On Learning Assumptions for Compositional Verification of Probabilistic Systems

Lu Feng
Trinity College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy in Computer Science
March 2013

Abstract

Probabilistic model checking is a powerful formal verification method that can ensure the correctness of real-life systems that exhibit stochastic behaviour. The work presented in this thesis aims to solve the *scalability* challenge of probabilistic model checking, by developing, for the first time, fully-automated compositional verification techniques for probabilistic systems. The contributions are novel approaches for automatically learning probabilistic assumptions for three different compositional verification frameworks.

The first framework considers systems that are modelled as Segala probabilistic automata, with assumptions captured by probabilistic safety properties. A fully-automated approach is developed to learn assumptions for various assume-guarantee rules, including an asymmetric rule (ASYM) for two-component systems, an asymmetric rule (ASYM-N) for n-component systems, and a circular rule (CIRC). This approach uses the L* and NL* algorithms for automata learning.

The second framework considers systems where the components are modelled as probabilistic I/O systems (PIOSs), with assumptions represented by Rabin probabilistic automata (RPAs). A new (complete) assume-guarantee rule (ASYM-PIOS) is proposed for this framework. In order to develop a fully-automated approach for learning assumptions and performing compositional verification based on the rule (ASYM-PIOS), a (semi-)algorithm to check language inclusion of RPAs and an L*-style learning method for RPAs are also proposed.

The third framework considers the compositional verification of discrete-time Markov chains (DTMCs) encoded in Boolean formulae, with assumptions represented as Interval DTMCs (IDTMCs). A new parallel operator for composing an IDTMC and a DTMC is defined, and a new (complete) assume-guarantee rule (ASYM-IDTMC) that uses this operator is proposed. A fully-automated approach is formulated to learn assumptions for rule (ASYM-IDTMC), using the CDNF learning algorithm and a new symbolic reachability analysis algorithm for IDTMCs.

All approaches proposed in this thesis have been implemented as prototype tools and applied to a range of benchmark case studies. Experimental results show that these approaches are helpful for automating the compositional verification of probabilistic systems through learning small assumptions, but may suffer from high computational complexity or even undecidability. The techniques developed in this thesis can assist in developing scalable verification frameworks for probabilistic models.

Acknowledgements

I would like to thank my supervisor Professor Marta Kwiatkowska, without whose support this thesis would not have been completed. Marta gave me a lot of guidance on doing research and writing papers. She was always available for discussion whenever I got stuck, giving me valuable insights and keeping me on the right track. She read every draft of my writing, provided suggestions for improvement, and even corrected my grammar and spelling mistakes. She was a wonderful supervisor who offered me opportunities to network with other researchers, for example, arranging for me to visit Dr Corina Pasareanu at NASA Ames Research Centre. Marta was also very helpful in my personal development: she wrote letters of recommendation for me; and when I competed at the UK ICT Pioneers Competition in London, she was there for me. Marta's attitude to science, enthusiasm for research, and patience with students will deeply influence me in my future academic career.

My thanks also go to Dr Dave Parker and Dr Tingting Han, who were coauthors of several papers with me. Dave was very patient with my endless questions about PRISM and taught me how to write scientific papers. Tingting helped me a great deal with the theoretical work, and we had many fruitful discussions.

I would like to express my appreciation of the EPSRC-funded UK Large Scale Complex IT Systems (LSCITS) initiative for their financial support. Special thanks go to Professor Dave Cliff, the director of LSCITS initiative, for giving me the opportunity to chair the LSCITS Postgraduate Workshop and writing recommendations for me.

I must also take this opportunity to thank all my colleagues at the Department of Computer Science and my friends at the Trinity College, for making my life pleasant and joyful during the past few years. Finally, I would like to thank my parents for their untiring love, support and encouragement.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

We are entering an era of ubiquitous computing, in which devices are becoming more and more autonomous. These smart devices and sensors are often connected through heterogeneous wired and wireless networks, and exhibit probabilistic behaviour due to the presence of failures (e.g. message loss in unreliable communication protocols) or the use of randomisation (e.g. symmetry breaker in distributed systems). Unfortunately, the breakdown of these devices, e.g. a bug in a piece of flight control software or a flaw in the medical cyber-physical systems, may lead to catastrophic risks and damages. *Formal verification* is an approach that provides mathematically rigorous guarantees about the correctness of such devices, by systematically exploring all the possible executions. In particular, *probabilistic model checking* is a formal verification technique that focuses on analysing quantitative, dependability properties of probabilistic systems, e.g. "what is the probability of an airbag failing to deploy?". Probabilistic model checking has been applied to a wide range of systems in many different application domains, from cloud computing [CKJ12], to biological cellular processes [KNP10], to communication protocols [DKN+10], to nanoscale computing devices [LPC+12], and

many more.

A key challenge of formal verification is *scalability*, because the increasing scale and complexity of real-life systems yields models that are orders of magnitude larger than those that are within the capacity of current techniques. A promising solution is to use *compositional verification*, in which the verification of large-scale complex systems is decomposed into sub-tasks of verifying each of its constituent components separately. In the non-probabilistic setting, several compositional verification frameworks (e.g. see [CGP03, PGB+08, CCF+10]) have been developed using the *assume-guarantee reasoning* approach. In assume-guarantee reasoning, each system component is verified in isolation under an *assumption* about its contextual environment; such assumptions can be generated automatically through algorithmic learning.

The compositional verification of probabilistic systems, however, only received scant attention prior to the start of my doctoral research. Kwiatkowska et al. [KNPQ10] proposed the first fully-automated assume-guarantee verification framework for Segala probabilistic automata, but this work requires non-trivial manual effort to derive assumptions, limiting its practical usage. To overcome this limitation, I proposed a novel approach to learn probabilistic assumptions automatically, which was published in a jointly authored paper [FKP10]. To the best of my knowledge, my doctoral research is pioneering work in the area of fully-automated assumption generation for compositional verification of probabilistic systems.

To date, as reported in this thesis, I have developed approaches for learning assumptions for three different probabilistic compositional verification frameworks, which differ not only in the format of their assume-guarantee rules, but also in the type of model and assumption. Apart from working with the framework proposed in [KNPQ10], I also actively contributed to the development of the other two frameworks.

A typical assume-guarantee reasoning rule looks like the following:

$$\frac{\mathcal{M}_1 \sim \mathcal{A} \qquad \mathcal{A} \| \mathcal{M}_2 \models \mathcal{G}}{\mathcal{M}_1 \| \mathcal{M}_2 \models \mathcal{G}}$$

In order to guarantee that a system composed from two components $\mathcal{M}_1$ and $\mathcal{M}_2$ satisfies a property $\mathcal{G}$, we need to

(a) find an appropriate assumption $\mathcal{A}$ related to one component, say $\mathcal{M}_1 \sim \mathcal{A}$,

(b) prove that $\mathcal{A} \| \mathcal{M}_2$ satisfies $\mathcal{G}$.

If both steps succeed, we can claim that property $\mathcal{G}$ is guaranteed on system $\mathcal{M}_1 \| \mathcal{M}_2$.

When defining a compositional verification framework, in order to formulate effective compositional reasoning rules, we need to carefully consider the models, assumptions, and the relations between them:

- the *model* (e.g. $\mathcal{M}_1$ and $\mathcal{M}_2$) should be realistic for real-life systems, and verifiable with efficient model checking techniques;
- the *assumption* (e.g. $\mathcal{A}$) should be expressive enough to capture the abstract behaviour of models and amenable to automatic generation via algorithmic learning;
- the *relation* (e.g. $\sim$) between assumptions and corresponding components should preserve compositionality, and be checkable with efficient procedures.

All approaches developed in my thesis follow the above guidelines. I have implemented these approaches as prototype tools and applied them to a range of benchmark case studies, which indicate that the results of my doctoral research can be used to improve the scalability of probabilistic model checking techniques.

1.2 Thesis Outline and Contributions

My doctoral research aims to develop fully-automated approaches for learning assumptions for compositional verification of probabilistic systems, with a focus on verifying probabilistic safety properties. This thesis reports the main results of my research. In the remainder of the thesis, I will review the related work in Chapter 2, introduce technical background in Chapter 3, present the contributions in Chapter 4-6, and lastly draw conclusions and point out potential directions for future work in Chapter 7. The following is a brief summary of the three main contributions.

Firstly, in Chapter 4, I propose a novel approach for learning assumptions that are represented as *probabilistic safety properties*, for the compositional verification framework of *asynchronous* probabilistic systems modelled as *Segala probabilistic automata*. This approach is adapted to handle three different assume-guarantee rules proposed in [KNPQ10]: an asymmetric rule for two-component systems (ASYM), an asymmetric rule for n-component systems (ASYM-N), and a circular rule (CIRC). In this chapter, I also consider and compare the use of two different learning algorithms, L* [Ang87a] for learning deterministic finite-state automata and NL* [BHKL09] for learning residual finite-state automata (i.e. a subclass of nondeterministic finite-state automata). A prototype tool is implemented and applied to several benchmark case studies such as a module from the flight software for JPL's Mars Exploration Rover, Aspnes & Herlihy's randomised consensus algorithm, and a sensor network model; small assumptions that enable the compositional verification of large models in these case studies are learnt successfully.

Secondly, in Chapter 5, I consider the learning of a class of more expressive assumptions formalised as *Rabin probabilistic automata* (RPAs). I first propose a new (complete) compositional verification framework that targets *synchronous* probabilistic systems composed of *probabilistic I/O systems* (PIOSs), i.e. an extension of *discrete*

Markov chains (DTMCs) with input/output actions. To build a fully-automated implementation of this new assume-guarantee rule (Asym-Pios), I develop a semi-algorithm for the undecidable problem of checking language inclusion between RPAs and a novel L*-style active learning algorithm for RPAs. A prototype tool is also implemented and experimental results on benchmark case studies, including the contract signing protocol and the bounded retransmission protocol, are reported in this chapter.

Thirdly, in Chapter 6, I develop a novel approach for learning *implicit assumptions* for the compositional verification of DTMC models. I formulate the notion of assumptions as *interval DTMCs* (IDTMCs), which are encoded implicitly as Boolean functions using an eager encoding scheme, and present a new (complete) assume-guarantee rule (Asym-Idtmc) for verifying probabilistic safety properties on DTMCs compositionally. I also propose a symbolic value-iteration algorithm for computing reachability probabilities on IDTMCs using the data structure of multi-terminal binary decision diagrams (MTBDDs). A fully-automated implementation is built based on the rule (Asym-Idtmc) using the CDNF learning algorithm [Bsh95], and applied to case studies such as the contract signing protocol and a client-server model.

1.3 Other Publications and Credits

Some of the work in this thesis has previously been published in jointly authored papers. However, I present the relevant content in the thesis in my own words and point out the credits of the technical chapters below.

Chapter 4 is an extension of [FKP10] and [FKP11], which are jointly authored papers with Marta Kwiatkowska and David Parker. The compositional verification framework presented in Section 4.1 was originally proposed by Marta Kwiatkowska and David Parker et al. in [KNPQ10]. The probabilistic counterexample for PAs in Section 4.2 first appeared in [FKP10] and was David Parker's contribution. I worked

actively with David Parker on the learning approach for rule (Asym) [FKP10], and rewrote this part in Section 4.3 based on my own understanding. Among the three extensions presented in Section 4.4, the use of NL* algorithm and the generalisation of learning for (Asym-N) were briefly mentioned in [FKP11], and are extended here with more details and additional running examples; the generalisation of learning for rule (Circ) is new. The prototype implementation and experiments of case studies reported in Section 4.5 were my work, and David Parker helped me with some of the case studies.

Chapter 5 is based on [FHKP11a] and the technical report [FHKP11b]. I worked jointly with Tingting Han, Marta Kwiatkowska and David Parker on the compositional verification framework, and presented in Section 5.1 with my own words. The semi-algorithm of checking language inclusion for RPAs in Section 5.2 and the L*-style learning for RPAs in Section 5.3 were my independent work; the presentation in the thesis contains more details than in the paper (e.g. proofs and running examples). I was also solely responsible for the development of the assumption learning approach in Section 5.4 and the prototype implementation in Section 5.5.

Chapter 6 is based on recent joint work with Tingting Han, Marta Kwiatkowska and David Parker, and has not been published yet. The encoding scheme of translating probabilistic models into Boolean functions presented in Section 6.1 was my independent work. I worked actively with Tingting Han and David Parker on defining the compositional verification framework presented in Section 6.2. The value iteration algorithm proposed in Section 6.3 was worked out jointly with Tingting Han, and the symbolic algorithm based on MTBDDs was my own work. I was solely responsible for the learning approach in Section 6.4 and the prototype implementation in Section 6.5.

Chapter 2

Review of Related Work

This chapter provides a review of the related work. Section 2.1 covers various topics of model checking, Section 2.2 provides a bibliography note of relevant learning algorithms, and Section 2.3 introduces applications of using learning in model checking.

2.1 Model Checking

In this section, we survey the main model checking techniques that are relevant to this thesis. We start with the techniques of checking temporal properties for models such as labelled transition systems, discrete-time Markov chains, Markov decision processes, probabilistic automata and interval discrete-time Markov chains. Then, we introduce preorder and equivalence relations such as simulation and bisimulation for comparing the behaviour of models. Moreover, we also review the techniques of compositional verification and counterexample generation.

2.1.1 Property Checking

Model checking is a formal verification technique for automatic verification of finite-state systems against correctness properties, where systems are usually modelled as

labelled transition systems (Kripke structures) and properties are often specified in temporal logics. A model checker exhaustively explores all relevant execution paths from the initial state in a model and analyses whether these paths satisfy a given property specification. Model checking was pioneered independently by Clarke and Emerson [CE81] and by Queille and Sifakis [QS82] in the early 1980's; more recently, Clarke, Emerson and Sifakis shared the 2007 A.M. Turing Award for their original work in this area. Today, model checking is widely used to detect and diagnose errors in hardware and software designs. However, the main difficulty with model checking is the *state explosion problem*, i.e. the number of states grows exponentially with the number of components, while model checking algorithms are linear or worse in the number of such states. Hence, apply model checking to real-world problems which often have enormous state spaces can be difficult. Several approaches have been proposed to address this problem, such as symbolic model checking with BDDs [BCM$^+$90], bounded model checking using SAT solvers [CBRZ01], and counterexample-guided abstraction refinement (CEGAR) [CGJ$^+$00]. Compositional verification, which is the main focus of this thesis, is another popular approach to tackle the state explosion problem.

Since many real-world systems exhibit probabilistic behaviour (e.g. due to randomisation and uncertainty), model checking has been extended to handle probabilistic models, and is referred to as *probabilistic model checking*. The simplest probabilistic model is the *discrete-time Markov chains* (DTMCs), modelling purely probabilistic systems. DTMCs can be considered as an extension of labelled transition systems, in which each labelled transition leads to a set of states based on a probabilistic distribution rather than a single state. Model checking DTMCs involves computing the exact probability of a set of paths satisfying a given property, which can be expressed in *linear-time logic*, e.g. linear temporal logic (LTL), or *branching-time logic*, e.g. probabilistic computation tree logic (PCTL). The first algorithm for LTL model checking for DTMCs was proposed by Vardi [Var85], and an improved algorithm was developed later by

Courcoubetis and Yannakakis [CY88, CY95]. The algorithm for PCTL model checking for DTMCs was proposed by Hansson and Jonsson [HJ94], where the problem was reduced to computing the reachability probability via solving a linear equation system.

A more complex type of probabilistic models is the *Markov decision processes* (MDPs), which extend DTMCs with nondeterminism and can be used to model systems exhibiting both probabilistic and nondeterministic behaviour. Nondeterminism is an essential tool for modelling concurrency and underspecification. A transition between states in an MDP occurs in two steps: firstly, a nondeterministic choice between one or more transition labels is made; and then, a probabilistic choice of successor states is taken similarly as in a DTMC. Model checking MDPs relies on the resolution of nondeterminism via *adversaries*. An adversary picks one enabled labelling action for each state in an MDP. Under a given adversary, the behaviour of an MDP is purely probabilistic and induces a DTMC. We check whether a given temporal logic property holds in an MDP by computing the minimum and maximum probability of satisfying the property over all adversaries. Model checking algorithms for probabilistic properties of MDPs were first proposed by Courcoubetis and Yannakakis [CY90] and by Bianco and de Alfaro [BdA95], and involved graph-based analysis in conjunction with *linear programming*. An alternative approach is *value iteration* [Put94], which provides approximate results but has better scalability than linear programming. Later, Parker [Par02] demonstrated that MDPs can be verified efficiently via an implementation of value iteration with symbolic data structures. In addition, Etessami and Kwiatkowska et al. [EKVY07] proposed *multi-objective* model checking techniques, enabling the analysis of trade-offs between multiple linear-time properties of MDPs. Multi-objective model checking for MDPs reduces to solving a linear programming problem.

In this thesis, we also consider the model checking of *probabilistic automata* (PAs) defined by Segala [Seg95], which is a slight generalisation of MDPs. The essential difference between PAs and MDPs is that a state of a PA permits the nondeterminism

between equally-labelled outgoing transitions. Indeed, MDPs can be considered as a subclass of PAs. The model checking algorithms for MDPs can be adapted for PAs.

Interval discrete-time Markov chains (IDTMCs) is another type of models that can be used to capture the uncertainty of probabilistic systems. IDTMCs are extensions of DTMCs in the sense that the exact transition probabilities are unspecified but restricted by intervals. There are two common semantic interpretations of IDTMCs: *uncertain Markov chains* (UMCs) [JL91] and *interval Markov decision processes* (IMDPs) [KU02]. With the UMC interpretation, an IDTMC is considered as a family of DTMCs whose transition probabilities lie within the transition intervals of the IDTMC; once a DTMC is picked at the very beginning, the behaviour of IDTMC follows the chosen DTMC. By contrast, with the IMDP interpretation, the uncertainty of IDTMC is resolved step by step; each time a state is visited, a transition distribution is picked nondeterministically based on the given intervals. Sen et al [SVA06, CSH08] studied the problem of PCTL model checking for IDTMCs (with both UMC and IMDP interpretations), and the complexity results of the problem have been improved recently by Chen, Han and Kwiatkowska [CHK13]. For the LTL model checking of IDTMCs, a practical algorithm based on the expectation-maximisation procedure was proposed by Benedikt, Lenhardt and Worrell [BLW13]. Note that there are also models that can be considered as variants of IDTMCs, for example, three-valued abstraction of DTMCs [KKLW12], Constraint Markov Chains [CDL$^+$11] and Markov set-chains [Har98, HS94]. IDTMCs were also studied in the representation of a central DTMC with an interval confidence score matrix for the purpose of approximate model checking in [GDK$^+$12].

There are several software tools available for probabilistic model checking, such as PRISM [KNP11], LiQuor [CB06] and MRMC [KZH$^+$09]. A extensive list of these tools can be found at http://www.prismmodelchecker.org/other-tools.php. This thesis mainly uses PRISM, which is one of the most widely used probabilistic model checkers. PRISM supports the model checking of DTMCs, MDPs and PAs as reviewed above.

It can also be used to analyse continuous-time models such as continuous-time Markov chains (CTMCs) and probabilistic timed automata (PTAs), which are not considered in this thesis. PRISM implements state-of-the-art symbolic techniques for probabilistic model checking, based on the data structures of BDDs (Binary Decision Diagrams) [Bry92] and MTBDDs (Multi-Terminal Binary Decision Diagrams) [KNP04, Par02].

2.1.2 Preorder and Equivalence Relations

We compare the behaviour of two models via preorder and equivalence relations. Such relations can also serve as a basis of compositional verification, if they preserve linear-time or branching-time properties through composition.

Given two labelled transition systems *TS* and *TS'*, if all (infinite) traces of *TS* are exhibited in *TS'*, then we say that there is a *trace inclusion* preorder relation between *TS* and *TS'*. Note that *TS'* may have some traces that *TS* does not have. If all traces of *TS'* are also exhibited in *TS*, then these two models are related by *trace equivalence*. The trace inclusion and equivalence relations preserve linear-time properties. For example, given trace-equivalent *TS* and *TS'*, if *TS* satisfies a linear-time property P, then *TS'* also satisfies P. Trace theory was first introduced by Hoare [Hoa78] and has been further developed over the past decades. A comprehensive survey of trace inclusion and trace equivalence notions was given by Bruda [Bru04].

On the other hand, *simulation* preorder and *bisimulation* equivalence relations are often used to compare the branching-time behaviour of labelled transition systems. Simulation between two labelled transition systems *TS* and *TS'* requires that the initial state of *TS* can mimic all stepwise behaviour of the initial state of *TS'*, whereas bisimulation requires *TS* and *TS'* mutually mimic all transitions. There are *strong* and *weak* variants of such relations: strong relations consider every transition, while weak relations only take into account "observable" transitions but ignore "silent" (or "internal") transitions. The concepts of simulation and bisimulation were originated by

Milner [Mil71, Mil80], and efficient decision algorithms for simulation preorders were proposed in [BP95] and [HHK95]. A comprehensive comparison of various trace-based and (bi)simulation relations was provided by van Glabbeek [vG90, vG93].

Simulation and bisimulation relations were extended for probabilistic models such as DTMCs by Jonsson and Larsen [JL91], which require the distributions of two DTMCs to have a stepwise mapping via *weight functions*. Segala and Lynch [SL95] further extended simulation and bisimulation relations to PAs, where each distribution in a PA must have a corresponding equal-labelled distribution in the other related PA; they also define the concept of *probabilistic simulation*, which is less strict than simulation in the sense that the stepwise mapping is allowed between a convex combination of multiple distributions rather than a single distribution. Effective decision algorithms for (strong) simulation relations for both DTMCs and PAs have been developed [Zha09]. Segala [Seg95] also studied trace-based relations for PAs, but he observed that *trace distribution inclusion*, which is a conservative extension of trace inclusion preorder in labelled transition systems, is not a precongruence and hence not compositional. The simulation relation for IDTMCs was first introduced by Jonsson and Larsen [JL91], but the problem of deciding whether there is a simulation relation between two given IDTMCs had remained open until recently [DLL$^+$11].

2.1.3 Compositional Verification

Compositional verification addresses the state explosion problem of model checking by decomposing systems into components and verifying each component separately under contextual assumptions. The compositional verification techniques of *non-probabilistic* systems via *assume-guarantee reasoning* have been widely studied, e.g. in [EDK89, GL94, Jon83, Pnu85].

For probabilistic systems, a number of compositional verification techniques have also been developed. Segala and Lynch [Seg95, SL95] proposed several compositional

proof techniques for probabilistic automata (PAs) based on variants of simulation relations. There are also compositional frameworks built on top of trace-based relations, for example, the synchronous parallel composition of probabilistic Reactive Modules by de Alfaro et al. [dAHJ01] and the switched I/O automata by Cheung et al. [CLSV04]. However, none of these techniques has presently a practical implementation.

Kwiatkowska et al. [KNPQ10] proposed the first fully-automated assume-guarantee verification framework for PAs based on multi-objective model checking techniques [EKVY07]. This framework focused on the verification of probabilistic safety properties, and was further extended in [FKN$^+$11] for a richer class of properties such as liveness. A limitation of these approaches is that they require non-trivial human effort to find appropriate assumptions for assume-guarantee reasoning.

2.1.4 Counterexample Generation

Counterexamples are of crucial importance in model checking, because they provide valuable diagnostic feedback about the property violation; they are also essential for techniques such as counterexample-guided abstraction refinement and learning-based assume-guarantee model checking. In the non-probabilistic setting, a counterexample is usually a single path leading to some "bad" state. The counterexample generation techniques for non-probabilistic systems have been investigated extensively, see e.g. [dAHM00, BNR03, JRS04].

By contrast, counterexamples for probabilistic model checking are often more complex and sometimes cannot be captured by a single path due to the stochastic nature of probabilistic models. Han et al. [HKD09] defined a probabilistic counterexample as a set of paths, with the sum of path probabilities indicating the violation of a property, and proposed counterexample generation techniques for LTL and PCTL model checking of probabilistic models such as DTMCs and MDPs. An alternative method was proposed by Aljazzar et al. in [AL10], where probabilistic counterexamples are represented

in a more compact format of diagnostic subgraphs; based on this method, a practical tool named DiPro [ALFLS11] was implemented. Fecher and Huth et al. [FHPW10] considered counterexamples in the branching-time setting, and proposed a game-theoretic method in which diagnostic information for truth and falsity of model checking PCTL on countable labeled Markov chains are encoded as monotone strategies. More recently, Braitling and Wimmer et al. [BWB+11] proposed a counterexample generation method for DTMCs using SMT-based bounded model checking.

2.2 Learning

Grammatical inference [dlH10], also called *learning*, refers to the problem of learning a formal grammar (usually represented by automata) that characterises a set of given data (e.g. strings, trees, graphs). From the wide spectrum of learning techniques, we survey three topics relevant to this thesis: learning finite-state automata, learning probabilistic automata and learning Boolean functions. We consider techniques that learn from strings only (rather than trees or graphs).

2.2.1 Learning Finite-State Automata

The learnability of *finite-state automata* was first studied by Gold [Gol67] under the model of *identification in the limit* (i.e. a learning algorithm identifies an exactly correct grammar for the given data after a finite number of wrong hypotheses). Gold [Gol78] showed that the representation class of *deterministic finite-state automata* (DFAs) is identified in the limit in polynomial time and data, while de la Higuera [dlH97] proved a negative result for *nondeterministic finite-state automata* (NFAs). Apart from the learning model of identification in the limit, Valiant [Val84] proposed the *probably approximately correct* (PAC) learning model, which learns a grammar with low generalization error ("approximately correct") given any arbitrary distribution of the samples.

In this thesis, we are interested in learning exact grammars rather than PAC learning, because, to enable the assume-guarantee verification, we need to learn the exact representation of assumptions.

Biermann and Feldman [BF72] proposed one of the first *offline* algorithms that learn a DFA from a *fixed set* of examples, such that the learnt automaton accepts the positive examples and rejects the negative examples. Another well-known offline algorithm for learning DFAs is the RPNI algorithm proposed in [OG92]. An alternatively approach is the *online* algorithms that work incrementally and have the possibility of asking for further examples in the learning process, for example, the L* algorithm proposed by Angluin [Ang87a] that learns a *minimal* DFA for a regular language in polynomial time. The L* algorithm was later improved by Rivest and Schapire [RS93]. Angluin's work pioneered a classical online learning model named the *minimal adequate teacher* (MAT), in which a *teacher* capable of answering certain types of queries (e.g. membership and equivalence queires) proposed by the learning algorithm is employed. All the learning algorithms that we will use in later chapters of this thesis follow the MAT model; we also call them *active* learning algorithms since they are actively asking queries.

The learning of NFAs is hard since the class of NFAs does not have the characterisation of right-congruence, i.e. there is no unique minimal NFA for a given regular language. Research in this area has been mainly focused on a subclass of NFAs called *residual finite-state automata* (RFSAs), which exhibit the properties of right-congruence. For learning RFSAs, an offline algorithm named DeLeTe2 and an online algorithm named NL* (following the MAT model of L*) were proposed in [DLT04] and [BHKL09], respectively.

2.2.2 Learning Probabilistic Automata

Many learning techniques have been developed to learn probabilistic automata. Many methods aim to learn deterministic probabilistic automata (PDFAs) that represent a

probabilistic distribution over finite words. For example, [CO94] proposed the well-known ALERGIA algorithm to learn PDFAs using a state merging method, and [dlHO04] discussed the active learning of PDFAs with queries. In addition, Tzeng [Tze92a] presented an approach to learn a more general class of probabilistic automata via asking queries that require the state space of the target automaton to be known as a priori.

My co-authored paper [FHKP11a] proposed a novel active learning method for Rabin probabilistic automata (RPAs) [Rab63] (with all states accepting). Our method was inspired by [BV96], which considered the active learning of *multiplicity automata* (a generalisation of RPAs by replacing transition probabilities with arbitrary rationals), following the MAT learning model.

A recent paper [MCJ+11] proposed a variant of the ALERGIA algorithm to learn probabilistic automata from a sequence of observed system behaviours. In comparison with our method [FHKP11a], their work follows the style of *black-box* learning, i.e. the target is a black-box system and the learning can only provide approximate results, whereas our method is *white-box* learning and the learnt result is exactly correct.

2.2.3 Learning Boolean Functions

There are many results concerning the learnability of Boolean functions: for example, [AK91, AHP92] showed that learning Boolean functions in polynomial time w.r.t. the size of the *disjunctive normal form* (DNF) and the number of variables is hard (i.e. not learnable with membership and equivalence queries), and [Ang87b, AP92, BR92] investigated subclasses of Boolean functions that are learnable in their DNF representations.

Bshouty [Bsh95] developed an active learning method, called the CDNF algorithm, based on monotone Boolean functions. The CDNF algorithm follows Angluin's MAT learning model and interacts with a teacher via asking membership and equivalence

queries. It learns any arbitrary Boolean function with a polynomial number of queries in the sizes of its corresponding minimal DNF, minimal CNF (conjunctive normal form) and the variable set. Bshouty's CDNF learning algorithm assumes that the target Boolean function is over a fixed set of variables. A recent work [CW12] improved this algorithm and proposed an incremental method of learning Boolean functions over indefinitely many variables, which is essential for real-world applications such as loop invariant generation where the variable set is not fixed.

2.3 Using Learning in Model Checking

Over the past decade, learning techniques have become popular in the domain of model checking to address problems such as learning system models and learning program invariants. In this section, we review a number of such applications, with emphasis on the learning of assumptions for compositional verification since this is the main topic of this thesis.

2.3.1 Learning Assumptions for Compositional Verification

Compositional verification, as mentioned in Section 2.1.3, is a promising approach to tackle the state explosion problem of model checking. However, in practice, it often requires a considerable human effort to derive *assumptions* for the assume-guarantee reasoning of composition verification.

Cobleigh, Giannakopoulou and Pasareanu [CGP03] proposed an automatic method to generate assumptions, where the L* learning algorithm [RS93] was applied to learning assumptions for an asymmetric assume-guarantee rule of two-component labelled transition systems. This work has been generalised for the verification of systems with n components and for learning assumptions of other assume-guarantee rules (e.g. symmetric and circular) in [BGP03, PG06, GGP07, PGB+08].

Following the seminal work of [CGP03], the research on learning assumptions for compositional verification in the *non-probabilistic* setting has flourished. For example, [CS07] suggested a few optimisations to the L*-based assumption generation method, including the computation of a minimal alphabet for assumptions; [NMA08] presented an assumption generation approach using the symbolic BDD implementation of L*; [CFC+09] formulated the assumption generation problem as the learning of the smallest automaton separating two regular languages, and proposed a new learning algorithm named L^{Sep} to learn such minimal separating automata; [CCF+10] proposed an approach for generating assumptions encoded implicitly as Boolean functions using the CDNF learning algorithm [Bsh95]; and [LLS+12] considered the assume-guarantee verification of timed systems, using the TL* learning algorithm [LAD+11] to generate timed assumptions modelled as event-recording automata.

For probabilistic systems, however, this problem remained open until 2010 when my co-authored paper [FKP10] was published. In this work, we proposed an approach for learning assumptions, represented as *probabilistic safety properties*, for an asymmetric assume-guarantee rule of two-component systems modelled as probabilistic automata (PAs) using the L* learning algorithm. We further extended this approach in [FKP11] for the compositional verification of systems with n components, and investigated the use of an alternative learning algorithm named NL* [BHKL09], which learns a subclass of nondeterministic finite automata. Later, we studied learning of a more expressive class of assumptions represented as *probabilistic finite automata* in [FHKP11a], for the compositional verification of systems modelled as discrete-time Markov chains.

The research into the automated generation of probabilistic assumptions has received increasing attention recently. Apart from our work, Komuravelli, Pasareanu and Clarke [KPC12a, KPC12b] considered the compositional verification of PAs with respect to strong simulation conformance. In [KPC12b], they proposed a semi-algorithm for learning PAs from tree samples and applied it to learn assumptions; however,

no practical implementation or experimental results of this work were reported. In [KPC12a], they presented an alternative way of generating assumptions based on the use of *abstraction refinement* instead of learning. Although [KPC12a] reported experimental results on the same set of benchmark case studies as those used in our work [FKP10, FKP11], we cannot compare their results with ours directly because [KPC12a] considered strong simulation relations between PAs, whereas our work verified PAs against probabilistic safety properties.

2.3.2 Other Applications

We also mention some other popular applications of learning in the domain of model checking, such as learning system models, mining specifications and learning invariants for program analysis.

A major barrier to the practicality of model checking is that system models and specifications are often unknown or inaccurate. [PVY99] developed a method for model checking black-box systems, where models are unknown, by firstly learning an initial model through observing system executions and then refining the model with counterexamples provided by model checking. [GPY02] proposed an approach named *adaptive model checking*, which is capable of updating an inaccurate model automatically by learning from counterexamples. Along similar lines, several other methods have been developed to learn various models using statistical learning techniques in the context of model checking, see e.g. [SVA04, MCJ+11, MJ12, CMJ+12]. In addition, [CJR11] proposed a new Bayesian technique to learn state transition probabilities of discrete-time Markov chains for the model checking of quality-of-service properties.

Learning has also been applied in software verification for mining specifications: for example, [ABL02] developed a mining tool which learns temporal specifications from the static or runtime behaviour of programs, [AvMN05] used the L* algorithm to synthesise interface specifications for Java classes, and [GS08] proposed a symbolic

BDD-based approach which expands the tractability of the specification mining by orders of magnitude.

More recent work applied learning techniques to automatically find loop invariants, which is one of the most important problems in the static analysis of imperative programs. [JKWY10] proposed an automated technique for finding loop invariants in propositional formulae using the CDNF learning algorithm that we also employ in this thesis. This work was extended for solving the predicate generation problem in [JLWY11] and for program termination analysis in [LWY12].

Chapter 3

Preliminaries

This chapter introduces the necessary technical background material for this thesis. Section 3.1 reviews finite-state automata and regular languages. Section 3.2 covers three different types of probabilistic models, including discrete-time Markov chains, interval discrete-time Markov chains and Segala probabilistic automata. Section 3.3 shows how properties can be specified and verified on probabilistic models. Section 3.4 introduces the concept of probabilistic counterexamples. Section 3.5 describes active learning algorithms relevant for the remainder of the thesis. Finally, Section 3.6 reviews two settings for the learning of non-probabilistic assumptions for compositional verification.

The material presented in this chapter is mainly based on the textbook [BK08] and the tutorial paper [FKNP11], as well as various other papers that will be cited later.

3.1 Formal Languages and Finite-State Automata

First of all, we review some basic concepts of formal languages used in this thesis. Let $\mathbb{N} = \{0, 1, 2, \dots\}$ be the set of natural numbers. An *alphabet* α is a finite non-empty set of symbols. The elements of α are called *letters*. An *infinite word* over α is an infinite sequence of letters $a_0 a_1 a_2 \dots$ where $a_i \in \alpha$ for all $i \in \mathbb{N}$; and similarly, a *finite word*

over α is a finite sequence $a_0 a_1 \dots a_n$. The length of a finite word w, denoted by $|w|$, is the number of letters in it. A special case of $|w| = 0$ is allowed; we call it *empty word* and denote it by ϵ. The *concatenation* of two words u and v, denoted by $u \cdot v$ or uv, yields a new word w of length $|u| + |v|$ such that

$$w[i] = \begin{cases} u[i] & \text{if } 0 \leq i < |u| \\ v[i] & \text{if } |u| \leq i < |u| + |v|. \end{cases}$$

We denote by α^ω and α^* the set of all infinite and finite words over alpabet α, respectively. A *language* $\mathcal{L}$ over α contains a set of finite words $w \in \alpha^*$. Two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are *equivalent* if they contain the same set of finite words; and, if $\mathcal{L}_1 \subseteq \mathcal{L}_2$, then they are related by *language inclusion*. The concatenation of words is lifted to languages such that $\mathcal{L}_1 \cdot \mathcal{L}_2 = \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2\}$.

Next, we introduce the classical concept of *finite-state automata*, which accept finite words. Note that the automata on infinite words, e.g. deterministic Rabin automata, will be introduced later (see Definition 3.10).

Definition 3.1 (NFA/DFA) *A* nondeterministic finite automaton *(NFA) is a tuple* $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$ *where* S *is a finite set of states,* $\overline{s} \in S$ *is an initial state,* α *is an alphabet,* $\delta \subseteq S \times \alpha \times S$ *is a transition relation, and* $F \subseteq S$ *is a set of accepting states.* $\mathcal{A}$ *is a* deterministic finite automaton *(DFA) if* $|\delta(s, a)| \leq 1$ *for all states* $s \in S$ *and actions* $a \in \alpha$*, where* $|\delta(s, a)|$ *is the size of the set* $\delta(s, a)$*.* $\mathcal{A}$ *is called a* complete *DFA if* $|\delta(s, a)| = 1$ *for all* $s \in S$ *and* $a \in \alpha$*.*

We also use $s \xrightarrow{a} s'$ to represent a transition $\delta(s, a) = s'$. A *run* of $\mathcal{A}$ is a finite sequence of transitions $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ where $s_0 = \overline{s}, s_i \in S$ and $a_i \in \alpha$ for all $i \in \mathbb{N}$; it is called *accepting* if $s_n \in F$. We call the finite sequence of observable actions $a_0 a_1 \dots a_{n-1}$ a *trace* of $\mathcal{A}$. A language accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is a set of finite words such that each word $w \in \mathcal{L}(\mathcal{A})$ corresponds to the trace of an accepting run

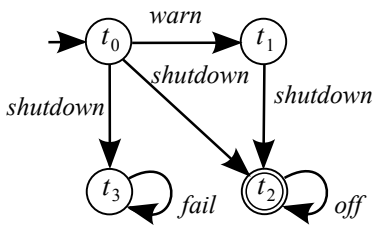


Figure 3.1: A 4-state NFA $\mathcal{A}$

of $\mathcal{A}$. A language is *regular* if and only if it is accepted by some finite-state automaton. Two automata $\mathcal{A}_1, \mathcal{A}_2$ are *equivalent* if they accept the same language, i.e. $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$. Given a NFA $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$, we can apply the standard subset construction algorithm [RS59] to convert it into an equivalent DFA $\mathcal{A}' = (S', \overline{s}', \alpha, \delta', F')$, where $S' = \{Q : Q \subseteq S\}$, $\overline{s}' = \{\overline{s}\}$, $\delta'(Q, a) = \bigcup_{s \in Q} \delta(s, a)$ for all $Q \subseteq S$ and $a \in \alpha$, and $F' = \{Q \subseteq S : Q \cap F \neq \emptyset\}$. We call a DFA *minimal* if there is no equivalent DFA with strictly fewer states. For each regular language $\mathcal{L}$, there exists a unique minimal DFA $\mathcal{A}$ that accepts it, i.e. $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

Example 3.2 *Figure 3.1 shows an NFA $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$ with four states $S = \{t_0, t_1, t_2, t_3\}$ and alphabet $\alpha = \{warn, shutdown, fail, off\}$. It represents a device that powers down correctly if it receives a "warn" signal before a "shutdown" command; otherwise, the device may "fail" upon receiving "shutdown" in the initial state $\overline{s} = t_0$. The transition relation δ is defined by:*

$$\delta(t_0, warn) = \{t_1\} \qquad \delta(t_0, shutdown) = \{t_2, t_3\} \qquad \delta(t_2, off) = \{t_2\}$$
$$\delta(t_3, fail) = \{t_3\} \qquad \delta(t_1, shutdown) = \{t_2\}$$

The set $F = \{t_2\}$ is accepting (the accepting state t_2 is drawn with a double circle). The accepting runs of $\mathcal{A}$ include, for instance, $t_0t_1t_2$ over a word $\langle warn, shutdown \rangle$ and $t_0t_2t_2$ over a word $\langle shutdown, off \rangle$. However, the word $\langle shutdown, fail \rangle$ would not be accepted by $\mathcal{A}$, because there is no corresponding run containing t_2. If we remove the

transition $t_0 \overset{shutdown}{\longrightarrow} t_2$*, then* $\mathcal{A}$ *becomes a DFA.*

3.2 Probabilistic Models

Here we introduce probabilistic models which, in contrast to finite-state automata, incorporate information about the likelihood of transitions occurring between states. We consider three different types of probabilistic models in this thesis: discrete-time Markov chains, interval discrete-time Markov chains and Segala probabilistic automata.

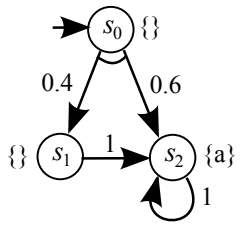
For the rest of this thesis, we denote by $Dist(S)$ the set of probability distributions over a set S, i.e. functions $\mu : S \to [0,1]$ satisfying $\sum_{s \in S} \mu(s) = 1$, and by $SDist(S)$ the set of *sub-distributions* over S, i.e. functions $\mu : S \to [0,1]$ satisfying $\sum_{s \in S} \mu(s) \leq 1$. We use η_s for the point distribution on $s \in S$ and $\mu_1 \times \mu_2$ for the product distribution of μ_1 and μ_2, defined by $\mu_1 \times \mu_2((s_1, s_2)) \stackrel{\text{def}}{=} \mu_1(s_1) \cdot \mu_2(s_2)$. We assume probabilities are rational numbers.

3.2.1 Discrete-Time Markov Chains

We first consider so called *discrete-time Markov chains* (DTMCs), which model fully probabilistic systems.

Definition 3.3 (DTMC) *A* discrete-time Markov chain *is a tuple* $\mathcal{D} = (S, \overline{s}, \alpha, \delta, L)$ *where* S *is a finite set of states,* $\overline{s} \in S$ *is an initial state,* α *is an alphabet,* $\delta : S \times (\alpha \cup \{\tau\}) \to Dist(S)$ *is a (partial) transition relation such that, for any state* $s \in S$*,* $\delta(s, a)$ *is defined for at most one action* $a \in \alpha \cup \{\tau\}$*, where* τ *denotes a "silent" (or "internal") action, and* $L : S \to 2^{AP}$ *is a labelling function mapping states to atomic propositions from a set* AP*.*

The behaviour of a DTMC $\mathcal{D}$ in each state s is represented by the function δ. If $\delta(s, a) = \mu$, then the DTMC can make a transition, labelled with action a, and move to state s' with probability $\mu(s')$. Since $\delta(s, a)$ is defined for at most one action $a \in \alpha \cup \{\tau\}$



$$\mathbf{P} = \begin{pmatrix} 0 & 0.4 & 0.6 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3.2: A 3-state DTMC $\mathcal{D}$ and its transition probability matrix $\mathbf{P}$

in each state s, the sum of probabilities of all outgoing transitions from s equals one. Therefore, we can use a matrix $\mathbf{P} : S \times S \to [0, 1]$ to represent the transition probabilities between states, such that $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all states $s \in S$. Sometimes we also write a DTMC as $\mathcal{D} = (S, \overline{s}, \mathbf{P}, L)$ with the alphabet α of action labels omitted.

An *infinite path* through a DTMC $\mathcal{D}$ is a sequence $\pi = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ where $\delta(s_i, a_i) > 0$ (also written $\mathbf{P}(s_i, s_{i+1}) > 0$) for all $i \in \mathbb{N}$. The *trace* of a path π, denoted $tr(\pi)$, is defined as the sequence of action labels $a_0 a_1 \ldots$ after removal of any "silent" τ actions. A *finite path* $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ is a prefix of an infinite path ending in a state s_n. We use $IPaths_{\mathcal{D},s}$ and $FPaths_{\mathcal{D},s}$ to denote the set of all infinite and finite paths starting from state s of $\mathcal{D}$, respectively.

To reason about the behaviour of $\mathcal{D}$, we need to determine the probability of certain paths occurring. In the following, we construct a *probability space* over the set of infinite paths $IPaths_{\mathcal{D},s}$ for each state s. We start by defining the probability of a finite path $\rho = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} s_n$ as $\mathbf{P}(\rho) \stackrel{\text{def}}{=} \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$. For each finite path $\rho \in FPaths_{\mathcal{D},s}$, we define a *basic cylinder* C_ρ which consists of all infinite paths starting with ρ as a prefix. Using properties of cylinders [KSK76], we can then construct the probability space $(IPaths_{\mathcal{D},s}, \mathcal{F}_{\mathcal{D},s}, Pr_{\mathcal{D},s})$ where $\mathcal{F}_{\mathcal{D},s}$ is the smallest σ-algebra generated by the basic cylinder sets $\{C_\rho | \rho \in FPaths_{\mathcal{D},s}\}$ and $Pr_{\mathcal{D},s}$ is the unique measure such that $Pr_{\mathcal{D},s}(C_\rho) = \mathbf{P}(\rho)$ for all finite paths $\rho \in FPaths_{\mathcal{D},s}$.

Example 3.4 *Figure 3.2 shows a DTMC $\mathcal{D}$ and its transition probability matrix $\mathbf{P}$. The states set is $S = \{s_0, s_1, s_2\}$, and the initial state s_0 is indicated by an incoming*

arrow. The labelling function L is given by $L(s_0) = L(s_1) = \emptyset$ and $L(s_2) = \{a\}$ where a is an atomic proposition. The transition action labels are omitted. We give an example of a finite path as $\rho_1 = s_0 \to s_1 \to s_2$ with the path probability $\mathbf{P}(\rho_1) = 0.4 \times 1 = 0.4$. And $\pi_1 = s_0 \to s_1 \to s_2 \to s_2 \to \cdots$ is an infinite path starting with ρ_1 as a prefix.

3.2.2 Probabilistic Automata

We consider a second type of model, *probabilistic automata* (PAs) [Seg95], for systems that exhibit both probabilistic and nondeterministic behaviour. To distinguish this model with the probabilistic automata defined by Rabin [Rab63], which will be used in Chapter 5, we call Segala's model as PAs and Rabin's model as RPAs in this thesis.

Definition 3.5 (PA) *A* probabilistic automaton *is a tuple $\mathcal{M} = (S, \overline{s}, \alpha, \delta, L)$ where S is a set of states, $\overline{s} \in S$ is an initial state, α is an alphabet, $\delta \subseteq S \times (\alpha \cup \{\tau\}) \times Dist(S)$ is a probabilistic transition relation with τ a "silent" action, and $L : S \to 2^{AP}$ is a labelling function mapping states to a set AP of atomic propositions.*

One or more *transitions*, denoted $s \xrightarrow{a} \mu$, are available in a state s of a PA $\mathcal{M}$, where $a \in \alpha \cup \{\tau\}$ is an action label, μ is a probability distribution over states and $(s, a, \mu) \in \delta$. There may exist multiple outgoing distributions in a state labelled with the same action. Note that a popularly studied model called *Markov decision processes* (MDPs) is a subclass of PAs, where a state in an MDP can have at most one outgoing probability distribution for a given action.

In each step of an execution of the PA, firstly, a *nondeterministic* choice is made between available transitions from s, and then a *probabilistic* choice of successor state is determined according to the chosen distribution μ. It is possible to allow μ to be sub-distributions (i.e. sum to less than 1), with the interpretation that a PA may choose to deadlock in the source state with certain probability [Seg95]. We call such models *sub-stochastic PAs*.

An *infinite path* through the PA $\mathcal{M}$ is a sequence $\pi = s_0 \xrightarrow{a_0,\mu_0} s_1 \xrightarrow{a_1,\mu_1} \cdots$ where $s_i \in S$ and $s_i \xrightarrow{a_i} \mu_i$ is a transition with $\mu_i(s_{i+1}) > 0$ for all $i \in \mathbb{N}$. A *finite path* ρ is a prefix of an infinite path ending in a state, and we denote $last(\rho)$ its last state. The *trace* $tr(\pi)$ of a path π is defined the same as for DTMCs. And we denote by $tr(\pi)\restriction_{\alpha'}$ the restriction of a trace to an alphabet $\alpha' \subseteq \alpha$. The set of infinite (resp. finite) paths starting from state s of $\mathcal{M}$ is denoted by $IPaths_{\mathcal{M},s}$ (resp. $FPaths_{\mathcal{M},s}$). And the sets of all infinite and finite paths in $\mathcal{M}$ are denoted as $IPaths_{\mathcal{M}}$ and $FPaths_{\mathcal{M}}$, respectively.

We use the notion of *adversaries* (also known as 'schedulers' or 'policies') to reason about PAs. An adversary resolves the nondeterministic choices in a PA, based on its execution history. Formally, an adversary is a function $\sigma : FPaths_{\mathcal{M}} \to Dist(\alpha \times Dist(S))$ such that, for any finite path $\rho \in FPaths_{\mathcal{M}}$, the distribution $\sigma(\rho)$ only assigns non-zero probabilities to action-distribution pairs (a, μ) for which $(last(\rho), a, \mu) \in \delta$. We call adversaries defined in this manner as *complete* adversaries, and, by contrast, we call an adversary *partial* if it contains some sub-distributions. Intuitively, a partial adversary can opt to take none of the available transitions and remain in the current state (with some probability). We denote by $Adv_{\mathcal{M}}$ the set of all possible adversaries for $\mathcal{M}$. Under an adversary σ, the behaviour of PA $\mathcal{M}$ is purely probabilistic and can be captured by a (countably infinite-state) DTMC, each state of which corresponds to a finite path of $\mathcal{M}$. We denote by $IPaths^{\sigma}_{\mathcal{M},s}$ (resp. $FPaths^{\sigma}_{\mathcal{M},s}$) the set of infinite (resp. finite) paths through $\mathcal{M}$ that start in state s and under σ. We can define a probability measure $Pr^{\sigma}_{\mathcal{M},s}$ over paths $IPaths^{\sigma}_{\mathcal{M},s}$ in a similar manner as for DTMCs (Section 3.2.1). Note that s can be dropped if it is clear from the context.

We distinguish several different classes of adversaries. An adversary is *deterministic* if it selects a unique action for the current state; otherwise, if it selects enabled actions probabilistically, the adversary is *randomised*. In this thesis, we consider deterministic adversaries only. We say that an adversary is *memoryless* if it always selects the same action in a given state and the choice is independent of the execution history, i.e. which

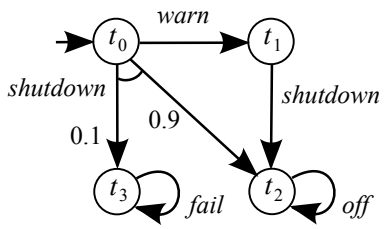


Figure 3.3: A 4-state PA $\mathcal{M}$ (taken from [KNPQ10])

path led to the current state; otherwise, the adversary is *history-dependent*. A variant of memoryless adversaries are *finite-memory* adversaries, in which the selection of actions depends on the current state of the PA and the current state (called *mode*) of memory stored in a DFA. Under a deterministic, memoryless adversary σ, the behaviour of a PA $\mathcal{M}$ with states S can be represented by another PA over the same state space, denoted $\mathcal{M}^\sigma$, in which each $s \in S$ contains only the choices made by σ in s. Similarly, under a deterministic, finite-memory adversary σ, the behaviour of a PA $\mathcal{M}$ with states S can be represented by a PA $\mathcal{M}^\sigma$ with states $S \times Q$ where Q is the set of modes of σ's memory DFA.

Example 3.6 *Figure 3.3 shows a PA $\mathcal{M} = (S, \overline{s}, \alpha, \delta, L)$, where $S = \{t_0, t_1, t_2, t_3\}$, $\overline{s} = t_0$, and $\alpha = \{warn, shutdown, fail, off\}$. The labelling function L over states is omitted here. Each transition of $\mathcal{M}$ is labelled with an action and a probability (omitted if probability is 1). For example, from state t_0, there is a nondeterministic choice of distributions between $\delta(t_0, warn) = [t_1 \to 1]$ and $\delta(t_0, shutdown) = [t_2 \to 0.9, t_3 \to 0.1]$.*

The PA $\mathcal{M}$ models a device whose nondeterministic behaviour is controlled by the signal that it receives. If a "warn" signal is received before "shutdown", then the device powers down correctly; otherwise, a failure may occur with probability 0.1.

3.2.3 Interval Discrete-Time Markov Chains

The third type of model we consider, named *interval discrete-time Markov chains* (IDTMCs) [JL91, KU02], is a generalisation of DTMCs with the transition probabilities

unspecified but assumed to lie within an interval.

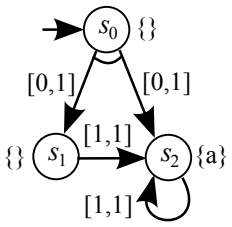
Definition 3.7 (IDTMC) *An* interval discrete-time Markov chain *is a tuple* $\mathcal{I} = (S, \overline{s}, \mathbf{P}^l, \mathbf{P}^u, L)$ *where* S *is a finite set of states,* $\overline{s} \in S$ *is an initial state,* $\mathbf{P}^l, \mathbf{P}^u : S \times S \to [0,1]$ *are matrices representing the lower/upper bounds of transition probabilities such that* $\sum_{s' \in S} \mathbf{P}^l(s,s') \leq 1 \leq \sum_{s' \in S} \mathbf{P}^u(s,s')$ *and* $\mathbf{P}^l(s,s') \leq \mathbf{P}^u(s,s')$ *for all states* $s, s' \in S$*, and* $L : S \to 2^{AP}$ *is a labelling function assigning atomic propositions from a set* AP *to states. (Note that the alphabet* α *of transition actions is omitted.)*

Recall from Section 2.1.1 that there are two common semantic interpretations of IDTMCs: *uncertain Markov chains* (UMCs) and *interval Markov decision processes* (IMDPs). In this thesis, we adopt the latter, where each IDTMC defines an IMDP.

Definition 3.8 (IDTMC semantics) *An IDTMC* $\mathcal{I} = (S, \overline{s}, \mathbf{P}^l, \mathbf{P}^u, L)$ *defines an IMDP* $\lceil \mathcal{I} \rceil = (S, \overline{s}, \delta, L)$*, where* $S, \overline{s}, L$ *are the same as in* $\mathcal{I}$*, and* $\delta \subseteq S \times Dist(S)$ *is the transition relation such that the set of available distributions in state* s *is given by* $\delta(s) = \{\mu \in Dist(S) \mid \forall s' \in S, \mathbf{P}^l(s,s') \leq \mu(s') \leq \mathbf{P}^u(s,s')\}$.

IMDPs are very similar to MDPs (a subclass of PAs) mentioned before. The only difference is that a state of an IMDP may have an *infinite* number of available distributions, whereas the set of available distributions in an MDP state is finite. However, as stated in [CHK13], IMDPs can be treated as MDPs in terms of model checking. Therefore, we do not distinguish IMDPs from MDPs in this thesis. We use the same notions of paths, adversaries and probabilistic measures defined in Section 3.2.2 here for IMDPs (with the transition actions omitted).

Example 3.9 *Figure 3.4 shows an IDTMC* $\mathcal{I} = (S, \overline{s}, \mathbf{P}^l, \mathbf{P}^u, L)$ *and its lower (resp. upper) transition bound matrix* $\mathbf{P}^l$ *(resp.* $\mathbf{P}^u$*). The state set is* $S = \{s_0, s_1, s_2\}$*, the initial state is* $\overline{s} = s_0$*, and the labelling function* L *assigns* $L(s_0) = L(s_1) = \emptyset$*,* $L(s_2) = \{a\}$ *where* a *is an atomic proposition. The transitions of* $\mathcal{I}$ *are labelled with the correspond-*



$$\mathbf{P}^l = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{P}^u = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 3.4: A 3-state IDTMC $\mathcal{I}$ and the probability bounds matrices $\mathbf{P}^l, \mathbf{P}^u$

ing intervals. For instance, $\mathbf{P}^l(s_0, s_1) = 0$ *and* $\mathbf{P}^u(s_0, s_1) = 1$*, and thus the transition probability between states* s_0 *and* s_1 *is bounded by interval* $[0, 1]$*.*

The IMDP $\lceil \mathcal{I} \rceil = (S, \overline{s}, \delta, L)$ *induced by* $\mathcal{I}$ *has the same* $S, \overline{s}, L$ *as above. The transition relation* δ *is given by* $\delta(s_0) = \{\mu \,|\, \mu(s_0) = 0, 0 \leq \mu(s_1) \leq 1, 0 \leq \mu(s_2) \leq 1, \text{and } \mu(s_1) + \mu(s_2) = 1\}$*, and* $\delta(s_1) = \delta(s_2) = \{\mu \,|\, \mu(s_0) = 0, \mu(s_1) = 0, \mu(s_2) = 1)\}$*.*

3.3 Probabilistic Model Checking

We now describe how to specify and verify quantitative properties for DTMCs and PAs. We will discuss the model checking for IDTMCs separately in Chapter 6. The material presented in this section is based on the textbook [BK08] and tutorial paper [FKNP11].

3.3.1 Specifying Properties

There are various different specification formalisms for expressing properties of probabilistic models. In this thesis, we mainly focus on three types of properties: probabilistic reachability, probabilistic LTL specifications and probabilistic safety properties.

Probabilistic Reachability

A fundamental property of probabilistic models is *probabilistic reachability*, i.e. the probability of reaching a set of target states when starting from a given state.

For a DTMC $\mathcal{D}$, let $reach_s(T)$ be the set of all infinite paths that start from state s and contain a target state from the set $T \subseteq S$, formally:

$$reach_s(T) \stackrel{\text{def}}{=} \{\pi \in IPaths_{\mathcal{D},s} \mid \pi(i) \in T \text{ for some } i \in \mathbb{N}\}.$$

We write $reach(T)$ instead of $reach_s(T)$ when s is clear from the context. $reach_s(T)$ is measurable for any $T \subseteq S$, because $reach_s(T)$ is the union of all basic cylinders of finite paths from s ending in T and each element of this union is measurable. Thus, we can define the reachability probability formally as $Pr_{\mathcal{D},s}(reach_s(T))$, where $Pr_{\mathcal{D},s}$ is the probability measure over paths as defined in Section 3.2.1.

For a PA $\mathcal{M}$, we define $reach_s(T)$ similarly as a set of infinite paths from s to T, i.e. $\{\pi \in IPaths_{\mathcal{M},s} \mid \pi(i) \in T \text{ for some } i \in \mathbb{N}\}$. Due to the nondeterministic behaviour of PA, the reachability probability is not a single value; rather, we define the *minimum* and *maximum* probability, when starting from a state s, of reaching a set of target states $T \subseteq S$ as:

$$Pr^{\min}_{\mathcal{M},s}(reach_s(T)) \stackrel{\text{def}}{=} \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(reach_s(T))$$
$$Pr^{\max}_{\mathcal{M},s}(reach_s(T)) \stackrel{\text{def}}{=} \sup_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(reach_s(T))$$

where $Pr^{\sigma}_{\mathcal{M},s}$ is the probabilistic measure capturing the behaviour of $\mathcal{M}$ from state s under adversary σ (defined in Section 3.2.2).

Probabilistic LTL Specifications

Linear temporal logic (LTL) [Pnu77] is a widely used temporal logic for linear time properties. The syntax of LTL is as follows:

$$\psi ::= \mathsf{true} \mid a \mid \psi \wedge \psi \mid \neg\psi \mid X\,\psi \mid \psi\,\mathcal{U}\,\psi$$

where $a \in AP$ is an atomic proposition, and X (*next*) and $\mathcal{U}$ (*until*) are standard temporal logic operators. For an infinite path π of DTMC/PA, the *satisfaction relation* $\models$ is defined inductively by:

$$\begin{array}{lcll}
\pi \models \mathsf{true} & & & always \\
\pi \models a & \iff & a \in L(\pi(0)) & \\
\pi \models \psi_1 \wedge \psi_2 & \iff & \pi \models \psi_1 \wedge \pi \models \psi_2 & \\
\pi \models \neg\psi & \iff & \pi \not\models \psi & \\
\pi \models X\,\psi & \iff & \pi[1\ldots] \models \psi & \\
\pi \models \psi_1\ \mathcal{U}\ \psi_2 & \iff & \exists j \in \mathbb{N}.\ \pi[j\ldots] \models \psi_2 \text{ and } \pi[i\ldots] \models \psi_1, \text{for all } i < j &
\end{array}$$

where $L(\pi(0))$ is the labelling of the first state in path π, and $\pi[i\ldots]$ is a suffix of path π starting from the i-th state.

In this thesis, we also use temporal operators $\Diamond$ (*future*) and $\Box$ (*globally*), which can be derived as follows:

$$\Diamond\psi \stackrel{\text{def}}{=} \mathsf{true}\ \mathcal{U}\ \psi$$
$$\Box\psi \stackrel{\text{def}}{=} \neg(\Diamond\neg\psi)$$

$\Diamond\psi$ means that ψ is eventually satisfied, and $\Box\psi$ represents that ψ is always satisfied. We can use $\Diamond T$ to represent the reachability $reach(T)$ of hitting a set of states T.

The probability measure of satisfying an LTL formula ψ in state s of DTMC $\mathcal{D}$ is defined as:

$$Pr_{\mathcal{D},s}(\psi) \stackrel{\text{def}}{=} Pr_{\mathcal{D},s}(\{\pi \in IPaths_{\mathcal{D},s} \mid \pi \models \psi\}).$$

Similarly, for a PA $\mathcal{M}$, it is straightforward to define the probability of satisfying an LTL formula ψ in state s under adversary σ as:

$$Pr^{\sigma}_{\mathcal{M},s}(\psi) \stackrel{\text{def}}{=} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \in IPaths^{\sigma}_{\mathcal{M},s} \mid \pi \models \psi\}).$$

The minimum and maximum probabilities of satisfaction over all adversaries of $\mathcal{M}$ are:

$$Pr^{\min}_{\mathcal{M},s}(\psi) \stackrel{\text{def}}{=} \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\psi)$$
$$Pr^{\max}_{\mathcal{M},s}(\psi) \stackrel{\text{def}}{=} \sup_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\psi).$$

A *probabilistic LTL specification* is a formula $\mathsf{P}_{\bowtie p}[\psi]$ where $\bowtie \in \{\geq, >, \leq, <\}$, $p \in [0,1]$ is a rational probability and ψ is an LTL formula. A probabilistic LTL specification $\mathsf{P}_{\bowtie p}[\psi]$ is satisfied in a state s of DTMC $\mathcal{D}$ iff $Pr_{\mathcal{D},s}(\psi) \bowtie p$. Analogously, $\mathsf{P}_{\bowtie p}[\psi]$ is satisfied in a state s of PA $\mathcal{M}$ iff $Pr^{\sigma}_{\mathcal{M},s}(\psi) \bowtie p$ for all adversaries $\sigma \in Adv_{\mathcal{M}}$. In particular, if $\bowtie \in \{\geq, >\}$, then $s \models \mathsf{P}_{\bowtie p}[\psi]$ iff $Pr^{\min}_{\mathcal{M},s}(\psi) \bowtie p$; while if $\bowtie \in \{\leq, <\}$, then $s \models \mathsf{P}_{\bowtie p}[\psi]$ iff $Pr^{\max}_{\mathcal{M},s}(\psi) \bowtie p$. We say a DTMC $\mathcal{D}$ (resp. a PA $\mathcal{M}$) satisfies a probabilistic LTL specification $\mathsf{P}_{\bowtie p}[\psi]$, denoted $\mathcal{D} \models \mathsf{P}_{\bowtie p}[\psi]$ (resp. $\mathcal{M} \models \mathsf{P}_{\bowtie p}[\psi]$), iff the specification is satisfied in the initial state $\overline{s}$.

Probabilistic Safety Properties

Safety properties characterise the requirements that "the bad thing should never happen". Typical examples of safety properties include the *mutual exclusion property* (i.e. the bad scenario of having two or more processes in their critical section simultaneously never occurs) and the *deadlock freedom property* (i.e. the unwanted deadlock should be avoided). Formally, a safety property is defined as a set of infinite words, none of which has a *bad prefix* (i.e. a finite word where the bad thing has happened). Any word that violates a safety property has a bad prefix. If all the bad prefixes of a safety property constitute a regular language (i.e. accepted by a finite-state automaton, see Section 3.1), then the property is called a *regular safety property.*

Safety properties can be defined over a set of atomic propositions AP or an alphabet α of action labels. In this thesis, if a regular safety property is defined over atomic propositions, we represent it as an LTL formula ψ; for example, $\psi = \Box(\neg \Diamond err)$ means

that states labelled with the atomic proposition *err* should never be reached. Otherwise, we use an automaton representation G, where the set of bad prefixes are stored in a (complete) DFA G^{err} over alphabet α. We define language $\mathcal{L}(G)$ as the set of infinite words $w \in \alpha^\omega$ such that no prefix of w is in the regular language characterised by G^{err}. An (infinite) path π of DTMC/PA satisfies G, denoted $\pi \models G$, if its trace $tr(\pi)$ is in $\mathcal{L}(G)$.

A *probabilistic safety property*, denoted by $\mathsf{P}_{\geq p}[\psi]$ (or $\langle G \rangle_{\geq p}$), is given by a regular safety property ψ (or automaton G), and a rational *lower* probability bound p. For example, "the device always receives a *warn* signal before *shutdown* with probability at least 0.98" is a probabilistic safety property. The satisfaction of $\mathsf{P}_{\geq p}[\psi]$ for DTMC/PA follows the semantics of probabilistic LTL specifications as discussed previously.

In the following, we explain the semantics of satisfying $\langle G \rangle_{\geq p}$ for DTMCs and PAs. We define the probability of a DTMC $\mathcal{D}$ satisfying G as:

$$Pr_{\mathcal{D}}(G) \stackrel{\text{def}}{=} Pr_{\mathcal{D},\overline{s}}(\{\pi \in IPaths_{\mathcal{D},\overline{s}} \mid \pi \models G\})$$

where $\overline{s}$ is the initial state of $\mathcal{D}$. We say that a probabilistic safety property $\langle G \rangle_{\geq p}$ is satisfied by $\mathcal{D}$, denoted $\mathcal{D} \models \langle G \rangle_{\geq p}$, if $Pr_{\mathcal{D}}(G) \geq p$.

Similarly, the probability of a PA $\mathcal{M}$ satisfying G under an adversary σ is given by:

$$Pr^{\sigma}_{\mathcal{M}}(G) \stackrel{\text{def}}{=} Pr^{\sigma}_{\mathcal{M},\overline{s}}(\{\pi \in IPaths^{\sigma}_{\mathcal{M},\overline{s}} \mid \pi \models G\})$$

where $\overline{s}$ is the initial state of $\mathcal{M}$. We say that a probabilistic safety property $\langle G \rangle_{\geq p}$ is satisfied by $\mathcal{M}$, denoted $\mathcal{M} \models \langle G \rangle_{\geq p}$, if the probability of satisfying G is at least p for any adversary:

$$\begin{aligned} \mathcal{M} \models \langle G \rangle_{\geq p} &\Leftrightarrow \forall \sigma \in Adv_{\mathcal{M}} \,.\, Pr^{\sigma}_{\mathcal{M}}(G) \geq p \\ &\Leftrightarrow Pr^{\min}_{\mathcal{M}}(G) \geq p \end{aligned}$$

where $Pr^{\min}_{\mathcal{M}}(G) \stackrel{\text{def}}{=} \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M}}(G)$. According to [BK08], it suffices to consider deterministic, finite-memory adversaries for checking $\mathcal{M} \models \langle G \rangle_{\geq p}$ on a PA $\mathcal{M}$, i.e. there always exists such an adversary σ for which $Pr^{\sigma}_{\mathcal{M}}(G) = Pr^{\min}_{\mathcal{M}}(G)$.

3.3.2 Model Checking for DTMCs

In the following, we introduce the model checking techniques for DTMCs, including the computation of probabilistic reachability and the verification of probabilistic LTL specifications, as well as probabilistic safety properties.

Computing Probabilistic Reachability

The following is an efficient way of computing reachability probabilities for a DTMC $\mathcal{D} = (S, \overline{s}, \mathbf{P}, L)$. Let variable x_s denote $Pr_{\mathcal{D},s}(reach_s(T))$, the probability of reaching target states T from state $s \in S$, then x_s can be computed as the unique solution of the following linear equation system [CY88]:

$$x_s = \begin{cases} 1 & \text{if } s \in T \\ 0 & \text{if } T \text{ is not reachable from } s \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot x_{s'} & \text{otherwise.} \end{cases}$$

It is straightforward to find the set of states that can reach T through graph analysis (e.g. backward depth-first or breath-first search from T). Then we can (approximately) solve the above linear equation system using an iterative method such as the Gauss-Seidel value iteration [BK08].

Verifying Probabilistic LTL Specifications

Recall from Section 3.3.1 that model checking a probabilistic LTL specification $\mathsf{P}_{\bowtie p}[\psi]$ in a state s of a DTMC $\mathcal{D}$ requires computing the probability of a set of infinite paths

in $\mathcal{D}$ for which ψ holds, denoted $Pr_{\mathcal{D},s}(\psi)$. In the following, we introduce an automata-based approach, in which an LTL formula ψ is represented by means of a *deterministic Rabin automaton* (DRA) $\mathcal{A}_\psi$ and the computation of $Pr_{\mathcal{D},s}(\psi)$ reduces to computing reachability probabilities in the product of $\mathcal{D}$ and $\mathcal{A}_\psi$, denoted by $\mathcal{D} \otimes \mathcal{A}_\psi$.

Definition 3.10 (DRA) *A* deterministic Rabin automaton *(DRA) is a tuple* $\mathcal{A} = (S, \overline{s}, \alpha, \delta, Acc)$*, where* S *is a finite set of states,* $\overline{s} \in S$ *is an initial state,* α *is an alphabet of actions,* $\delta : S \times \alpha \to S$ *is a transition function, and* $Acc = \{(L_i, K_i)\}_{i=1}^{k}$ *is an acceptance condition where* $k \in \mathbb{N}$ *and* $L_i, K_i \subseteq S$ *for* $1 \leq i \leq k$*.*

For each infinite word $w = a_0 a_1 a_2 \cdots$ over alphabet α, there is a unique corresponding run $\overline{s} \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ of DRA $\mathcal{A}$ (since $\mathcal{A}$ has a deterministic transition function). We say that an infinite word $w \in \alpha^\omega$ is accepted by $\mathcal{A}$ if its corresponding path contains *finitely* many states from L_i and *infinitely* many states from K_i for some $1 \leq i \leq k$. The language of the DRA $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the set of infinite words accepted by $\mathcal{A}$. For any LTL formula ψ, we can construct a corresponding DRA $\mathcal{A}_\psi$ such that an infinite path $\pi \models \psi$ if and only if $L(\pi) \in \mathcal{A}_\psi$, where $L(\pi) = L(s_0)L(s_1)\ldots$ is the sequence of atomic propositions over states in path π. We refer the reader to [BK08] for the details of how to construct such a DRA $\mathcal{A}_\psi$ from ψ.

Now we define the product $\mathcal{D} \otimes \mathcal{A}_\psi$. Since we consider LTL formulae over atomic propositions only, the transition actions of DTMCs are omitted for convenience. Given a DTMC $\mathcal{D} = (S, \overline{s}, \mathbf{P}, L)$ and a DRA $\mathcal{A}_\psi = (Q, \overline{q}, 2^{AP}, \delta, Acc)$ with $Acc = \{(L_i, K_i)\}_{i=1}^{k}$, their product is a DTMC $\mathcal{D} \otimes \mathcal{A}_\psi = (S \times Q, (\overline{s}, q_{\overline{s}}), \mathbf{P}', L')$ where $q_{\overline{s}} = \delta(\overline{q}, L(\overline{s}))$, the transition matrix $\mathbf{P}'$ is given by

$$\mathbf{P}'((s,q),(s',q')) = \begin{cases} \mathbf{P}(s,s') & \text{if } q' = \delta(q, L(s)) \\ 0 & \text{otherwise} \end{cases}$$

for states $(s,q),(s',q') \in S \times Q$, and the labelling function is defined as $L'((s,q)) = L(s) \cup H$ if $q \in H$ for any $H \in \{L_1,\ldots,L_k,K_1,\ldots,K_k\}$, and $L'((s,q)) = L(s)$ otherwise.

The computation of $Pr_{\mathcal{D},s}(\psi)$ reduces to computing the probability of reaching certain *bottom strongly connected components* (BSCCs) in the product $\mathcal{D} \otimes \mathcal{A}_\psi$. Such BSCCs, named *accepting* BSCCs, are defined such that the set of states $T \subseteq S \times Q$ of an accepting BSCC should fulfil $T \cap (S \times L_i) = \emptyset$ and $T \cap (S \times K_i) \neq \emptyset$ for some $1 \leq i \leq k$. See [BK08] for the formal definition of BSCCs and how to detect them using graph analysis algorithms.

In summary, model checking a probabilistic LTL specification $\mathsf{P}_{\bowtie p}[\psi]$ in a state s of a DTMC $\mathcal{D}$ involves the following steps:

(1) generate a DRA $\mathcal{A}_\psi$ for ψ,
(2) construct the product DTMC $\mathcal{D} \otimes \mathcal{A}_\psi$,
(3) identify accepting BSCCs of $\mathcal{D} \otimes \mathcal{A}_\psi$,
(4) compute $Pr_{\mathcal{D},s}(\psi)$ as the probability of reaching accepting BSCCs from state (s,q_s) in $\mathcal{D} \otimes \mathcal{A}_\psi$ where $q_s = \delta(\overline{q}, L(s))$,
(5) check whether $Pr_{\mathcal{D},s}(\psi) \bowtie p$ is satisfied.

The complexity of checking LTL specification ψ on DTMC $\mathcal{D}$ is doubly exponential in the size of formula ψ and polynomial in the size of model $\mathcal{D}$.

Verifying Probabilistic Safety Properties

Recall that probabilistic safety properties can be represented as a probabilistic LTL formula $\mathsf{P}_{\geq p}[\psi]$, or as $\langle G \rangle_{\geq p}$ where the set of bad prefixes are stored in a DFA G^{err}. The verification of probabilistic LTL formulae has just been explained. In the following, we describe how to check $\langle G \rangle_{\geq p}$ for a DTMC $\mathcal{D}$.

Firstly, we construct the product $\mathcal{D} \otimes G^{err}$. The intuition is to synchronise common actions. Given a DTMC $\mathcal{D} = (S, \overline{s}, \alpha_\mathcal{D}, \delta_\mathcal{D}, L)$ and DFA $G^{err} = (Q, \overline{q}, \alpha_G, \delta_G, F)$ with

$\alpha_G \subseteq \alpha_{\mathcal{D}}$, their product is a DTMC $\mathcal{D} \otimes G^{err} = (S \times Q, (\overline{s}, \overline{q}), \alpha_{\mathcal{D}}, \delta', L')$, where the transition function δ' is given by

$$\delta'((s,q),a) = \begin{cases} \delta_{\mathcal{D}}(s,a) \times \eta_{q'} & \text{if } a \in \alpha_G \cap A(s) \\ \delta_{\mathcal{D}}(s,a) \times \eta_{q} & \text{if } a \in (\alpha_{\mathcal{D}} \setminus \alpha_G) \cap A(s) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

with $q' = \delta_G(q,a)$ and $A(s) = \{a \in \alpha_{\mathcal{D}} \mid \delta_{\mathcal{D}}(s,a) \text{ is defined}\}$, and the labelling function $L'((s,q)) = L(s) \cup \{err\}$ if $q \in F$ and $L'((s,q)) = L(s)$ otherwise.

Let T be the set of states labelled with err in $\mathcal{D} \otimes G^{err}$. We then have the probability of satisfying G on DTMC $\mathcal{D}$ as:

$$Pr_{\mathcal{D}}(G) = 1 - Pr_{\mathcal{D} \otimes G^{err}}(reach(T))$$

where $Pr_{\mathcal{D} \otimes G^{err}}(reach(T))$ is the probability of reaching states T from the initial state $(\overline{s}, \overline{q})$ in $\mathcal{D} \otimes G^{err}$. If $Pr_{\mathcal{D}}(G) \geq p$, then $\mathcal{D} \models \langle G \rangle_{\geq p}$.

3.3.3 Model Checking for PAs

In the following, we describe several model checking techniques for PAs, including the computation of reachability probabilities, the verification of probabilistic LTL formulae and probabilistic safety properties, and the multi-objective model checking.

Computing Probabilistic Reachability

It has been demonstrated [CY90, dA97] that the computation of minimum and maximum reachability probabilities of PAs can be phrased as *linear programming* problems.

For a PA $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$, we first define the following sets of states:

$$S^0_{\min} \stackrel{\text{def}}{=} \{s \in S \mid Pr^{\min}_{\mathcal{M},s}(reach_s(T)) = 0\}$$
$$S^1_{\min} \stackrel{\text{def}}{=} \{s \in S \mid Pr^{\min}_{\mathcal{M},s}(reach_s(T)) = 1\}$$
$$S^0_{\max} \stackrel{\text{def}}{=} \{s \in S \mid Pr^{\max}_{\mathcal{M},s}(reach_s(T)) = 0\}$$
$$S^1_{\max} \stackrel{\text{def}}{=} \{s \in S \mid Pr^{\max}_{\mathcal{M},s}(reach_s(T)) = 1\}$$

where $Pr^{\min}_{\mathcal{M},s}(reach_s(T))$ and $Pr^{\max}_{\mathcal{M},s}(reach_s(T))$ are the minimum and maximum probabilities of reaching target states $T \subseteq S$ from state s in $\mathcal{M}$, respectively. We also denote by $A(s)$ the set of actions available in state s, i.e. $\{a \in \alpha_{\mathcal{M}} \mid \delta_{\mathcal{M}}(s,a) \text{ is defined}\}$.

Let variable $x_s = Pr^{\min}_{\mathcal{M},s}(reach_s(T))$, then the minimum reachability probabilities can be computed as unique solution x_s of the following linear program:

maximise $\sum_{s \in S} x_s$ subject to the constraints:	
$x_s = 1$	for all $s \in S^1_{\min}$
$x_s = 0$	for all $s \in S^0_{\min}$
$x_s \leq \sum_{s' \in S} \delta_{\mathcal{M}}(s,a,\mu)(s') \cdot x_{s'}$	for all $s \notin S^1_{\min} \cup S^0_{\min}$, $a \in A(s)$ and $\mu \in \delta_{\mathcal{M}}(s,a)$

Similarly, the following linear program yields a unique solution for the maximum reachability probabilities $x_s = Pr^{\max}_{\mathcal{M},s}(reach_s(T))$:

minimise $\sum_{s \in S} x_s$ subject to the constraints:	
$x_s = 1$	for all $s \in S^1_{\max}$
$x_s = 0$	for all $s \in S^0_{\max}$
$x_s \geq \sum_{s' \in S} \delta_{\mathcal{M}}(s,a,\mu)(s') \cdot x_{s'}$	for all $s \notin S^1_{\max} \cup S^0_{\max}$, $a \in A(s)$ and $\mu \in \delta_{\mathcal{M}}(s,a)$

The state sets $S^0_{\min}, S^1_{\min}, S^0_{\max}$ and $S^1_{\max}$ can be obtained by standard (non-probabilistic) model checking, and the linear programs can be solved using LP solvers, or by approximate value iteration or policy iteration algorithms. We refer the details of these algorithms to [BK08].

Verifying Probabilistic LTL Specifications

Recall from Section 3.3.1 that checking whether a probabilistic LTL specification $\mathsf{P}_{\bowtie p}[\psi]$ is satisfied in a state s of a PA $\mathcal{M}$ requires the computation of $Pr^{\max}_{\mathcal{M},s}(\psi)$ if $\bowtie \in \{\leq, <\}$ or $Pr^{\min}_{\mathcal{M},s}(\psi)$ if $\bowtie \in \{\geq, >\}$. We show below that the computation of $Pr^{\min}_{\mathcal{M},s}(\psi)$ can be easily reduced to computing $Pr^{\max}_{\mathcal{M},s}(\neg\psi)$, due to the fact that a path π of $\mathcal{M}$ satisfies either ψ or $\neg\psi$. The reduction is shown below:

$$\begin{aligned}
Pr^{\min}_{\mathcal{M},s}(\psi) &= \inf_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \models \psi\}) \\
&= \inf_{\sigma \in Adv_{\mathcal{M}}} (1 - Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \not\models \psi\})) \\
&= \inf_{\sigma \in Adv_{\mathcal{M}}} (1 - Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \models \neg\psi\})) \\
&= 1 - \sup_{\sigma \in Adv_{\mathcal{M}}} Pr^{\sigma}_{\mathcal{M},s}(\{\pi \mid \pi \models \neg\psi\}) \\
&= 1 - Pr^{\max}_{\mathcal{M},s}(\neg\psi).
\end{aligned}$$

Now we describe how to compute $Pr^{\max}_{\mathcal{M},s}(\psi)$, the maximum probability of satisfying an LTL formula ψ in a state s of PA $\mathcal{M}$. Similarly to checking LTL formulae for DTMCs, we first convert the LTL formula ψ to a DRA $\mathcal{A}_\psi$ (Definition 3.10). Then we construct a PA-DRA product $\mathcal{M} \otimes \mathcal{A}_\psi$. Given a PA $\mathcal{M}=(S, \overline{s}, \alpha, \delta_{\mathcal{M}}, L)$ and a DRA $\mathcal{A}_\psi = (Q, \overline{q}, 2^{AP}, \delta_{\mathcal{A}}, Acc)$ with $Acc = \{(L_i, K_i)\}_{i=1}^{k}$, their product is a PA $\mathcal{M} \otimes \mathcal{A}_\psi = (S \times Q, (\overline{s}, q_{\overline{s}}), \alpha, \delta', L')$, where $q_{\overline{s}} = \delta(\overline{q}, L(\overline{s}))$, the transition relation δ' is defined such that we have $(s,q) \xrightarrow{a} \mu \times \eta_{q'}$ if and only if $s \xrightarrow{a} \mu$ and $q' = \delta_{\mathcal{A}}(q, L(s))$ holds, and the labelling function is $L'((s,q)) = L(s) \cup H$ if $q \in H$ for any $H \in \{L_1, \ldots, L_k, K_1, \ldots, K_k\}$ and $L'((s,q)) = L(s)$ otherwise.

Recall from Section 3.3.2 that checking LTL formulae for DTMCs reduces to computing the probabilistic reachability of accepting BSCCs. For PAs, we have an analogous notion of BSCCs in DTMCs, named *end components*. An end component is a strongly connected sub-PA, which comprises a subset of PA states and a partial tran-

sition function of those states. We refer the reader to the tutorial paper [FKNP11] for the formal definition and detecting algorithms of end components.

We say that an end component of $\mathcal{M} \otimes \mathcal{A}_\psi$ is *accepting* if its set of states $T \subseteq S \times Q$ satisfies $T \cap (S \times L_i) = \emptyset$ and $T \cap (S \times K_i) \neq \emptyset$ for some $1 \leq i \leq k$, which is related to the Rabin accepting condition of $\mathcal{A}_\psi$ that states in L_i should be visited finitely often and states in K_i be visited infinitely often. We have

$$Pr^{\max}_{\mathcal{M},s}(\psi) = Pr^{\max}_{\mathcal{M}\otimes\mathcal{A}_\psi,(s,\overline{q})}(reach_{(s,\overline{q})}(T))$$

where state $(s,q) \in T$ if and only if (s,q) appears in some accepting end components of $\mathcal{M} \otimes \mathcal{A}_\psi$. Thus, verifying probabilistic LTL specifications for PAs reduces to computing probabilistic reachability of accepting end components.

Verifying Probabilistic Safety Properties

Recall that we can represent a probabilistic safety property as $\mathsf{P}_{\geq p}[\psi]$ or $\langle G \rangle_{\geq p}$. Model checking $\mathsf{P}_{\geq p}[\psi]$ for PAs just follows the LTL verification techniques. We now describe how to verify $\langle G \rangle_{\geq p}$ on a PA $\mathcal{M}$. Recall from Section 3.3.1 that $\mathcal{M} \models \langle G \rangle_{\geq p}$ if and only if $Pr^{\min}_{\mathcal{M}}(G) \geq p$. The computation of $Pr^{\min}_{\mathcal{M}}(G)$ reduces to checking probabilistic reachability in the product $\mathcal{M} \otimes G^{err}$, where G^{err} is a DFA recognising all the bad prefixes of regular safety property G.

Given a PA $\mathcal{M}=(S, \overline{s}, \alpha_\mathcal{M}, \delta_\mathcal{M}, L)$ and a DFA $G^{err}=(Q, \overline{q}, \alpha_G, \delta_G, F)$ with $\alpha_G \subseteq \alpha_\mathcal{M}$, their product is a PA $\mathcal{M} \otimes G^{err} = (S \times Q, (\overline{s}, \overline{q}), \alpha_\mathcal{M}, \delta', L')$, where the transition function δ' is defined such that we have $(s,q) \xrightarrow{a} \mu \times \eta_{q'}$ if and only if one of the following holds:

- $s \xrightarrow{a} \mu$, $q' = \delta_G(q,a)$ and $a \in \alpha_G$
- $s \xrightarrow{a} \mu$, $q' = q$ and $a \notin \alpha_G$

and $L'((s,q)) = L(s) \cup \{err\}$ if $q \in F$ and $L'((s,q)) = L(s)$ otherwise. Let T be the set

of states labelled with *err* in $\mathcal{M} \otimes G^{err}$. It is proved in [KNPQ10] that

$$Pr^{\min}_{\mathcal{M}}(G) = 1 - Pr^{\max}_{\mathcal{M} \otimes G^{err}}(reach(T))$$

and thus,

$$\mathcal{M} \models \langle G \rangle_{\geq p} \Leftrightarrow Pr^{\max}_{\mathcal{M} \otimes G^{err}}(reach(T)) \leq 1 - p$$

where $Pr^{\max}_{\mathcal{M} \otimes G^{err}}(reach(T))$ is the maximum probability of reaching error states T from the initial state $(\overline{s}, \overline{q})$ of $\mathcal{M} \otimes G^{err}$.

Multi-objective Model Checking

The *multi-objective* model checking approach allows us to analyse the trade-offs between several linear-time properties, e.g. "the probability of reaching error states is at most 0.02 *and*, with probability at least 0.9, the process always stays in good states".

Given k predicates of the form $Pr^{\sigma}_{\mathcal{M},s}(\psi_i) \sim_i p_i$ where ψ_i is an LTL formula, $p_i \in [0, 1]$ is a rational probability bound and $\sim_i \in \{\geq, >\}$, then, using the techniques in [EKVY07], we can verify whether there exists an adversary satisfying their conjunction

$$\exists \sigma \in Adv_{\mathcal{M}} \; . \; \wedge^{k}_{i=1} \left(Pr^{\sigma}_{\mathcal{M},s}(\psi_i) \sim_i p_i \right)$$

by a reduction to a linear programming problem. The generalisations to checking existential or universal queries over a Boolean combination of predicates for which $\sim_i \in \{\geq, >, \leq, <\}$ are also available. In all cases, if an adversary satisfying the predicates exists, then it can also be constructed.

We can also check *quantitative* multi-objective queries [KNPQ10, FKN$^+$11]. Given a multi-objective LTL query $\theta = \wedge^{k}_{i=1} Pr^{\sigma}_{\mathcal{M},s}(\psi_i) \sim_i p_i$ and an additional LTL formula ψ, we can compute the maximum probability of satisfying ψ whilst maintaining the

satisfaction of θ:

$$Pr^{\max}_{\mathcal{M},s}(\psi \,|\, \theta) = \sup\{Pr^{\sigma}_{\mathcal{M},s}(\psi) \,|\, \sigma \in Adv_{\mathcal{M}} \wedge \sigma, s \models \theta)\}\,.$$

This is done by adding an objective function to the set of linear inequalities given by θ and solving the new linear program.

3.4 Probabilistic Counterexamples

Unlike in the non-probabilistic setting, where the violation of a property such as "an error never occurs" can be captured by a counterexample of a single finite path to an error state, the probabilistic counterexamples are often more complex. In this section, we introduce the notion of probabilistic counterexamples for DTMCs based on [HKD09]. Probabilistic counterexamples for PAs will be introduced later in Section 4.2.

Recall from Section 3.3.2 that model checking DTMCs against LTL formulae or probabilistic safety properties reduces to checking probabilistic reachability properties. Therefore, in the following, we only consider finding counterexamples for the violation of (upper bound) probabilistic reachability properties of DTMCs. As shown in [HKD09], the generation of counterexamples for lower bound probabilistic reachability properties of DTMCs reduces to the case of upper probability bounds.

Let us consider the violation of properties of the form $\mathsf{P}_{\leq p}[\Diamond T]$, where p is a probability value and T is a set of target states. The property is refuted in state s of DTMC $\mathcal{D}$ whenever the total probability mass of all paths that start in s and reach any state $s' \in T$ exceeds p. Let $FPaths^{\min}_{\mathcal{D},s}(T)$ be the set of finite paths in DTMC $\mathcal{D}$ that start in s and end in a target state $s' \in T$. Any finite path $\rho \in FPaths^{\min}_{\mathcal{D},s}(T)$ is an *evidence* for violating $\mathsf{P}_{\leq p}[\Diamond T]$ in state s of $\mathcal{D}$, because it might contribute to the probability mass of violation. And the evidence that contributes the most (i.e. the one with the largest path probability) is called the *strongest evidence*.

A *counterexample* for violating $\mathsf{P}_{\leq p}[\Diamond T]$ in state s is a set C of evidences such that $C \subseteq FPaths^{\min}_{\mathcal{D},s}(T)$ and $Pr_{\mathcal{D},s}(c) > p$. Intuitively, for a probabilistic property such as "an error state is reached with probability at most p", a single path may not be enough to serve as a counterexample on its own; indeed, a probabilistic counterexample usually consists of a *set* of such paths whose the combined probability exceeds p. In practice, we are interested in generating counterexamples that are succinct in representation and most distinctive to indicate the violation. The notion of *smallest counterexample* is thus defined as the one that deviates most from the probability bound p given that it has the smallest number of paths.

Generating strongest evidences or smallest counterexamples for $\mathsf{P}_{\leq p}[\Diamond T]$ on a DTMC reduces to finding (k-)shortest paths as follows. Firstly, we need to adapt the DTMC $\mathcal{D}$ by adding an extra state t such that all outgoing transitions from a target state $s \in T$ are replaced by a transition to t with probability 1. The next step is to convert the adapted DTMC $\mathcal{D}' = (S, \overline{s}, \mathbf{P}, L)$ into a weighted digraph $\mathcal{G}_{\mathcal{D}'} = (\mathrm{V}, \mathrm{E}, \mathbf{w})$, where the vertex set $\mathrm{V} = S$, the edges set E is defined by $(v, v') \in \mathrm{E}$ iff $\mathbf{P}(v, v') > 0$, and the edge weight between two vertices v, v' is $\mathbf{w}(v, v') = -\log \mathbf{P}(v, v')$. Then, finding the strongest evidence for violating $\mathsf{P}_{\leq p}[\Diamond T]$ in state s of DTMC $\mathcal{D}$ reduces to the shortest path problem, i.e. determine a finite path ρ from s to t such that $\mathbf{w}(\rho) \leq \mathbf{w}(\rho')$ for any path ρ' from s to t in $\mathcal{G}_{\mathcal{D}'}$, which can be solved using various algorithms (e.g. Dijkstra's algorithm [Dij59]). Finding the smallest counterexample reduces to the k-shortest paths (KSP) problem, i.e. finding k distinct paths $\rho_1, \ldots, \rho_k$ between s and t in $\mathcal{G}_{\mathcal{D}'}$ such that (1) for $1 \leq i < j \leq k$, $w(\rho_i) \leq w(\rho_j)$ and (2) for every ρ between s and t, if $\rho \notin \{\rho_1, \ldots, \rho_k\}$, then $w(\rho) \geq w(\rho_k)$. Note that the value of k should be determined *on-the-fly*, because the number of paths in the smallest counterexample is not known as a priori. Eppstein's algorithm [Epp98] can be applied to solve this problem. In this thesis (Section 6.4), we also implement a symbolic solution [GSS10] to the KSP problem based on the data structure of MTBDDs.

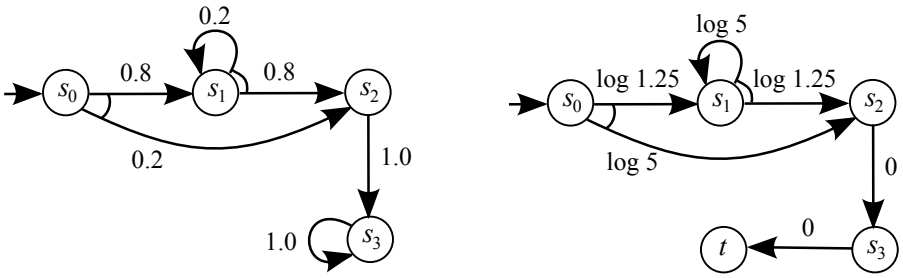


Figure 3.5: A DTMC $\mathcal{D}$ and its corresponding weighted digraph $\mathcal{G}_{\mathcal{D}'}$

Example 3.11 *Consider the DTMC $\mathcal{D}$ shown in Figure 3.5. We want to verify if its initial state s_0 satisfies the probabilistic reachability property $\mathsf{P}_{\leq 0.8}[\Diamond s_3]$. We first adapt $\mathcal{D}$ into the DTMC $\mathcal{D}'$ by adding an extra state t and replacing the self-loop on state s_3 in $\mathcal{D}$ with a transition $s_3 \xrightarrow{1.0} t$ in $\mathcal{D}'$. Then we convert $\mathcal{D}'$ into a weighted digraph $\mathcal{G}_{\mathcal{D}'}$ as shown Figure 3.5, by taking the negation of the logarithm of the corresponding transition probabilities in $\mathcal{D}'$ as the edge weights. The shortest path from the initial state s_0 to t in $\mathcal{G}_{\mathcal{D}'}$ is $s_0 \to s_1 \to s_2 \to s_3 \to t$, thus the strongest evidence for the violation of $\mathsf{P}_{\leq 0.8}[\Diamond s_3]$ is the finite path $\rho_1 = s_0 \xrightarrow{0.8} s_1 \xrightarrow{0.8} s_2 \xrightarrow{1.0} s_3$ with probability $Pr(\rho_1) = 0.64$. The second shortest path from s_0 to t in $\mathcal{G}_{\mathcal{D}'}$ is $s_0 \to s_2 \to s_3 \to t$, and the corresponding path in $\mathcal{D}$ is $\rho_2 = s_0 \xrightarrow{0.2} s_2 \xrightarrow{1.0} s_3$ with probability $Pr(\rho_2) = 0.2$. Since $Pr(\rho_1) + Pr(\rho_2) = 0.84 > 0.8$, the smallest counterexample for $\mathsf{P}_{\leq 0.8}[\Diamond s_3]$ in state s_0 of DTMC $\mathcal{D}$ is $C = \{\rho_1, \rho_2\}$.*

3.5 Learning Algorithms

In this section, we describe several *active learning* algorithms which are used in this thesis for learning assumptions. These algorithms all follow the active learning model, in which a *learner* actively interacts with a *teacher* by asking *membership* and *equivalence queries*, though the queries are different depending on the algorithm used.

We introduce, firstly, the L* algorithm for learning minimal DFAs in Section 3.5.1, then the NL* algorithm for learning *residual finite-state automata* (RFSAs), a sub-

class of NFAs, in Section 3.5.2, and finally the CDNF algorithm for learning arbitrary Boolean functions in Section 3.5.3.

3.5.1 The L* Algorithm

The L* algorithm aims to learn a DFA with a *minimal* number of states that accepts an unknown regular language. It proposes two kinds of queries to a *teacher*: membership queries (i.e. whether some word is in the target language) and equivalence queries (i.e. whether a conjectured DFA accepts the target language). If the conjectured DFA is not correct, the teacher would return a counterexample to L* to refine the automaton being learnt. The L* algorithm was first proposed by Angluin [Ang87a] and later improved by Rivest and Schapire [RS93]. We describe both versions of the algorithm below.

Angluin's L* Algorithm

Algorithm 1 shows the pseudo code of the original L* algorithm by Angluin. We use this algorithm in Chapter 4 to learn assumptions for compositional verification of asynchronous probabilistic systems.

At the implementation level, L* maintains an observation table (U, V, T). Here U is a nonempty finite prefix-closed set of words, V is a nonempty finite suffix-closed set of words, and T is a finite function mapping $((U \cup U \cdot \alpha) \cdot V) \to \{+, -\}$, where α is the alphabet of the target language and $+, -$ represent whether or not a word is accepted by the target language. The observation table can be visualised as a two-dimensional array with rows labelled by elements of $(U \cup U \cdot \alpha)$ and columns labelled by elements of V, while the entry for row u and column v is equal to $T(uv)$.

As shown in Algorithm 1, L* initially sets U, V to $\{\epsilon\}$ where ϵ is the empty word and asks membership queries to fill $T(w)$ for all words $w \in (U \cup U \cdot \alpha) \cdot V$. It then, from line 3 to 14, makes sure that the observation table (U, V, T) is *closed* (i.e. for all $u \in U$, $v \in V$ and $a \in \alpha$, there is a $u' \in U$ such that $T(uav) = T(u'v)$) and *consistent* (i.e.

Algorithm 1 Angluin's L* learning algorithm [Ang87a]

Input: alphabet α, a teacher who knows the target language $\mathcal{L}$
Output: DFA $\mathcal{A}$

1: initialise (U, V, T): let $U = V = \{\epsilon\}$, ask membership queries and fill $T(w)$ for all words $w \in (U \cup U \cdot \alpha) \cdot V$
2: **repeat**
3: **while** (U, V, T) is not closed or not consistent **do**
4: **if** (U, V, T) is not closed **then**
5: find $u \in U$, $a \in \alpha$, and $v \in V$ such that $T(uav) \neq T(u'v)$ for any $u' \in U$,
6: add ua to U,
7: extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
8: **end if**
9: **if** (U, V, T) is not consistent **then**
10: find $a \in \alpha$, $v \in V$ and $u, u' \in U$ s.t. $T(uv) = T(u'v)$ but $T(uav) \neq T(u'av)$,
11: add av to V,
12: extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
13: **end if**
14: **end while**
15: construct a conjectured DFA $\mathcal{A}$ and ask an equivalence query
16: **if** a counterexample $c \in (\mathcal{L}(\mathcal{A}) \backslash \mathcal{L}) \cup (\mathcal{L} \backslash \mathcal{L}(\mathcal{A}))$ is provided **then**
17: add c and all its prefixes to U,
18: extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
19: **else**
20: the correct DFA $\mathcal{A}$ has been learnt such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
21: **end if**
22: **until** $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
23: **return** $\mathcal{A}$

for all $u, u' \in U$, $v \in V$ and $a \in \alpha$, $T(uv) = T(u'v)$ implies $T(uav) = T(u'av)$). Once (U, V, T) is both closed and consistent, L* builds a conjectured DFA $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$, where the set of states S corresponds to the set of distinct rows labelled by $u \in U$, the initial state $\overline{s}$ corresponds to the row labelled by the empty word ϵ, the transition relation δ is defined as $\delta(u, a) = u'$ with $\forall v \in V : T(uav) = T(u'v)$, and the accepting set F consists of states corresponding to $u \in U$ with $T(u) = +$. An equivalence query is made for $\mathcal{A}$ in line 15. If the teacher's answer is "yes", then $\mathcal{A}$ accepts the target language $\mathcal{L}$, i.e. $\mathcal{L}(\mathcal{A}) = \mathcal{L}$, and the algorithm terminates; otherwise, a counterexample word c that witnesses the symmetric difference of $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}$ is provided. L* adds c

and all its prefixes to U, and extends T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries. It then continues to make sure that (U, V, T) is closed and consistent. The above procedure repeats until a correct DFA $\mathcal{A}$ is learnt.

L* is guaranteed to terminate with a minimal DFA for a target regular language [Ang87a]. The conjectured DFAs built by L* strictly increase in size, that is, each conjectured DFA is smaller than the next one and all incorrect DFAs are smaller than the correct DFA $\mathcal{A}$. Therefore, if $\mathcal{A}$ has n states, L* makes at most $n-1$ equivalence queries. Moreover, the number of membership queries is bounded by $\mathcal{O}(kmn^2)$ where k is the size of alphabet α and m is the length of the longest counterexample.

$\mathcal{T}_1$	ϵ
ϵ	−
a	−
b	−

$\mathcal{T}_2$	ϵ
ϵ	−
b	−
ba	+
a	−
bb	+
baa	−
bab	−

$\mathcal{T}_3$	ϵ	a
ϵ	−	−
b	−	+
ba	+	−
a	−	−
bb	+	+
baa	−	−
bab	−	+

$\mathcal{T}_4$	ϵ	a
ϵ	−	−
b	−	+
ba	+	−
bb	+	+
a	−	−
baa	−	−
bab	−	+
bba	+	−
bbb	+	+

Figure 3.6: Learning language $\mathcal{L} = \alpha^* b \alpha$ with Angluin's L*

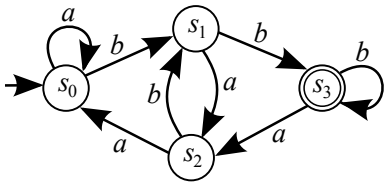


Figure 3.7: Minimal DFA accepting language $\mathcal{L} = \alpha^* b \alpha$ with 4 states

Example 3.12 *Figure 3.6 shows a run of Angluin's L* to learn a regular language* $\mathcal{L} = \alpha^* b \alpha$ *over the alphabet* $\alpha = \{a, b\}$*. The observation table* $\mathcal{T}_1$ *is closed and consistent, but does not represent the correct DFA because the word* ba*, which is in* $\mathcal{L}$*, is not accepted. Therefore, L* adds the word* ba *and its prefix* b *to* U*. The updated observation table* $\mathcal{T}_2$

is closed but not consistent, because $T(\epsilon) = T(b)$ but $T(a) \neq T(ba)$. A column labelled by a is added in $\mathcal{T}_3$. However, $\mathcal{T}_3$ is not closed, since there is no equivalent row labelled by elements of U for bb. After making $\mathcal{T}_3$ closed, we obtain $\mathcal{T}_4$, which is both closed and consistent. Based on $\mathcal{T}_4$, a DFA with 4 states is constructed as shown in Figure 3.7. It passes the equivalence query and, therefore, is the minimal DFA for language $\alpha^* b \alpha$. In this example run, the total number of equivalence queries is 2 and the number of membership queries is 18 (given by the entries of $\mathcal{T}_4$).

The Improved L* Algorithm

Rivest and Schapire [RS93] proposed a variant of Angluin's L* algorithm to improve the worst-case number of membership queries. In this thesis, we call this algorithm the *improved L** and use it to learn assumptions in Section 3.6.1.

Algorithm 2 shows the pseudo code of the improved L* algorithm. It maintains an observation table (U, V, T) in the same way as Angluin's L*. Indeed, the whole learning process is very similar to that of Angluin's algorithm. There are two essential differences. Firstly, when a counterexample c is found in the improved L*, only a suffix v of c would be added to V to update the observation table (line 11-13); on the contrary, in Angluin's L*, the counterexample word c and all its prefixes would be added to U (see line 17 of Algorithm 1). Secondly, the improved L* only checks whether (U, V, T) is *closed*, as there is no need to check the *consistency* condition because it is satisfied by construction; however, in Angluin's L*, both closedness and consistency conditions have to be checked explicitly.

Now we describe how to analyse a counterexample word c and find a suffix v to update (U, V, T). This is done by finding the earliest point in c at which the conjectured DFA $\mathcal{A}$ and the DFA that would accept $\mathcal{L}$ diverge in behaviour. This point i is found by determining $\zeta_i \neq \zeta_{i+1}$, where ζ_i is computed as follows:

1. for $0 \leq i \leq |c|$, let words p_i, r_i be such that $c = p_i \cdot r_i$, and $|p_i| = i$;

Algorithm 2 The improved L* learning algorithm by Rivest and Schapire [RS93]

Input: alphabet α, a teacher who knows the target language $\mathcal{L}$
Output: DFA $\mathcal{A}$

initialise (U, V, T): let $U = V = \{\epsilon\}$, ask membership queries and fill $T(w)$ for all words $w \in (U \cup U \cdot \alpha) \cdot V$
repeat
  while (U, V, T) is not closed **do**
    find $u \in U$, $a \in \alpha$, and $v \in V$ such that $T(uav) \neq T(u'v)$ for any $u' \in U$,
    add ua to U,
    extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
  end while
  construct a conjectured DFA $\mathcal{A}$ and ask an equivalence query
  if a counterexample $c \in (\mathcal{L}(\mathcal{A}) \backslash \mathcal{L}) \cup (\mathcal{L} \backslash \mathcal{L}(\mathcal{A}))$ is provided **then**
    determine a suffix v of c that distinguishes $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}$,
    add v to V,
    extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
  else
    the correct DFA $\mathcal{A}$ has been learnt such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
  end if
until $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
return $\mathcal{A}$

2. let s_i be the state reached in $\mathcal{A}$ after the first i actions of c have been executed; by construction, this state s_i corresponds to a row $u \in U$ of the observation table;

3. perform a membership query on the word $u \cdot r_i$;

4. return the result of the membership query as ζ_i.

By using binary search, the point i where $\zeta_i \neq \zeta_{i+1}$ can be found in $\mathcal{O}(\log m)$ queries (m is the length of the counterexample c).

Therefore, the worst-case number of overall membership queries for the improved L* has a bound of $\mathcal{O}(kn^2 + n \log m)$ where n is the size of a minimal DFA and k is the size of the alphabet α. Recall that the membership queries of Angluin's L* are bounded by $\mathcal{O}(kmn^2)$, so there is indeed a significant improvement.

Example 3.13 *Figure 3.8 shows a run of the improved L* to learn a regular language $\alpha^* b \alpha$ over the alphabet $\alpha = \{a, b\}$, which is the language learnt in Example 3.12. The*

$\mathcal{T}_1$	ϵ
ϵ	$-$
a	$-$
b	$-$

$\mathcal{T}_2$	ϵ	a
ϵ	$-$	$-$
a	$-$	$-$
b	$-$	$+$

$\mathcal{T}_3$	ϵ	a
ϵ	$-$	$-$
b	$-$	$+$
ba	$+$	$-$
bb	$+$	$+$
a	$-$	$-$
baa	$-$	$-$
bab	$-$	$+$
bba	$+$	$-$
bbb	$+$	$+$

Figure 3.8: Learning language $\mathcal{L} = \alpha^* b \alpha$ with the improved L*

first conjectured DFA based on the closed and consistent observation table $\mathcal{T}_1$ does not pass the equivalence query, because it does not accept the word ba, which is actually in the target language. Unlike Angluin's L, which adds the counterexample ba and all its prefixes to U, the improved L* adds a suffix a of ba to the column V and obtains $\mathcal{T}_2$. The updated observation table $\mathcal{T}_2$ is not closed, because there is no row in U equivalent to $row(b)$. By making $\mathcal{T}_2$ closed, a new $\mathcal{T}_3$ is built. Note that this $\mathcal{T}_3$ is exactly the same as $\mathcal{T}_4$ shown in Figure 3.6. Therefore, the improved L* learns the same minimal DFA for language $\alpha^* b \alpha$ as shown in Figure 3.7, using 2 equivalence queries and 18 membership queries. Compared to the run of Angluin's L* in Example 3.12, the improved L* does not show any improvement over the number of membership queries in this simple example. However, it may improve the worst-case scenario in some large examples.*

3.5.2 The NL* Algorithm

The NL* algorithm [BHKL09] is the first active learning algorithm for *residual finite-state automata* (RFSAs) [DLT02, DLT04], a subclass of NFAs. RFSAs share important properties with DFAs: for every regular language, there is a unique minimal RFSA accepting it. This characterisation enables L*-style learning of RFSAs. Indeed, RFSAs are the preferred class for learning regular languages, because they can be exponentially more succinct than the corresponding DFAs [DLT02] (for example, the minimal DFA

recognising word $\alpha^*0\alpha^n$ has 2^n states while its equivalent RFSA only needs $n+2$ states). We use NL* to learn assumptions represented as RFSAs later in Section 4.4.1.

For a regular language $\mathcal{L}$ over alphabet α, a language $\mathcal{L}' \subseteq \alpha^*$ is a *residual language* of $\mathcal{L}$ if there is a $u \in \alpha^*$ with $\mathcal{L}' = u^{-1}\mathcal{L} = \{v \in \alpha^* | uv \in \mathcal{L}\}$. We denote by $Res(\mathcal{L})$ the set of residual languages of $\mathcal{L}$. A *residual finite-state automaton (RFSA)* is an NFA whose states correspond to residual languages of the language it recognised. Formally, given a RFSA $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$, for each state $s \in S$, let $\mathcal{L}_s$ be the set of words $w \in \alpha^*$ with $\delta(s, w) \cap F \neq \emptyset$, then we have $\mathcal{L}_s \in Res(\mathcal{L}(\mathcal{A}))$. Intuitively, the states of an RFSA are a subset of the states of the corresponding minimal DFA; with nondeterminism, certain states of the DFA are no longer needed, since they correspond to the union of languages of other states. A residual is *composed* if it is the union of other residuals; otherwise, it is called *prime*.

Similarly to L*, the NL* algorithm also maintains an observation table (U, V, T) where U is a prefix-closed set of words, V is a suffix-closed set of words, and T is a finite function mapping $((U \cup U \cdot \alpha) \cdot V) \to \{+, -\}$. We associate each word $u \in (U \cup U \cdot \alpha)$ with a mapping $row(u) : V \to \{+, -\}$ such that $row(u)(v) = T(uv)$. We denote by $Rows$ the set of rows in observation table (U, V, T), and by $Rows^U$ the set $\{row(u) | u \in U\}$. The join $(r_1 \sqcup r_2) : V \to \{+, -\}$ of two rows $r_1, r_2 \in Rows$ is defined componentwise for each $v \in V : (r_1 \sqcup r_2) \cdot v = (r_1 \cdot v) \sqcup (r_2 \cdot v)$ where $(- \sqcup -) = -$ and $(+ \sqcup +) = (+ \sqcup -) = (- \sqcup +) = +$. A row $r \in Rows$ is called *composed* if there are rows $r_1, \ldots, r_n \in (Rows \setminus \{r\})$ such that $r = r_1 \sqcup \cdots \sqcup r_n$. Otherwise, r is called a *prime* row. The set of prime rows in (U, V, T) is denoted by $Primes$. We denote by $Primes^U = Primes \cap Rows^U$. For two rows $r, r' \in Rows$, r is covered by r', denoted by $r \sqsubseteq r'$, if for all $v \in V$, $T(rv) = +$ implies $T(r'v) = +$; moreover, if $r' \neq r$, then r is strictly covered by r', denoted as $r \sqsubset r'$.

Algorithm 3 illustrates the pseudo code of NL*, which follows a similar pattern as Angluin's L* (see Algorithm 1). Starting with the sets U, V only containing the

Algorithm 3 The NL* learning algorithm [BHKL09]

Input: alphabet α, a teacher who knows the target language $\mathcal{L}$
Output: RFSA $\mathcal{A}$

1: initialise (U, V, T): let $U = V = \{\epsilon\}$, ask membership queries and fill $T(w)$ for all words $w \in (U \cup U \cdot \alpha) \cdot V$
2: **repeat**
3: **while** (U, V, T) is not RFSA-closed or not RFSA-consistent **do**
4: **if** (U, V, T) is not RFSA-closed **then**
5: find $u \in U$ and $a \in \alpha$ such that $row(ua) \in Primes \backslash Primes^U$,
6: add ua to U,
7: extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
8: **end if**
9: **if** (U, V, T) is not RFSA-consistent **then**
10: find $u \in U, a \in \alpha$ and $v \in V$ such that $T(uav) = -$ and $T(u'av) = +$ for some $u' \in U$ with $row(u') \sqsubseteq row(u)$,
11: add av to V,
12: extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
13: **end if**
14: **end while**
15: construct a conjectured RFSA $\mathcal{A}$ and ask an equivalence query
16: **if** a counterexample $c \in (\mathcal{L}(\mathcal{A}) \backslash \mathcal{L}) \cup (\mathcal{L} \backslash \mathcal{L}(\mathcal{A}))$ is provided **then**
17: add c and all its suffixes to V,
18: extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
19: **else**
20: the correct RFSA $\mathcal{A}$ has been learnt such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
21: **end if**
22: **until** $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
23: **return** $\mathcal{A}$

empty word ϵ, the NL* fills the observation table incrementally by asking membership queries for all words $w \in (U \cup U \cdot \alpha) \cdot V$. Then, from line 3 to 14, NL* makes sure that the observation table (U, V, T) is *RFSA-closed* (i.e. for each row $r \in (Rows \backslash Rows^U)$, $r = \bigsqcup\{r' \in Primes^U | r' \sqsubseteq r\}$) and *RFSA-consistent* (i.e. for all $u, u' \in U$ and $a \in \alpha$, $row(u) \sqsubseteq row(u)$ implies $row(u'a) \sqsubseteq row(ua)$). Once (U, V, T) is both RFSA-closed and RFSA-consistent, a conjectured RFSA $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$ is constructed such that the states set S corresponds to the set of rows in $Primes^U$, the initial state $\overline{s}$ corresponds to $row(\epsilon)$, the transition relation δ is defined by $\delta(row(u), a) = \{r \in S | r \sqsubseteq row(ua)\}$ for $u \in U$ with $row(u) \in S$ and $a \in \alpha$, and the accepting state set

$\mathcal{T}_1$	ϵ
ϵ	$-$
b	$-$
a	$-$

$\mathcal{T}_2$	ϵ	ba	a
ϵ	$-$	$+$	$-$
a	$-$	$+$	$-$
b	$-$	$+$	$+$

$\mathcal{T}_3$	ϵ	ba	a
ϵ	$-$	$+$	$-$
b	$-$	$+$	$+$
ba	$+$	$+$	$-$
a	$-$	$+$	$-$
bb	$+$	$+$	$+$
baa	$-$	$+$	$-$
bab	$-$	$+$	$+$

Figure 3.9: Learning language $\mathcal{L} = \alpha^* b \alpha$ with NL*

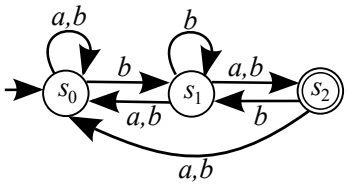


Figure 3.10: Minimal RFSA accepting language $\mathcal{L} = \alpha^* b \alpha$ with 3 states

$F = \{r \in S | row(r)(\epsilon) = +\}$. An equivalence query is asked for $\mathcal{A}$, and if the teacher's answer is "yes", then the correct RFSA has been learnt and the algorithm terminates; otherwise, a counterexample c illustrating the difference between the language $\mathcal{L}(\mathcal{A})$ and the target $\mathcal{L}$ is provided. NL* adds c and all its suffixes to V and continues the learning by updating (U, V, T).

Let n be the number of states of the minimal DFA for a regular language, m be the length of the longest counterexample, and k be the size of alphabet α. The theoretical complexity of the NL* algorithm is restricted by at most $\mathcal{O}(n^2)$ equivalence queries and $\mathcal{O}(kmn^3)$ membership queries, which is worse than the complexity of the L* algorithm (see Section 3.5.1). However, NL* often needs *fewer* equivalence and membership queries than L* in practice [BHKL09].

Example 3.14 *Figure 3.9 shows a run of NL* to learn a regular language* $\mathcal{L} = \alpha^* b \alpha$ *over alphabet* $\alpha = \{a, b\}$. *The observation table* $\mathcal{T}_1$ *is RFSA-closed and RFSA-consistent, but its corresponding RFSA does not accept the word* ba, *which actually belongs to the language* $\mathcal{L}$. *Thus, the counterexample word* ba *and its suffix* a *are added*

to V. The updated observation table $\mathcal{T}_2$ is not RFSA-closed, because $row(b)$ cannot be represented using $row(\epsilon)$. By making $\mathcal{T}_2$ RFSA-closed, a new observation table $\mathcal{T}_3$ is obtained. $\mathcal{T}_3$ is both RFSA-closed and RFSA-consistent, and its corresponding RFSA (Figure 3.10) passes the equivalence query. We can observe that the size of this RFSA (3 states) is smaller than the corresponding minimal DFA (4 states, see Figure 3.7). The total number of equivalence queries of this NL run is 2, and the number of membership queries is 22 (given by the entries of $\mathcal{T}_3$ and the additional check for determining counterexample suffix).*

3.5.3 The CDNF Algorithm

The CDNF algorithm [Bsh95] is an exact learning algorithm for Boolean functions. It learns a formula in *conjunctive disjunctive normal form* (CDNF) for a target Boolean function $\lambda(\mathbf{x})$ over a fixed set of Boolean variables $\mathbf{x}$. Later, we use this algorithm to learn assumptions for systems represented by Boolean functions in Section 3.6.2 and Chapter 6.

Before we describe the CDNF algorithm, we first introduce some terminology. We call a Boolean variable or its negation a *literal*. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions of literals, and a formula is in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions of literals. A CDNF formula is a conjunction of DNF formulae.

Similarly to the L* and NL* algorithms introduced in previous sections, the CDNF algorithm also follows the active learning model. It interacts with a *teacher* who knows about the target function $\lambda(\mathbf{x})$ via asking *membership* and *equivalence* queries. The membership query, denoted $MQ(\nu)$, asks whether a valuation ν over Boolean variables $\mathbf{x}$ satisfies the target function $\lambda(\mathbf{x})$. We use $\lambda[\nu]$ to denote the result of evaluating $\lambda(\mathbf{x})$ by replacing each $x \in \mathbf{x}$ with $\nu(x)$. If $\lambda[\nu] = \mathsf{true}$, then the teacher answers the membership query with "yes", denoted by $MQ(\nu) \rightarrow YES$; otherwise, the teacher

Algorithm 4 The CDNF learning algorithm [Bsh95]

Input: a teacher who knows about the target Boolean function $\lambda(\mathbf{x})$
Output: a CDNF formula $\theta(\mathbf{x})$ such that $\theta(\mathbf{x}) = \lambda(\mathbf{x})$

1: $t := 0$
2: **if** $EQ(\mathsf{true}) \to YES$ **then**
3: $\quad\theta(\mathbf{x}) := \mathsf{true}$
4: **else**
5: $\quad$let ν be the counterexample obtained from $EQ(\mathsf{true}) \to \nu$
6: $\quad t := t + 1$
7: $\quad(H_t, S_t, a_t) := (\mathsf{false}, \emptyset, \nu)$
8: $\quad$**if** $EQ(\bigwedge_{i=1}^{t} H_i) \to YES$ **then**
9: $\quad\quad\theta(\mathbf{x}) := \bigwedge_{i=1}^{t} H_i$
10: $\quad$**else**
11: $\quad\quad$let ν be the counterexample obtained from $EQ(\bigwedge_{i=1}^{t} H_i) \to \nu$
12: $\quad\quad I := \{i \mid H_i[\nu] = \mathsf{false}\}$
13: $\quad\quad$**if** $I = \emptyset$ **then**
14: $\quad\quad\quad$**goto** line 6
15: $\quad\quad$**else**
16: $\quad\quad\quad$**for each** $i \in I$ **do**
17: $\quad\quad\quad\quad s := \mathbf{walkTo}(a_i, \nu, \mathbf{x})$
18: $\quad\quad\quad\quad S_i := S_i \cup \{s \oplus a_i\}$
19: $\quad\quad\quad$**end for**
20: $\quad\quad\quad$**for each** $1 \leq i \leq t$ **do**
21: $\quad\quad\quad\quad H_i := M_{DNF}(S_i)[\mathbf{x} \to \mathbf{x} \oplus a_i]$
22: $\quad\quad\quad$**end for**
23: $\quad\quad$**end if**
24: $\quad$**end if**
25: $\quad$**goto** line 8
26: **end if**
27: **return** $\theta(\mathbf{x})$

answers "no", denoted by $MQ(\nu) \to NO$. The equivalence query, denoted $EQ(\theta(\mathbf{x}))$, asks whether a conjectured Boolean function $\theta(\mathbf{x})$ is equivalent to $\lambda(\mathbf{x})$. If so, the teacher answers "yes", denoted by $EQ(\theta(\mathbf{x})) \to YES$; otherwise, the teacher provides a counterexample ν, which is a valuation over Boolean variables $\mathbf{x}$ such that $\theta[\nu] \neq \lambda[\nu]$, denoted by $EQ(\theta(\mathbf{x})) \to \nu$.

At the implementation level, unlike L* and NL*, the CDNF algorithm does not maintain an observation table. Instead, it builds a conjunction of DNF formulae as

a conjecture for each equivalence query. As illustrated in Algorithm 4, the algorithm uses a variable t to record the number of DNF formulae in the current conjecture. The initial value of t is set as 0 (line 1), and the corresponding conjecture is degenerated to true. If the answer to the equivalence query $EQ(\mathsf{true})$ is *YES* (line 2), then true is a representation of the target function $\lambda(\mathbf{x})$ and the learning terminates. Otherwise, the teacher would provide a counterexample ν such that $\lambda[\nu] = \mathsf{false}$. The learning algorithm continues to refine the conjecture by adding a new DNF formula (line 6).

The algorithm uses three variables (H_i, S_i, a_i) to keep track of every DNF formula in the conjecture, where H_i represents the i-th DNF formula, S_i is a set of valuations over Boolean variables $\mathbf{x}$, and a_i is just one valuation over $\mathbf{x}$. When a new DNF formula H_t is added to the conjecture, the algorithm sets H_t as false, S_t as an empty set $\emptyset$, and the valuation $a_t = \nu$ (line 7). If the current conjecture $\bigwedge_{i=1}^{t} H_i$ yields *YES* for the equivalence query (line 8), then the algorithm terminates and returns the learnt CDNF formula as $\theta(\mathbf{x}) = \bigwedge_{i=1}^{t} H_i$.

If, on the other hand, the equivalence query $EQ(\bigwedge_{i=1}^{t} H_i)$ fails, then the algorithm refines the conjecture with the following procedure (line 10-24 of Algorithm 4). Given a counterexample ν of $EQ(\bigwedge_{i=1}^{t} H_i)$, the algorithm checks if the set $\{i \mid H_i[\nu] = \mathsf{false}\}$ is empty (line 12-13). If it is an empty set, then no DNF formula H_i in the current conjecture can be further refined and the conjecture has to be updated by adding a conjunction of a new DNF formula (line 14). If there exists some $H_i[\nu] = \mathsf{false}$, then the algorithm refines the i-th DNF formula by adding $s \oplus a_i$ to the set S_i (line 18), where s is the result of $\mathbf{walkTo}(a_i, \nu, \mathbf{x})$ (see Algorithm 5) and $\oplus$ is the component-wise exclusive-or operator (we define that by $(s \oplus a_i)(x) = s(x) \oplus a_i(x)$ for each Boolean variable $x \in \mathbf{x}$). The refined i-th DNF formula H_i is derived by function $M_{DNF}(S_i)[\mathbf{x} \to \mathbf{x} \oplus a_i]$ (line 21), where $M_{DNF}(S_i) = \bigvee_{\nu \in S_i} M_{DNF}(\nu)$ and $M_{DNF}(\nu) = \bigwedge x$ for all $x \in \mathbf{x}$ with $\nu(x) = \mathsf{true}$. Moreover, given a Boolean function $\theta(\mathbf{x})$, a new Boolean function $\theta[\mathbf{x} \to \mathbf{x} \oplus a_i]$ is obtained by replacing $x \in \mathbf{x}$ with $\neg x$ if $a_i(x) = \mathsf{true}$. The algorithm then

Algorithm 5 Function **walkTo**$(a, \nu, \mathbf{x})$ used in Algorithm 4

Input: valuations a, ν over a set of Boolean variables $\mathbf{x}$
Output: a valuation ν' closest to a such that $MQ(\nu') \to YES$

```
1: ν' := ν, j := 1
2: while j ≤ |x| do
3:    if ν'(x_j) = a(x_j) then
4:       j := j + 1
5:    else
6:       ν'(x_j) := a(x_j)
7:       if MQ(ν') → NO then
8:          ν'(x_j) := ¬a(x_j), j := j + 1
9:       else
10:         j := 0
11:      end if
12:   end if
13: end while
14: return ν'
```

asks a new equivalence query $EQ(\bigwedge_{i=1}^{t} H_i)$ for the refined conjecture (line 25). The learning continues until a correct representation $\theta(\mathbf{x})$ that is equivalent to the target function $\lambda(\mathbf{x})$ is learnt.

We now explain the details of **walkTo**$(a, \nu, \mathbf{x})$ as shown in Algorithm 5. The function **walkTo**$(a, \nu, \mathbf{x})$ computes a valuation ν' over Boolean variables $\mathbf{x}$ such that ν' is the closest (measured in Hamming distance) valuation to a and $MQ(\nu') \to YES$. Starting with $\nu' = \nu$, the algorithm finds a variable $x \in \mathbf{x}$ with $\nu'(x) \neq a(x)$ and flips the value of $\nu'(x)$. If the membership query $MQ(\nu')$ of the new valuation ν' yields NO, the algorithm reverts $\nu'(x)$ to its old value and keeps flipping another value; otherwise, the algorithm continues to flip other values of ν different from a.

As proved by [Bsh95], the CDNF algorithm learns an unknown Boolean function $\lambda(\mathbf{x})$ within $\mathcal{O}(mnk^2)$ membership and $\mathcal{O}(mn)$ equivalence queries, where m (resp. n) is the minimal size of all CNF (resp. DNF) formulae representing $\lambda(\mathbf{x})$, and k is the size of the Boolean variable set $\mathbf{x}$.

Example 3.15 *Figure 3.11 shows a run of the CDNF algorithm to learn a Boolean*

EQ	answer	I	S_i	H_i	a_i
true	$\nu(x_1x_2) = 00$		$S_1 = \emptyset$	$H_1 = \mathsf{false}$	$a_1(x_1x_2) = 00$
false	$\nu(x_1x_2) = 10$	$\{1\}$	$S_1 = \{10\}$	$H_1 = x_1$	
x_1	$\nu(x_1x_2) = 11$	$\emptyset$	$S_2 = \emptyset$	$H_2 = \mathsf{false}$	$a_2(x_1x_2) = 11$
$x_1 \wedge \mathsf{false}$	$\nu(x_1x_2) = 10$	$\{2\}$	$S_2 = \{01\}$	$H_2 = \neg x_2$	
$x_1 \wedge \neg x_2$	*YES*				

Figure 3.11: Learning Boolean function $x_1 \wedge \neg x_2$ with the CDNF algorithm

function $x_1 \wedge \neg x_2$*. The algorithm first makes an equivalence query* $EQ(\mathsf{true})$*, and obtains a counterexample valuation* $\nu(x_1x_2) = 00$*. Since* $t = 1$*, it sets* $S_1 = \emptyset$*,* $H_1 = \mathsf{false}$ *and* $a_1(x_1x_2) = \nu(x_1x_2) = 00$*. Next, an equivalence query* $EQ(\mathsf{false})$ *is made and another counterexample* $\nu(x_1x_2) = 10$ *is returned. The set* I *equals* $\{1\}$ *because* $H_1[10] = \mathsf{false}$*. The algorithm now walks from* $\nu(x_1x_2) = 10$ *to* $a_1(x_1x_2) = 00$*, and the result is* $\nu(x_1x_2) = 10$*. The set* S_1 *is expanded to* $S_1 = \emptyset \cup \{10 \oplus 00\} = \{10\}$*, and* H_1 *is refined as* x_1*. A counterexample* $\nu(x_1x_2) = 11$ *is provided as the answer to equivalence query* $EQ(x_1)$*. Since* $H_1[11] \neq \mathsf{false}$*, we have* $I = \emptyset$ *and the value of* t *increases to 2, which means that a second approximation* $H_2 = \mathsf{false}$ *would be added. The algorithm continues by asking equivalence query for* $H_1 \wedge H_2$*, i.e.* $EQ(x_1 \wedge \mathsf{false})$*. With the counterexample* $\nu(x_1x_2) = 10$*,* H_2 *is refined to* $\neg x_2$*. Now, the conjunction of approximations* H_1 *and* H_2 *gives us the target Boolean function* $x_1 \wedge \neg x_2$*. Therefore, the algorithm terminates.*

3.6 Learning Non-Probabilistic Assumptions

In this section, we describe two settings of learning *non-probabilistic* assumptions for compositional verification. The first setting is taken from [CGP03], in which the L* algorithm (Section 3.5.1) is used to learn assumptions represented as DFAs. The second setting is based on [CCF+10], where the CDNF algorithm (Section 3.5.3) is used to learn assumptions encoded implicitly as Boolean functions. In the following presentation, we use slightly different notations from the original papers for consistency reasons.

3.6.1 Learning assumptions using the L* Algorithm

[CGP03] shows how the L* learning algorithm could be adapted to learn assumptions for compositional verification of *finite labelled transition systems* (LTSs), which can be viewed as finite-state automata with all states accepting (Section 3.1).

The compositional verification is based on the following *asymmetric* assume-guarantee reasoning rule:

$$\frac{\langle true\rangle \mathcal{M}_1 \langle A\rangle \quad \langle A\rangle \mathcal{M}_2 \langle G\rangle}{\langle true\rangle \mathcal{M}_1 \parallel \mathcal{M}_2 \langle G\rangle} \qquad \text{(ASYM-LTS)}$$

where components $\mathcal{M}_1, \mathcal{M}_2$ and the assumption A are all LTSs, and G is a regular safety property (see Section 3.3.1). The assume-guarantee rule reasons about triples of the form $\langle A\rangle \mathcal{M} \langle G\rangle$, which means that "whenever component $\mathcal{M}$ is part of a system satisfying the assumption A, then the system is guaranteed to satisfy property G". The satisfaction check of $\langle A\rangle \mathcal{M} \langle G\rangle$ reduces to the standard (non-probabilistic) model checking of $A \| \mathcal{M} \models G$. When the assumption is absent from the triple, denoted $\langle true\rangle \mathcal{M} \langle G\rangle$, it means that "the component $\mathcal{M}$ always satisfies the property G". Thus, based on the assume-guarantee rule (ASYM-LTS), verifying a safety property G on a two-component LTS $\mathcal{M}_1 \| \mathcal{M}_2$ reduces to two separate checks on the models of $\mathcal{M}_1$ and $A \| \mathcal{M}_2$, which can be more efficient if the size of A is smaller than $\mathcal{M}_1$.

L* is adapted to automatically learn the assumption A for rule (ASYM-LTS). This is done by phrasing the problem in a language-theoretic setting: the set of (finite) traces of A forms a regular language over the alphabet $\alpha = \alpha_{\mathcal{M}_1} \cap (\alpha_{\mathcal{M}_2} \cup \alpha_G)$, where $\alpha_{\mathcal{M}_1}$, $\alpha_{\mathcal{M}_2}$ and α_G are the alphabets of $\mathcal{M}_1$, $\mathcal{M}_2$ and G, respectively. The target language $\mathcal{L}$ is defined by the notion of *weakest assumption*, i.e. the set of all possible traces of a process that, when put in parallel with $\mathcal{M}_2$, do not violate the regular safety property G. The approach presented in [CGP03] uses the improved L* algorithm by Rivest and

Schapire (see Section 3.5.1), and uses the process algebra model checker LTSA [JJ99] as a *teacher* to answer membership and equivalence queries.

A membership query asks whether a trace t is in the target language $\mathcal{L}$. The teacher answers the query by checking whether $t \parallel \mathcal{M}_2 \models G$ holds, where t denotes the LTS comprising the single trace t. An equivalence query asks if a conjecture A is a correct assumption for rule (ASYM-LTS). To answer this query, the teacher needs to check both premises of the rule. The teacher first verifies $\langle A \rangle \mathcal{M}_2 \langle G \rangle$ via model checking $A \parallel \mathcal{M}_2 \models G$. If a violation and a counterexample trace c is detected, then the trace $c{\restriction_\alpha}$ (obtained by removing any action not in α from c) is returned to L* to refine the conjecture; otherwise, the teacher continues to verify premise $\langle true \rangle \mathcal{M}_1 \langle A \rangle$ via checking $\mathcal{M}_1 \models A$. If both premises hold, then A is an assumption of rule (ASYM-LTS) that proves the satisfaction of G on the system $\mathcal{M}_1 \| \mathcal{M}_2$. Note that the learnt assumption A can be stronger than the weakest assumption, i.e. $\mathcal{L}(A) \subseteq \mathcal{L}$.

However, if $\mathcal{M}_1 \not\models A$ and a trace c of $\mathcal{M}_1$ is found as a counterexample such that $c{\restriction_\alpha} \notin \mathcal{L}(A)$, then the teacher needs to perform a *counterexample analysis* to determine whether c would really cause the violation of G on the full system $\mathcal{M}_1 \| \mathcal{M}_2$. This is done by checking $c{\restriction_\alpha} \parallel \mathcal{M}_2 \models G$. If it is true, then the trace $c{\restriction_\alpha}$ is in the target language, i.e. $c{\restriction_\alpha} \in \mathcal{L}$. Thus, $c{\restriction_\alpha}$ witnesses a difference between $\mathcal{L}$ and $\mathcal{L}(A)$, and should be returned to L* to refine A. On the other hand, if $c{\restriction_\alpha} \parallel \mathcal{M}_2 \not\models G$, then we can claim that $\mathcal{M}_1 \parallel \mathcal{M}_2 \not\models G$ because c is a trace of $\mathcal{M}_1$.

The assumption learning approach described above tends to perform particularly well in practice when the size of a generated assumption and the size of its alphabet remain small. There are also a variety of subsequent improvements and extensions developed beyond this basic technique (see e.g. [PGB+08] for details).

3.6.2 Learning assumptions using the CDNF Algorithm

In contrast to L*-based assumption learning approaches, [CCF+10] uses the CDNF algorithm (Section 3.5.3) to learn assumptions encoded implicitly as Boolean functions. This has the advantage of generating more succinct assumptions than L*-based methods, and thus is more scalable.

This approach is based on the following assume-guarantee rule:

$$\frac{\mathcal{M}_1 \preceq \mathcal{A} \qquad \mathcal{A} \| \mathcal{M}_2 \models \phi}{\mathcal{M}_1 \| \mathcal{M}_2 \models \phi} \qquad \textit{(ASYM-BOOL)}$$

where $\mathcal{M}_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \delta_i(\mathbf{x}_i, \mathbf{x}_i'))$ for $i = 1, 2$ and $\mathcal{A} = (\mathbf{x}_1, \iota_\mathcal{A}(\mathbf{x}_1), \delta_\mathcal{A}(\mathbf{x}_1, \mathbf{x}_1'))$ are transition systems with state variables $\mathbf{x}_i$, initial predicates $\iota_i(\mathbf{x}_i), \iota_\mathcal{A}(\mathbf{x}_1)$ and transition relations $\delta_i(\mathbf{x}_i, \mathbf{x}_i'), \delta_\mathcal{A}(\mathbf{x}_1, \mathbf{x}_1')$, and ϕ is a state predicate over $\mathbf{x}_1 \cup \mathbf{x}_2$. We say that $\mathcal{M}_1$ refines $\mathcal{A}$, denoted $\mathcal{M}_1 \preceq \mathcal{A}$, if the following implication conditions hold: $\forall \mathbf{x}_1.\ \iota_1(\mathbf{x}_1) \implies \iota_\mathcal{A}(\mathbf{x}_1)$ and $\forall \mathbf{x}_1 \mathbf{x}_1'.\ \delta_1(\mathbf{x}_1, \mathbf{x}_1') \implies \delta_\mathcal{A}(\mathbf{x}_1, \mathbf{x}_1')$. The composition of $\mathcal{M}_1$ and $\mathcal{M}_2$ is given by $\mathcal{M}_1 \| \mathcal{M}_2 = (\mathbf{x}_1 \cup \mathbf{x}_2, \iota_1(\mathbf{x}_1) \cup \iota_2(\mathbf{x}_2), \delta_1(\mathbf{x}_1, \mathbf{x}_1') \cup \delta_2(\mathbf{x}_2, \mathbf{x}_2'))$.

A *trace* $t = \nu_0 \nu_1 \cdots \nu_n$ of $\mathcal{M} = (\mathbf{x}, \iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{x}'))$ is a finite sequence of valuations ν_j over $\mathbf{x}$ such that $\iota[\nu_0] = \mathsf{true}$ and $\delta[\nu_j, \nu_{j+1}] = \mathsf{true}$ for $0 \leq j < n$. We denote by $\mathit{Traces}(\mathcal{M})$ the set of traces in $\mathcal{M}$. Let a state predicate $\phi(\mathbf{x})$ be a Boolean function over variables $\mathbf{x}$. We say that $\mathcal{M}$ satisfies $\phi(\mathbf{x})$, denoted $\mathcal{M} \models \phi(\mathbf{x})$ if, for any trace $t = \nu_0 \nu_1 \cdots \nu_n \in \mathit{Traces}(\mathcal{M})$, we have $\phi[\nu_j] = \mathsf{true}$ for $0 \leq j \leq n$. A witness to $\mathcal{M} \not\models \phi(\mathbf{x})$ is a trace $\nu_0 \nu_1 \cdots \nu_n$ of $\mathcal{M}$ such that $\phi[\nu_j] = \mathsf{true}$ for $0 \leq j < n$ but $\phi[\nu_n] = \mathsf{false}$.

In order to learn the assumption $\mathcal{A} = (\mathbf{x}_1, \iota_\mathcal{A}(\mathbf{x}_1), \delta_\mathcal{A}(\mathbf{x}_1, \mathbf{x}_1'))$, two instances of the CDNF algorithm are used: one for the initial predicate $\iota_\mathcal{A}(\mathbf{x}_1)$, and the other for the transition relation $\delta_\mathcal{A}(\mathbf{x}_1, \mathbf{x}_1')$. Let ν be a valuation over $\mathbf{x}_1$. The membership query $MQ(\nu)$ asks if ν is a satisfying valuation for the initial predicate $\iota_\mathcal{A}(\mathbf{x}_1)$ of an unknown assumption $\mathcal{A}$. The teacher checks whether $\iota_1[\nu] = \mathsf{true}$ holds. If so, then $\iota_\mathcal{A}[\nu] = \mathsf{true}$

because $\forall \mathbf{x}_1.\ \iota_1(\mathbf{x}_1) \implies \iota_{\mathcal{A}}(\mathbf{x}_1)$ due to $\mathcal{M}_1 \preceq \mathcal{A}$. Thus, the answer to $MQ(\nu)$ is *YES*. Otherwise, the teacher simply returns *NO* for the sake of termination. Similarly, given valuations ν, ν' over $\mathbf{x}_1$ and $\mathbf{x}'_1$ respectively, the answer to membership query $MQ(\nu, \nu')$ for the target transition relation $\delta_{\mathcal{A}}(\mathbf{x}_1, \mathbf{x}'_1)$ is *YES* if $\delta_1[\nu, \nu'] = \mathsf{true}$ and *NO* otherwise.

Let $C = (\mathbf{x}_1, \iota(\mathbf{x}_1), \delta(\mathbf{x}_1, \mathbf{x}'_1))$ be a conjectured assumption. The answer to an equivalence query $EQ(\iota, \delta)$ is *YES* if and only if $\mathcal{M}_1 \preceq C$ and $C \| \mathcal{M}_2 \models \phi$. To verify $\mathcal{M}_1 \preceq C$, the teacher first checks if $\iota_1(\mathbf{x}_1) \wedge \neg\iota(\mathbf{x}_1)$ is satisfiable. If there is a valuation ν such that $\iota_1[\nu] \wedge \neg\iota[\nu] = \mathsf{true}$, then $\forall \mathbf{x}_1.\ \iota_1(\mathbf{x}_1) \implies \iota(\mathbf{x}_1)$ does not hold and hence $\mathcal{M}_1 \not\preceq C$. The valuation ν is returned to the CDNF learning instance for $\iota_{\mathcal{A}}(\mathbf{x}_1)$ as a counterexample. Similarly, if $\delta_1(\mathbf{x}_1, \mathbf{x}'_1) \wedge \neg\delta(\mathbf{x}_1, \mathbf{x}'_1)$ is satisfied by valuation $\nu\nu'$, then $\nu\nu'$ is returned as a counterexample to the CDNF learning of $\delta_{\mathcal{A}}(\mathbf{x}_1, \mathbf{x}'_1)$. Assume that $\mathcal{M}_1 \preceq C$ holds, the teacher continues to check if $C \| \mathcal{M}_2 \models \phi$ by using the interpolation-based model checking algorithm [McM03]. If the model checking result is true, then both premises of the assume-guarantee rule are fulfilled, and thus we can conclude that $\mathcal{M}_1 \| \mathcal{M}_2 \models \phi$. Otherwise, the model checking algorithm returns a witness t to $C \| \mathcal{M}_2 \models \phi$. However, t may not necessarily be a witness for $\mathcal{M}_1 \| \mathcal{M}_2 \not\models \phi$ because $t{\restriction_{\mathbf{x}_1}}$ may not be trace of $\mathcal{M}_1$, where $t{\restriction_{\mathbf{x}_1}}$ is the *restriction* of trace t on $\mathbf{x}_1$. Therefore, an additional check is needed to inspect if $t{\restriction_{\mathbf{x}_1}}$ is a trace of $\mathcal{M}_1$. If so, then $\mathcal{M}_1 \| \mathcal{M}_2 \not\models \phi$; otherwise, the conjecture $C = (\mathbf{x}_1, \iota(\mathbf{x}_1), \delta(\mathbf{x}_1, \mathbf{x}'_1))$ needs to be modified.

[CCF+10] reports very encouraging experimental results of this assumption learning approach, which performs better than the monolithic interpolation-based model checking in three parametrised case studies: the MSI cache coherence protocol, synchronous bus arbiters, and dining philosophers.

Chapter 4

Learning Assumptions for Asynchronous Probabilistic Systems

In this chapter, we present a novel approach to automatically learn probabilistic assumptions for compositional verification of systems modelled as probabilistic automata (PAs). Such a system is always composed from two or more PA components using the *asynchronous* parallel operator, modelling the concurrent behaviour of multiple probabilistic processes.

Our approach builds upon the compositional verification framework of [KNPQ10], in which several assume-guarantee reasoning rules for PAs are proposed. In these rules, assumptions and guarantees are represented as probabilistic safety properties. This framework has been successfully applied on a set of large case studies, including cases where non-compositional verification is infeasible due to the size of models. A limitation, however, is that it requires non-trivial manual effort to construct appropriate assumptions to enable the compositional verification.

We address this limitation by proposing a fully-automated approach to generate assumptions for the compositional verification framework. Inspired by the work of learning non-probabilistic assumptions with L* [CGP03, PGB+08], our approach also uses active learning algorithms (e.g. L* and NL*) to generate a series of conjectured finite-state automata for assumptions.

In this chapter, we first introduce the compositional verification framework of [KNPQ10] in Section 4.1. We define the notion of probabilistic counterexamples for the model checking of PAs in Section 4.2. In Section 4.3, we present our approach for learning assumptions for a basic assume-guarantee rule (Asym) using the L* algorithm. Then, we discuss three extensions in Section 4.4, including learning assumptions using the NL* algorithm and generalisations for learning assumptions for rule (Asym-N) and (Circ). We illustrate the applicability of our approach through experiments on a range of case studies in Section 4.5. Finally, we summarise the strengths and weaknesses of our approach in Section 4.6.

This work was previously published in two jointly authored papers [FKP10] and [FKP11], and is presented here with more explanations and discussions, additional running examples and extensions. See Section 1.3 for the credits of this chapter.

4.1 Compositional Verification for PAs

In this section, we introduce the compositional verification framework for PAs based on [KNPQ10], including a few concepts needed for reasoning about systems consisting of multiple PA components (e.g. parallel composition, adversary projection), and three different assume-guarantee rules (Asym), (Asym-N) and (Circ).

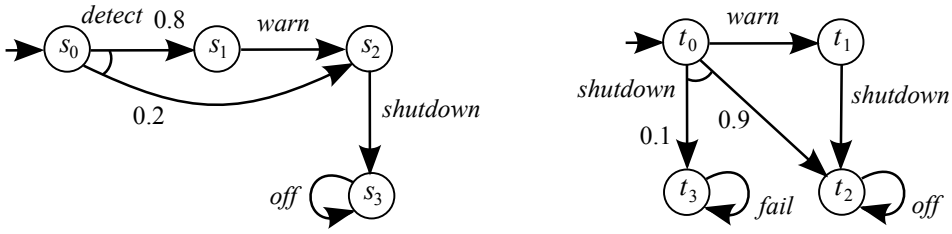


Figure 4.1: A pair of PAs $\mathcal{M}_1, \mathcal{M}_2$ (taken from [KNPQ10])

4.1.1 Concepts for Compositional Reasoning about PAs

Firstly, we describe how two or more PAs (Definition 3.5) are composed together. This is done by using the standard *asynchronous* parallel operator defined by [Seg95], where PAs synchronise over shared actions and interleave otherwise.

Definition 4.1 (Parallel composition of PAs) *Let $\mathcal{M}_1, \mathcal{M}_2$ be two PAs such that $\mathcal{M}_i = (S_i, \overline{s}_i, \alpha_i, \delta_i, L_i)$ for $i = 1, 2$. Their parallel composition is the PA $\mathcal{M}_1 \| \mathcal{M}_2 = (S_1 \times S_2, (\overline{s}_1, \overline{s}_2), \alpha_1 \cup \alpha_2, \delta, L)$, where the transition relation δ is defined such that $(s_1, s_2) \xrightarrow{a} \mu_1 \times \mu_2$ if and only if one of the following holds:*

- $s_1 \xrightarrow{a} \mu_1$, $s_2 \xrightarrow{a} \mu_2$ *and* $a \in \alpha_1 \cap \alpha_2$
- $s_1 \xrightarrow{a} \mu_1$, $\mu_2 = \eta_{s_2}$ *and* $a \in (\alpha_1 \backslash \alpha_2) \cup \{\tau\}$
- $s_2 \xrightarrow{a} \mu_2$, $\mu_1 = \eta_{s_1}$ *and* $a \in (\alpha_2 \backslash \alpha_1) \cup \{\tau\}$

and $L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$.

Example 4.2 *Figure 4.1 shows a pair of PAs: $\mathcal{M}_1$ represents a sensor which issues a "warn" signal followed by "shutdown" when it detects some unusual condition, but it may also issue "shutdown" directly with probability 0.2; $\mathcal{M}_2$ represents a device which powers down correctly when receiving "warn" before "shutdown", and may cause system failure with probability 0.1 otherwise. The product PA $\mathcal{M}_1 \| \mathcal{M}_2$ obtained from their parallel composition is shown in Figure 4.2, where $\mathcal{M}_1, \mathcal{M}_2$ synchronise over their shared actions "warn", "shutdown" and "off", and interleave with "detect" and "fail".*

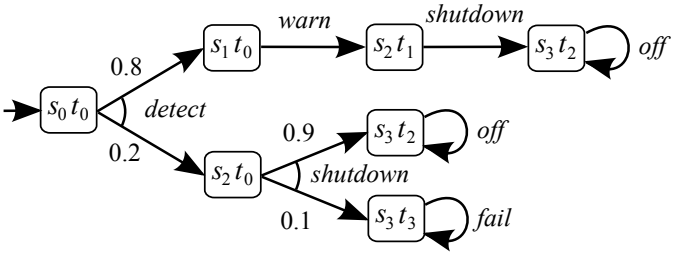


Figure 4.2: Parallel composition product $\mathcal{M}_1 \| \mathcal{M}_2$ (taken from [KNPQ12])

We also need the notion of *projections* to decompose models that are constructed through parallel composition. Given any state $s = (s_1, s_2)$ of $\mathcal{M}_1 \| \mathcal{M}_2$, the projection of s onto a component $\mathcal{M}_i$, denoted by $s\lceil_{\mathcal{M}_i}$, equals s_i. We can extend this notation to distributions over $\mathcal{M}_1 \| \mathcal{M}_2$ in the standard manner. The projection of any path π of $\mathcal{M}_1 \| \mathcal{M}_2$ onto $\mathcal{M}_i$, denoted $\pi\lceil_{\mathcal{M}_i}$, is the path obtained from π by projecting each state of π onto $\mathcal{M}_i$ and removing all the actions not in the alphabet of $\mathcal{M}_i$ together with the subsequent states. Note that the projection of an infinite path may be finite. Recall from Section 3.2.2 that PAs use adversaries to resolve the nondeterminism; the projection of a (complete) adversary σ of $\mathcal{M}_1 \| \mathcal{M}_2$ onto one component $\mathcal{M}_i$ is a (partial) adversary of $\mathcal{M}_i$, formally:

Definition 4.3 (Adversary projection) *Let $\mathcal{M}_1, \mathcal{M}_2$ be PAs and σ an adversary of $\mathcal{M}_1 \| \mathcal{M}_2$. The* projection *of σ onto $\mathcal{M}_i$, denoted $\sigma\lceil_{\mathcal{M}_i}$, is the (partial) adversary on $\mathcal{M}_i$, where, for any finite path ρ_i of $\mathcal{M}_i$, we have $\sigma\lceil_{\mathcal{M}_i}(\rho_i)(a, \mu_i)$ equals*

$$\frac{\sum\{\!|\, Pr^{\sigma}_{\mathcal{M}_1 \| \mathcal{M}_2}(\rho) \cdot \sigma(\rho)(a,\mu) \mid \pi \in FPaths^{\sigma}_{\mathcal{M}_1 \| \mathcal{M}_2} \ \wedge\ \rho\lceil_{\mathcal{M}_i} = \rho_i \ \wedge\ \mu\lceil_{\mathcal{M}_i} = \mu_i \,|\!\}}{Pr^{\sigma\lceil_{\mathcal{M}_i}}_{\mathcal{M}_i}(\rho_i)}$$

In the above definition, the projected adversary $\sigma\lceil_{\mathcal{M}_i}$ on component $\mathcal{M}_i$ assigns non-zero probabilities to the action-distribution pair (a, μ_i) followed the finite path ρ_i, which equals to the corresponding probabilities mass on $\mathcal{M}_1 \| \mathcal{M}_2$ dividing the probability of path ρ_i on $\mathcal{M}_i$ under the adversary $\sigma\lceil_{\mathcal{M}_i}$.

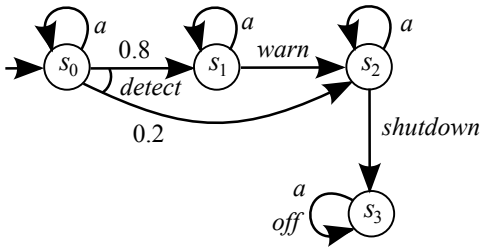


Figure 4.3: PA $\mathcal{M}_1[\alpha]$ for $\mathcal{M}_1$ in Figure 4.1 and $\alpha = \{a\}$

Example 4.4 *Consider the PA $\mathcal{M}_1 \| \mathcal{M}_2$ in Figure 4.2. The projection of state s_0t_0 onto $\mathcal{M}_2$ is t_0. The projection of path*

$$\rho = s_0t_0 \xrightarrow{detect,0.2} s_2t_0 \xrightarrow{shutdown,0.1} s_3t_3 \xrightarrow{fail,1} s_3t_3$$

onto $\mathcal{M}_2$ is given by $\rho\lfloor_{\mathcal{M}_2} = t_0 \xrightarrow{shutdown,0.1} t_3 \xrightarrow{fail,1} t_3$. Suppose σ is an adversary of $\mathcal{M}_1 \| \mathcal{M}_2$ such that $\sigma(\rho)(fail, \mu) = 1$, then $\sigma\lfloor_{\mathcal{M}_2}(\rho\lfloor_{\mathcal{M}_2})(fail, \mu') = 1$, where μ is a distribution over the state space of $\mathcal{M}_1 \| \mathcal{M}_2$ with $\mu(s_3t_3) = 1$, and μ' is a projection of μ on $\mathcal{M}_2$ such that $\mu = \eta_{s_3} \times \mu'$.

When reasoning about the behavior of multi-component system, the assumption and guarantee may contain actions that are not present in an individual component. The notion of *alphabet extension* adds self-loops labelled with all the actions from an additional alphabet to all states of a PA $\mathcal{M}$.

Definition 4.5 (Alphabet extension) *For any PA $\mathcal{M} = (S, \overline{s}, \alpha_{\mathcal{M}}, \delta_{\mathcal{M}}, L)$ and set of actions α, we extend the alphabet of $\mathcal{M}$ to include α, denoted as $\mathcal{M}[\alpha] = (S, \overline{s}, \alpha_{\mathcal{M}} \cup \alpha, \delta_{\mathcal{M}[\alpha]}, L)$ where $\delta_{\mathcal{M}[\alpha]} = \delta_{\mathcal{M}} \cup \{(s, a, \eta_s) \mid s \in S \wedge a \in (\alpha \backslash \alpha_{\mathcal{M}})\}$.*

Example 4.6 *Consider the PA $\mathcal{M}_1$ shown in Figure 4.1 and an alphabet $\alpha = \{a\}$. Figure 4.3 shows a new PA $\mathcal{M}_1[\alpha]$ as the result of alphabet extension, where self-loops labelled with actions a are added to all states.*

Recall from Section 3.6.1 that the compositional verification of (non-probabilistic) labelled transition systems uses assume-guarantee triples of the form $\langle A\rangle\mathcal{M}\langle G\rangle$, which means that "whenever component $\mathcal{M}$ is part of a system satisfying the assumption A, then the system is guaranteed to satisfy property G". Analogously, for the compositional verification of PAs, we use *probabilistic assume-guarantee triples* of the form $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$, where the component $\mathcal{M}$ is a PA, and both the assumption $\langle A\rangle_{\geq p_A}$ and the guarantee $\langle G\rangle_{\geq p_G}$ are probabilistic safety properties as defined in Section 3.3.1. Informally, the triple means that "whenever $\mathcal{M}$ is part of a system satisfying A with probability at least p_A, then the system is guaranteed to satisfy G with probability at least p_G". Formally:

Definition 4.7 (Probabilistic assume-guarantee triple for PA) *Let $\mathcal{M}$ be a PA, and $\langle A\rangle_{\geq p_A}$, $\langle G\rangle_{\geq p_G}$ be a pair of probabilistic safety properties with the alphabets $\alpha_G \subseteq \alpha_A \cup \alpha_{\mathcal{M}}$. Then $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$ is a* probabilistic assume-guarantee triple *with the following semantics:*

$$\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G} \;\Leftrightarrow\; \forall\sigma{\in}Adv_{\mathcal{M}[\alpha_A]}\;.\;\Big(Pr^{\sigma}_{\mathcal{M}[\alpha_A]}(A){\geq}p_A \implies Pr^{\sigma}_{\mathcal{M}[\alpha_A]}(G){\geq}p_G\Big)$$

where $\mathcal{M}[\alpha_A]$ is $\mathcal{M}$ with its alphabet extended to include α_A.

Note that the alphabet extension $\mathcal{M}[\alpha_A]$ is necessary because the assumption and guarantee may contain actions that are not in $\alpha_{\mathcal{M}}$ when reasoning about systems with multiple components containing $\mathcal{M}$. To check the safety property G on $\mathcal{M}[\alpha_A]$, we restrict the alphabet $\alpha_G \subseteq \alpha_A \cup \alpha_{\mathcal{M}}$. We check whether $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$ is *not* true by checking for the existence of an adversary that satisfies the assumption $\langle A\rangle_{\geq p_A}$ but does not satisfy the guarantee $\langle G\rangle_{\geq p_G}$.

Proposition 4.8 ([KNPQ10]) *Let $\mathcal{M}$ be a PA, $\langle A\rangle_{\geq p_A}$ and $\langle G\rangle_{\geq p_G}$ be a pair of probabilistic (regular) safety properties. Let $\mathcal{M}' = \mathcal{M}[\alpha_A]{\otimes}A^{err}{\otimes}G^{err}$. The probabilistic*

assume-guarantee triple $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$ *holds if and only if:*

$$\neg\exists\sigma'\in Adv_{\mathcal{M}'}\ .\ \left(Pr^{\sigma'}_{\mathcal{M}'}(\Box\neg err_A)\geq p_A \wedge Pr^{\sigma'}_{\mathcal{M}'}(\Diamond err_G)>1-p_G\right)$$

where A^{err}, G^{err} *are DFAs representing safety properties* A *and* G *respectively, and* err_A, err_G *are accepting states in corresponding DFAs. (Note that the operator* $\otimes$ *for composing a PA and a DFA is defined in Section 3.3.3.)*

Thus, checking the satisfaction of triple $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$ reduces to the multi-objective model checking problem (Section 3.3.3), which can be checked in time polynomial in $|\mathcal{M}'|$ by solving an LP problem. When the assumption is absent from the triple, denoted by $\langle true\rangle\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$, the satisfaction check reduces to verifying $\mathcal{M}\models\langle G\rangle_{\geq p_G}$. Recall from Section 3.3.3 that checking a probabilistic safety property $\langle G\rangle_{\geq p_G}$ on PA $\mathcal{M}$ reduces to computing the maximum reachability probabilities on PA $\mathcal{M}\otimes G^{err}$.

We also consider *quantitative* queries about the triples: $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{I_G=?}$ and $\langle A\rangle_{I_A=?}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$, which ask for the tightest lower-bounded interval $I_G\subseteq[0,1]$ for which the triple $\langle A\rangle_{\geq p_A}\,\mathcal{M}\,\langle G\rangle_{I_G}$ holds with a fixed value of p_A, and the widest lower-bounded interval $I_A\subseteq[0,1]$ for which the triple $\langle A\rangle_{I_A}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$ holds with a fixed value of p_G, respectively. Both queries can be checked using the quantitative multi-objective queries as described in Section 3.3.3. Intuitively, these queries enable us to compute the strongest possible guarantee that can be obtained for some assumption $\langle A\rangle_{\geq p_A}$ and the weakest possible assumption (not to be confused with the notion of "weakest assumption" in Section 3.6.1) that guarantees a particular property $\langle G\rangle_{\geq p_G}$. Note that the answer to the query $\langle A\rangle_{I_A=?}\,\mathcal{M}\,\langle G\rangle_{\geq p_G}$ can be an empty interval, i.e. $I_A=\emptyset$; this occurs when we cannot guarantee that $\langle G\rangle_{\geq p_G}$ holds even under the strongest possible assumption for A.

4.1.2 Assume-Guarantee Reasoning Rules

Based on the probabilistic assume-guarantee triples (Definition 4.7), we now present three assume-guarantee reasoning rules for the compositional verification of systems built from two or more PA components.

The first rule we consider is an *asymmetric* assume-guarantee rule (Asym) for two-component PA systems, which is an analogue of rule (Asym-Lts) in Section 3.6.1.

Theorem 4.9 ([KNPQ10]) *If $\mathcal{M}_1, \mathcal{M}_2$ are PAs and $\langle A \rangle_{\geq p_A}, \langle G \rangle_{\geq p_G}$ are probabilistic safety properties such that their alphabets satisfy $\alpha_A \subseteq \alpha_{\mathcal{M}_1}$ and $\alpha_G \subseteq \alpha_{\mathcal{M}_2} \cup \alpha_A$, then the following proof rule holds:*

$$\frac{\begin{array}{c}\langle true \rangle\, \mathcal{M}_1\, \langle A \rangle_{\geq p_A} \\ \langle A \rangle_{\geq p_A}\, \mathcal{M}_2\, \langle G \rangle_{\geq p_G}\end{array}}{\langle true \rangle\, \mathcal{M}_1 \parallel \mathcal{M}_2\, \langle G \rangle_{\geq p_G}} \qquad \text{(Asym)}$$

Note that the alphabet restrictions are given by Definition 4.7. With rule (Asym) and an appropriate assumption $\langle A \rangle_{\geq p_A}$, the verification of property $\langle G \rangle_{\geq p_G}$ on $\mathcal{M}_1 \| \mathcal{M}_2$ can be done compositionally by

(1) verifying a probabilistic safety property on $\mathcal{M}_1$, which reduces to computing reachability probabilities on $\mathcal{M}_1 \otimes A^{err}$,

(2) checking a probabilistic assume-guarantee triple on $\mathcal{M}_2$, which reduces to the multi-objective model checking on $\mathcal{M}_2[\alpha_A] \otimes A^{err} \otimes G^{err}$.

Thus, the construction and verification of the full system $\mathcal{M}_1 \| \mathcal{M}_2$ can be avoided. If the size of the assumption's DFA A^{err} is much smaller than the size of the corresponding component $\mathcal{M}_1$, we can expect significant gains of the verification performance.

The second rule we consider is an asymmetric assume-guarantee rule (Asym-N) for

n-component systems, which is obtained through repeated application of rule (ASYM).

Theorem 4.10 ([KNPQ10]) *If $\mathcal{M}_1, \ldots, \mathcal{M}_n$ are PAs and $\langle A_1 \rangle_{\geq p_1}, \ldots, \langle A_{n-1} \rangle_{\geq p_{n-1}}$, $\langle G \rangle_{\geq p_G}$ are probabilistic safety properties with their alphabets satisfying $\alpha_{A_1} \subseteq \alpha_{\mathcal{M}_1}$, $\alpha_{A_2} \subseteq \alpha_{\mathcal{M}_2} \cup \alpha_{A_1}, \ldots,$ and $\alpha_G \subseteq \alpha_{\mathcal{M}_n} \cup \alpha_{A_{n-1}}$, then the following proof rule holds:*

$$\frac{\begin{array}{c}\langle \mathit{true} \rangle\, \mathcal{M}_1\, \langle A_1 \rangle_{\geq p_1} \\ \langle A_1 \rangle_{\geq p_1}\, \mathcal{M}_2\, \langle A_2 \rangle_{\geq p_2} \\ \ldots \\ \langle A_{n-1} \rangle_{\geq p_{n-1}}\, \mathcal{M}_n\, \langle G \rangle_{\geq p_G}\end{array}}{\langle \mathit{true} \rangle\, \mathcal{M}_1 \parallel \ldots \parallel \mathcal{M}_n\, \langle G \rangle_{\geq p_G}} \quad (\text{ASYM-N})$$

Similarly to rule (ASYM), given appropriate assumptions $\langle A_1 \rangle_{\geq p_1}, \ldots, \langle A_{n-1} \rangle_{\geq p_{n-1}}$, we can use rule (ASYM-N) to verify $\langle G \rangle_{\geq p_G}$ on the n-component system $\mathcal{M}_1 \parallel \ldots \parallel \mathcal{M}_n$ compositionally, without constructing the full system. The compositional verification involves one instance of the standard model checking of probabilistic safety property $\langle A_1 \rangle_{\geq p_1}$ on $\mathcal{M}_1$, and $(n-1)$ instances of multi-objective model checking for the assume-guarantee triples on components $\mathcal{M}_i$ for $2 \leq i \leq n$. This rule is particularly useful when the construction and verification of the full system $\mathcal{M}_1 \parallel \ldots \parallel \mathcal{M}_n$ is not feasible due to the large size of the model.

The third rule we consider is a *circular* rule (CIRC). We present here a version for two-component systems, but it is straightforward to adapt it for n-component systems.

Theorem 4.11 ([KNPQ10]) *If $\mathcal{M}_1, \mathcal{M}_2$ are PAs, and $\langle A_1 \rangle_{\geq p_1}$, $\langle A_2 \rangle_{\geq p_2}$ and $\langle G \rangle_{\geq p_G}$ are probabilistic safety properties with their alphabets satisfying $\alpha_{A_1} \subseteq \alpha_{\mathcal{M}_2}$, $\alpha_{A_2} \subseteq$*

$\alpha_{\mathcal{M}_1} \cup \alpha_{A_1}$ *and* $\alpha_G \subseteq \alpha_{\mathcal{M}_2} \cup \alpha_{A_2}$*, then the following proof rule holds:*

$$\frac{\begin{array}{c}\langle true\rangle\ \mathcal{M}_2\ \langle A_1\rangle_{\geq p_1} \\ \langle A_1\rangle_{\geq p_1}\ \mathcal{M}_1\ \langle A_2\rangle_{\geq p_2} \\ \langle A_2\rangle_{\geq p_2}\ \mathcal{M}_2\ \langle G\rangle_{\geq p_G}\end{array}}{\langle true\rangle\ \mathcal{M}_1 \parallel \mathcal{M}_2\ \langle G\rangle_{\geq p_G}} \quad \text{(Circ)}$$

Recall that, in the asymmetric rule (Asym), only one assumption is made about component $\mathcal{M}_1$; however, we may not be able to show that the assumption is true for $\mathcal{M}_1$ without making additional assumptions about $\mathcal{M}_2$, if these two components have a "circular" dependency (i.e. $\mathcal{M}_1$ depends on $\mathcal{M}_2$ and vice versa). This limitation can be overcome by using rule (Circ), where two assumptions $\langle A_1\rangle_{\geq p_1}$ and $\langle A_2\rangle_{\geq p_2}$ are used. Note that this rule is very similar to rule (Asym-N) with $n = 3$, where the first and the last components coincide (justifying the name "circular").

We conclude this section by pointing out that the compositional verification framework described above is *incomplete*, in the sense that, if a multi-component system satisfies a certain property, there may not exist appropriate assumptions that can be used to verify it compositionally. For frameworks in which assumptions are expressed in the same formalism as the models, completeness can be trivially achieved by using the component itself as the assumption in the worst-case. However, in this framework, the formalisms of assumptions (probabilistic safety properties) and components (PAs) are distinct; thus, the above completeness argument cannot be applied here.

4.2 Probabilistic Counterexamples for PAs

In this section, we introduce the notion of probabilistic counterexamples for model checking PAs, based on a jointly authored paper [FKP10]. Recall that, in Section 3.4, counterexamples for probabilistic reachability properties of DTMCs are formulated as a

set of finite paths; in particular, the notion of *smallest counterexample* is characterised, which can be generated via solving the k-shortest paths problem. In the following, we show how these basic techniques can be extended for generating counterexamples for probabilistic safety properties and probabilistic assume-guarantee triples of PAs.

We first consider the counterexamples for probabilistic safety properties of PAs. Recall from Section 3.3.1 that a PA $\mathcal{M}$ satisfies a probabilistic safety property $\langle G \rangle_{\geq p_G}$ if and only if $Pr^{\sigma}_{\mathcal{M}}(G) \geq p_G$ for all adversaries σ of $\mathcal{M}$. So, to refute $\langle G \rangle_{\geq p_G}$, we would need to find an adversary $\sigma \in Adv_{\mathcal{M}}$ for which $Pr^{\sigma}_{\mathcal{M}}(G) \geq p_G$ does not hold and a set C of paths that illustrates the violation under adversary σ. Formally, a *counterexample* for the violation of probabilistic safety property $\langle G \rangle_{\geq p_G}$ on PA $\mathcal{M}$ is a pair (σ, C) of a (deterministic, finite-memory) adversary σ for $\mathcal{M}$ with $Pr^{\sigma}_{\mathcal{M}}(G) < p_G$, and a set C of finite paths in PA $\mathcal{M}^{\sigma}$ such that $Pr(C) > 1 - p_G$ and $\rho\lfloor_{\mathcal{M}} \not\models G$ for all $\rho \in C$, where $\rho\lfloor_{\mathcal{M}}$ denotes the projection of a path ρ of $\mathcal{M}^{\sigma}$ onto $\mathcal{M}$ and $\rho\lfloor_{\mathcal{M}} \in FPaths^{\sigma}_{\mathcal{M}}$.

Such a pair (σ, C) can be obtained as follows. Recall from Section 3.3.3 that model checking $\langle G \rangle_{\geq p_G}$ on PA $\mathcal{M}$ requires the computation of $Pr^{\min}_{\mathcal{M}}(G)$, which reduces to computing the maximum probability of reaching an accepting state in the product PA $\mathcal{M} \otimes G^{err}$. The (deterministic, finite-memory) adversary σ for $\mathcal{M}$ is obtained directly from the (deterministic, memoryless) adversary of $\mathcal{M} \otimes G^{err}$, under which the required maximum reachability probability is above $1 - p_G$. The set of paths C is then obtained from $\mathcal{M}^{\sigma}$, which is actually a DTMC since σ is deterministic (see Section 3.2.2), by using the counterexample generation techniques for upper bound probabilistic reachability properties of DTMCs as described in Section 3.4. Given a counterexample (σ, C), we can construct a *fragment* of the PA $\mathcal{M}$, denoted $\mathcal{M}^{\sigma,C}$, which is a (sub-stochastic) PA obtained from $\mathcal{M}^{\sigma}$ by removing all transitions that do not appear in any path of C. PA fragments have the following useful properties.

Proposition 4.12 ([FKP10]) *For PAs $\mathcal{M}$ and $\mathcal{M}'$, a PA fragment $\mathcal{M}^{frag}$ of $\mathcal{M}$ and a probabilistic safety property $\langle G \rangle_{\geq p_G}$, we have:*

(a) $\mathcal{M} \models \langle G \rangle_{\geq p_G} \Rightarrow \mathcal{M}^{frag} \models \langle G \rangle_{\geq p_G}$

(b) $\mathcal{M} \| \mathcal{M}' \models \langle G \rangle_{\geq p_G} \Rightarrow \mathcal{M}^{frag} \| \mathcal{M}' \models \langle G \rangle_{\geq p_G}$

(c) $\mathcal{M}^{frag} \| \mathcal{M}' \not\models \langle G \rangle_{\geq p_G} \Rightarrow \mathcal{M} \| \mathcal{M}' \not\models \langle G \rangle_{\geq p_G}$.

The proof of the above Proposition was contributed by David Parker [FKP10].

We now consider counterexamples for probabilistic assume-guarantee triples defined in Definition 4.7. Recall that a triple $\langle A \rangle_{\geq p_A} \mathcal{M} \langle G \rangle_{\geq p_G}$ does not hold if, under some adversary of $\mathcal{M}[\alpha_A]$, the assumption $\langle A \rangle_{\geq p_A}$ is satisfied but the guarantee $\langle G \rangle_{\geq p_G}$ is refuted. We need the notion of *witness*: given a PA $\mathcal{M}$ and a probabilistic safety property $\langle A \rangle_{\geq p_A}$, a witness for $\mathcal{M} \models \langle A \rangle_{\geq p_A}$ is a pair (σ, W) comprising a (deterministic, finite-memory) adversary σ for $\mathcal{M}$ with $Pr^{\sigma}_{\mathcal{M}}(A) \geq p_A$ and a set W of infinite paths in $\mathcal{M}^{\sigma}$ such that $Pr(W) \geq p_A$ and $\pi\lfloor_{\mathcal{M}} \models A$ for all $\pi \in W$. Then, a *counterexample* for the refutation of $\langle A \rangle_{\geq p_A} \mathcal{M} \langle G \rangle_{\geq p_G}$ is a tuple (σ, W, C), such that (σ, W) is a witness for $\langle A \rangle_{\geq p_A}$ in $\mathcal{M}[\alpha_A]$ and (σ, C) is a counterexample for $\langle G \rangle_{\geq p_G}$ in $\mathcal{M}[\alpha_A]$. To generate a counterexample (σ, W, C): firstly, we obtain σ as an adversary on the product PA $\mathcal{M} \otimes A^{err} \otimes G^{err}$ via multi-objective model checking (Proposition 4.8); then, we obtain paths W from $\mathcal{M}[\alpha_A]^{\sigma}$ using the counterexample generation techniques for lower bound probabilistic reachability properties of DTMCs (see Section 3.4); and finally, we obtain paths C with the same procedure for generating (σ, C) as described above.

4.3 Learning Assumptions for Rule (Asym)

In this section, we present a novel approach that was first published in [FKP10] for the automated generation of assumptions for rule (Asym) of Theorem 4.9 via algorithmic learning. This approach requires the input of two PA components $\mathcal{M}_1, \mathcal{M}_2$ and a probabilistic safety property $\langle G \rangle_{\geq p_G}$. The aim is to verify whether $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq p_G}$ is true by learning a probabilistic assumption $\langle A \rangle_{\geq p_A}$ about $\mathcal{M}_1$.

The key idea is that we reduce the task of learning probabilistic assumptions $\langle A \rangle_{\geq p_A}$ to the problem of learning *non-probabilistic* assumptions A. Recall from Theorem 4.9 that an appropriate assumption $\langle A \rangle_{\geq p_A}$ should satisfy both premises of rule (ASYM), i.e. $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq p_A}$ and $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$. If the former holds for a particular value of p_A, then it also holds for any lower value of p_A; conversely, if the latter holds for some p_A, then it must hold for any higher value. Thus, given a regular safety property A, we can determine a (lower) probability bound p_A by finding the lowest value of p_A such that $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$ holds and checking if it suffices for $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq p_A}$. Alternatively, we can find the highest possible value p_A for $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq p_A}$ and check if $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$ holds.

We adapt the L*-based techniques of learning non-probabilistic assumptions (see Section 3.6.1) here for learning probabilistic assumptions $\langle A \rangle_{\geq p_A}$. An overview of our approach is illustrated in Figure 4.4. Our approach builds on top of the L* algorithm (Section 3.5.1) shown on the left-hand side, which interacts with a *teacher* (on the right-hand side) through *membership* and *equivalence* queries, and generates a succession of conjectures A. The teacher in our case is a probabilistic model checker, as opposed to a conventional model checker. Note that our approach uses the original version of the L* algorithm by Angluin [Ang87a] rather than the improved version by Rivest and Schapire [RS93], because our usage is slightly non-standard, which will be explained later.

Recall from Section 3.6.1 that a notion of *weakest assumptions* is defined and used as the target language for L* when learning non-probabilistic assumptions. However, an analogous notion does not exist in our approach because, as explained in Section 4.1.2, the underlying compositional verification framework is *incomplete*, i.e. even if property $\langle G \rangle_{\geq p_G}$ is satisfied in the system, there may exist no appropriate assumption to verify it compositionally. An important observation about the approach in Section 3.6.1 is that it does not actually construct the weakest assumption in practice; instead, the aim is

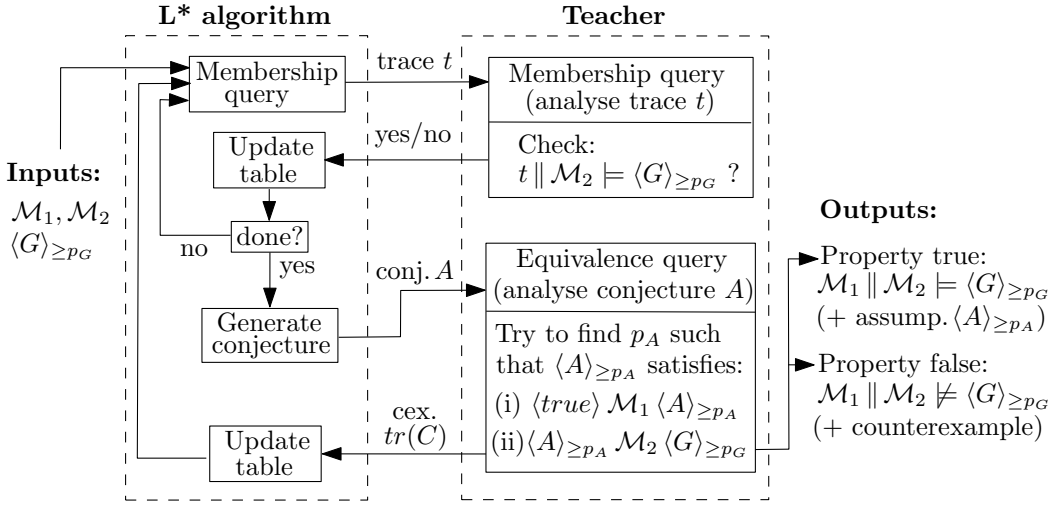


Figure 4.4: Learning probabilistic assumptions for rule (ASYM)

to either find a stronger assumption that is sufficient to verify the property or generate a counterexample that refutes it during the learning process. Therefore, we can still adopt a similar approach, where the feedback provided by the teacher is considered as *heuristics* that guides L* towards an appropriate assumption.

Answering Membership Queries

As shown in Figure 4.4, L* asks *membership queries* about whether certain traces t should be included in the assumption A being generated. Here, we use a probabilistic model checker to simulate the teacher and check if $t \| \mathcal{M}_2 \models \langle G \rangle_{\geq p_G}$ is true, where t is represented by a linear, $|t|+1$ state PA in which each transition has probability 1. This criterion can be seen as an analogue of the non-probabilistic case in Section 3.6.1.

If $t \| \mathcal{M}_2 \not\models \langle G \rangle_{\geq p_G}$, then the teacher answers "no" to L*. Any A containing such a trace t would cause the violation of triple $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$, no matter what the value of p_A is. Thus, t should definitely be excluded from A.

On the other hand, if $t \| \mathcal{M}_2 \models \langle G \rangle_{\geq p_G}$, then the teacher's answer is "yes". However, this is only a heuristic. It is possible that *multiple* traces within an assumption which

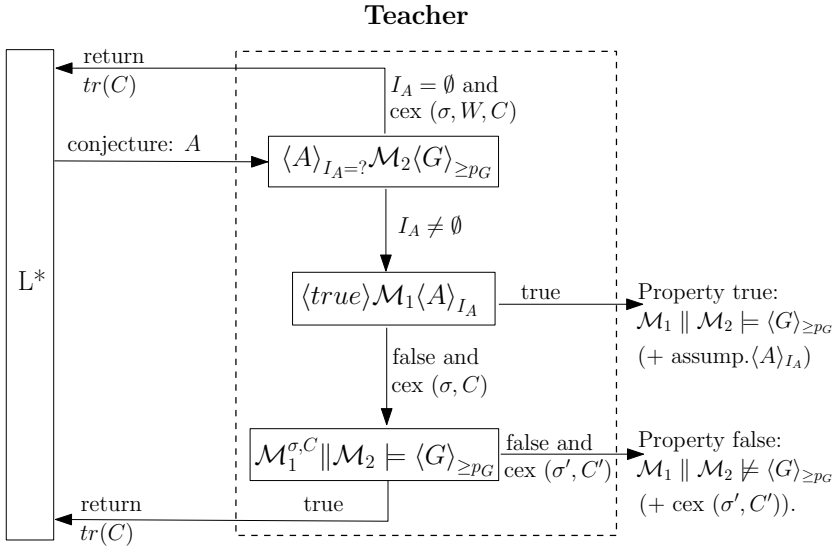


Figure 4.5: Implementation of a teacher for equivalence queries of rule (Asym)

do not violate property $\langle G \rangle_{\geq p_G}$ individually but, when combined, cause the violation. We cannot tell this from the membership query that analyses a single trace t, thus we assume that it is safe to include t in A for now. Later, when answering equivalence queries, if t is found to be contributing to the violation, then we will remove t from A. As we demonstrate in Section 4.5, this heuristic seems to work well in practice.

There is a possibility that the learning procedure may successively add and remove certain trace t from A, because, as mentioned before, our approach cannot guarantee to find a weakest assumption that may not exist. In this case, the learning procedure halts and generates a pair of lower/upper bounds on probability $Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G)$. We will discuss the generation of such bounds later.

Answering Equivalence Queries

Once a conjecture A is generated by L*, the teacher answers an *equivalence query* by trying to determine an appropriate probability value p_A such that $\langle A \rangle_{\geq p_A}$ satisfies both premises of rule (Asym), i.e. $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq p_A}$ and $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$. This process

is only briefly illustrated in Figure 4.4, but more details are revealed in Figure 4.5.

Given a conjecture A, the teacher first checks the quantitative assume-guarantee query $\langle A \rangle_{I_A=?}\, \mathcal{M}_2\, \langle G \rangle_{\geq p_G}$ by determining the widest interval $I_A \subseteq [0,1]$ for which the triple is true with assumption A. Recall from Section 4.1.1 that the resulting I_A would be either $\emptyset$, or a non-empty, closed, lower-bounded interval. The former case of $I_A{=}\emptyset$ indicates that $\mathcal{M}_2$ would not satisfy $\langle G \rangle_{\geq p_G}$ even under the probabilistic assumption $\langle A \rangle_{\geq 1}$, which means that A is too strong and we need to remove from A the trace(s) that causes the violation of $\langle G \rangle_{\geq p_G}$ on $\mathcal{M}_2$. A probabilistic counterexample (σ, W, C) illustrating the refutation of $\langle A \rangle_{\geq 1}\, \mathcal{M}_2\, \langle G \rangle_{\geq p_G}$ is generated using techniques in Section 4.2. Let $tr(C) = \{tr(\rho)\restriction_{\alpha_A} \mid \rho \in C\}$ be the set of traces taken from paths in the set C and restricted to the alphabet α_A. Suppose that $tr(C)$ only contains a single trace t, since t corresponds to a path through $\mathcal{M}_2$ that cause the violation of $\langle G \rangle_{\geq p_G}$, then any assumption A with trace t would make the triple $\langle A \rangle_{\geq p_A}\, \mathcal{M}_2\, \langle G \rangle_{\geq p_G}$ false for any value of p_A. Thus, t should be removed from A. In the case that $tr(C)$ contains *multiple* traces, each trace may not cause the violation of $\langle G \rangle_{\geq p_G}$ on its own; however, since they contribute to the probability mass of counterexample C, we remove all of them from A, as mentioned in the reasoning for membership queries. Note that, in the standard L* algorithm, only a single trace is returned as a counterexample for each equivalence query; here we adapt L* to accept all traces of $tr(C)$ at once, although it is very likely that $tr(C)$ only has a single trace by choosing C to be the *smallest counterexample* (see Section 3.4).

In the latter case that the interval I_A is non-empty, the teacher proceeds to check the first premise of (Asym), i.e. it verifies whether $\langle true \rangle\, \mathcal{M}_1\, \langle A \rangle_{I_A}$ is true. If so, then an assumption $\langle A \rangle_{I_A}$ that satisfies both premises of (Asym) has been found, proving that $\mathcal{M}_1 \| \mathcal{M}_2$ satisfies $\langle G \rangle_{\geq p_G}$, and we can terminate the learning algorithm. If, on the other hand, $\langle true \rangle\, \mathcal{M}_1\, \langle A \rangle_{I_A}$ is false, then a counterexample (σ, C) showing $\mathcal{M}_1 \not\models \langle A \rangle_{I_A}$ is generated as described in Section 4.2. The teacher constructs a PA

fragment $\mathcal{M}_1^{\sigma,C}$ and verifies $\mathcal{M}_1^{\sigma,C}\|\mathcal{M}_2$ against property $\langle G\rangle_{\geq p_G}$ to see if (σ, C) is a spurious counterexample on $\mathcal{M}_1\|\mathcal{M}_2$. If $\mathcal{M}_1^{\sigma,C}\|\mathcal{M}_2 \not\models \langle G\rangle_{\geq p_G}$ with a counterexample (σ', C'), then we can conclude that $\mathcal{M}_1\|\mathcal{M}_2 \not\models \langle G\rangle_{\geq p_G}$ based on Proposition 4.12(c) and terminate the learning process; otherwise, (σ, C) is a spurious counterexample and L* needs to use it to refine A. Consider as beforehand that $tr(C) = \{tr(\rho)\restriction_{\alpha_A} \mid \rho \in C\}$ comprises a single trace t. Since t comes from a counterexample showing $\mathcal{M}_1 \not\models \langle A\rangle_{\geq I_A}$, it is not in A. And we have $t\|\mathcal{M}_2 \models \langle G\rangle_{\geq p_G}$ because of $\mathcal{M}_1^{\sigma,C}\|\mathcal{M}_2 \models \langle G\rangle_{\geq p_G}$, thus trace t is added to the assumption A. Similarly, as for (σ, W, C), if the set $tr(C)$ contains multiple traces, we return all of them to L*.

Generating Lower and Upper Bounds

Recall that, in the approach of Section 3.6.1, it is guaranteed that we find a trace illustrating an inconsistency between the current assumption and the weakest assumption if a conjecture A fails an equivalence query. However, in our approach, this cannot be guaranteed. This is because, as discussed previously, our approach is based on an incomplete compositional verification framework and there is no equivalent notion of weakest assumption. To address this limitation, we equip our approach with the ability to generate a lower and upper bound on the minimum probability of $\mathcal{M}_1\|\mathcal{M}_2$ satisfying G. This means that, when the feedback returned by the teacher cannot help L* to update A, we can terminate the algorithm and provide this valuable quantitative information to the user based on the assumption A learnt so far.

Now, we describe how to compute these bounds. For any conjectured assumption A, we denote by $lb(A, G)$ and $ub(A, G)$ the lower and upper bounds on the (minimum) probability of satisfying the safety property G, respectively. Firstly we compute $p_A^* = Pr_{\mathcal{M}_1}^{\min}(A)$ and generate an adversary $\sigma \in Adv_{\mathcal{M}_1}$ that achieves this minimum probability. Next, we check the quantitative assume-guarantee query $\langle A\rangle_{\geq p_A^*}\, \mathcal{M}\, \langle G\rangle_{I_G=?}$ and, from the resulting interval, take $lb(A, G) = min(I_G)$. For the

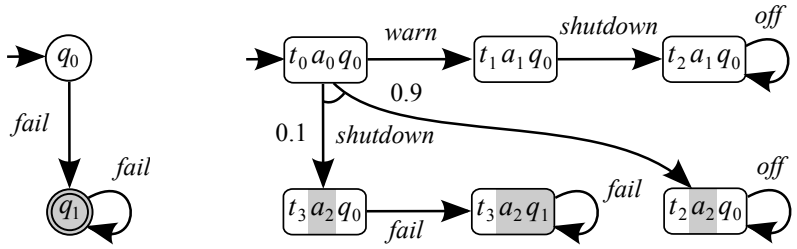


Figure 4.6: A DFA G^{err} and a product PA $\mathcal{M}_2 \otimes A_2^{err} \otimes G^{err}$ (taken from [KNPQ12])

upper bound, we compute: $ub(A, G) = Pr^{\min}_{\mathcal{M}_1^\sigma \| \mathcal{M}_2}(G)$ using the adversary σ from above.

Correctness and termination. As discussed earlier, since the underlying compositional verification framework is incomplete, we cannot guarantee that our approach will terminate with a definitive answer showing whether $\mathcal{M}_1 \| \mathcal{M}_2$ satisfies $\langle G \rangle_{\geq p_G}$. However, when the learning gets stuck, our approach can generate lower and upper bounds on the minimum probability of satisfying regular safety property G as an alternative exit for the loop. If the learning approach terminates with an output claiming that $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq p_G}$ is true or false, the correctness is guaranteed by rule (Asym) of Theorem 4.9. The correctness for the provision of lower/upper bounds $lb(A, G)$ and $ub(A, G)$ is shown below.

Proposition 4.13 ([FKP10]) $lb(A, G) \leq Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G) \leq ub(A, G)$.

Proof: For the lower bound, both $\mathcal{M}_1 \models \langle A \rangle_{\geq p_A^*}$ and $\langle A \rangle_{\geq p_A^*} M \langle G \rangle_{\geq lb(A,G)}$ hold by construction. Thus, $lb(A, G) \leq Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G)$ follows from rule (Asym).

For the upper bound, since $\mathcal{M}_1^\sigma$ is a fragment of $\mathcal{M}_1$, we have $Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G) \leq Pr^{\min}_{\mathcal{M}_1^\sigma \| \mathcal{M}_2}(G) = ub(A, G)$ based on Proposition 4.12(b). ■

Example 4.14 *This is a running example of learning probabilistic assumptions for rule (Asym). Consider the pair of PAs $\mathcal{M}_1, \mathcal{M}_2$ shown in Section 4.1, where we want to verify if $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.98}$ is true. G is a regular safety property with its corresponding DFA G^{err} shown in Figure 4.6 and means that "fail" should never occur.*

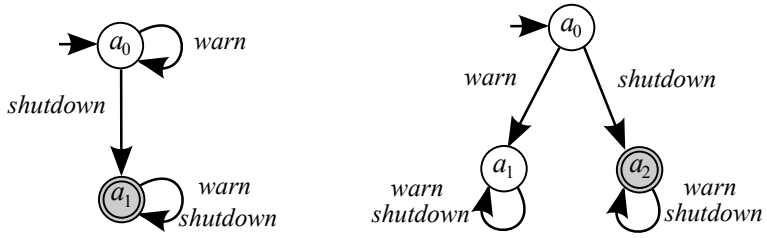


Figure 4.7: Two learnt DFAs A_1^{err} and A_2^{err} (taken from [FKP10])

$\mathcal{T}_1$	ϵ
ϵ	$-$
$shutdown$	$+$
$warn$	$-$
$shutdown, warn$	$+$
$shutdown, shutdown$	$+$

$\mathcal{T}_2$	ϵ	$shutdown$
ϵ	$-$	$+$
$shutdown$	$+$	$+$
$warn$	$-$	$-$
$warn, shutdown$	$-$	$-$
$warn, warn$	$-$	$-$
$shutdown, warn$	$+$	$+$
$shutdown, shutdown$	$+$	$+$

Figure 4.8: Observation tables corresponding to DFAs A_1^{err}, A_2^{err} in Figure 4.7

Given alphabet $\alpha_A = \{warn, shutdown\}$, the first closed and consistent observation table learnt by L is $\mathcal{T}_1$ shown in Figure 4.8. A DFA A_1^{err} (see Figure 4.7) is built based on $\mathcal{T}_1$. The teacher checks $\langle A_1 \rangle_{I_1=?}\, \mathcal{M}_2\, \langle G \rangle_{\geq 0.98}$ and the result is $I_1 \in [0.8, 1]$, which means that $\mathcal{M}_2$ satisfies $\langle G \rangle_{\geq 0.98}$ under the assumption that "shutdown" should never occur with probability at least 0.8. The teacher proceeds to check whether the minimum probability of satisfying A_1 on $\mathcal{M}_1$ is at least 0.8, i.e. $\langle true \rangle\, \mathcal{M}_1\, \langle A_1 \rangle_{\geq 0.8}$. But this check fails with a counterexample (σ, C) indicating that "shutdown" eventually occurs in $\mathcal{M}_1$ with probability 1. Figure 4.9 shows PA $\mathcal{M}_1^{\sigma}$ and PA fragment $\mathcal{M}_1^{(\sigma,C)}$. The teacher verifies $\mathcal{M}_1^{(\sigma,C)} \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.98}$ for counterexample analysis, and the result yields true. Thus, (σ, C) is a spurious counterexample and the trace $\langle warn, shutdown \rangle$ from $tr(C)$ is returned to L* to refine the conjecture.*

L updates the observation table till it obtains a second closed and consistent table $\mathcal{T}_2$ (Figure 4.8). A new DFA A_2^{err} as shown in Figure 4.7 is constructed based on $\mathcal{T}_2$. Model checking $\langle A_2 \rangle_{I_2=?}\, \mathcal{M}_2\, \langle G \rangle_{\geq 0.98}$ is performed on the product PA $\mathcal{M}_2 \otimes A_2^{err} \otimes G^{err}$ shown*

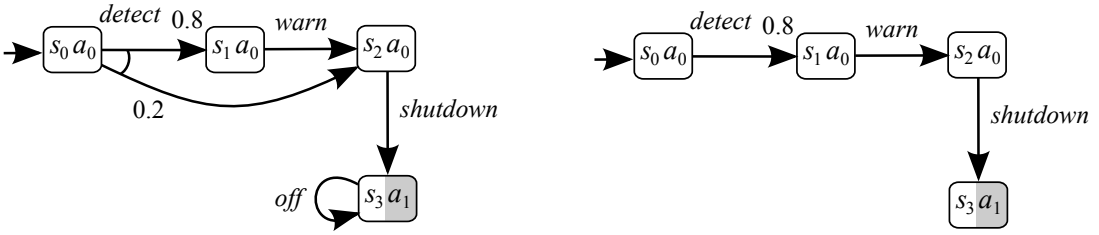


Figure 4.9: PA $\mathcal{M}_1^{\sigma}$ and PA fragment $\mathcal{M}_1^{(\sigma,C)}$ (taken from [FKP10])

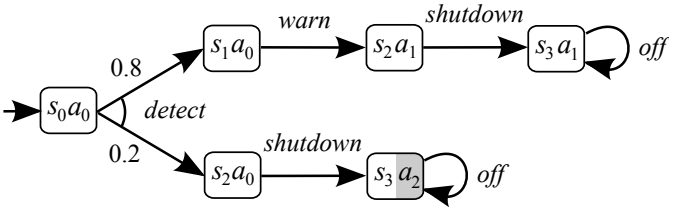


Figure 4.10: Product PA $\mathcal{M}_1 \otimes A_2^{err}$ (taken from [KNPQ12])

in Figure 4.6, yielding a result of $I_2 \in [0.8, 1]$. *Checking* $\langle true \rangle\, \mathcal{M}_1\, \langle A_2 \rangle_{\geq 0.8}$ *reduces to computing maximum probability of reaching error states on product PA* $\mathcal{M}_1 \otimes A_2^{err}$. *As shown in Figure 4.10, the error state* s_3a_2 *can only be reached with probability 0.2, and thus* $\langle true \rangle\, \mathcal{M}_1\, \langle A_2 \rangle_{\geq 0.8}$ *holds.*

Therefore, we conclude that $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.98}$ *is true, which can be verified compositionally with rule (*Asym*) by using the probabilistic assumption* $\langle A_2 \rangle_{\geq 0.8}$.

Example 4.15 *Consider two PA components* $\mathcal{M}_1, \mathcal{M}_2$ *(Figure 4.11) of a simple communication channel that satisfies the safety property "input and output should be sent in order" with probability at least 0.82, i.e.* $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.82}$ *with the corresponding DFA* G^{err} *shown in Figure 4.12. In the following, we show that we cannot find an appropriate assumption to verify this property compositionally with rule (*Asym*), but we can compute a pair of lower/upper bounds that restrict the probability* $Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G)$.

We initialise the learning approach in Figure 4.4 and set the assumption alphabet as $\alpha_A = \{output, send, ack\}$. *L* fills the observation table with the teacher's answers to*

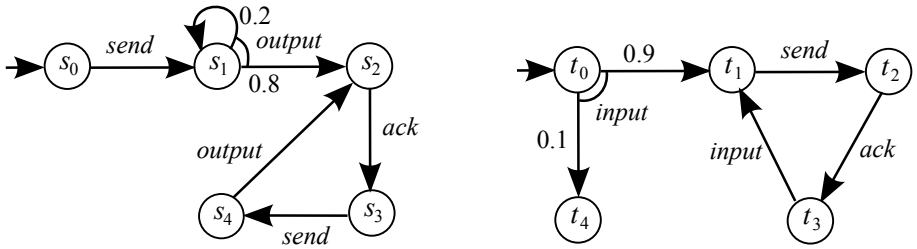


Figure 4.11: Two PAs $\mathcal{M}_1, \mathcal{M}_2$ for Example 4.15

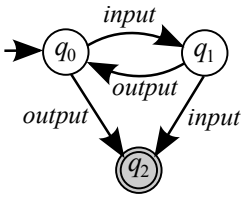


Figure 4.12: DFA G^{err} for Example 4.15

membership queries, which are obtained by checking $t \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.82}$ *for each trace* t*. The first closed and consistent observation table is* $\mathcal{T}_1$ *in Figure 4.14, which corresponds to the DFA* A_1^{err} *shown in Figure 4.13. For simplicity, we denote actions by their initial letters in the observation table. To answer the equivalence query for* A_1*, the teacher checks* $\langle A_1 \rangle_{I_1=?} \mathcal{M}_2 \langle G \rangle_{\geq 0.82}$*, which yields* $I_1 = \emptyset$*. This is because there is a trace* $\langle send, ack \rangle$ *in* A_1 *that will cause the violation of* $\langle G \rangle_{\geq 0.82}$ *on* $\mathcal{M}_2$*, e.g. following the path* $t_0q_0 \xrightarrow{input,0.9} t_1q_1 \xrightarrow{send} t_2q_1 \xrightarrow{ack} t_3q_1 \xrightarrow{input} t_1q_2$ *in* $\mathcal{M}_2 \otimes G^{err}$*, the error state* t_1q_2 *would be reached with probability* 0.9*. Thus, the teacher returns trace* $\langle send, ack \rangle$ *to* L^* *to refine learning.*

After the refinement, L^* *obtains a second closed and consistent observation table* $\mathcal{T}_2$ *as shown in Figure 4.14, corresponding to the DFA* A_2^{err} *in Figure 4.13. The equivalence query yields an empty interval* I_2 *for* $\langle A_2 \rangle_{I_2=?} \mathcal{M}_2 \langle G \rangle_{\geq 0.82}$ *and a counterexample trace* $\langle send, output, ack, output \rangle$*. This counterexample corresponds to the path* $t_0q_0 \xrightarrow{input,0.9} t_1q_1 \xrightarrow{send} t_2q_1 \xrightarrow{output} t_2q_0 \xrightarrow{ack} t_3q_0 \xrightarrow{output} t_3q_2$ *in* $\mathcal{M}_2 \otimes G^{err}$*, where the error state* t_3q_2 *is reachable with probability* 0.9*, and thus the property* $\langle G \rangle_{\geq 0.82}$ *is violated.*

After refinement with the counterexample trace $\langle send, output, ack, output \rangle$*,* L^*

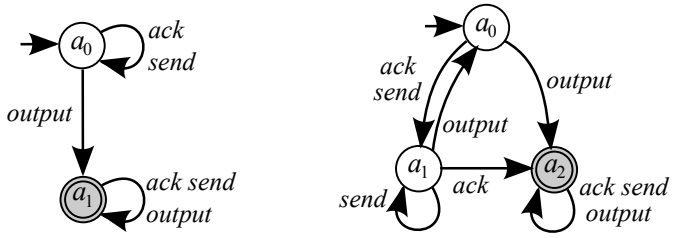


Figure 4.13: Learnt DFAs A_1^{err}, A_2^{err} for Example 4.15

$\mathcal{T}_1$	ϵ
ϵ	$-$
o	$+$
s	$-$
a	$-$
oo	$+$
os	$+$
oa	$+$

$\mathcal{T}_2$	ϵ	o
ϵ	$-$	$+$
o	$+$	$+$
s	$-$	$-$
sa	$+$	$+$
a	$-$	$-$
oo	$+$	$+$
os	$+$	$+$
oa	$+$	$+$
so	$-$	$+$
ss	$-$	$-$
sao	$+$	$+$
sas	$+$	$+$
saa	$+$	$+$

Figure 4.14: Observation tables corresponding to A_1^{err} and A_2^{err} in Figure 4.13

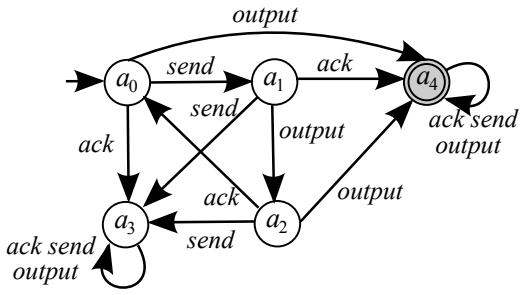


Figure 4.15: Learnt DFA A^{err} for Examples 4.15 and 4.17

learns a new conjecture A^{err} as shown in Figure 4.15. The quantitative query $\langle A \rangle_{I=?}\, \mathcal{M}_2\, \langle G \rangle_{\geq 0.82}$ yields the non-empty interval $I = [0.82, 1]$, and the teacher continues to check the triple $\langle true \rangle\, \mathcal{M}_1\, \langle A \rangle_{\geq 0.82}$, which is actually false due to $Pr^{\min}_{\mathcal{M}_1}(A) = 0.8$. The probabilistic counterexample illustrating this violation consists of the set of traces

$\langle send, output^n, output\rangle$ *with* $n \geq 1$. L^* *cannot refine* A *with these traces because they do not illustrate any difference between the target language and* $\mathcal{L}(A)$*, i.e. for any of these traces* t*, the membership query yields* $t \| \mathcal{M}_2 \not\models \langle G \rangle_{\geq 0.82}$*, while* A *also rejects it. The learning gets stuck with conjecture* A*. We terminate the learning and compute the lower bound of the minimum probability of satisfying safety property* G *on* $\mathcal{M}_1 \| \mathcal{M}_2$ *by checking* $\langle A \rangle_{\geq 0.8} \mathcal{M}_2 \langle G \rangle_{I_G=?}$*, which yields* 0.8*. The upper bound is obtained by computing* $Pr^{\min}_{\mathcal{M}_1^\sigma \| \mathcal{M}_2}(G)$*, where* σ *is the adversary generated during the previous check for* $\langle true \rangle \mathcal{M}_1 \langle A \rangle_{\geq 0.82}$*, and the result is* 0.82*.*

In conclusion, although we cannot find an appropriate assumption to verify that $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.82}$ *is true, we are sure that the probability* $Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G)$ *is in* $[0.8, 0.82]$.

4.4 Extensions

In the previous section, we considered learning assumptions for rule (Asym) using the L* algorithm. Here, we explore three extensions to this approach, with the hope to learn more succinct representation of assumptions and to handle more complex systems. First, we investigate learning assumptions using an alternative learning algorithm, the NL* algorithm, in Section 4.4.1. Then, we generalise the assumption learning approach for the asymmetric n-component rule (Asym-N) and the circular rule (Circ) in Section 4.4.2 and Section 4.4.3, respectively.

4.4.1 Learning Assumptions using the NL* Algorithm

Recall that the L* algorithm (Section 3.5.1) learns a minimal DFA that accepts a regular language, whereas the NL* algorithm (Section 3.5.1) learns a minimal *residual finite-state automaton* (RFSA), which is a subclass of NFAs and can be exponentially more succinct than the corresponding minimal DFA for the same regular language. Thus,

we might be able to learn more succinct assumptions using NL*. Since NL* works similarly to L*, i.e. interacting with a teacher via asking the same kind of membership and equivalence queries, we can substitute it into the assumption learning approach shown in Figure 4.4 straightforwardly.

Nevertheless, we need to determinise the learnt RFSAs when answering equivalence queries in Figure 4.4, i.e. checking whether $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq p_A}$ and $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$ are true. Recall from Section 4.1.2 that, when the assumption is represented by a DFA, verifying $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq p_A}$ reduces to computing maximum reachability probabilities on the product PA $\mathcal{M}_1 \otimes A^{err}$ and checking $\langle A \rangle_{\geq p_A} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$ reduces to the multi-objective model checking on the product PA $\mathcal{M}_2[\alpha_A] \otimes A^{err} \otimes G^{err}$, where the operator $\otimes$ for composing a PA and a DFA is as defined in Section 3.3.3. We cannot adapt the operator for composing a PA and a RFSA, because the nondeterminism of RFSA will add extra nondeterministic choices in the product PA. Conventionally [1], the automata specifications for probabilistic model checking need to be determinised [dA97]. So, we convert the RFSAs learnt by the NL* algorithm into DFAs using the subset construction method as mentioned in Section 3.1.

Although we will not be able to obtain more succinct assumptions because of the determinising step, the hope is that NL* may lead to a faster learning procedure than L*. As we demonstrate through experiments in Section 4.5, NL* does perform better than L* in some large case studies due to the smaller size of learnt RFSAs and fewer equivalence queries.

4.4.2 Generalisation to Rule (Asym-N)

Recall from Section 4.1.2 that we can verify a probabilistic safety property $\langle G \rangle_{\geq p_G}$ on an n-component PA system $\mathcal{M}_1 \parallel \ldots \parallel \mathcal{M}_n$ compositionally with the assume-guarantee rule (Asym-N), which requires multiple assumptions $\langle A_1 \rangle_{\geq p_1}, \ldots, \langle A_{n-1} \rangle_{\geq p_{n-1}}$. The

[1] More recently, Henzinger and Piterman [HP06] show that a nondeterministic automaton is good for solving games, and thus it may not always necessary to determinise the specification automaton.

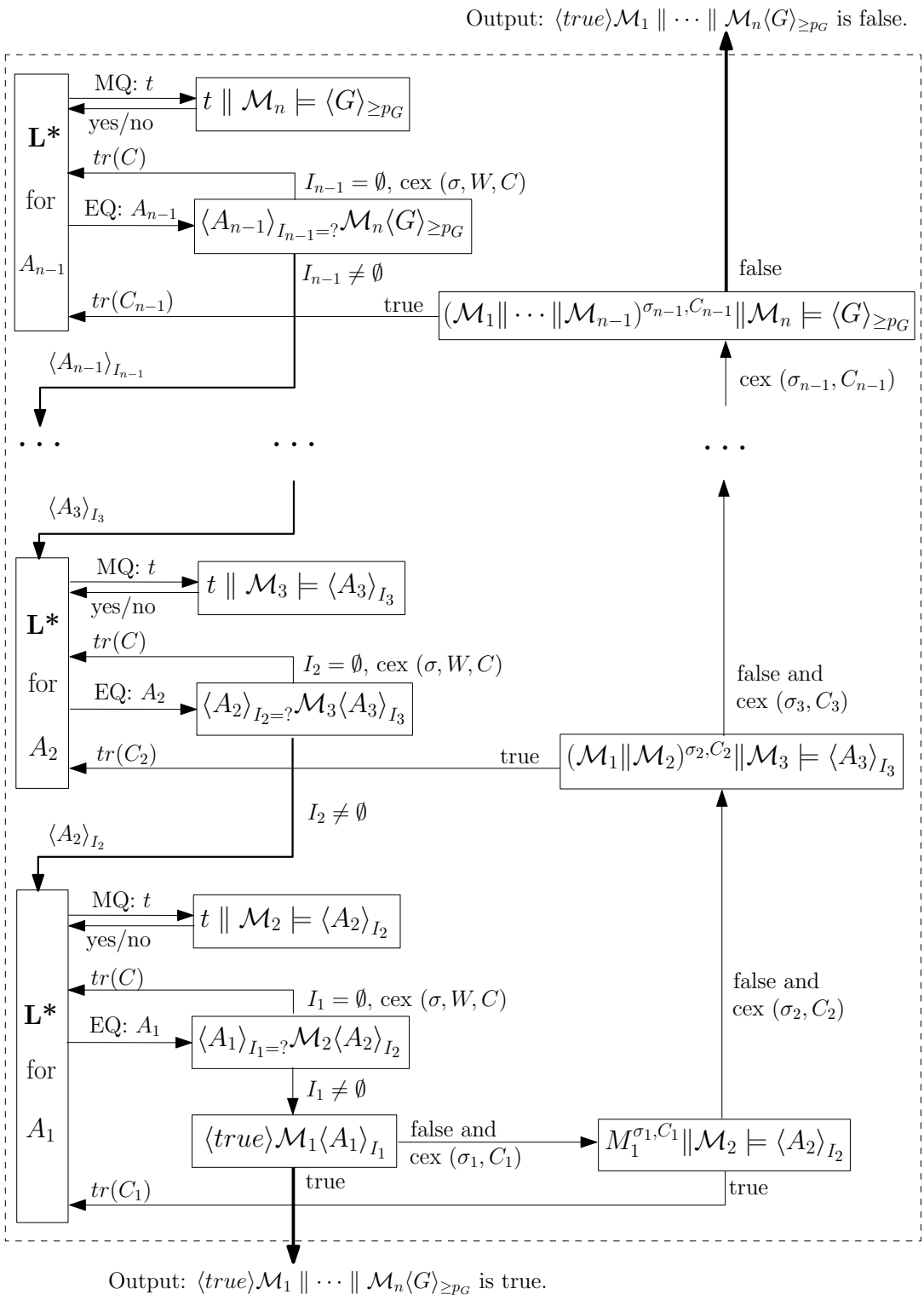


Figure 4.16: Learning probabilistic assumptions for rule (Asym-N)

idea of learning assumptions for rule (Asym-N) is inspired by [PGB+08]. To check if system $\mathcal{M}_1 \| \ldots \| \mathcal{M}_n$ satisfies $\langle G \rangle_{\geq p_G}$, we decompose it into $\mathcal{M}_1 \| \ldots \| \mathcal{M}_{n-1}$ and $\mathcal{M}_n$, and then apply the assumption learning approach for rule (Asym) recursively. At each recursive invocation for $\mathcal{M}_1 \| \ldots \| \mathcal{M}_{j-1}$ and $\mathcal{M}_j$, we find assumption $\langle A_j \rangle_{\geq p_j}$ such that the following premises are both true:

- $\langle true \rangle \, \mathcal{M}_1 \| \ldots \| \mathcal{M}_{j-1} \, \langle A_j \rangle_{\geq p_j}$ and
- $\langle A_j \rangle_{\geq p_j} \, \mathcal{M}_j \, \langle A_{j+1} \rangle_{\geq p_{j+1}}$.

Here $\langle A_{j+1} \rangle_{\geq p_{j+1}}$ is the assumption for $\mathcal{M}_{j+1}$ and plays the role of the property for the current recursive call.

As shown in Figure 4.16, the learning approach for rule (Asym-N) employs multiple L* instances, one for each assumption. In the learning instance for A_j, the membership query for a trace t is resolved by checking $t \| \mathcal{M}_{j+1} \models \langle A_{j+1} \rangle_{I_{j+1}}$, where $\langle A_{j+1} \rangle_{I_{j+1}}$ is learnt through the previous recursive L* instance and $I_{j+1} \subseteq [0,1]$ is a non-empty, closed, lower-bounded probability interval. When $j = n-1$, the property $\langle A_{j+1} \rangle_{I_{j+1}}$ is replaced by $\langle G \rangle_{\geq p_G}$.

The equivalence query for A_j is resolved by first checking the quantitative multi-objective query $\langle A_j \rangle_{I_j = ?} \, \mathcal{M}_{j+1} \, \langle A_{j+1} \rangle_{I_{j+1}}$. If $I_j = \emptyset$, then L* must refine the conjecture A_j. More specifically, as described in Section 4.3, the teacher would produce a counterexample (σ, W, C) from the multi-objective query and return a set of traces $tr(C) = \{tr(\rho) \restriction_{\alpha_A} \mid \rho \in C\}$ to L*. If $I_j \neq \emptyset$, then the next recursive call for learning assumption $\langle A_{j-1} \rangle_{I_{j-1}}$ would be invoked, which uses $\langle A_j \rangle_{I_j}$ as the property.

Eventually, when a non-empty interval I_1 is found for $\langle A_1 \rangle_{I_1 = ?} \, \mathcal{M}_2 \, \langle A_2 \rangle_{I_2}$, the teacher checks whether $\langle true \rangle \, \mathcal{M}_1 \, \langle A_1 \rangle_{I_1}$ holds. If so, then we have found a series of assumptions $\langle A_1 \rangle_{I_1}, \ldots, \langle A_{n-1} \rangle_{I_{n-1}}$ that make all premises of rule (Asym-N) satisfied. Thus, the system $\mathcal{M}_1 \| \ldots \| \mathcal{M}_n$ satisfies $\langle G \rangle_{\geq p_G}$. However, if $\mathcal{M}_1 \not\models \langle A_1 \rangle_{I_1}$, then the teacher would provide a counterexample (σ_1, C_1) illustrating the violation. We

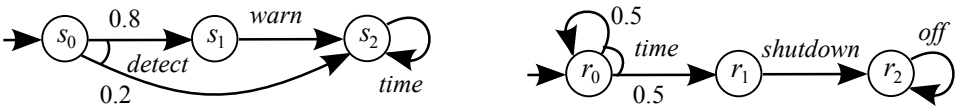


Figure 4.17: PAs $\mathcal{M}_1$ and $\mathcal{M}_2$ for Example 4.16

need to perform counterexample analysis to check if (σ_1, C_1) is a real counterexample for $\mathcal{M}_1 \| \mathcal{M}_2 \not\models \langle A_2 \rangle_{I_2}$. To do so, we construct a PA fragment $\mathcal{M}_1^{\sigma_1, C_1}$ similarly as in Section 4.3 and check if $\mathcal{M}_1^{\sigma_1, C_1} \| \mathcal{M}_2 \models \langle A_2 \rangle_{I_2}$ holds. If the model checking result is true, then (σ_1, C_1) is a spurious counterexample and a set of traces $tr(C_1)$ extracted from paths in the set C_1 are returned to L* to refine the learning of A_1. Otherwise, the teacher produces a new counterexample (σ_2, C_2) showing that $\mathcal{M}_1 \| \mathcal{M}_2 \not\models \langle A_2 \rangle_{I_2}$ and performs counterexample analysis. The counterexample generation and analysis procedure continues recursively until a spurious counterexample is found for the L* instance of some assumption, or a counterexample (σ_{n-1}, C_{n-1}) is generated such that $(\mathcal{M}_1 \| \cdots \| \mathcal{M}_{n-1})^{\sigma_{n-1}, C_{n-1}} \| \mathcal{M}_n \not\models \langle G \rangle_{\geq p_G}$. In the latter case, we can conclude that the system $\mathcal{M}_1 \| \ldots \| \mathcal{M}_n$ does not satisfy $\langle G \rangle_{\geq p_G}$.

Recall from Section 4.3 that, when learning assumptions for rule (Asym), if the conjecture A cannot be updated any more, we can generate lower/upper probability bounds of $Pr^{\min}_{\mathcal{M}_1 \| \mathcal{M}_2}(G)$ based on the computation of $p^*_A = Pr^{\min}_{\mathcal{M}_1}(A)$. However, in the case of learning for rule (Asym-N), we do not have a feasible plan for computing such bounds. It is not efficient to compute $p^*_A = Pr^{\min}_{\mathcal{M}_1 \| \ldots \| \mathcal{M}_{j-1}}(A)$ directly by composing $\mathcal{M}_1 \| \ldots \| \mathcal{M}_{j-1}$. A preferred approach to compute this value would be in a recursive manner, but this is not straightforward. For example, if we get stuck in the learning of A_{n-1} (due to $I_{n-1} = \emptyset$), then we would want to compute the bounds and terminate the learning; however, at this point, the learning of other assumptions has not been invoked, so we cannot compute the value of $p_{A^*_{n-1}}$ recursively.

Example 4.16 *Consider verifying the system $\mathcal{M}_1 \| \mathcal{M}_2 \| \mathcal{M}_3$ against a probabilistic safety property $\langle G \rangle_{\geq 0.98}$, where PAs $\mathcal{M}_1, \mathcal{M}_2$ are shown in Figure 4.17, and PA*

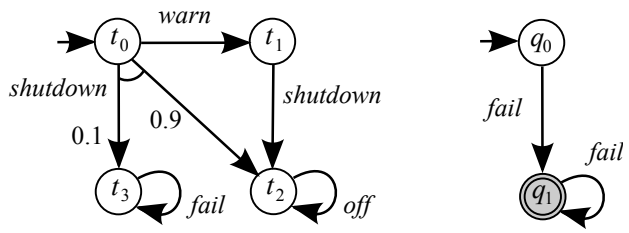


Figure 4.18: PA $\mathcal{M}_3$ and a DFA G^{err} for Example 4.16

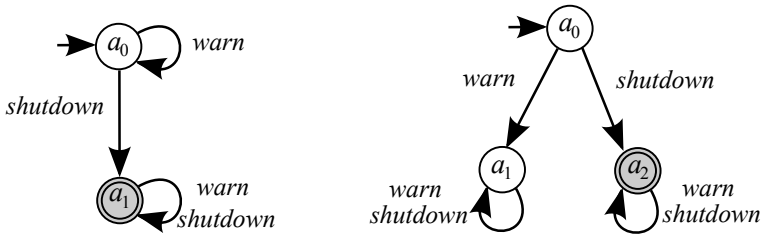


Figure 4.19: DFAs $A^{err}_{2'}, A^{err}_{2}$ learnt by the L* instance for A_2 in Example 4.16

$\mathcal{M}_3$ and the DFA G^{err} for the safety property are shown in Figure 4.18. We show the whole procedure of applying the approach in Figure 4.16 to perform the compositional verification and learn assumptions with alphabet $\alpha_{A_1} = \{warn, time\}$ and $\alpha_{A_2} = \{warn, shutdown\}$.

First of all, one instance of L is initialised for learning A_2. The teacher starts with answering membership queries by checking $t \| \mathcal{M}_3 \models \langle G \rangle_{\geq 0.98}$ for traces t. The first conjecture produced by L* is the DFA $A^{err}_{2'}$ shown in Figure 4.19. An equivalence query is asked for it and the teacher checks the quantitative multi-objective query $\langle A_{2'} \rangle_{I_2 = ?} \mathcal{M}_3 \langle G \rangle_{\geq 0.98}$, which yields a non-empty interval and results in a probabilistic assumption $\langle A_{2'} \rangle_{\geq 0.8}$.*

Now, another L instance is invoked for learning A_1. The teacher answers membership queries by checking $t \| \mathcal{M}_2 \models \langle A_{2'} \rangle_{\geq 0.8}$ and produces a conjectured DFA $A^{err}_{1'}$ as shown in Figure 4.20. The equivalence query for $A^{err}_{1'}$ yields the non-empty interval $I_1 = [0.8, 1]$ for $\langle A_{1'} \rangle_{I_1 = ?} \mathcal{M}_2 \langle A_{2'} \rangle_{\geq 0.8}$. But the triple $\langle true \rangle \mathcal{M}_1 \langle A_{1'} \rangle_{\geq 0.8}$ is false, because there is a counterexample path $s_0 \xrightarrow{detect, 0.8} s_1 \xrightarrow{warn} s_2 \xrightarrow{time} s_2$ through $\mathcal{M}_1$ such*

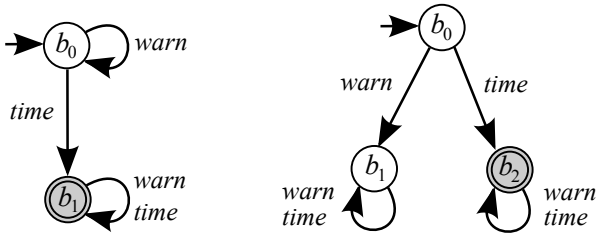


Figure 4.20: DFAs $A^{err}_{1'}, A^{err}_{1}$ learnt by the L* instance for A_1 in Example 4.16

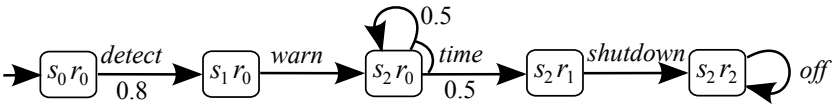


Figure 4.21: PA fragment $\mathcal{M}_1^{\sigma_1,C_1} \| \mathcal{M}_2$ for Example 4.16

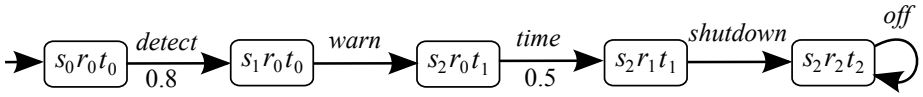


Figure 4.22: PA fragment $(\mathcal{M}_1 \| \mathcal{M}_2)^{\sigma_2,C_2} \| \mathcal{M}_3$ for Example 4.16

that the "time" action occurs with probability 0.8. *A fragment PA* $\mathcal{M}_1^{\sigma_1,C_1}$ *containing this path is constructed and the teacher performs counterexample analysis by verifying* $\mathcal{M}_1^{\sigma_1,C_1} \| \mathcal{M}_2 \models \langle A_{2'} \rangle_{\geq 0.8}$. *Since the "shutdown" action occurs in* $\mathcal{M}_1^{\sigma_1,C_1} \| \mathcal{M}_2$ *with probability at least 0.4 (see* $s_0 r_0 \xrightarrow{detect,0.8} s_1 r_0 \xrightarrow{warn} s_2 r_0 \xrightarrow{time,0.5} s_2 r_1 \xrightarrow{shutdown} s_2 r_2$ *in Figure 4.21),* $\langle A_{2'} \rangle_{\geq 0.8}$ *is violated in* $\mathcal{M}_1^{\sigma_1,C_1} \| \mathcal{M}_2$ *and a new counterexample* (σ_2, C_2) *illustrating the violation is generated. As shown in Figure 4.22,* $(\mathcal{M}_1 \| \mathcal{M}_2)^{\sigma_2,C_2} \| \mathcal{M}_3$ *satisfies probabilistic safety property* $\langle G \rangle_{\geq 0.98}$ *because the "fail" action never occurs. Thus, the teacher returns to the L* instance for* A_2 *a set of counterexample traces* $tr(C_2)$, *which actually only contains a single trace* $\langle warn, shutdown \rangle$.

The second conjecture learnt by the L instance for* A_2 *is the DFA* A_2^{err} *shown in Figure 4.19, and the query* $\langle A_2 \rangle_{I_2=?} \mathcal{M}_3 \langle G \rangle_{\geq 0.98}$ *yields a valid probabilistic assumption* $\langle A_2 \rangle_{\geq 0.8}$. *A new L* instance is invoked for learning* A_1, *and the teacher answers membership queries by checking* $t \| \mathcal{M}_2 \models \langle A_2 \rangle_{\geq 0.8}$. *The first conjecture produced by this new L* instance is the DFA* $A^{err}_{1'}$ *in Figure 4.20, which is the same as the first*

DFA learnt by the previous L instance for A_1. The teacher answers an equivalence query for it by first checking $\langle A_{1'}\rangle_{I_1=?}\, \mathcal{M}_2\, \langle A_2\rangle_{\geq 0.8}$, which yields $\langle A_{1'}\rangle_{\geq 0.8}$, and then checking $\langle true\rangle\, \mathcal{M}_1\, \langle A_{1'}\rangle_{\geq 0.8}$, which is false with counterexample (σ_1, C_1) as described before. We can see from Figure 4.21 that $\mathcal{M}_1^{\sigma_1,C_1} \| \mathcal{M}_2$ satisfies $\langle A_2\rangle_{\geq 0.8}$, because the probability that "shutdown" occurs before "warn" is 0. Thus, (σ_1, C_1) is a spurious counterexample and a trace $\langle warn, time\rangle$ is returned to the L* instance for A_1. After the refinement, a new conjectured DFA A_1^{err} as shown in Figure 4.20 is generated. The teacher checks that $\langle A_1\rangle_{\geq 0.8}\, \mathcal{M}_2\, \langle A_2\rangle_{\geq 0.8}$ and $\langle true\rangle\, \mathcal{M}_1\, \langle A_1\rangle_{\geq 0.8}$ both hold.*

*Therefore, we conclude that $\mathcal{M}_1 \| \mathcal{M}_2 \| \mathcal{M}_3 \models \langle G\rangle_{\geq 0.98}$ is true, based on the compositional verification with rule (*Asym-N*) and two assumptions $\langle A_1\rangle_{\geq 0.8}, \langle A_2\rangle_{\geq 0.8}$ (the corresponding DFAs A_1^{err}, A_2^{err} are shown in Figure 4.20 and Figure 4.19, respectively).*

4.4.3 Generalisation to Rule (Circ)

Recall from Section 4.1.2 that the assume-guarantee rule (Circ) verifies two-component systems in a circular fashion and overcomes the limitation of rule (Asym), which may not be applicable for systems that exhibit circular dependency between components. Since rule (Circ) has a similar format to rule (Asym-N), it turns out that learning assumptions for rule (Circ) can be treated as a special case of learning for (Asym-N) with $n = 3$ (see Figure 4.16).

In Example 4.15, we show that an appropriate assumption cannot be found for rule rule (Asym) to verify that $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G\rangle_{\geq 0.82}$ is true. This problem can be solved via learning assumptions for rule (Circ), which we demonstrate in the following example.

Example 4.17 *Consider PAs $\mathcal{M}_1, \mathcal{M}_2$ shown in Figure 4.11 and the DFA G^{err} shown in Figure 4.12; we want to verify $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G\rangle_{\geq 0.82}$ compositionally with rule (*Circ*). More specifically, we adapt the assumption learning approach illustrated in Figure 4.16 to learn two probabilistic assumptions $\langle A\rangle_{\geq p}, \langle A'\rangle_{\geq p'}$ such that the following are all true:*

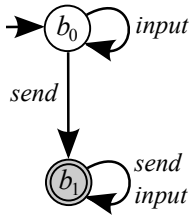


Figure 4.23: The DFA for assumption A' in Example 4.17

(i) $\langle true \rangle \, \mathcal{M}_2 \, \langle A' \rangle_{\geq p'}$,

(ii) $\langle A' \rangle_{\geq p'} \, \mathcal{M}_1 \, \langle A \rangle_{\geq p}$,

(iii) $\langle A \rangle_{\geq p} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$.

We first initialise one instance of L^* to learn A, and set the alphabet $\alpha_A = \{output, send, ack\}$. The membership queries for traces t are answered by checking whether $t \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.82}$ is true. L^* proposes conjectures based on closed and consistent observation tables and asks quantitative queries $\langle A \rangle_{I=?} \, \mathcal{M}_2 \, \langle G \rangle_{\geq p_G}$ until a conjecture is learnt with a non-empty interval I. The procedure is exactly the same as in Example 4.15, and eventually a probabilistic assumption $\langle A \rangle_{\geq 0.82}$ is learnt with the corresponding DFA A^{err} shown in Figure 4.15.

Now, instead of checking the satisfaction of $\langle true \rangle \, \mathcal{M}_1 \, \langle A \rangle_{\geq 0.82}$ as in Figure 4.15, we involve another L^* instance for learning A' with alphabet $\alpha_{A'} = \{input, send\}$. The membership queries are answered by checking $t \| \mathcal{M}_1 \models \langle A \rangle_{\geq 0.82}$. And L^* learns a DFA for assumption A' as shown in Figure 4.23. The teacher answers equivalence query for A' by first checking $\langle A' \rangle_{I'=?} \, \mathcal{M}_1 \, \langle A \rangle_{\geq 0.82}$, which yields the non-empty interval $I' = [0.1, 1]$, and then verifying $\langle true \rangle \, \mathcal{M}_2 \, \langle A' \rangle_{\geq 0.1}$, which turns out to be also true.

*Thus, we can verify that $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.82}$ is true, by using rule (*CIRC*) and two assumptions $\langle A \rangle_{\geq 0.82}, \langle A' \rangle_{\geq 0.1}$ whose corresponding DFAs are shown in Figure 4.15 and Figure 4.23, respectively.*

4.5 Implementation and Case studies

We have built a prototype tool that implements the fully-automated learning-based compositional verification approach proposed in this chapter, targeting three different assume-guarantee rules (ASYM), (ASYM-N) and (CIRC). Our prototype uses the libalf learning library [BKK+10] for the implementation of the L* and NL* learning algorithms. It also employs the probabilistic model checker PRISM [KNP11] to act as the teacher to answer membership and equivalence queries for the learning process. To generate probabilistic counterexamples for model checking, we first build adversaries during model checking with PRISM (sparse engine), and then use CARMEL (http://www.isi.edu/licensed-sw/carmel/), which implements Eppstein's algorithm [Epp98] for finding k-shortest paths. We have applied our prototype tool to a set of benchmark case studies and run experiments on a 2.80GHz PC with 32GB RAM running 64-bit Fedora. We describe these case studies briefly below. See Appendix C for detailed models and properties in the PRISM modelling language.

Client-Server Model. This case study is a variant of the client-server model from [PGB+08]. It models a server and N clients. The server can *grant* or *deny* a client's *request* for using a common resource; once a client receives permission to access the resource, it can either *use* it or *cancel* the reservation. Failures might occur with certain probability in one or multiple clients, causing the violation of the mutual exclusion property (i.e. conflict in using resources between clients). In the experiments for rule (ASYM), we model the server as a PA component $\mathcal{M}_1$ and the interleaved composition of N clients as the other PA component $\mathcal{M}_2$; for rule (ASYM-N), we model the server as a PA $\mathcal{M}_1$ and each client as an individual PA $\mathcal{M}_i$ for $2 \leq i \leq N+1$. In all cases, we verify whether the composed system satisfies the mutual exclusion property with certain minimum probability.

Randomised Consensus Algorithm. This case study is based on Aspnes & Herlihy's randomised consensus algorithm [AH90] and modelled in [KNS01]. It models N distributed processes trying to reach consensus and employs, at each round, a shared coin protocol parameterised by K. We apply the assumption learning approach for rule (Asym) to this case study and verify the minimum probability of consensus being reached within R rounds, where the system is decomposed into two PA components: $\mathcal{M}_1$ for the coin protocol and $\mathcal{M}_2$ for the interleaving of N processes.

Sensor Network. This case study is first presented in [FKP10]. It models a network of N sensors, which are sending data to a processor through a channel. The network might crash if some data packets are lost. We decompose the system into two PA components for using rule (Asym): $\mathcal{M}_1$, modelling the asynchronous composition of sensors and the channel, and $\mathcal{M}_2$, modelling the processor. We check the minimum probability that the network never crashes.

Mars Exploration Rovers (mer). This case study is modified from a module of the flight software for JPL's Mars Exploration Rovers, which was studied as a non-probabilistic model in [PG06]. The module is a resource arbiter that grants and rescinds access rights of R shared resources to N user threads as required. Each user thread models a different application on the rover, e.g. imaging, driving and communicating with earth. We adapt the module to a probabilistic model by adding the possibility of faulty behaviour, e.g. due to the message lost when the arbiter grants an access for a user thread. When experimenting with rule (Asym), we decompose the system into two PA components: $\mathcal{M}_1$ for the arbiter and $\mathcal{M}_2$ for the join of N user threads; when learning for rule (Asym), we model the arbiter and each user thread as an individual PA component. We verify the minimum probability of the mutual exclusion property that "the arbiter would never grant permissions for communication and driving simultaneously" holds.

Case study [parameters]		Component sizes		Compositional (L*)				Compositional (NL*)				Non-comp.
		$\|\mathcal{M}_2 \otimes G^{err}\|$	$\|\mathcal{M}_1\|$	$\|A\|$	MQ	EQ	Time	$\|A\|$	MQ	EQ	Time	Time
client-server (1 failure) [N]	3	81	16	5	99	3	**5.4**	6	223	4	6.9	0.02
	5	613	36	7	465	5	**19.8**	8	884	5	24.3	0.04
	7	4,733	64	9	1,295	7	483.1	10	1,975	5	**402.7**	0.08
client-server (N failures) [N]	3	229	16	5	99	3	**6.3**	6	192	3	7.2	0.04
	4	1,121	25	6	236	4	**25.2**	7	507	4	31.9	0.12
	5	5,397	36	7	465	5	**190.8**	8	957	5	200.4	0.28
consensus [N R K]	2 3 20	391	3,217	6	149	5	25.6	7	161	3	**14.7**	104.2
	2 4 4	573	431,649	12	2,117	8	415.1	12	1,372	5	**101.2**	2.59
	3 3 20	8,843	38,193	11	471	6	438.6	15	1,231	5	**409.7**	>24h
sensor network [N]	1	42	72	3	17	2	**3.2**	4	31	2	3.6	0.03
	2	42	1,184	3	17	2	**3.7**	4	31	2	4.0	0.25
	3	42	10,662	3	17	2	**4.5**	4	31	2	4.8	2.01
mer [N R]	2 2	961	85	4	113	3	**8.3**	7	1,107	5	27.2	0.09
	2 5	5,776	427,363	4	113	3	**29.6**	7	1,257	5	151.6	1.96
	3 2	16,759	171	4	173	3	**210.5**	–	–	–	mem-out	0.42

Figure 4.24: Performance of the learning-based compositional verification using rule (Asym)

Case study [parameters]		Component sizes		(Asym)		(Asym-N)		Non-comp.
		$\|\mathcal{M}_2 \otimes G^{err}\|$	$\|\mathcal{M}_1\|$	$\|A\|$	Time	$\|A\|$	Time	Time
client-server (N failures) [N]	6	25,801	49	–	mem-out	8	38.7	0.7
	7	123,053	64	–	mem-out	24	161.2	1.7
mer [N R]	3 5	224,974	1,949,541	–	mem-out	4	**29.4**	48.2
	4 5	7,949,992	6,485,603	–	mem-out	4	**120.6**	mem-out
	5 5	265,559,722	17,579,131	–	mem-out	4	**3,876.3**	mem-out

Figure 4.25: Performance of the learning-based compositional verification using rule (Asym-N)

Results

Figure 4.24 compares the performance of our L*-based and NL*-based compositional verification approach for rule (Asym), with the non-compositional verification using PRISM (sparse engine). For each case study, we report the "Component sizes", i.e. the number of states of product PA $\mathcal{M}_2 \otimes G^{err}$ and PA $\mathcal{M}_1$. We also report the size $|A|$ of learnt assumptions: in the L*-based method, this means the number of states of the learnt DFAs, while, in the NL*-based methods, this means the size of DFAs that are obtained by converting the learnt RFSAs via subset construction. The performance of different methods is measured in terms of the total run time (in seconds), and the number of membership queries (MQ) and equivalence queries (EQ) needed for learning.

We observe that both L*-based and NL*-based methods successfully learned correct assumptions that are much more compact than components in almost all cases. For example, in the consensus (2,4,4) model, the learnt assumption A has only 12 states while the corresponding component $\mathcal{M}_1$ has 431,649 states; in the mer (2,5) model, the component size is $427,363$ and the L*-based method learns an assumption with $|A| = 4$. This kind of reduction is quite significant, demonstrating that our approach can be used to generate succinct assumptions for the compositional verification of large systems with rule (Asym).

We also observe that the L*-based method generally performs better than the NL*-based method; however, on several of the larger case studies (e.g. the consensus models), NL* has better performance because it requires fewer equivalence queries. Note that NL* often needs a larger number of membership queries than L*, but answering membership queries is less costly than answering equivalence queries which require performing the multi-objective model checking.

The primary focus of our current research is on the feasibility of learning appropriate assumptions automatically and on the quality of learnt assumptions (e.g. their size). Thus, we are not particularly interested in comparing the total run time of our

prototype tool with the non-compositional verification using PRISM, which is a state-of-the-art symbolic probabilistic model checker with highly optimised performance. Nevertheless, it is worth noting that compositional verification is actually faster than non-compositional verification in two of the consensus models (2,3,20) and (3,3,20). In the latter case, PRISM cannot finish the non-compositional verification within 24 hours, but our prototype tool can verify the property compositionally within less than 7 minutes.

When we scale up the case studies, some of the models become so large that decomposing the system into two components is not good enough and the compositional verification with rule (Asym) runs out of memory. Thus, we decompose the system into multiple smaller components and apply the compositional verification with rule (Asym-N). Figure 4.25 shows the performance of our approach for learning-based compositional verification using rule (Asym-N) on two case studies: the client-server model with N clients exhibiting the potential of failures, and the Mars exploration rover (mer). In all cases, we report the component sizes in terms of PAs $\mathcal{M}_2 \otimes G^{err}$ and $\mathcal{M}_1$ for rule (Asym). For the use of rule (Asym-N), we decompose $\mathcal{M}_2$ further into separate client (for client-server) or user thread (for mer) models. We also report the largest size $|A|$ of a series of assumptions learnt by rule (Asym-N), and the total run time in seconds. We do not include a benchmark example rule (Circ), which is a special case of rule (Asym-N).

The experimental results demonstrate that we can successfully learn small assumptions and perform compositional verification using rule (Asym-N) for models which are beyond the scalability of rule (Asym). Moreover, in two cases of mer (4,5) and (5,5), where the non-compositional verification is not feasible, we can still verify the models using our learning-based compositional verification with rule (Asym-N). Note that the compositional verification for mer (5,5) runs for more than 1 hour, because the model is quite huge. Indeed, the component $\mathcal{M}_1$ has about 17.5 million states and

$\mathcal{M}_2 \otimes G^{err}$ has more than 265 million states. So, it takes a long time to perform the model checking for answering membership and equivalence queries, even though the learnt assumptions are very small (with a maximum of 4 states).

We observe from Figures 4.24 and 4.25 that non-compositional verification using PRISM is generally faster than compositional verification, this is because PRISM is a highly optimized tool based on the symbolic data structure of MTBDDs and performs particularly well for small to medium sized models. On the other hand, compositional verification requires iterations of model checking for answering membership and equivalence queries, its performance is relatively slow (but mostly still managed to complete within 1-2 minutes). The advantage of composition verification is more obvious when it comes to lager models, such as consensus models (2,3,20) and (3,3,20), and mer (4,5) and (5,5), where non-compositional verification is significantly slower or even not feasible. Since most real-world applications employ ultra-large models, we would expect the compositional verification to be extremely useful for such applications.

4.6 Summary and Discussion

In this chapter, we have presented a fully-automated approach for learning probabilistic assumptions for the compositional verification of systems composed of PAs. In this assume-guarantee framework, the formats of assumptions and guarantees are the same. Since we aim to verify probabilistic safety properties, the assumptions are in the format represented by a DFA plus a lower probability bound [1]. Our approach can handle three different assume-guarantee rules: an asymmetric rule (ASYM) for two-component systems, an asymmetric rule (ASYM-N) for n-component systems, and a circular rule (CIRC). These three rules are probabilistic analogs of the most important rules in the non-probabilistic setting [CGP03, PGB+08].

[1] There is a thought that, instead of restricting to lower bounds, we may find an interval for the probability of certain regular language. However, it is not clear how such assumptions can be used in the compositional verification framework.

We also considered the use of two different learning methods, i.e. the L* and NL* algorithms. We build a prototype tool that implements our approach and applied it successfully to a set of benchmark case studies. The experimental results are very encouraging, which demonstrated that our approach can learn significantly smaller assumptions about large components in most cases, e.g. a 4-state assumption is learnt for a component that has more than 17.5 million states in the mer (5,5) case shown in Figure 4.25. Furthermore, our approach enables compositional verification for some large models where non-compositional verification is not feasible, e.g. the consensus (3,3,20) case in Figure 4.24 and the mer (4,5) case in Figure 4.25.

One weakness of our approach is that it is built on top of an incomplete compositional framework of [KNPQ10], which means that, even if a property holds in a system, there may not necessarily exist appropriate assumptions to verify it compositionally. As described in Section 4.3, we equip our assumption learning approach for rule (Asym) with the ability to generate a pair of lower/upper bounds on the minimum probability of the system $\mathcal{M}_1 \| \mathcal{M}_2$ satisfying the regular safety property G. Another solution for solving this problem is to develop a complete compositional verification framework that uses richer classes of probabilistic assumptions, which we will explore in Chapter 5.

As mentioned in Section 4.5, our current focus is on the feasibility of learning assumptions rather than optimising perormance of our prototype tool. However, there are a few thoughts that might help to optimise our approach. The first idea is to adapt the alphabet refinement technique [PGB+08]. Currently, we set a fixed alphabet a priori for learning, which is usually the interface alphabet $\alpha_A = \alpha_{\mathcal{M}_1} \cap (\alpha_{\mathcal{M}_2} \cup \alpha_G)$. By determining a reduced alphabet, we might be able to learn a smaller assumption and improve the total run time. The second idea is to use a symbolic BDD-based implementation of the L* learning algorithm [NMA08]. The efficiency of BDD operations might help to optimise our prototype. So far, we have only considered the verification of systems against probabilistic safety properties. It may be possible to extend our as-

sumption learning approach to the assume-guarantee rules in [FKN$^+$11] for ω-regular properties and expected reward properties.

Chapter 5

Learning Assumptions for Synchronous Probabilistic Systems

Recall that, in Chapter 4, we proposed an approach to learn assumptions represented as probabilistic safety properties for the compositional verification of systems composed of PAs. Probabilistic safety properties have a limited expressiveness to capture the behaviour of probabilistic systems. Indeed, the underlying assume-guarantee rules used in Chapter 4 are *incomplete*, because the components modelled as PAs cannot be interchanged with assumptions formalised as probabilistic safety properties.

In this chapter, we present a new probabilistic assume-guarantee verification framework that uses a more expressive class of assumptions represented as *Rabin probabilistic automata* (RPAs), which capture trace probabilities of system components. It is well-known that developing compositional verification methods on top of trace-based equivalence relations for nondeterministic probabilistic systems (e.g. PAs) is difficult [Seg95]. Therefore, we restrict our work in this chapter to *purely* probabilistic systems.

We consider *discrete-time Markov chains* (DTMCs) built from components modelled as *probabilistic I/O systems* (PIOSs), which extend DTMCs with output actions and (nondeterministic) input actions, using a *synchronous* parallel operator. The relation between components and corresponding assumptions is captured by *(weak) language inclusion*, which abstracts away the unobservable behaviour of components to produce smaller assumptions.

In the remainder of this chapter, we first propose an asymmetric assume-guarantee rule (Asym-Pios) for verifying probabilistic safety properties on DTMCs composed of two PIOS components in Section 5.1. We give a semi-algorithm in Section 5.2 to check language inclusion between RPAs, which is an undecidable problem [BC01, MO05]. Next, we present a novel L*-style learning method for RPAs in Section 5.3. Then in Section 5.4, we build a fully-automated implementation of the assume-guarantee rule (Asym-Pios), in which the assumptions are learnt automatically as RPAs. We report on experimental results of our prototype implementation for a selection of benchmark case studies in Section 5.5, and summarise contributions of this chapter in Section 5.6.

The work presented in this Chapter was first published in a jointly authored paper [FHKP11a]. It is extended here with corrections, detailed proofs, running examples, and experimental results. Some of the notations have been modified for consistency of the thesis. See Section 1.3 for credits of this chapter.

5.1 Compositional Verification for PIOSs

In this section, we define a compositional verification framework for fully probabilistic systems composed of PIOSs based on a new assume-guarantee rule (Asym-Pios), in which assumptions are represented by RPAs. We first introduce the formalisms of PIOSs in Section 5.1.1 and RPAs in Section 5.1.2. Then, we propose the assume-guarantee rule (Asym-Pios) in Section 5.1.3.

5.1.1 Probabilistic I/O Systems

In the following, we introduce *probabilistic I/O system* (PIOSs), which generalise DTMCs (defined in Section 3.2.1) by distinguishing between different types (e.g. input, output) of actions. An individual PIOS component can respond to nondeterministic *input* actions issued by other components, and the synchronous composition of two PIOSs yields a fully probabilistic DTMC system. Note that, in this chapter, we allow sub-distributions in DTMCs and omit the atomic proposition labelling function L.

Definition 5.1 (PIOS) *A* probabilistic I/O system *(PIOS) is a tuple* $\mathcal{M} = (S, \overline{s}, \alpha, \delta)$*, where* S *is a finite set of states,* $\overline{s} \in S$ *is an initial state,* α *is an alphabet of action labels, and* $\delta : S \times (\alpha \cup \{\tau\}) \to \mathit{SDist}(S)$ *is a transition function satisfying:*

- α *is partitioned into three disjoint sets of* input*,* output *and* hidden *actions, denoted by* α^I, α^O *and* α^H*, respectively; input actions* α^I *are further partitioned into* m *disjoint* bundles $\alpha^{I,i}$ $(1 \leq i \leq m)$ *for some* m*;*
- *for each state* s*, the set of* enabled *actions* $A(s) \stackrel{\text{def}}{=} \{a \in \alpha \mid \delta(s,a) \text{ is defined}\}$ *satisfies either* $|A(s)| = 1$ *if* $A(s) \in \alpha^O \cup \alpha^H \cup \{\tau\}$*, or* $A(s) = \alpha^{I,i}$ *for some input action bundle* $\alpha^{I,i}$*.*

Transitions and paths of PIOSs are defined similarly as for DTMCs. From any state in a PIOS, there is either a single outgoing transition labelled with an output, hidden or τ action, or k transitions that correspond to certain bundle $\alpha^{I,i}$ comprising k input actions. Intuitively, each bundle $\alpha^{I,i}$ represents a set of possible input actions that can be communicated from other components. The nondeterministic behaviour of a PIOS is resolved by responding to such input actions. We denote a transition labelled with an input (resp. output) action a as $s \xrightarrow{a?} \mu$ (resp. $s \xrightarrow{a!} \mu$), where $s \in S$ and $\mu \in \mathit{SDist}(S)$. We call the set of input and output actions $\alpha^I \cup \alpha^O$ *external* actions. We write $s \not\to$ if

no action is enabled in a state s, i.e. $A(s) = \emptyset$. A PIOS can be considered a DTMC, if $|A(s)| \leq 1$ for any state s.

Typically, we need the notion of *adversaries* to reason about models exhibiting both probabilistic and nondeterministic behaviour (e.g. PAs in Section 3.2.2). For PIOSs, however, following a sequence of actions $w = a_1 \dots a_n$ dictates a unique adversary; this is because PIOSs only have nondeterminism on input actions, and, in each execution step, at most one input action from a bundle would be trigged by the external environment. Given a finite path $\rho = s_0 \xrightarrow{a_0} s_1 \cdots \xrightarrow{a_{n-1}} s_n$ in $\mathcal{M}$, the path probability is given by $Pr_{\mathcal{M}}(\rho) = \prod_{i=0}^{n-1} \delta(s_i, a_i)(s_{i+1})$. Let $act(\rho)$ denote the sequence of actions $a_0 \dots a_{n-1}$ obtained from ρ. The probability of a particular sequence of actions $w \in (\alpha \cup \{\tau\})^*$ being observed in PIOS $\mathcal{M}$ is well-defined: $Pr_{\mathcal{M}}(w) = \sum_{act(\rho)=w} Pr_{\mathcal{M}}(\rho)$. Let $st : (\alpha \cup \{\tau\})^*\alpha \to \alpha^*$ be the function that removes all τs from a finite word with a non-τ ending, and define the (weak) probability of accepting a τ-free word $\bar{w} \in \alpha^*$ in $\mathcal{M}$ as $Pr^{\mathrm{w}}_{\mathcal{M}}(\bar{w}) = \sum_{\bar{w}=st(w)} Pr_{\mathcal{M}}(w)$ for all $w \in (\alpha \cup \{\tau\})^*\alpha$. Note that $Pr^{\mathrm{w}}_{\mathcal{M}}(\bar{w})$ is a well-defined probability value, because all the corresponding paths of words $w \in (\alpha \cup \{\tau\})^*\alpha$ with $st(w) = \bar{w}$ belong to the same probability space.

Example 5.2 *Figure 5.1 shows a pair of PIOSs $\mathcal{M}_1$ and $\mathcal{M}_2$. $\mathcal{M}_1$ is a data communicator that can either "receive" or "send" data depending on a probabilistic choice. When $\mathcal{M}_1$ is ready to receive data, it outputs a "ready!" signal to the data generator $\mathcal{M}_2$. On the other hand, if $\mathcal{M}_1$ chooses to send data, then a failure would occur and a "fail!" signal would be sent out. There is an internal initialisation step labelled by "init" in $\mathcal{M}_2$ that, with probability 0.8, it waits for signals from $\mathcal{M}_1$; otherwise, it just starts to send the sequence of data packets immediately, which is modelled by the alternating actions d0 and d1. Note that input/output actions for $\mathcal{M}_1, \mathcal{M}_2$ are labelled with ?/! in the figure; all other actions are hidden. Each PIOS has a single input action bundle: $\alpha_1^{I,1} = \{d0, d1\}$, $\alpha_2^{I,1} = \{ready, fail\}$. A path*

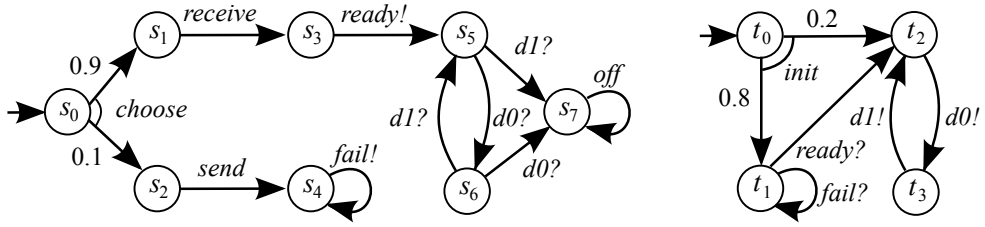


Figure 5.1: A pair of PIOSs $\mathcal{M}_1$ and $\mathcal{M}_2$ (taken from [FHKP11a])

$\rho = t_0 \xrightarrow{init,0.8} t_1 \xrightarrow{ready?,1} t_2 \xrightarrow{d0!,1} t_3 \xrightarrow{d1!,1} t_2$ *through* $\mathcal{M}_2$ *corresponds to a word* $w = \langle init, ready?, d0!, d1! \rangle$*, whose probability is given by* $Pr_{\mathcal{M}_2}(w) = 0.8$.

Next, we define synchronous parallel composition of PIOSs. For simplicity, we limit our attention to a binary parallel operator, which matches the assume-guarantee rule used in this chapter. However, it should be possible to adapt the operator for composing multiple PIOSs. Given two PIOSs $\mathcal{M}_1$, $\mathcal{M}_2$ with alphabets α_1, α_2, we say that $\mathcal{M}_1$ and $\mathcal{M}_2$ are *composable* if $\alpha_1^I = \alpha_2^O$, $\alpha_1^O = \alpha_2^I$ and $\alpha_1^H \cap \alpha_2^H = \emptyset$. Let $\perp$ denote an *idle* action, and $*$ be a binary operator joining two actions into a *combined* action, e.g. $a * b$ means that actions a and b occur simultaneously.

Definition 5.3 (Parallel composition of PIOSs) *The synchronous parallel composition of two composable PIOSs* $\mathcal{M}_i = (S_i, \overline{s}_i, \alpha_i, \delta_i)$ *for* i=1, 2 *is given by* $\mathcal{M}_1 \| \mathcal{M}_2 = (S_1 \times S_2, (\overline{s}_1, \overline{s}_2), \alpha, \delta)$*, where the alphabet is* $\alpha = \alpha_1^I \cup \alpha_1^O \cup \big((\alpha_1^H \cup \{\perp\}) * (\alpha_2^H \cup \{\perp\})\big)$ *and the transition relation* δ *is defined such that* $(s_1, s_2) \xrightarrow{\gamma} \mu_1 \times \mu_2$ *if and only if one of the following holds:*

- $s_1 \xrightarrow{a} \mu_1$, $s_2 \xrightarrow{a} \mu_2$, *and* $\gamma = a$
- $s_1 \xrightarrow{b_1} \mu_1$, $s_2 \xrightarrow{b_2} \mu_2$, *and* $\gamma = b_1 * b_2$
- $s_1 \xrightarrow{b_1} \mu_1$, $s_2 \xrightarrow{a}$ *(or* $s_2 \not\rightarrow$*)*, $\mu_2 = \eta_{s_2}$, *and* $\gamma = b_1 * \perp$
- $s_1 \xrightarrow{a}$ *(or* $s_1 \not\rightarrow$*)*, $s_2 \xrightarrow{b_2} \mu_2$, $\mu_1 = \eta_{s_1}$, *and* $\gamma = \perp * b_2$

for any $a \in \alpha_1^I \cup \alpha_1^O$, $b_1 \in \alpha_1^H \cup \{\tau\}$ *and* $b_2 \in \alpha_2^H \cup \{\tau\}$.

Note that, in the above definition, the $*$ operator is lifted for the set $(\alpha_1^H \cup \{\perp\}) * (\alpha_2^H \cup \{\perp\})$. We distinguish the following cases for the transition relation:

(1) if actions of both transitions in $\mathcal{M}_1$ and $\mathcal{M}_2$ are external actions and they are matching (e.g. $a \in \alpha_1^I$ and $a \in \alpha_2^O$), then they will be performed simultaneously;

(2) if both actions are hidden or τ actions, say $b_1 \in \alpha_1^H \cup \{\tau\}$ and $b_2 \in \alpha_2^H \cup \{\tau\}$, then they will be carried out by a combined action $b_1 * b_2$ simultaneously;

(3) if the transition of one component is labelled with an external action a (or no action is enabled in the start state), and the other component's transition is labelled with b, which is a hidden or τ action, then a will wait (or equivalently perform an idle action $\perp$) and a combined action $\perp * b$ will be executed.

We show by the following lemma that the synchronous parallel composition of two PIOSs $\mathcal{M}_1 \| \mathcal{M}_2$ results in a DTMC.

Lemma 5.4 *Given two composable PIOSs* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, their synchronous parallel composition product* $\mathcal{M}_1 \| \mathcal{M}_2$ *is a DTMC.*

Proof: Based on Definition 5.3, we distinguish three cases for the states in $\mathcal{M}_1 \| \mathcal{M}_2$.

1. $(s_1, s_2) \xrightarrow{a}$ where $a \in \alpha_1^I \cup \alpha_1^O$: one of the states s_1 and s_2 may have multiple input actions while the other state only has one output action; thus, there is only matching action enabled in the product state (s_1, s_2).

2. $(s_1, s_2) \xrightarrow{b_1 * b_2}$ where $b_1 \in \alpha_1^H \cup \{\tau\}$ and $b_2 \in \alpha_2^H \cup \{\tau\}$: since each of the states s_1, s_2 has exactly one hidden or τ action enabled, there is only one available combined action in state (s_1, s_2).

3. $(s_1, s_2) \xrightarrow{b_1 * \perp}$ or $(s_1, s_2) \xrightarrow{\perp * b_2}$: if state s_i performs an action $b_i \in \alpha_i^H \cup \{\tau\}$ and the other state has one or more enabled external actions (or no action enabled at all), then the product state (s_1, s_2) would take a combined action joined by b_i and the idle action $\perp$.

For any state, there is at most one enabled action. Thus, $\mathcal{M}_1 \| \mathcal{M}_2$ is a DTMC. ■

5.1.2 Rabin Probabilistic Automata

The model of *Rabin probabilistic automata* (RPAs) was originally proposed by Rabin [Rab63], also called *probabilistic automata* in some literature; however, we avoid this term to prevent confusion with the identically named model by Segala [Seg95], which is also used in this thesis (see Section 3.2.2 and Chapter 4).

Definition 5.5 (RPA) *A* Rabin probabilistic automaton *(RPA) is a tuple* $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P})$*, where* S *is a finite set of states,* $\overline{s} \in S$ *is an initial state,* α *is an alphabet, and* $\mathbf{P} : \alpha \to (S \times S \to [0, 1])$ *is a transition function mapping actions to probability matrices such that* $\sum_{s' \in S} \mathbf{P}[a](s, s') \in [0, 1]$ *for all* $a \in \alpha$ *and* $s \in S$*.*

Note that the above definition is slightly non-standard. Firstly, we allow $\sum_{s' \in S} \mathbf{P}[a](s, s') \in [0, 1]$, i.e. rows of matrices $\mathbf{P}[a]$ can sum to less than 1. This is based on the understanding that some action may not be enabled in a state or may lead to sub-distributions. We can always translate a RPA defined in this manner to the classical definition by adding a (non-accepting) sink state and additional transitions to make the probabilistic distributions full. Secondly, we omit the accepting condition of states, effectively making all states accepting; because we restrict our attention to RPAs that correspond to executions of probabilistic models.

A run of a finite word $w \in \alpha^*$ on a RPA $\mathcal{A}$ is a finite sequence $s_0 \xrightarrow{a_0} s_1 \cdots \xrightarrow{a_{n-1}} s_n$, where $w = a_0 a_1 \cdots a_{n-1}$ and $s_0 = \overline{s}$. The probability of accepting w in $\mathcal{A}$ is given

by $Pr_{\mathcal{A}}(w) = \vec{\iota}\mathbf{P}[w]\vec{\kappa}$, where $\vec{\iota}$ is a 0-1 row vector indexed by states S with $\vec{\iota}(s) = 1$ if $s = \overline{s}$ and $\vec{\iota}(s) = 0$ otherwise, $\vec{\kappa}$ is a column vector over S containing all 1s, and $\mathbf{P}[w] = \mathbf{P}[a_1]\mathbf{P}[a_2]\cdots\mathbf{P}[a_n]$ is obtained from the multiplication of transition matrices. Intuitively, $Pr_{\mathcal{A}}(w)$ is determined by tracing path(s) through $\mathcal{A}$ that correspond to w, with $\mathbf{P}[a_i](s, s')$ giving the probability of moving from state s to s' when reading action a_i for $i \in \mathbb{N}$.

RPAs are a generalisation of finite-state automata (Definition 3.1). Indeed, a finite-state automaton $\mathcal{A} = (S, \overline{s}, \alpha, \delta, F)$ can be considered as a RPA with (possibly) non-accepting states, where $\mathbf{P}[a](s, s') = 1$ if $\delta(s, a)$ is defined and $\mathbf{P}[a](s, s') = 0$ otherwise; and the probability $Pr_{\mathcal{A}}(w)$ of accepting a finite word w in a finite-state automaton $\mathcal{A}$ is either 1 or 0. The languages accepted by RPAs are called *stochastic languages*, which include regular languages (i.e. accepted by finite-state automata) as a subset. In the following, we generalise the notion of language inclusion/equivalence of finite-state automata (see Section 3.1) to RPAs, which will be used later to establish the relation between a PIOS and its assumption.

Definition 5.6 (Language inclusion/equivalence of RPAs) *Given two RPAs $\mathcal{A}_1$ and $\mathcal{A}_2$ with the same alphabet α, we say $\mathcal{A}_1$ and A_2 are related by (strong)* language inclusion *(resp.* language equivalence*), denoted $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ (resp. $\mathcal{A}_1 \equiv \mathcal{A}_2$), if for every word $w \in \alpha^*$, $Pr_{\mathcal{A}_1}(w) \leq Pr_{\mathcal{A}_2}(w)$ (resp. $Pr_{\mathcal{A}_1}(w) = Pr_{\mathcal{A}_2}(w)$).*

The problem of checking language equivalence for RPAs is decidable in polynomial time (see e.g. [Sch61, Tze92b, DHR08, KMO$^+$11]). However, checking language inclusion for RPAs is undecidable [BC01, MO05], because one can reduce the undecidable *nonemptiness with threshold* problem [Paz71] to it. Later, in Section 5.2, we propose a semi-algorithm to check language inclusion for RPAs; and in Section 5.3, we develop a new L*-style learning algorithm which learns a RPA $\mathcal{A}$ for a target stochastic language $\mathcal{L}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$.

5.1.3 Assume-Guarantee Reasoning Rule (Asym-Pios)

Before we present an assume-guarantee rule for the compositional verification of PIOSs, we first formulate a notion of *assumptions* about PIOSs.

Definition 5.7 (Assumption RPA) *Let $\mathcal{M}$ be a PIOS with alphabet $\alpha = \alpha^I \uplus \alpha^O \uplus \alpha^H$ and input action bundles $\alpha^I = \biguplus_{i=1}^{m} \alpha^{I,i}$. An* assumption $\mathcal{A}$ *about $\mathcal{M}$ is a RPA $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P})$ satisfying, for each state $s \in S$,*

- *either all or none of the actions in a bundle $\alpha^{I,i}$ $(1 \leq i \leq m)$ are enabled in s;*
- *$p^{\max}(s) \in [0,1]$, where:*

$$p^{\max}(s) \stackrel{\text{def}}{=} \sum_{a \in \alpha^O \cup \alpha^H} \sum_{s' \in S} \mathbf{P}[a](s,s') + \sum_{i=1}^{m} p_i^{\max}(s)$$

with $p_i^{\max}(s) \stackrel{\text{def}}{=} \max_{a \in \alpha^{I,i}} \sum_{s' \in S} \mathbf{P}[a](s,s')$.

Intuitively, an assumption about a PIOS is a RPA in which all states are accepting and, for any state, the sum of all its outgoing transition probabilities by output or hidden actions, together with the *maximum* outgoing probability sums among all the input actions in each bundle, yields at most 1. These characterisations are due to the fact that we are using a specific class of RPAs to capture the abstract behaviour (i.e. trace probabilities) of PIOSs. For simplicity, in the above definition we require that assumption $\mathcal{A}$ contains all actions $a \in \alpha$ from PIOS $\mathcal{M}$. As we will show later, to obtain smaller assumptions, we can actually rename all hidden actions α^H of $\mathcal{M}$ to τ; and thus, the assumption would only include external actions $\alpha^I \cup \alpha^O$ of $\mathcal{M}$.

The relation between a PIOS and its RPA assumption is formalised by the notion of *weak language inclusion* (it is a *weak* relation because of the ignorance of τ action).

Definition 5.8 (Weak language inclusion/equivalence) *Given a PIOS $\mathcal{M}$ with alphabet α and an assumption $\mathcal{A}$ about $\mathcal{M}$, we say that $\mathcal{M}$ and $\mathcal{A}$ are related by*

weak language inclusion (resp. equivalence), *denoted* $\mathcal{M} \sqsubseteq_{\mathrm{w}} \mathcal{A}$ *(resp.* $\mathcal{M} \equiv_{\mathrm{w}} \mathcal{A}$*), if* $Pr^{\mathrm{w}}_{\mathcal{M}}(w) \leq Pr_{\mathcal{A}}(w)$ *(resp.* $Pr^{\mathrm{w}}_{\mathcal{M}}(w) = Pr_{\mathcal{A}}(w)$*) for every word* $w \in \alpha^*$.

The problem of checking whether a RPA $\mathcal{A}$ is a *valid* assumption about a PIOS $\mathcal{M}$, i.e. whether $\mathcal{M} \sqsubseteq_{\mathrm{w}} \mathcal{A}$ is true, can be reduced to checking (strong) language inclusion between two RPAs. The reduction is formalised by the following proposition.

Proposition 5.9 *Let* $\mathcal{M} = (S, \overline{s}, \alpha, \delta)$ *be a PIOS and* $\mathcal{A}$ *be an assumption about* $\mathcal{M}$. *We denote by* $rpa(\mathcal{M})$ *the (straightforward) translation of* $\mathcal{M}$ *to a RPA, defined as* $(S, \overline{s}, \alpha \cup \{\tau\}, \mathbf{P})$, *where* $\mathbf{P}[a](s, s') = \delta(s, a)(s')$ *for* $a \in \alpha \cup \{\tau\}$. *Then, letting* $\mathcal{A}^\tau$ *denote the RPA derived from* $\mathcal{A}$ *by adding* τ *to its alphabet and a probability* 1 τ*-loop to every state, we have that:* $\mathcal{M} \sqsubseteq_{\mathrm{w}} \mathcal{A} \iff rpa(\mathcal{M}) \sqsubseteq \mathcal{A}^\tau$.

Proof: For each word w in $\mathcal{M}$, the following holds:

$$Pr_{\mathcal{M}}(w) \overset{(1)}{\leq} Pr^{\mathrm{w}}_{\mathcal{M}}(st(w)) \overset{(2)}{\leq} Pr_{A}(st(w)) \overset{(3)}{=} Pr_{A^\tau}(st(w)) \overset{(4)}{=} Pr_{A^\tau}(w),$$

where (1) holds because $Pr^{\mathrm{w}}_{\mathcal{M}}(st(w)) = \sum_{w'} Pr_{\mathcal{M}}(w')$ for $\mathcal{W} = \{w' \mid st(w') = st(w)\}$ and $w \in \mathcal{W}$; (2) is given by the condition that $\mathcal{M} \sqsubseteq_{\mathrm{w}} \mathcal{A}$ (see Definition 5.8); (3) is true because adding τ-loop to every state with probability 1 does not change the probability of accepting τ-free words; (4) is true because every τ in $\mathcal{A}^\tau$ is a self-loop with probability 1. It is also clear that $Pr_{\mathcal{M}}(w) = Pr_{rpa(\mathcal{A})}(w)$. The above (in)equalities hold from both directions, thus $\mathcal{M} \sqsubseteq_{\mathrm{w}} \mathcal{A}$ if and only if $rpa(\mathcal{M}) \sqsubseteq \mathcal{A}^\tau$. ■

For the purpose of compositional verification (which will be explained later), we also need to perform the conversion in the other direction, i.e. from an assumption $\mathcal{A}$ to a (weak) language equivalent PIOS, denoted by $pios(A)$.

Definition 5.10 (Assumption-to-PIOS conversion) *Given an assumption* $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P})$ *with the action partition* $\alpha = (\biguplus_{i=1}^{m} \alpha^{I,i}) \uplus \alpha^O \uplus \alpha^H$, *its conversion to*

PIOS is given by $pios(\mathcal{A}) = (S', \overline{s}, \alpha, \delta)$*, where the states set* $S' = S \uplus \{s^a | s \in S, a \in \alpha^H \cup \alpha^O\} \uplus \{s^i | s \in S, 1 \leq i \leq m\}$ *and the transition relation* δ *are constructed such that, for any transition* $s \xrightarrow{a} s'$ *in* $\mathcal{A}$*,*

- *if* $a \in \alpha^O \cup \alpha^H$*, then* $\delta(s,\tau)(s^a) = \frac{p}{p^{\max}(s)}$ *and* $\delta(s^a, a)(s') = p^{\max}(s)$*;*
- *if* $a \in \alpha^{I,i}$ *(for* $1 \leq i \leq m$*), then* $\delta(s,\tau)(s^i) = \frac{p_i^{\max}(s)}{p^{\max}(s)}$ *and* $\delta(s^i, a)(s') = p \cdot \frac{p^{\max}(s)}{p_i^{\max}(s)}$*.*

where $p = \mathbf{P}[a](s, s')$*, and* $p^{\max}(s), p_i^{\max}(s)$ *are defined as in Definition 5.7.*

Intuitively, an assumption may have a mixture of input, output and hidden actions enabled in the same state. It can be converted into a (weak) language equivalent PIOS (see Definition 5.1), in which a state has either a single outgoing transition labelled by an output or hidden action, or a bundle of transitions with input actions, by adding extra states and τ transitions. We show below that $pios(A)$ is well-defined.

Proposition 5.11 *Given an assumption* $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P})$ *and action partition* $\alpha = (\biguplus_{i=1}^{m} \alpha^{I,i}) \uplus \alpha^O \uplus \alpha^H$*, the conversion* $pios(\mathcal{A}) = (S', \overline{s}, \alpha, \delta)$ *is a well-defined PIOS.*

Proof: With the given action partition, we need to prove that, for any state s in $pios(\mathcal{A})$,

(1) if $A(s) \in \alpha^O \cup \alpha^H \cup \{\tau\}$, then $|A(s)| = 1$;

(2) if $A(s)$ contains some input actions from a bundle $\alpha^{I,i}$, then $A(s) = \alpha^{I,i}$;

where $A(s)$ is the set of enabled actions in state s.

We first consider the case (1) for all τ, output and hidden actions. Given that the assumption $\mathcal{A}$ does not have any τ-transition, by construction, if a state s in $pios(\mathcal{A})$ contains a τ action, then it will be the unique enabled action in that state. The transition $\delta(s,\tau)(s^a)$ is well-defined, because the newly created state s^a is unique for an output or hidden action $a \in \alpha^O \cup \alpha^H$, and $\frac{p}{p^{\max}(s)}$ is a probability due to $p \in [0,1]$, $p^{\max}(s) \in [0,1]$ and $p \leq p^{\max}(s)$. By construction, the transition from any state s^a to

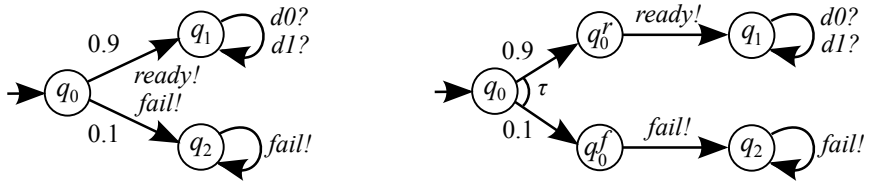


Figure 5.2: RPA $\mathcal{A}$ and its PIOS conversion $pios(\mathcal{A})$ (taken from [FHKP11a])

s' is unique for any output or hidden action a, so that $\delta(s^a, a)(s')$ is well-defined with probability $p^{\max}(s) \in [0, 1]$.

Now we consider the case (2) for input actions. Since there is at most one input action bundle $\alpha^{I,i}$ enabled for any state in $\mathcal{A}$, the newly created state s^i in $pios(\mathcal{A})$ is unique. Thus, the transition $\delta(s, \tau)(s^i)$ is well-defined and $\frac{p_i^{\max}(s)}{p^{\max}(s)}$ is a probability. For any state s^i, the transition to s' with action a is unique by construction; hence $\delta(s^i, a)(s')$ is well-defined and $p \cdot \frac{p^{\max}(s)}{p_i^{\max}(s)}$ is a probability. Since the input actions in a bundle will be enabled in an all-or-none fashion (Definition 5.1), we have $A(s) = \alpha^{I,i}$ if some input action bundle $\alpha^{I,i}$ is enabled in state s of $pios(\mathcal{A})$.

All in all, we have proved that $pios(A)$ is a well-defined PIOS.

■

Example 5.12 *Figure 5.2 shows a valid assumption $\mathcal{A}$ about PIOS $\mathcal{M}_1$ (Figure 5.1), i.e. $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A}$, and the converted PIOS $pios(\mathcal{A})$. In $\mathcal{A}$, state q_0 has two output actions "ready!" and "fail!" leading to respective sub-distributions. $\mathcal{A}$ is converted into a PIOS $pios(\mathcal{A})$ by adding two states q_0^{ready} and q_0^{fail} (abbreviated to q_0^r and q_0^f), which are linked to the initial state q_0 through a distribution labelled with τ.*

Now, we present a new assume-guarantee reasoning rule for verifying a probabilistic safety property $\langle G \rangle_{\geq p}$ on a DTMC $\mathcal{M}_1 \| \mathcal{M}_2$ composed of a pair of PIOSs. For simplicity, we assume that the property $\langle G \rangle_{\geq p}$ refers only to the input/output actions of $\mathcal{M}_1$ and $\mathcal{M}_2$; we also rename all hidden actions of $\mathcal{M}_1$ and $\mathcal{M}_2$ as τ actions, which affects neither

the structure nor the transition probabilities of $\mathcal{M}_1 \| \mathcal{M}_2$. To establish the assume-guarantee reasoning rule, we use *assume-guarantee triples* in the form of $\langle \mathcal{A} \rangle \, \mathcal{M} \, \langle G \rangle_{\geq p}$, meaning that "whenever component $\mathcal{M}$ is part of a system satisfying the assumption $\mathcal{A}$, then the system is guaranteed to satisfy $\langle G \rangle_{\geq p}$". Formally:

Definition 5.13 (Probabilistic assume-guarantee triple for PIOS) *If $\mathcal{M}$ is a PIOS with alphabet α, $\mathcal{A}$ is an assumption about $\mathcal{M}$, and $\langle G \rangle_{\geq p}$ is a probabilistic safety property, then $\langle \mathcal{A} \rangle \, \mathcal{M} \, \langle G \rangle_{\geq p}$ is an* assume-guarantee triple *with the following meaning:*

$$\langle \mathcal{A} \rangle \, \mathcal{M} \, \langle G \rangle_{\geq p} \iff \forall \mathcal{M}'. \, (\mathcal{M}' \sqsubseteq_{\mathrm{w}} \mathcal{A} \implies \mathcal{M}' \| \mathcal{M} \models \langle G \rangle_{\geq p})$$

where $\mathcal{M}'$ represents a PIOS that is composable with $\mathcal{M}$.

Based on the following proposition, we can check the satisfaction of $\langle \mathcal{A} \rangle \, \mathcal{M} \, \langle G \rangle_{\geq p}$ by first converting assumption $\mathcal{A}$ into $pios(\mathcal{A})$ and then using standard model checking techniques for DTMCs (see Section 3.3.2).

Proposition 5.14 *$\langle \mathcal{A} \rangle \, \mathcal{M} \, \langle G \rangle_{\geq p}$ holds if and only if $pios(\mathcal{A}) \| \mathcal{M} \models \langle G \rangle_{\geq p}$.*

Proof: See Appendix A. ■

Theorem 5.15 *Suppose that $\mathcal{M}_1$, $\mathcal{M}_2$ are two PIOSs, $\mathcal{A}$ is an assumption about $\mathcal{M}_1$, and $\langle G \rangle_{\geq p}$ is a probabilistic safety property $\langle G \rangle_{\geq p}$. Then the following proof rule holds:*

$$\frac{\begin{array}{c} \mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A} \\ \langle \mathcal{A} \rangle \, \mathcal{M}_2 \, \langle G \rangle_{\geq p} \end{array}}{\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq p}} \qquad \text{(Asym-Pios)}$$

Proof: The result follows directly from Definition 5.13. ■

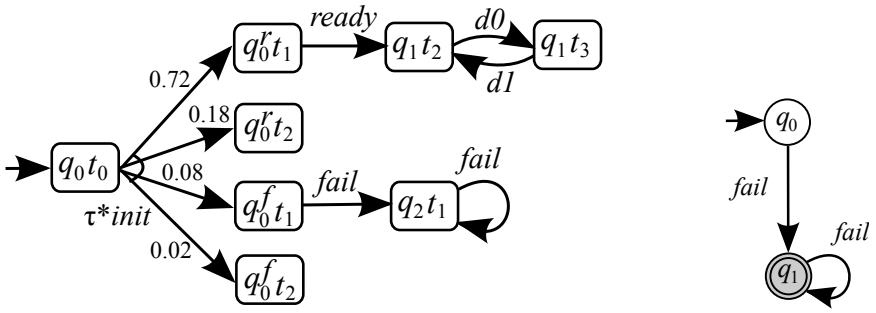


Figure 5.3: $pios(\mathcal{A})\|\mathcal{M}_2$ and a DFA G^{err} for Example 5.16

Therefore, the verification of a probabilistic safety property $\langle G\rangle_{\geq p}$ on $\mathcal{M}_1\|\mathcal{M}_2$ can be decomposed into two separate problems with an appropriate assumption $\mathcal{A}$ about $\mathcal{M}_1$: (1) checking if there is a weak language inclusion relation between $\mathcal{M}_1$ and $\mathcal{A}$, which reduces to (strong) language inclusion on RPAs (Proposition 5.9); (2) checking if $\langle\mathcal{A}\rangle\ \mathcal{M}_2\ \langle G\rangle_{\geq p}$ holds, which can be done by constructing the DTMC $pios(\mathcal{A})\|\mathcal{M}_2$ and verifying $\langle G\rangle_{\geq p}$ on $pios(\mathcal{A})\|\mathcal{M}_2$ (Proposition 5.13).

The compositional verification framework based on rule (Asym-Pios) is *complete* in the sense that, if $\mathcal{M}_1\|\mathcal{M}_2 \models \langle G\rangle_{\geq p}$ is true, we can always find an assumption $\mathcal{A}$ to verify it by simply taking $\mathcal{A}$ to be $rpa(\mathcal{M}_1)$ in the worst case.

Example 5.16 *Consider the pair of PIOSs $\mathcal{M}_1, \mathcal{M}_2$ shown in Figure 5.1 and a probabilistic safety property $\langle G\rangle_{\geq 0.9}$ with the DFA G^{err} shown in Figure 5.3, which means that with probability at least* 0.9 *"fail" never occurs. We use rule (*Asym-Pios*) to verify $\mathcal{M}_1\|\mathcal{M}_2 \models \langle G\rangle_{\geq 0.9}$ compositionally with the assumption $\mathcal{A}$ shown in Figure 5.2. From Example 5.12, we know that the first premise of rule (*Asym-Pios*) is true, i.e. $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A}$. We still need to check the second premise $\langle\mathcal{A}\rangle\ \mathcal{M}_2\ \langle G\rangle_{\geq 0.9}$, which reduces to checking $pios(\mathcal{A})\|\mathcal{M}_2 \models \langle G\rangle_{\geq 0.9}$. Observe from Figure 5.1 that "fail" occurs in $pios(\mathcal{A})\|\mathcal{M}_2$ with probability* 0.08*, thus $\langle G\rangle_{\geq 0.9}$ is satisfied. Since both premises of rule (*Asym-Pios*) are true, we conclude that $\mathcal{M}_1\|\mathcal{M}_2 \models \langle G\rangle_{\geq 0.9}$ holds.*

5.2 Checking Language Inclusion for RPAs

Recall from the previous section that checking the first premise of assume-guarantee rule (Asym-Pios), i.e. $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A}$, reduces to checking language inclusion for RPAs, which is an undecidable problem. Inspired by the algorithm of deciding language equivalence for RPAs [Tze92b], we now propose a method to check if two RPAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are related by language inclusion (i.e. $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$). Our method is a *semi*-algorithm in the sense that, termination is not guaranteed if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ is true; however, if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ does not hold, the algorithm will terminate and produce a lexicographically minimal counterexample word w_c illustrating that $Pr_{\mathcal{A}_1}(w_c) > Pr_{\mathcal{A}_2}(w_c)$.

Given a pair of RPAs $\mathcal{A}_i{=}(S_i, \overline{s}_i, \alpha, \mathbf{P}_i)$ for $i = 1, 2$, we first define vectors $\vec{\iota}_i, \vec{\kappa}_i$ as in Section 5.1.2. More specifically, $\vec{\iota}_i$ is a 0-1 row vector indexed over S_i where $\vec{\iota}_i(s) = 1$ if $s = \overline{s}_i$ and $\vec{\iota}_i(s) = 0$ otherwise; and $\vec{\kappa}_i$ is a column vector over S_i containing all 1s. Similarly to the language equivalence decision algorithm in [Tze92b], our method proceeds by expanding a tree. We denote each node of the tree as $(\vec{v}_1, \vec{v}_2, w)$, where $\vec{v}_i = \vec{\iota}_i \mathbf{P}_i[w]$ is a vector storing probabilities of reaching each state through tracing word w in $\mathcal{A}_i$ for $i = 1, 2$. Thus, the probability of accepting a word w in RPA $\mathcal{A}_i$ can be obtained as $Pr_{\mathcal{A}_i}(w) = \vec{v}_i \vec{\kappa}_i$.

As shown in Algorithm 6, we expand the tree in breadth-first order and keep track of the tree nodes by using a *queue*. During the tree expansion, we also maintain a set V of non-leaf nodes. Initially, both *queue* and V only contain the root node $(\vec{\iota}_1, \vec{\iota}_2, \epsilon)$, where ϵ is the empty word. If *queue* is not empty, we take the head node $(\vec{v}_1, \vec{v}_2, w)$ and consider each of its children nodes $(\vec{v}\,'_1, \vec{v}\,'_2, w')$, where $\vec{v}\,'_1 = \vec{v}_1 \mathbf{P}_1[a]$, $\vec{v}\,'_2 = \vec{v}_2 \mathbf{P}_2[a]$ and $w' = wa$ for all $a \in \alpha$ (Line 2-5). If there is a node $(\vec{v}\,'_1, \vec{v}\,'_2, w')$ satisfing $\vec{v}\,'_1 \vec{\kappa}_1 > \vec{v}\,'_2 \vec{\kappa}_2$, then we can conclude that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ does not hold and terminate the semi-algorithm with a counterexample word $w_c = w'$ (Line 6-7). Otherwise, we need to check if the node $(\vec{v}\,'_1, \vec{v}\,'_2, w')$ can be *pruned* (i.e. making it a leaf node) by using the following two

Algorithm 6 Semi-algorithm of checking language inclusion for RPAs

Input: A pair of RPAs $\mathcal{A}_i$=$(S_i, \overline{s}_i, \alpha, \mathbf{P}_i)$ for $i = 1, 2$
Output: *true* if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$, or a counterexample $w_c \in \alpha^*$ otherwise

1: *queue* := $\{(\vec{\iota}_1, \vec{\iota}_2, \epsilon)\}$, $V := \{(\vec{\iota}_1, \vec{\iota}_2, \epsilon)\}$
2: **while** *queue* is not empty **do**
3: remove $(\vec{v}_1, \vec{v}_2, w)$ from the head of *queue*
4: **for each** $a \in \alpha$ **do**
5: $\vec{v}\,'_1 := \vec{v}_1\mathbf{P}_1[a]$, $\vec{v}\,'_2 := \vec{v}_2\mathbf{P}_2[a]$, $w' := wa$
6: **if** $\vec{v}\,'_1\vec{\kappa}_1 > \vec{v}\,'_2\vec{\kappa}_2$ **then**
7: **return** $w_c = w'$
8: **else if** $(\vec{v}\,'_1, \vec{v}\,'_2, w')$ does not satisfy either (C1) or (C2) **then**
9: add $(\vec{v}\,'_1, \vec{v}\,'_2, w')$ to the tail of *queue*,
10: $V = V \cup \{(\vec{v}\,'_1, \vec{v}\,'_2, w')\}$
11: **end if**
12: **end for**
13: **end while**
14: **return** *true*

pruning criteria:

(C1) $\vec{v}\,'_1\vec{\kappa}_1 = 0$.

(C2) *There exists a set of non-negative rational numbers* λ_i *(for* $0 \leq i < |V|$*), each corresponding to a node* $(\vec{v}_{1,i}, \vec{v}_{2,i}, w_i) \in V$ *such that,*

$$\begin{cases} \vec{v}\,'_1 \leq \sum\limits_{0 \leq i < |V|} \lambda_i \vec{v}_{1,i} \\ \vec{v}\,'_2 \geq \sum\limits_{0 \leq i < |V|} \lambda_i \vec{v}_{2,i} \end{cases}$$

where $\leq$ *and* $\geq$ *denote pointwise comparisons between vectors.*

In practice, (C2) can be easily checked using an SMT solver. If node $(\vec{v}\,'_1, \vec{v}\,'_2, w')$ satisfies either of these two criteria, then it can be pruned as a leaf node; otherwise, we append it to the tail of *queue* and also add it to the non-leaf nodes set V (Line 9-10). The semi-algorithm terminates if no counterexample word can be found when *queue* becomes empty, concluding that $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$.

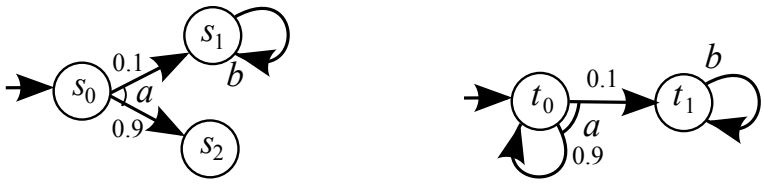


Figure 5.4: A pair of RPAs $\mathcal{A}_1, \mathcal{A}_2$ for Example 5.17

The essential difference between our method and the language equivalence decision algorithm [Tze92b] concerns the pruning criteria. For checking language equivalence, the set of non-leaf nodes can be maintained by calculating the span of the vector space. However, for checking language inclusion, we have to restrict the coefficients of linear combinations of vectors to be non-negative (as stated in (C2)) to prevent the flip of inequality signs.

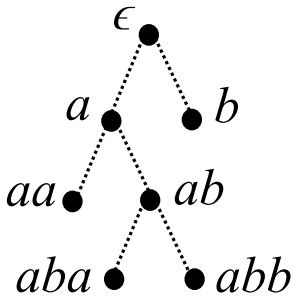
Example 5.17 *Consider the pair of RPAs $\mathcal{A}_1, \mathcal{A}_2$ shown in Figure 5.4; we apply Algorithm 6 to check if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$. For $\mathcal{A}_1$, we have $\vec{\iota}_1 = [1,0,0]$, $\vec{\kappa}_1 = [1,1,1]'$ and*

$$\mathbf{P}_1[a] = \begin{pmatrix} 0 & 0.1 & 0.9 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \mathbf{P}_1[b] = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

For $\mathcal{A}_2$, we have $\vec{\iota}_2 = [1,0]$, $\vec{\kappa}_2 = [1,1]'$ and

$$\mathbf{P}_2[a] = \begin{pmatrix} 0.9 & 0.1 \\ 0 & 0 \end{pmatrix} \quad \mathbf{P}_2[b] = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

Starting with the root node $(\vec{\iota}_1, \vec{\iota}_2, \epsilon)$, we expand the tree with a node for word "a", where $\vec{v}_1[a] = \vec{\iota}_1\mathbf{P}_1[a] = [0, 0.1, 0.9]$ and $\vec{v}_2[a] = \vec{\iota}_2\mathbf{P}_2[a] = [0.9, 0.1]$. Since $\vec{v}_1[a]\vec{\kappa}_1 = \vec{v}_2[a]\vec{\kappa}_2 = 1$, and the node does not satisfy $(C1)$ or $(C2)$, we add it as a non-leaf node to set V and also append it to the tail of queue. Next, we consider the node for word "b", where $\vec{v}_1[b] = \vec{\iota}_1\mathbf{P}_1[b] = [0,0,0]$ and $\vec{v}_2[b] = \vec{\iota}_2\mathbf{P}_2[b] = [0,0]$; this node satisfies $(C1)$



w	$\vec{v}_1$	$\vec{v}_2$	pruned?
ϵ	$[1, 0, 0]$	$[1, 0]$	-
a	$[0, 0.1, 0.9]$	$[0.9, 0.1]$	-
b	$[0, 0, 0]$	$[0, 0]$	(C1)
aa	$[0, 0, 0]$	$[0.81, 0.09]$	(C1)
ab	$[0, 0.1, 0]$	$[0, 0.1]$	-
aba	$[0, 0, 0]$	$[0, 0]$	(C1)
abb	$[0, 0.1, 0]$	$[0, 0.1]$	(C2)

Figure 5.5: Tree nodes for Example 5.17

since $\vec{v}_1[b]\vec{\kappa}_1 = 0$*, so we prune it as a leaf node. The complete process of expanding tree and pruning nodes is shown in Figure 5.5. Note that the node associated with word "abb" is pruned because it satisfies* $(C2)$ *with*

$$\begin{cases} \vec{v}'_1[abb] \leq 0 \cdot \vec{v}'_1[\epsilon] + 0 \cdot \vec{v}'_1[a] + 1 \cdot \vec{v}'_1[ab] \\ \vec{v}'_2[abb] \geq 0 \cdot \vec{v}'_2[\epsilon] + 0 \cdot \vec{v}'_2[a] + 1 \cdot \vec{v}'_2[ab] \end{cases}$$

The queue is now empty. Thus, we terminate the algorithm and conclude that the language inclusion $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ *is true.*

Correctness and termination. We show that the output of Algorithm 6 is correct when it terminates. On one hand, if the output is a counterexample word $w_c \in \alpha^*$, then, based on Line 5-7, $Pr_{\mathcal{A}_1}(w_c) = \vec{\iota}_1 \mathbf{P}_1[w_c]\vec{\kappa}_1 = \vec{v}'_1\vec{\kappa}_1 > \vec{v}'_2\vec{\kappa}_2 = \vec{\iota}_2\mathbf{P}_2[w_c]\vec{\kappa}_2 = Pr_{\mathcal{A}_2}(w_c)$. Thus, according to Definition 5.6, $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ does not hold.

On the other hand, if the output is *true*, we prove that $Pr_{\mathcal{A}_1}(w) \leq Pr_{\mathcal{A}_2}(w)$ for any word $w \in \alpha^*$. We distinguish the following three cases:

1. For any non-leaf node $(\vec{v}_1, \vec{v}_2, w) \in V$, based on Line 10, it is guaranteed that $Pr_{A_1}(w) = \vec{v}_1\vec{\kappa}_1 \leq \vec{v}_2\vec{\kappa}_2 = Pr_{A_2}(w)$.

2. For any leaf node $(\vec{v}_1, \vec{v}_2, w)$ satisfying (C1), we have $Pr_{\mathcal{A}_1}(w) = \vec{v}_1\vec{\kappa}_1 = 0$. Thus, $Pr_{\mathcal{A}_1}(w) \leq Pr_{\mathcal{A}_2}(w)$ for any probability value $Pr_{\mathcal{A}_2}(w)$. This result also extends

to any word $\hat{w} = wu$ with a suffix $u \in \alpha^*$, because $Pr_{\mathcal{A}_1}(\hat{w}) = \vec{v}_1\mathbf{P}[u]\vec{\kappa}_1 = 0$.

3. For any leaf node $(\vec{v}_1, \vec{v}_2, w)$ satisfying (C2), we prove that $Pr_{\mathcal{A}_1}(w) \leq Pr_{\mathcal{A}_2}(w)$ by Lemma 5.18. For any finite word $\hat{w} = wu$ with a suffix $u \in \alpha^*$, we prove in Lemma 5.19 that its corresponding node $(\vec{v}_1\mathbf{P}[u], \vec{v}_2\mathbf{P}[u], wu)$ also satisfies (C2). Thus, we have $Pr_{\mathcal{A}_1}(\hat{w}) \leq Pr_{\mathcal{A}_2}(\hat{w})$.

Therefore, when Algorithm 6 outputs *true*, $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ holds.

Lemma 5.18 *If a node* $(\vec{v}_1, \vec{v}_2, w)$ *satisfies* (C2)*, then* $Pr_{\mathcal{A}_1}(w) \leq Pr_{\mathcal{A}_2}(w)$.

Proof: Based on (C2), we have

$$Pr_{\mathcal{A}_1}(w) = \vec{v}_1\vec{\kappa}_1 \leq \sum_{0 \leq i < |V|} \lambda_i \vec{v}_{1,i}\vec{\kappa}_1 = \sum_{0 \leq i < |V|} \lambda_i Pr_{\mathcal{A}_1}(w_i),$$

and

$$Pr_{\mathcal{A}_2}(w) = \vec{v}_2\vec{\kappa}_2 \geq \sum_{0 \leq i < |V|} \lambda_i \vec{v}_{2,i}\vec{\kappa}_2 = \sum_{0 \leq i < |V|} \lambda_i Pr_{\mathcal{A}_2}(w_i),$$

where $(\vec{v}_{1,i}, \vec{v}_{2,i}, w_i) \in V$. Given that $Pr_{\mathcal{A}_1}(w_i) \leq Pr_{\mathcal{A}_2}(w_i)$ for all $0 \leq i < |V|$, thus

$$Pr_{\mathcal{A}_1}(w) \leq \sum_{0 \leq i < |V|} \lambda_i Pr_{\mathcal{A}_1}(w_i) \leq \sum_{0 \leq i < |V|} \lambda_i Pr_{\mathcal{A}_2}(w_i) \leq Pr_{\mathcal{A}_2}(w).$$

■

Lemma 5.19 *When Algorithm 6 terminates with* true*, if a node* $(\vec{v}_1, \vec{v}_2, w)$ *satisfies* (C2)*, then* $(\vec{v}_1\mathbf{P}_1[u], \vec{v}_2\mathbf{P}_2[u], wu)$ *also satisfies* (C2) *for any* $u \in \alpha^*$.

Proof: Let $V = \{(\vec{v}_{1,i}, \vec{v}_{2,i}, w_i)\}$ be the set of non-leaf nodes. When Algorithm 6 terminates with *true*, any child of a non-leaf node is either a non-leaf node, or a leaf node satisfying (C1) or (C2). If the child is a non-leaf node then it satisfies (C2),

because we can always find a set of non-negative rational numbers for it (by letting the number corresponding to the node be 1 and others be 0). If the child node satisfies (C1), then it also satisfies (C2) with the set of rational coefficients as all 0s. Thus, any child of a non-leaf node $(\vec{v}_{1,i}, \vec{v}_{2,i}, w_i)$ satisfies (C2); that is, for any $a \in \alpha$, there exist a set of non-negative rational numbers ξ_{ij} such that

$$\begin{cases} \vec{v}_{1,i}\mathbf{P}_1[a] \leq \sum\limits_{0 \leq j < |V|} \xi_{ij}\vec{v}_{1,j} \\ \vec{v}_{2,i}\mathbf{P}_2[a] \geq \sum\limits_{0 \leq j < |V|} \xi_{ij}\vec{v}_{2,j} \end{cases} \tag{†1}$$

We are about to prove by induction on the length of $|u|$ that $(\vec{v}_1\mathbf{P}_1[u], \vec{v}_2\mathbf{P}_2[u], wu)$ satisfies (C2) for any $u \in \alpha^*$.

Base case: $|u| = 0$, it is known that $(\vec{v}_1, \vec{v}_2, w)$ satisfies (C2).

Induction step: $|u| \geq 1$. Suppose that $(\vec{v}_1\mathbf{P}_1[u], \vec{v}_2\mathbf{P}_2[u], wu)$ satisfies (C2) for some finite word u. Then, there exist a set of rational numbers λ_i such that

$$\begin{cases} \vec{v}_1\mathbf{P}_1[u] \leq \sum\limits_{0 \leq i < |V|} \lambda_i\vec{v}_{1,i} \\ \vec{v}_2\mathbf{P}_2[u] \geq \sum\limits_{0 \leq i < |V|} \lambda_i\vec{v}_{2,i} \end{cases} \tag{†2}$$

For any $a \in \alpha$, we have

$$\begin{aligned} \vec{v}_1\mathbf{P}_1[ua] &= \vec{v}_1\mathbf{P}_1[u]\mathbf{P}_1[a] \\ &\overset{(\dagger 2)}{\leq} \sum_{0 \leq i < |V|} \lambda_i\vec{v}_{1,i}\mathbf{P}_1[a] \\ &\overset{(\dagger 1)}{\leq} \sum_{0 \leq i < |V|} \lambda_i \sum_{0 \leq j < |V|} \xi_{ij}\vec{v}_{1,j} \\ &= \sum_{0 \leq j < |V|} \big(\sum_{0 \leq i < |V|} \lambda_i\xi_{ij} \big)\vec{v}_{1,j} \end{aligned}$$

Let $\zeta_j = \sum_{0 \leq i < |V|} \lambda_i \xi_{ij}$, which is a set of nonnegative rational numbers. We get

$$\vec{v}_1 \mathbf{P}_1[ua] \leq \sum_{0 \leq i < |V|} \zeta_j \vec{v}_{1,j}$$

Symmetrically, we can deduce that

$$\vec{v}_2 \mathbf{P}_2[ua] \geq \sum_{0 \leq i < |V|} \zeta_j \vec{v}_{2,j}$$

Therefore, $(\vec{v}_1 \mathbf{P}_1[ua], \vec{v}_2 \mathbf{P}_2[ua], wua)$ satisfies (C2) for all $a \in \alpha$.

■

Now, we discuss the termination of Algorithm 6. If $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ does not hold, then, by Definition 5.6, there must exist at least one word w_c such that $Pr_{\mathcal{A}_1}(w_c) > Pr_{A_2}(w_c)$. We can always reach node $(\vec{v}_1, \vec{v}_2, w_c)$ by expanding the tree, and hence terminate the algorithm (Line 7). Moreover, since we traverse the tree following a breadth-first fashion, w_c is the lexicographically minimum counterexample word. On the other hand, if $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ is true, then termination is not guaranteed (which is acceptable because of the undecidability of the underlying problem). Potentially, we may keep discovering nodes that do not satisfy either (C1) or (C2), and add them to *queue* (Line 8-9); thus, *queue* may never be empty and Algorithm 6 will not terminate.

5.3 L*-style Learning for RPAs

Recall from Section 5.1 that we represent assumptions about PIOSs with a specific class of RPAs in which all states are accepting. To enable the automated generation of such assumptions, we propose an active learning algorithm that learns RPAs (with all states accepting) for a target stochastic language (corresponding to the trace probabilities of PIOSs). Our method is inspired by [BV96] for learning multiplicity automata and

Algorithm 7 L*-style learning algorithm for RPAs

Input: alphabet α, a teacher who knows the target stochastic language $\mathcal{L}$
Output: RPA $\mathcal{A}$

initialise (U, V, T): let $U = V = \{\epsilon\}$, ask membership queries and fill $T(w)$ for all words $w \in (U \cup U \cdot \alpha) \cdot V$
repeat
  while (U, V, T) is not RPA-closed or not RPA-consistent **do**
    if (U, V, T) is not RPA-closed **then**
      find $u \in U, a \in \alpha$ that make (U, V, T) not RPA-closed,
      add ua to U,
      extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
    end if
    if (U, V, T) is not RPA-consistent **then**
      find $a \in \alpha, v \in V$ that make (U, V, T) not RPA-consistent,
      add av to V,
      extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
    end if
  end while
  construct a conjectured RPA $\mathcal{A}$ and ask an equivalence query
  if a counterexample c is provided **then**
    add c and all its prefixes to U,
    extend T for all words $w \in (U \cup U \cdot \alpha) \cdot V$ using membership queries
  else
    the correct RPA $\mathcal{A}$ has been learnt such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
  end if
until $\mathcal{L}(\mathcal{A}) = \mathcal{L}$
return $\mathcal{A}$

works in a similar style to the L* algorithm described in Section 3.5.1. It constructs an *observation table* (of acceptance probabilities for each finite word) and asks a teacher two types of queries: *membership queries* about the probability of certain finite words in the target stochastic language; and *equivalence queries* regarding whether a conjectured RPA accepts exactly the target language.

Algorithm 7 shows the details of our method. Similarly to the L* algorithm, it builds an observation table (U, V, T), where U is a prefix-closed set of finite words, V is a suffix-closed set of finite words, and $T : ((U \cup U \cdot \alpha) \cdot V) \to [0, 1]$ is a function mapping finite words to probabilities. The rows of the observation table (U, V, T) are labelled

by elements in the prefix set $(U \cup U{\cdot}\alpha)$ and the columns are labelled by elements in the suffix set V. Each entry at row u and column v in the table has a value $T(uv)$, corresponding to the probability of accepting word uv in the target stochastic language. Recall that, in the L* algorithm, $T(uv)$ is a binary result indicating whether or not a word is accepted by the target regular language. For each $u \in (U \cup U{\cdot}\alpha)$, we define a row vector $\vec{v}[u]$ of length $|V|$ such that $\vec{v}[u](v) = T(uv)$. As we will show later, each state in the learnt RPA corresponds to at least one row vector $\vec{v}[u]$.

Recall that L* maintains a *closed* and *consistent* observation table. Analogously, we define the notions of *RPA-closed* and *RPA-consistent* below. Note that the notions used in L* can be considered as a special case of our definitions, where the observation table is filled with 1s and 0s only.

Definition 5.20 (RPA-closed) *Observation table* (U, V, T) *is* RPA-closed *if, for any* $u \in U, a \in \alpha$*, there exists a set of non-negative rational numbers* λ_i *such that*

$$\vec{v}[ua] = \sum_{u_i \in U} \lambda_i \vec{v}[u_i].$$

Intuitively, an observation table is RPA-closed if any row vector $\vec{v}[ua]$ can be represented as a *conical combination* of vectors in the set $\{\vec{v}[u] \mid u \in U\}$. Note that we restrict the linear coefficients to be non-negative, because they are related to transition probabilities in the learnt RPA, which we will show later. In [BV96], a similar condition without this restriction is used for learning multiplicity automata, where transitions labelled with negative numbers are allowed.

Definition 5.21 (RPA-consistent) *Observation table* (U, V, T) *is* RPA-consistent *if, given any rational number* ξ_i*,*

$$\sum_{u_i \in U} \xi_i \vec{v}[u_i] = \vec{0} \implies (\forall a \in \alpha.) \sum_{u_i \in U} \xi_i \vec{v}[u_i a] = \vec{0}.$$

The intuition is that each row vector in the observation table corresponds to a RPA state, and the linear dependencies between vectors (states) should be consistent after taking a transition with action a.

As shown in Algorithm 7, the observation table is initialised with $U = V = \{\epsilon\}$. We assume $T(\epsilon) = 1$, since all states of the learnt RPA are accepting. The algorithm keeps filling entries of the observation table (U, V, T) by asking membership queries and maintaining the RPA-closed and RPA-consistent conditions. If it finds a pair of $u \in U, a \in \alpha$ which make (U, V, T) not RPA-closed, i.e. not satisfying the condition in Definition 5.20, it adds the word ua to the prefix set U; similarly, if it finds a pair of $a \in \alpha, v \in V$ that make (U, V, T) not RPA-consistent, i.e. violating the condition in Definition 5.21, it adds the word av to the suffix set V. Once (U, V, T) satisfies both conditions, the algorithm constructs a RPA $\mathcal{A}$ and asks an equivalence query for it. If the teacher answers "yes", then the algorithm terminates and outputs $\mathcal{A}$; otherwise, the teacher provides a counterexample word c indicating that the target language $\mathcal{L}$ accepts c with a different probability from $Pr_{\mathcal{A}}(c)$. In the latter case, the algorithm adds c and all its prefixes to U, and continues to update the observation table.

Now we describe how to construct a RPA $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P}, F)$ based on a RPA-closed and RPA-consistent observation table (U, V, T). Firstly, we determine a minimal set $U' \subseteq U$ such that, for all $u \in U$, the vector $\vec{v}[u]$ can be represented as a conical combination of vectors in the set $\{\vec{v}[u'] \mid u' \in U'\}$. The states set S of RPA $\mathcal{A}$ is fixed by U', in the sense that there is a one-to-one mapping between state $s \in S$ and vector $\vec{v}[u']$ for $u' \in U'$. The initial state $\overline{s}$ corresponds to vector $\vec{v}[\epsilon]$. The transition matrix $\mathbf{P}[a]$ for $a \in \alpha$ is given by

$$\mathbf{P}[a](i, j) = \lambda_j \cdot \frac{T(u_j)}{T(u_i)}$$

where the set of non-negative rational numbers λ_j are obtained by solving the following

linear equation:

$$\vec{v}[u_i a] = \sum_{u_j \in U'} \lambda_j \vec{v}[u_j]$$

The set F of accepting states is defined for row vectors with $\vec{v}[u](\epsilon) \neq 0$.

Lemma 5.22 *Suppose (U, V, T) is a RPA-closed and RPA-consistent observation table. The conjecture $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P}, F)$ based on (U, V, T) is a well-defined RPA with all states accepting and satisfies $Pr_{\mathcal{A}}(uv) = T(uv)$ for all $u \in (U \cup U{\cdot}\alpha), v \in V$.*

Proof: We first show that $\mathcal{A} = (S, \overline{s}, \alpha, \mathbf{P}, F)$ is a well-defined RPA. The states set S is well-defined because (U, V, T) is RPA-closed and, based on Definition 5.20, we can always find such a conical set U'. Since $T(\epsilon) = 1$, the vector $\vec{v}[\epsilon]$ is an extreme point in the conical hull and will always be included in the set U'; hence, the initial state $\overline{s}$ is well-defined. Now we show that the transition matrices $\mathbf{P}$ are well-defined, i.e. , $\sum_j \mathbf{P}[a](i, j) \in [0, 1]$ for all $a \in \alpha$. We have

$$\sum_j \mathbf{P}[a](i,j) \overset{(\dagger 1)}{=} \sum_j \lambda_j \cdot \frac{T(u_j)}{T(u_i)} \overset{(\dagger 2)}{=} \sum_j \lambda_j \cdot \frac{\vec{v}[u_j](\epsilon)}{\vec{v}[u_i](\epsilon)}$$
$$\overset{(\dagger 3)}{=} \frac{\vec{v}[u_i a](\epsilon)}{\vec{v}[u_i](\epsilon)} \overset{(\dagger 4)}{=} \frac{T(u_i a)}{T(u_i)} \overset{(\dagger 5)}{\in} [0, 1]$$

(†1) and (†3) are by definition of $\mathbf{P}[a](i, j)$. (†2) and (†4) are due to $T(u) = \vec{v}[u](\epsilon)$ for any word u. Finally, (†5) is because the probability of accepting word $u_i a$ should be no more than accepting its prefix u_i. The set of accepting states is well-defined with $F = S$. This is because, for any non-empty word u, if $\vec{v}[u](\epsilon) = 0$ then $\vec{v}[u] = \vec{0}$ (i.e. if $T(u) = 0$ then $T(uv) = 0$ with any suffix v), which always satisfies Definition 5.20 and hence u would never be added into U.

In the following, we show that $Pr_{\mathcal{A}}(uv) = T(uv)$ for all $u \in (U \cup U{\cdot}\alpha)$ and $v \in V$.

$$
\begin{aligned}
Pr_{\mathcal{A}}(uv) &\overset{(\ddagger 1)}{=} \vec{\iota}\mathbf{P}(uv)\vec{\kappa} \overset{(\ddagger 2)}{=} \sum_{j} \mathbf{P}(uv)[0,j] \\
&\overset{(\ddagger 3)}{=} \sum_{j}\sum_{k} \mathbf{P}(u)[0,k]\mathbf{P}(v)[k,j] \\
&\overset{(\ddagger 4)}{=} \sum_{j}\sum_{k} \lambda^{u}_{0,k}\frac{T(u_k)}{T(\epsilon)}\lambda^{v}_{k,j}\frac{T(u_j)}{T(u_k)} \\
&\overset{(\ddagger 5)}{=} \sum_{k} \lambda^{u}_{0,k}\big(\sum_{j} \lambda^{v}_{k,j}T(u_j)\big) \\
&\overset{(\ddagger 6)}{=} \sum_{k} \lambda^{u}_{0,k}T(u_k \cdot v) \overset{(\ddagger 7)}{=} T(\epsilon \cdot uv) \;=\; T(uv).
\end{aligned}
$$

($\ddagger$1) is by definition of $Pr_{\mathcal{A}}(uv)$. ($\ddagger$2) is because we assume $\vec{\iota} = [1, 0, \cdots, 0]$ for simplicity, and $\vec{\kappa}$ is a column vector of 1s given that all states of $\mathcal{A}$ are accepting. ($\ddagger$3) is due to matrix multiplication. ($\ddagger$4) is by definition of transition matrices $\mathbf{P}[u]$ and $\mathbf{P}[v]$. ($\ddagger$5) is because $T(\epsilon) = 1$. ($\ddagger$6) and ($\ddagger$7) are due to $\vec{v}[u_k \cdot v] = \sum_j \lambda^{v}_{k,j}\vec{v}[u_j]$, and $\vec{v}(u) = \sum_k \lambda^{u}_{0,k}\vec{v}(u_k)$, respectively.

■

Correctness and termination. When Algorithm 7 terminates, its output is correct because the result of equivalence query guarantees that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$ (Line 20), i.e. the learnt RPA $\mathcal{A}$ accepts the target stochastic language $\mathcal{L}$.

Nevertheless, we don't know whether Algorithm 7 terminates. The basis of the termination proof for the L* algorithm [Ang87a] is that, for any regular language, there exists a unique minimal accepting DFA. But an analogous property does not exist for arbitrary stochastic language and RPAs. According to [BV96], a smallest multiplicity automaton (i.e. weighted automaton with rational transition weights) *can* be learnt for a given stochastic language. However, converting a multiplicity automaton to a RPA with all states accepting is not always possible.

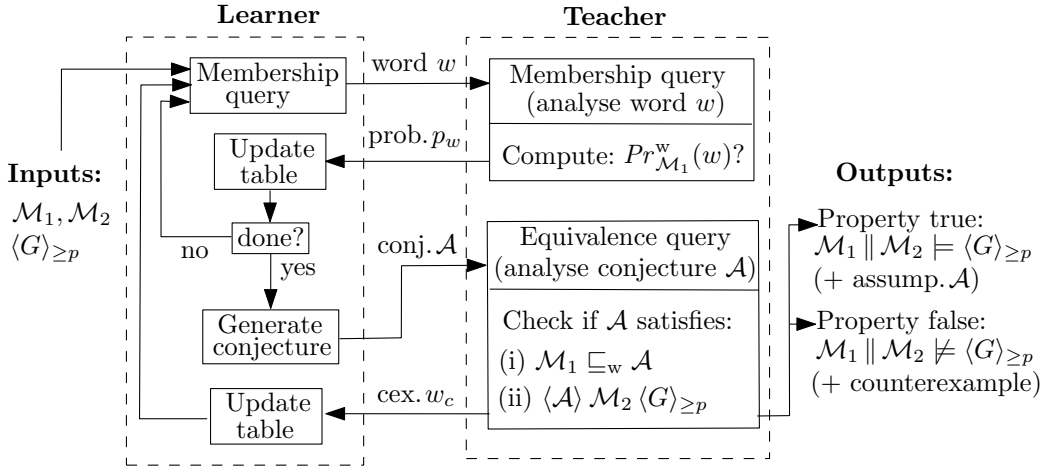


Figure 5.6: Learning probabilistic assumptions for rule (Asym-Pios)

5.4 Learning Assumptions for Rule (Asym-Pios)

We build a fully-automated implementation of the (complete) compositional verification framework proposed in Section 5.1, which aims to verify (or refute) $\mathcal{M}_1 \parallel \mathcal{M}_2 \models \langle G \rangle_{\geq p}$ for two PIOSs $\mathcal{M}_1, \mathcal{M}_2$ and a probabilistic safety property $\langle G \rangle_{\geq p}$. This is done using assume-guarantee rule (Asym-Pios) (Theorem 5.15) with the required RPA assumption $\mathcal{A}$ about component $\mathcal{M}_1$ being generated through learning.

Figure 5.6 summarises the overall structure of our approach: the left-hand side shows a *learner*, which is the active learning algorithm for RPAs proposed in Section 5.3; and the right-hand side shows a *teacher* who answers membership and equivalence queries asked by the learner. This structure is very similar to the approach presented in Section 4.3, where the L* algorithm is employed to learn assumptions for rule (Asym). However, here we use a completely different learner and teacher.

To answer membership queries (i.e. the probability of accepting a certain finite word w in the assumption $\mathcal{A}$), we use a probabilistic model checker to simulate the teacher, which computes $Pr^{\mathrm{w}}_{\mathcal{M}_1}(w)$. Based on Definition 5.8, to satisfy the first premise of rule (Asym-Pios), i.e. $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A}$, we need to have $Pr^{\mathrm{w}}_{\mathcal{M}_1}(w) \leq Pr_{\mathcal{A}}(w)$ for all $w \in \alpha^*$.

Thus, we can use the value of $Pr^{\mathrm{w}}_{\mathcal{M}_1}(w)$ as the weakest answer to membership queries.

Answering equivalence queries (i.e. whether a conjectured RPA $\mathcal{A}$ is an appropriate assumption for rule (Asym-Pios)) involves two steps, each of which corresponds to a premise of rule (Asym-Pios). Firstly, to check if $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} A$ is true, based on Proposition 5.9, we reduce the problem to checking language inclusion between two RPAs $rpa(\mathcal{M}_1)$ and $\mathcal{A}^{\tau}$, and then applying the semi-algorithm proposed in Section 5.2. Secondly, to check whether the assume-guarantee triple $\langle \mathcal{A} \rangle\, \mathcal{M}_2\, \langle G \rangle_{\geq p}$ holds, based on Proposition 5.14, we verify $pios(\mathcal{A}) \parallel \mathcal{M}_2 \models \langle G \rangle_{\geq p}$ instead, by using standard techniques of model checking probabilistic safety properties on DTMCs (see Section 3.3.2).

If both premises are true, we can conclude that $\mathcal{M}_1 \parallel \mathcal{M}_2$ satisfies property $\langle G \rangle_{\geq p}$ and $\mathcal{A}$ is an appropriate assumption to verify it with rule (Asym-Pios). However, if any premise fails, the teacher needs to return a counterexample word w_c to the learning algorithm. If the first premise $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} A$ fails, we can obtain a counterexample word w_c from the (weak) language inclusion check as described in Section 5.2 such that $Pr^{\mathrm{w}}_{\mathcal{M}_1}(w_c) > Pr_{\mathcal{A}}(w_c)$. If the second premise $\langle \mathcal{A} \rangle\, \mathcal{M}_2\, \langle G \rangle_{\geq p}$ fails, i.e. $pios(\mathcal{A}) \parallel \mathcal{M}_2 \not\models \langle G \rangle_{\geq p}$, a probabilistic counterexample C containing a *set* of finite paths is generated using the counterexample generation techniques for DTMCs (see Section 3.4). Given a path $\rho \in C$, we can obtain a τ-free word $\bar{w} = tr(\rho\restriction_{pios(A)})$ in A, which may correspond to a set of words $W_\rho = \{w \mid st(w) = \bar{w}\}$ in $\mathcal{M}_1$; thus, based on C, we can construct a (small) fragment of $\mathcal{M}_1$, denoted $\mathcal{M}_1^C$, by removing transitions in $\mathcal{M}_1$ that do not appear in any path corresponding to word $w \in W_\rho$ for all $\rho \in C$. If $\mathcal{M}_1^C \| \mathcal{M}_2 \not\models \langle G \rangle_{\geq p}$, then we conclude that $\mathcal{M}_1 \| \mathcal{M}_2 \not\models \langle G \rangle_{\geq p}$ and terminate the learning. Otherwise, C is a spurious counterexample, i.e. $Pr^{\mathrm{w}}_{\mathcal{M}_1}(w_c) < Pr_{\mathcal{A}}(w_c)$ for some word w_c in $\mathcal{A}$ that corresponds to a path $\rho \in C$, so the teacher returns w_c as a counterexample to the learning algorithm to refine $\mathcal{A}$.

Correctness and termination. When the learning loop terminates, the correctness of output $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq p}$ or $\mathcal{M}_1 \| \mathcal{M}_2 \not\models \langle G \rangle_{\geq p}$ is guaranteed by the assume-guarantee

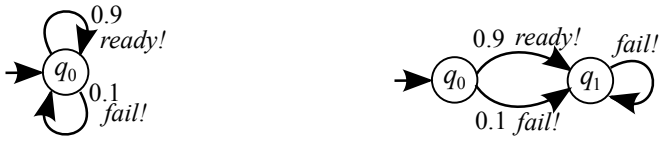


Figure 5.7: Two learnt RPAs $\mathcal{A}_1, \mathcal{A}_2$ for Example 5.23

rule (Asym-Pios). However, since this approach is driven by the RPA learning algorithm of Section 5.3 whose termination we cannot prove due to the learnability of stochastic languages, we cannot guarantee that the loop always terminates. Moreover, this approach uses the semi-algorithm of Section 5.2 to perform weak language inclusion checks, which also does not guarantee termination.

$\mathcal{T}_1$	ϵ
ϵ	1
$fail!$	0.1
$ready!$	0.9
$d0?$	0
$d1?$	0

$\mathcal{T}_2$	ϵ
ϵ	1
$fail!$	0.1
$fail!, fail!$	0.1
$ready!$	0.9
$d0?$	0
$d1?$	0
$fail!, ready!$	0
$fail!, d0?$	0
$fail!, d1?$	0
$fail!, fail!, fail!$	0.1
$fail!, fail!, ready!$	0
$fail!, fail!, d0?$	0
$fail!, fail!, d1?$	0

Figure 5.8: Observation tables corresponding to RPAs $\mathcal{A}_1, \mathcal{A}_2$ in Figure 5.7

Example 5.23 *Consider the compositional verification of $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.9}$ shown in Example 5.16. We show the process of learning an assumption $\mathcal{A}$ to verify the above specification using the learning approach in Figure 5.6.*

Given an alphabet $\alpha_{\mathcal{A}} = \{fail!, ready!, d0?, d1?\}$, the first RPA-closed and RPA-consistent observation table obtained by the RPA learning algorithm (Section 5.3) is $\mathcal{T}_1$ shown in Figure 5.8. Each entry of $\mathcal{T}_1$ is filled with the probability of accepting the corresponding τ-free word in $\mathcal{M}_1$. A conjectured RPA $\mathcal{A}_1$ is built from $\mathcal{T}_1$,

shown in Figure 5.7, and an equivalence query is asked for it. The teacher provides a counterexample word $c_1 = \langle fail!, fail! \rangle$ *indicating that* $\mathcal{M}_1 \not\sqsubseteq_{\mathrm{w}} \mathcal{A}_1$*, because* $Pr^{\mathrm{w}}_{\mathcal{M}_1}(c_1) = 0.1 > 0.01 = Pr_{\mathcal{A}_1}(c_1)$.

The learning algorithm updates the observation table by adding c_1 *and its prefix to* U*. A second RPA-closed and RPA-consistent observation table is obtained as* $\mathcal{T}_2$ *in Figure 5.8, which corresponds to* $\mathcal{A}_2$ *shown in Figure 5.7. This time the teacher finds a counterexample* $c_2 = \langle ready!, d0? \rangle$*, because* $Pr^{\mathrm{w}}_{\mathcal{M}_1}(c_2) = 0.9 > 0 = Pr_{\mathcal{A}_2}(c_2)$.

The learning algorithm continues to update the observation table with c_2*. A new RPA is built, which corresponds to* $\mathcal{A}$ *in Figure 5.2. As described in Example 5.16,* $\mathcal{A}$ *is a good assumption that can be used in rule (*Asym-Pios*) to verify* $\mathcal{M}_1 \| \mathcal{M}_2 \models \langle G \rangle_{\geq 0.9}$ *compositionally. Thus, the learning loop terminates.*

5.5 Implementation and Case Studies

We have built a prototype tool which implements the fully-automated compositional verification and assumption generation approach described in Section 5.4. The tool also implements the semi-algorithm for checking language inclusion for RPAs (Section 5.2) and the L*-style active learning algorithm for RPAs (Section 5.3). Our prototype uses the probabilistic model checker PRISM [KNP11] to answer membership queries and check probabilistic assume-guarantee triples (Definition 5.13) during the assumption learning procedure. It also uses the SMT solver Yices 2 (http://yices.csl.sri.com/) to solve the linear arithmetic problems that arise in the language inclusion check and in RPA learning. Experiments were run on a 2.80GHz PC with 32GB RAM running 64-bit Fedora for the following benchmark case studies.

Contract Signing (egl). This case study is modified from the contract signing protocol of Even, Goldreich and Lempel [EGL85]. It considers the problem of exchanging data fairly between two parties A and B over a network, i.e. if A has received all data

from B, then B should be able to obtain all data from A regardless of A's action, and vice versa. A trusted third party named *counter* is involved in the communication between A and B, facilitating the simultaneity of data exchange. In this case study, we focus on the scenario when two parties are exchanging N pairs of secrets and each secret contains L bits. The data exchange involves two phases: in the first phase, the parties use oblivious transfer to probabilistically reveal one secret from each pair; then, in the second phase, they release all secrets bit by bit. A party is considered committed if the opponent knows both secrets in at least one pair. To enable compositional verification using assume-guarantee rule (Asym-Pios), we model the behaviour of the counter as a PIOS $\mathcal{M}_1$ and the behaviour of parties A, B together as the other PIOS $\mathcal{M}_2$. We ask for the maximum probability that "A is committed while B is not".

Bounded Retransmission Protocol (brp). Originally proposed by Helmink, Sellink and Vaandrager [HSV94], this protocol aims to communicate messages over unreliable channels. It sends a file in N chunks, but allows only a bounded (Max) number of retransmissions of each chunk. We developed two case studies based on it. The first one, labelled as *brp-1* in Figure 5.9, models the sender module as a PIOS $\mathcal{M}_1$ and the receiver as the other PIOS $\mathcal{M}_2$. The second one, labelled as *brp-2*, models the receiver as $\mathcal{M}_1$ and the sender as $\mathcal{M}_2$. In both cases, we verify the maximum probability of violating the safety property that "no more than one file should be sent".

Client-Server Model (cs). This case study is a variant of the model from [PGB+08]), where a server is managing the resources for N clients. Each of the clients may request the use of a certain resource, and the server would grant the request if the resource is available, and would deny it if some other client is currently using the resource. However, with a small probability, the server may behave incorrectly and cause the resource conflict between clients. We model the server as a PIOS $\mathcal{M}_1$ and the clients organised via a round robin scheduling as the other PIOS $\mathcal{M}_2$. We check the probability

of the mutual exclusion property being violated.

Results

Figure 5.9 shows the experimental results. For each case study, we report the number of states of components $|\mathcal{M}_1|$ and $|\mathcal{M}_2 \otimes G^{err}|$, the size of learning assumption $|\mathcal{A}|$ and converted PIOS $pios(\mathcal{A})$. To illustrate the efficiency of our assumption learning approach, we also show the number of requested equivalence queries, $|EQ|$, which corresponds to the number of iterations in the learning loop (Figure 5.6). Finally, we compare the model checking results and total run-time of our learning-based compositional verification approach with the non-compositional verification by PRISM (using the sparse engine).

We can see from Figure 5.9 that, in all but one case, our approach successfully learns assumptions that are considerably smaller than the components (even after converting to PIOSs). We also observe that the model checking results obtained from the compositional verification are identical to the non-compositional verification. The above two insights illustrate that our approach has the advantage of learning assumptions of good quality (i.e. compact representation and sufficient for model checking).

However, in the brp-2 (4,4) model, the learning loop does not terminate. Investigations show that this is due to round-off errors in the numerical computation performed by PRISM being converted to rationals for the SMT solver. We plan to investigate the use of arbitrary precision arithmetic (i.e. the precision of numbers' digits are limited only by the available memory of the host system) to alleviate this problem. Currently our implementation is a prototype and (unsurprisingly) results in slower run-times than non-compositional verification using (highly-optimised) PRISM.

Case study [parameters]			Component sizes		Compositional					Non-compositional	
			$\lvert\mathcal{M}_2 \otimes G^{err}\rvert$	$\lvert\mathcal{M}_1\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert pios(\mathcal{A})\rvert$	$\lvert EQ\rvert$	Result	Time (s)	Result	Time (s)
egl [*N L*]	5	8	1,274,641	361	**20**	31	5	0.386718	29.7	0.386718	2.47
	6	8	8,265,625	433	**24**	37	5	0.514648	39.8	0.514648	5.39
	7	8	48,399,849	505	**28**	43	5	0.015380	58.6	0.015380	5.58
	8	2	31,573,161	193	**32**	49	5	0.503845	67.6	0.503845	4.24
brp-1 [*N Max*]	16	5	642	191	**2**	3	2	1.121E-8	1.3	1.121E-8	0.50
	32	5	1,282	191	**2**	3	2	2.241E-8	1.5	2.241E-8	0.92
	64	5	2,562	191	**2**	3	2	4.482E-8	1.6	4.482E-8	1.79
brp-2 [*N Max*]	2	2	764	50	**19**	25	9	1.792E-2	55.5	1.792E-2	0.04
	2	3	764	63	**39**	57	7	3.740E-3	172.2	3.740E-3	0.05
	4	4	764	146	-	-	-	-	-	1.556E-3	0.17
client-server [*N*]	2		72	274	**17**	17	10	0.079999	61.1	0.079999	0.019
	3		372	416	**31**	31	14	0.079999	325.9	0.079999	0.118
	4		1,728	562	**49**	49	37	0.079999	3,155.5	0.079999	0.247

Figure 5.9: Performance of the learning-based compositional verification using rule (ASYM-PIOS)

5.6 Summary and Discussion

In this chapter, we presented a novel (complete) assume-guarantee framework in which system components are modelled as probabilistic I/O systems (PIOSs) and assumptions are Rabin probabilistic automata (RPAs). We also developed new techniques for checking RPA language inclusion and learning RPAs from stochastic languages, which enable a fully-automated approach of learning assumptions and performing compositional verification based on the proposed assume-guarantee framework. A prototype tool is implemented and experiments on a set of benchmark case studies show that our approach is capable of learning assumptions that are compact yet still sufficient for producing exact model checking results.

A weakness of our approach is that it may not terminate, which is due to the semi-algorithm for checking language inclusion of RPAs and the (non-terminating) active learning algorithm for RPAs. In future, we may investigate the termination conditions for these algorithms to address this weakness. One possibility is to learn multiplicity automata instead of RPAs as assumptions. Given a stochastic language, there is a polynomial algorithm [BV96] to learn a smallest multiplicity automata using membership and equivalence queries. However, more investigations are needed to establish the connection between a PIOS component and a multiplicity automaton assumption whose transition weights may be negative rational numbers.

The approach presented in this chapter is methodologically similar to the approach proposed in Chapter 4, in the sense that they both learn assumptions via membership and equivalence queries. However, they contrast each other in a few technical aspects. Firstly, they are based on different assume-guarantee frameworks, one is for asynchronous systems while the other is for synchronous systems. Secondly, they adopt different learning algorithms and their learnt assumptions are in different formats (DFA vs. RPA). Thirdly, their implementation of the teacher for answering membership and

equivalence queries are also different: the approach in Chapter 4 uses a model checker as a teacher, while the approach in this chapter requires the language inclusion check.

Chapter 6

Learning Implicit Assumptions

Motivated by the work of [CCF+10], which learns assumptions encoded implicitly as Boolean functions for non-probabilistic compositional verification, we develop a novel approach for learning implicit assumptions for probabilistic systems modelled as DTMCs and present the approach in this chapter.

Similarly to [CCF+10], we use the CDNF algorithm [Bsh95] to automatically learn assumptions represented as Boolean functions. The target function of the learning algorithm is a DTMC system component represented as a Boolean formula over a set of variables encoding its state space and transition probabilities. In order to obtain more succinct assumptions about DTMC components, we need to abstract the behaviour of components by reducing the number of states or transitions. Recall from Section 3.5.3 that the CDNF algorithm learns a representation of the target function in *conjunctive disjunctive normal form* (CDNF) over the same set of Boolean variables. Thus, the learnt assumption would not result in a reduction in the size of the state space. The idea is that, instead, we will be able to obtain a succinct representation. We propose to represent assumptions as IDTMCs, whose transition probabilities are unspecified but assumed to lie within an interval (see Section 3.2.3), to abstract the behaviour of component DTMCs. A new (complete) assume-guarantee rule (Asym-Idtmc) is

developed for the compositional verification.

For the remainder of this chapter, we first describe how to encode DTMCs and IDTMCs as Boolean functions in Section 6.1. Then we present the new assume-guarantee rule (Asym-Idtmc) in Section 6.2, which verifies probabilistic safety properties on two-component systems built from DTMCs using synchronous parallel composition, with assumptions represented as IDTMCs. In Section 6.3, we develop novel techniques, which include a new value iteration/adversary generation algorithm and a symbolic variant of the algorithm based on MTBDDs, to compute the (maximum) reachability probability of IDTMCs that arise as a synchronous parallel composition product of an IDTMC and a DTMC as defined in Section 6.2. Note that, in [CCF+10], a similar reachability analysis for (non-probabilistic) transition systems represented as Boolean functions is done by using SAT-based model checking. However, SAT-based model checking for probabilistic systems are still in early stages of development, performing poorly compared to MTBDDs (see e.g. [TF11]). Thus, we convert Boolean representations of DTMCs/IDTMCs to MTBDDs, which enable the reuse of mature symbolic implementation techniques for probabilistic model checking. In Section 6.4, we build a fully-automated assumption generation framework for the assume-guarantee rule (Asym-Idtmc), using the CDNF algorithm to learn assumptions automatically. In Section 6.5, we present the experimental results of our prototype implementation on several case studies. Lastly, we summarise the strengths and weaknesses of our approach in Section 6.6. See Section 1.3 for credits of this chapter.

6.1 Implicit Encoding of Probabilistic Models

In this section, we describe how to encode DTMCs and IDTMCs in the form of multi-terminal binary decision diagrams (MTBDDs) and Boolean functions.

6.1.1 Encoding Models as MTBDDs

In this following, we give an overview of MTBDDs and describe how they can be used to represent probabilistic models such as DTMCs and IDTMCs.

Introduction to MTBDDs

MTBDDs are symbolic data structures first proposed in [CFM+93] as a generalisation of binary decision diagrams (BDDs) [Bry92], and later widely used for symbolic implementation of probabilistic model checking techniques [Par02, KNP04]. Similarly to BDDs, an MTBDD M is a rooted, directed acyclic graph with its vertex set partitioned into *non-terminal* and *terminal* vertices (also called *nodes*). A non-terminal node m is labelled by a variable $var(\mathsf{m}) \in \{\mathsf{x}_1, \ldots, \mathsf{x}_n\}$ where $\mathsf{x}_1 < \cdots < \mathsf{x}_n$ is a set of distinct, totally ordered, Boolean variables. Each non-terminal node has exactly two children nodes, denoted $then(\mathsf{m})$ and $else(\mathsf{m})$. A terminal node m is labelled by a real number $val(\mathsf{m})$ and has no children. The Boolean variable ordering $<$ is imposed onto the graph by requiring that a child m' of a non-terminal node m is either terminal, or is non-terminal and satisfies $var(\mathsf{m}) < var(\mathsf{m}')$.

An MTBDD M represents a function $f_{\mathsf{M}}(\mathsf{x}_1, \ldots, \mathsf{x}_n) : \mathbb{B}^n \to \mathbb{R}$, the value of which is determined by traversing M from the root node and following the edge of each non-terminal node m to $then(\mathsf{m})$ (resp. $else(\mathsf{m})$) if $var(\mathsf{m})$ is 1 (resp. 0) until reaching a terminal node and obtaining its labelled value. Note that BDDs are in fact a special case of MTBDDs whose terminal nodes are labelled by 1 or 0. MTBDDs can provide compact storage for real-valued functions because they are stored in reduced form. If nodes m and m' are identical (i.e. , $var(\mathsf{m}) = var(\mathsf{m}')$, $then(\mathsf{m}) = then(\mathsf{m}')$ and $else(\mathsf{m}) = else(\mathsf{m}')$), then only one copy is stored. Furthermore, if a node m satisfies $then(\mathsf{m}) = else(\mathsf{m})$, it is removed and any incoming edges are redirected to its unique child. With a fixed ordering of Boolean variables, there is a *canonical* MTBDD representing a given function.

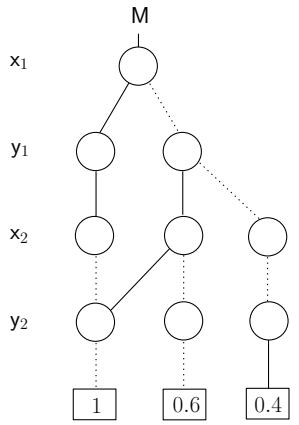
The MTBDD size (number of nodes) is extremely sensitive to the ordering of its Boolean variables, which has a direct effect on both the storage requirements and the time needed to perform operations. In the worst case, the MTBDD size could be exponential in the number of variables and deriving the optimal ordering for a given MTBDD is an NP-complete problem (which follows from NP-hardness of the same problem on BDDs [Bry92]). However, in practice, MTBDDs could be very efficient by applying heuristics to minimise the graph size (see e.g. [Par02]).

MTBDD representation of probabilistic models

In [CFM+93] it is shown how to encode real-valued vectors and matrices. Consider a real-valued vector $\mathbf{v}$ of length 2^n; we can think of it as a mapping from indices $\{1,\ldots,2^n\}$ to the reals $\mathbb{R}$. Given an encoding of $\{1,\ldots,2^n\}$ in terms of Boolean variables $\{\mathsf{x}_1,\ldots,\mathsf{x}_n\}$, we can represent $\mathbf{v}$ by an MTBDD M over $\{\mathsf{x}_1,\ldots,\mathsf{x}_n\}$. Similarly, a square matrix $\mathbf{M}$ of size 2^n by 2^n can be considered as a mapping from $\{1,\ldots,2^n\}\times\{1,\ldots,2^n\}$ to $\mathbb{R}$. Using Boolean variables $\{\mathsf{x}_1,\ldots,\mathsf{x}_n\}$ to range over row indices and variables $\{\mathsf{y}_1,\ldots,\mathsf{y}_n\}$ to range over column indices, $\mathbf{M}$ can be represented by an MTBDD over $\{\mathsf{x}_1,\ldots,\mathsf{x}_n,\mathsf{y}_1,\ldots,\mathsf{y}_n\}$ with an ordering of $\mathsf{x}_1<\mathsf{y}_1<\mathsf{x}_2<\cdots\mathsf{x}_n<\mathsf{y}_n$. Note that the row and column variables are ordered alternately; this is a well-known heuristic to reduce the size of MTBDDs.

Recall from Section 3.2 that DTMCs can be represented by transition probabilities matrices $\mathbf{P}$, and hence it is straightforward to encode DTMCs as MTBDDs. For IDTMCs, which are given by pairs of matrices $\mathbf{P}^l$ and $\mathbf{P}^u$ for the lower/upper transition probability bounds, we could encode them as a pair of MTBDDs M^l and M^u, representing $\mathbf{P}^l$ and $\mathbf{P}^u$ respectively.

Example 6.1 *Figure 6.1 shows an MTBDD* M *representing the transition matrix* $\mathbf{P}$ *of the DTMC introduced in Figure 3.2. The non-terminal nodes of* M *are drawn as circles and arranged in horizontal levels, with their associated Boolean variables displayed at*



x_1	y_1	x_2	y_2	f_{M}	Entry in $\mathbf{P}$
0	0	0	1	0.4	$(0,1) = 0.4$
0	1	0	0	0.6	$(0,2) = 0.6$
0	1	1	0	1	$(1,2) = 1$
1	1	0	0	1	$(2,2) = 1$

Figure 6.1: An MTBDD M representing the transition matrix $\mathbf{P}$ in Figure 3.2

the left end of each level. From each non-terminal node m*, two edges are directed to its children: a solid line for then*(m) *and a dashed line for else*(m)*. Terminal nodes are drawn as squares at the bottom, and are labelled with their value val*(m)*. The terminal node with value* 0 *and any edges which lead directly to it are omitted here for clarity.*

In this example, we encode the 3×3 *matrix* $\mathbf{P}$ *representing integers in binary (i.e.* $, 0 \to 00, 1 \to 01, 2 \to 10$*) with two row variables* (x_1, x_2) *and two column variables* (y_1, y_2)*. The table on the right illustrates the encoding details. Each valuation over* $\{x_1, y_1, x_2, y_2\}$ *in the table represents a non-zero entry in the matrix* $\mathbf{P}$*. Consider, for example, the valuation* $f_{\mathsf{M}}(0,0,0,1) = 0.4$*. In this case,* $x_1 = 0$ *and* $x_2 = 0$*, so the row index is* 0*. Similarly,* $y_1 = 0$ *and* $y_2 = 1$*, giving a column index of* 1*. This corresponds to the matrix entry* $(0,1) = 0.4$*.*

The transition matrices of DTMCs and IDTMCs are indexed by states. So we would need an efficient scheme to encode the model's state space into MTBDD variables. One approach is to assign each state a unique integer and then use the standard binary encoding for integers as described in the previous example. However, in practice, we adopt a more efficient encoding approach [Par02], which is not only fast but can also lead to very compact MTBDD structures. This approach targets models described using the PRISM language, in which a model's state space is defined by a number of integer-

valued PRISM variables. Each PRISM variable is then encoded with its own set of MTBDD variables. The close correspondence between PRISM and MTBDD variables helps to preserve the regularity of model structure in the high-level description, which could be reflected in an increasing number of shared nodes in the low-level MTBDD representation and, hence, a smaller size of MTBDD. As mentioned before, the MTBDD size is also very sensitive to the ordering of its variables. In [Par02], several effective variable ordering heuristics to minimise MTBDD size are discussed, for example, placing closely related MTBDD variables as near to each other as possible.

6.1.2 Encoding Models as Boolean Functions

In the non-probabilistic setting, Boolean encoding of transition systems are popular and used, for example, in SAT-based model checking [McM03, CCF+10]. However, these encoding methods cannot be directly applied to probabilistic models such as DTMCs and IDTMCs, because they do not handle the encoding of transition probabilities. In this section, we propose a novel scheme to encode IDTMCs as Boolean functions. The same scheme can be applied to encode DTMCs as well, by treating them as a special case of IDTMCs.

We encode the transition relation of an IDTMC as a disjunction over a set of transition formulae, with each formula corresponding to a transition between two states. Suppose there is transition in an IDTMC from state s to state s' with the transition probability bounds $[\mathbf{P}^l(s,s'), \mathbf{P}^u(s,s')]$; we represent the transition with the formula:

$$\mathbf{x}(s) \wedge \mathbf{y}(s') \wedge (p \geq \mathbf{P}^l(s,s')) \wedge (p \leq \mathbf{P}^u(s,s'))$$

where $\mathbf{x}(s)$ and $\mathbf{y}(s')$ are the encoding of states s and s' over Boolean variable sets $\mathbf{x}$ (for start states) and $\mathbf{y}$ (for end states) respectively, and p is a real variable whose value is bounded by a pair of predicates: $p \geq \mathbf{P}^l(s,s')$ and $p \leq \mathbf{P}^u(s,s')$. Note that DTMCs

can be considered as a special case of IDTMCs, where the transition probability matrix $\mathbf{P} = \mathbf{P}^l = \mathbf{P}^u$. Thus, to encode a transition of a DTMC we only need to replace $\mathbf{P}^l(s, s')$ and $\mathbf{P}^u(s, s')$ in the above formula with $\mathbf{P}(s, s')$, which is then equivalent to $\mathbf{x}(s) \wedge \mathbf{y}(s') \wedge (p = \mathbf{P}(s, s'))$.

Since the transition relation formulae obtained above contain a mixture of Boolean and real variables, we reduce them into pure Boolean functions by using the eager encoding approach [SSB02]. This reduction includes two steps: firstly, new Boolean variables are introduced to encode predicates containing the real variable p; secondly, constraints on these newly added variables are imposed to preserve the transitivity of the original predicates. We use two sets of Boolean variables, denoted $\mathbf{e}^l$ and $\mathbf{e}^u$, to encode predicates of the form $p \geq \mathbf{P}^l(s, s')$ and $p \leq \mathbf{P}^u(s, s')$ respectively. The size of set $\mathbf{e}^l$ and $\mathbf{e}^u$ is determined by the number of real values $\mathbf{P}^l(s, s')$ and $\mathbf{P}^u(s, s')$ in the IDTMC. After replacing predicates in the original formula with the corresponding variables in $\mathbf{e}^l$ and $\mathbf{e}^u$, we also need to impose transitivity constraints on these variables. For each pair of variables e^l and e^u taken from $\mathbf{e}^l$ and $\mathbf{e}^u$, if the corresponding $\mathbf{P}^l(s, s') \leq \mathbf{P}^u(s, s')$, we add the constraint $(e^l \vee e^u)$ to the formulae with conjunction; otherwise, a constraint $\neg(e^l \wedge e^u)$ would be conjuncted. For example, if e^l represents $p \geq 0.8$ and e^u represents $p \leq 0.5$, then the constraint $\neg(e^l \wedge e^u)$ is added because a real variable p should not be defined as greater than 0.8 and less than 0.5 simultaneously.

Definition 6.2 (Boolean representation of IDTMCs) *An IDTMC $\mathcal{I}$ is represented as a pair of Boolean functions $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$, where $\iota(\mathbf{x})$ is the* initial state predicate *in the form of a conjunction of literals over the set of Boolean variables $\mathbf{x}$ encoding states, and $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$ represents the* transition relation *with Boolean variable sets $\mathbf{x}$, $\mathbf{y}$ encoding states and $\mathbf{e}^l$, $\mathbf{e}^u$ encoding predicates for the lower and upper bounds of transition probabilities, repectively.*

Note that $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$ does not correspond to arbitrary Boolean functions, since it has

x_1	x_2	y_1	y_2	Probability Bounds in $\mathcal{I}$
0	0	0	1	[0,1]
0	0	1	0	[0,1]
0	1	1	0	[1,1]
1	0	1	0	[1,1]

Boolean Variable	Predicate
$\mathbf{e}_1^l$	$p \geq 0$
$\mathbf{e}_2^l$	$p \geq 1$
$\mathbf{e}_1^u$	$p \leq 0$
$\mathbf{e}_2^u$	$p \leq 1$

Figure 6.2: Boolean Encoding Scheme for the IDTMC $\mathcal{I}$ in Figure 3.4

to satisfy the eager encoding constraints over variables $\mathbf{e}^l$ and $\mathbf{e}^u$ as mentioned above.

Example 6.3 *We encode the IDTMC $\mathcal{I}$ shown in Figure 3.4 as a pair of Boolean functions $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$. The initial predicate $\iota(\mathbf{x}) = \neg\mathbf{x}_1 \wedge \neg\mathbf{x}_2$ is the encoding of initial state s_0 over variables $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2\}$. The left table of Figure 6.2 displays the encoding of start/end states for all four non-empty transitions of $\mathcal{I}$ and their associated probability bounds; the right table shows a mapping between Boolean variables and real value predicates. The transition relation of $\mathcal{I}$ is encoded as:*

$$\begin{aligned}\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) = \quad & ((\neg\mathbf{x}_1 \wedge \neg\mathbf{x}_2 \wedge \neg\mathbf{y}_1 \wedge \mathbf{y}_2 \wedge \mathbf{e}_1^l \wedge \mathbf{e}_2^u) \\ & \vee(\neg\mathbf{x}_1 \wedge \neg\mathbf{x}_2 \wedge \mathbf{y}_1 \wedge \neg\mathbf{y}_2 \wedge \mathbf{e}_1^l \wedge \mathbf{e}_2^u) \\ & \vee(\neg\mathbf{x}_1 \wedge \mathbf{x}_2 \wedge \mathbf{y}_1 \wedge \neg\mathbf{y}_2 \wedge \mathbf{e}_2^l \wedge \mathbf{e}_2^u) \\ & \vee(\mathbf{x}_1 \wedge \neg\mathbf{x}_2 \wedge \mathbf{y}_1 \wedge \neg\mathbf{y}_2 \wedge \mathbf{e}_2^l \wedge \mathbf{e}_2^u)) \\ & \wedge(\mathbf{e}_1^l \vee \mathbf{e}_1^u) \wedge (\mathbf{e}_1^l \vee \mathbf{e}_2^u) \wedge \neg(\mathbf{e}_2^l \wedge \mathbf{e}_1^u) \wedge (\mathbf{e}_2^l \vee \mathbf{e}_2^u)\end{aligned}$$

where the first four lines show a disjunction set of transition formulae and the last line conjuncts constraints on Boolean variables which are used to encode predicates for probability bounds. Note that empty transitions with probability bound $[0, 0]$ are omitted here for clarity of presentation. In practice, we encode all transitions including empty tranisitions to obtain the complete transition relation.

6.1.3 Conversion between MTBDDs and Boolean functions

We can convert the MTBDD representation of an IDTMC into a Boolean function $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$ by first converting the MTBDDs into lower/upper transi-

tion probability matrices $\mathbf{P}^l, \mathbf{P}^u$, and then applying the encoding scheme described in Section 6.1.2.

The conversion from Boolean encoding $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$ to MTBDDs is as follows. The initial predicate $\iota(\mathbf{x})$ can be straightforwardly converted to a BDD over Boolean variables $\mathbf{x}$ by preserving all the $\vee$ and $\wedge$ relations between variables. The transition relation $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$ is translated into a pair of MTBDDs M^l, M^u over Boolean variables $\mathbf{x}$ and $\mathbf{y}$, such that $f_{\mathsf{M}^l}(\mathbf{x}, \mathbf{y}) = \mathsf{P}(\mathbf{e}^l)$, $f_{\mathsf{M}^u}(\mathbf{x}, \mathbf{y}) = \mathsf{P}(\mathbf{e}^u)$, where $\mathsf{P}(\mathbf{e}^l)$ and $\mathsf{P}(\mathbf{e}^u)$ represent the lower/upper probability bounds of the original predicates encoded by $\mathbf{e}^l$ and $\mathbf{e}^u$.

6.2 Compositional Verification for DTMCs

In this section, we present a new assume-guarantee rule (Asym-Idtmc) for verifying two-component DTMC systems against probabilistic safety properties, with the assumptions captured by IDTMCs.

6.2.1 Refinement between DTMCs and IDTMCs

Firstly, we define a refinement relation between DTMCs and IDTMCs to indicate the correspondence between components and assumptions.

Definition 6.4 (Refinement relation) *A DTMC* $\mathcal{D} = (S, \overline{s}, \mathbf{P}, L)$ refines *an IDTMC* $\mathcal{I} = (S, \overline{s}, \mathbf{P}^l, \mathbf{P}^u, L)$, *denoted* $\mathcal{D} \preceq \mathcal{I}$, *if* $\mathbf{P}^l(s, s') \leq \mathbf{P}(s, s') \leq \mathbf{P}^u(s, s')$ *for all states* $s, s' \in S$.

Example 6.5 *Consider the DTMC* $\mathcal{D}_1$ *shown in Figure 6.3 and the IDTMC* $\mathcal{I}$ *shown in Figure 6.4. They have identical initial state* s_0 *and labelling function* L *over the same state space* $\{s_0, s_1, s_2\}$. *In addition, the transition probability between any two states in* $\mathcal{D}_1$ *lies within the corresponding transition probability interval in* $\mathcal{I}$. *For example, the*

transition probability between s_0 and s_1 in $\mathcal{D}_1$ is 0.4, which falls into the interval $[0, 1]$ *labelled with the transition between s_0 and s_1 in $\mathcal{I}$. Thus, we have $\mathcal{D}_1 \preceq \mathcal{I}$.*

We can infer the refinement relation between a DTMC and an IDTMC based on their Boolean encodings with the following propsition.

Proposition 6.6 *Given the Boolean encodings of a DTMC $\mathcal{D}$ and an IDTMC $\mathcal{I}$ over the same sets of variables, denoted as $\mathbb{B}(\mathcal{D}) = (\iota_{\mathcal{D}}(\mathbf{x}), \delta_{\mathcal{D}}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$ and $\mathbb{B}(\mathcal{I}) = (\iota_{\mathcal{I}}(\mathbf{x}), \delta_{\mathcal{I}}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$, if $\forall \mathbf{x}.\ \iota_{\mathcal{D}}(\mathbf{x}) \iff \iota_{\mathcal{I}}(\mathbf{x})$ and $\forall \mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u.\ \delta_{\mathcal{D}}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) \implies \delta_{\mathcal{I}}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$, then $\mathcal{D} \preceq \mathcal{I}$.*

Proof: Suppose the initial state of $\mathcal{D}$ is $\overline{s}$, then $\iota_{\mathcal{D}}(\mathbf{x}_{\overline{s}}) = \mathsf{true}$ where $\mathbf{x}_{\overline{s}}$ is the boolean encoding of $\overline{s}$ over variables $\mathbf{x}$. Since $\forall \mathbf{x}.\ \iota_{\mathcal{D}}(\mathbf{x}) \iff \iota_{\mathcal{I}}(\mathbf{x})$, we have $\iota_{\mathcal{I}}(\mathbf{x}_{\overline{s}}) = \mathsf{true}$ and therefore $\overline{s}$ is also the initial state of $\mathcal{I}$.

For any transition in $\mathcal{D}$ between states s and s' with probability $p = \mathbf{P}(s, s')$, there is a set of valuations $\mathbf{x}_s, \mathbf{y}_{s'}, \mathbf{e}^l_p, \mathbf{e}^u_p$ ensuring that $\delta_{\mathcal{D}}(\mathbf{x}_s, \mathbf{y}_{s'}, \mathbf{e}^l_p, \mathbf{e}^u_p) = \mathsf{true}$, where $\mathbf{x}_s$, $\mathbf{y}_{s'}$ are the Boolean encoding of states s and s' over variables $\mathbf{x}$ and $\mathbf{y}$ respectively, and $\mathbf{e}^l_p$, $\mathbf{e}^u_p$ is a set of valuations over Boolean variables $\mathbf{e}^l$, $\mathbf{e}^u$ which ensure the variables encoding the predicates $p \geq \mathbf{P}(s, s')$ and $p \leq \mathbf{P}(s, s')$ are true. Since $\delta_{\mathcal{D}}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) \implies \delta_{\mathcal{I}}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$, we have $\delta_{\mathcal{I}}(\mathbf{x}_s, \mathbf{y}_{s'}, \mathbf{e}^l_p, \mathbf{e}^u_p) = \mathsf{true}$, and hence, for any transition between states s and s' in $\mathcal{I}$, we must have $\mathbf{P}^l(s, s') \leq \mathbf{P}(s, s')$ and $\mathbf{P}^u(s, s') \geq \mathbf{P}(s, s')$, based on the eager encoding constraints (see Section 6.1).

Therefore, $\mathcal{D} \preceq \mathcal{I}$. ■

6.2.2 Synchronous Parallel Composition for DTMCs/IDTMCs

Secondly, we discuss the parallel composition between two DTMC components, and between an IDTMC assumption and a DTMC component.

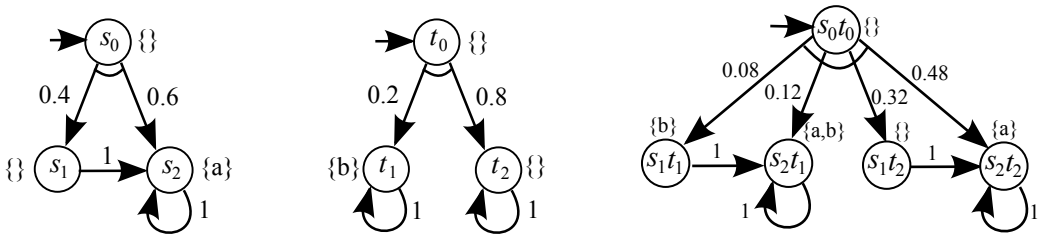


Figure 6.3: Two DTMCs $\mathcal{D}_1$, $\mathcal{D}_2$ and their parallel composition product $\mathcal{D}_1 \| \mathcal{D}_2$

Definition 6.7 (Parallel composition of DTMCs) *The (synchronous) parallel composition product of two DTMCs $\mathcal{D}_i = (S_i, \overline{s}_i, \mathbf{P}_i, L_i)$ for $i = 1, 2$ is given by a DTMC $\mathcal{D}_1 \| \mathcal{D}_2 = (S_1 \times S_2, (\overline{s}_1, \overline{s}_2), \mathbf{P}, L)$, where $\mathbf{P}((s_1, s_2), (s_1', s_2')) = \mathbf{P}_1(s_1, s_1') \cdot \mathbf{P}_2(s_2, s_2')$ and $L((s_1, s_2)) = L(s_1) \cup L(s_2)$ for any $s_1, s_1' \in S_1$ and $s_2, s_2' \in S_2$.*

Example 6.8 *Figure 6.3 shows two DTMCs $\mathcal{D}_1$, $\mathcal{D}_2$ and their synchronous parallel composition product $\mathcal{D}_1 \| \mathcal{D}_2$. The state space of the DTMC $\mathcal{D}_1 \| \mathcal{D}_2$ is over $\{s_0, s_1, s_2\} \times \{t_0, t_1, t_2\}$, and the initial state is (s_0, t_0), abbreviated as s_0t_0 in the graph. The transition probabilities in $\mathcal{D}_1 \| \mathcal{D}_2$ are given by the multiplication of components' transition probabilities. For example, $\mathbf{P}((s_0, t_0), (s_1, t_2)) = \mathbf{P}_1(s_0, s_1) \cdot \mathbf{P}_2(t_0, t_2) = 0.4 \cdot 0.8 = 0.32$. The labelling function of each product state is defined as a union of sets of atomic propositions attached to the corresponding component states, e.g. $L(s_2t_1) = L_1(s_2) \cup L_2(t_1) = \{a, b\}$. The probability of reaching the target state s_2t_1 from the initial state s_0t_0 is 0.2 (obtained by summing up the path probabilities of $s_0t_0 \xrightarrow{0.12} s_2t_1$ and $s_0t_0 \xrightarrow{0.08} s_1t_1 \xrightarrow{1} s_2t_1$).*

Straightforwardly, the synchronous parallel composition between an IDTMC $\mathcal{I} = (S_\mathcal{I}, \overline{s}_\mathcal{I}, \mathbf{P}^l_\mathcal{I}, \mathbf{P}^u_\mathcal{I}, L_\mathcal{I})$ and a DTMC $\mathcal{D} = (S_\mathcal{D}, \overline{s}_\mathcal{D}, \mathbf{P}_\mathcal{D}, L_\mathcal{D})$ may be defined as an IDTMC $\mathcal{I} \|_s \mathcal{D} = (S_\mathcal{I} \times S_\mathcal{D}, (\overline{s}_I, \overline{s}_\mathcal{D}), \mathbf{P}^l, \mathbf{P}^u, L)$, where $\mathbf{P}^l((s,t),(s',t')) = \mathbf{P}^l_I(s,s') \cdot \mathbf{P}_D(t,t')$, $\mathbf{P}^u((s,t),(s',t')) = \mathbf{P}^u_I(s,s') \cdot \mathbf{P}_D(t,t')$ and $L((s,t)) = L(s) \cup L(t)$ for any $s, s' \in S_\mathcal{I}$ and $t, t' \in S_\mathcal{D}$. However, the IMDP semantics of IDTMCs (Definition 3.8) is not closed under this style of synchronous parallel composition, i.e. the IMDP $\lceil \mathcal{I} \|_s \mathcal{D} \rceil$ may contain

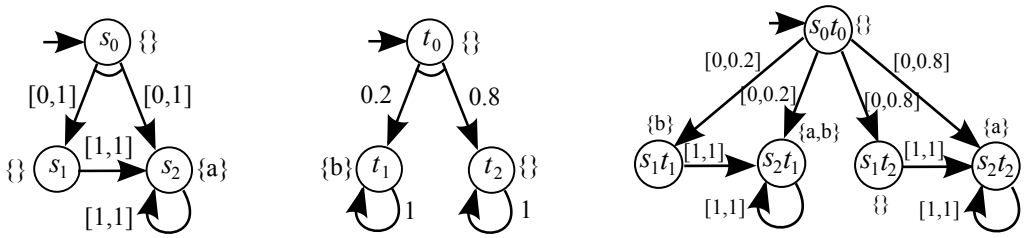


Figure 6.4: An IDTMC $\mathcal{I}$, a DTMC $\mathcal{D}_2$, and their synchronous product $\mathcal{I} \|_s \mathcal{D}_2$

distributions that cannot be obtained as a product of distributions from the IMDP $\lceil \mathcal{I} \rceil$ and the DTMC $\mathcal{D}$. We illustrate this by the following example.

Example 6.9 *Figure 6.4 shows an IDTMC* $\mathcal{I} = (S_\mathcal{I}, \overline{s}_\mathcal{I}, \mathbf{P}^l_\mathcal{I}, \mathbf{P}^u_\mathcal{I}, L_\mathcal{I})$*, a DTMC* $\mathcal{D}_2 = (S_{\mathcal{D}_2}, \overline{s}_{\mathcal{D}_2}, \mathbf{P}_{\mathcal{D}_2}, L_{\mathcal{D}_2})$*, and their synchronous product IDTMC* $\mathcal{I} \|_s \mathcal{D}_2$*. The transition relation of the IMDP* $\lceil \mathcal{I} \|_s \mathcal{D}_2 \rceil$ *gives*

$$\delta(s_0t_0) = \{\mu \in \mathit{Dist}(S_\mathcal{I} \times S_{\mathcal{D}_2}) \mid \forall s \in S_\mathcal{I}, t \in S_{\mathcal{D}_2}.\ \mathbf{P}^l(s_0t_0, st) \leq \mu(st) \leq \mathbf{P}^u(s_0t_0, st)\}.$$

Thus, $\lceil \mathcal{I} \|_s \mathcal{D}_2 \rceil$ *contains a distribution* μ *in state* s_0t_0 *such that* $\mu(s_1t_1) = \mu(s_2t_1) = 0.2$*,* $\mu(s_1t_2) = 0.6$ *and* $\mu(s_2t_2) = 0$*. If we project* μ *onto the state* s_0 *of* $\mathcal{I}$*, then we have*

$$\mu'(s_1) = \frac{\mu(s_1t_1)}{\mathbf{P}(t_0, t_1)} = \frac{0.2}{0.2} = 1, \quad \mu'(s_2) = \frac{\mu(s_2t_1)}{\mathbf{P}(t_0, t_1)} = \frac{0.2}{0.2} = 1.$$

Apparently, μ' *is not a distribution since it sums up to more than 1, and hence cannot be in the IMDP* $\lceil \mathcal{I} \rceil$*.*

In the following, we define a new parallel operator for composing an IDTMC $\mathcal{I}$ and a DTMC $\mathcal{D}$, which is closed under the IMDP semantics by construction.

Definition 6.10 (Parallel composition of an IDTMC and a DTMC) *Given an IDTMC* $\mathcal{I} = (S_\mathcal{I}, \overline{s}_\mathcal{I}, \mathbf{P}^l_\mathcal{I}, \mathbf{P}^u_\mathcal{I}, L_\mathcal{I})$ *and a DTMC* $\mathcal{D} = (S_\mathcal{D}, \overline{s}_\mathcal{D}, \mathbf{P}_\mathcal{D}, L_\mathcal{D})$*, their synchronous parallel composition product is an IDTMC* $\mathcal{I} \| \mathcal{D}$ *which defines an IMDP* $\lceil \mathcal{I} \| \mathcal{D} \rceil = (S_\mathcal{I} \times S_\mathcal{D}, (\overline{s}_I, \overline{s}_\mathcal{D}), \delta, L)$ *such that, for all states* $(s,t) \in S_\mathcal{I} \times S_\mathcal{D}$*, the*

transition relation is $\delta((s,t)) = \{\mu \in Dist(S_{\mathcal{I}} \times S_{\mathcal{D}}) \,|\, \mu = \mu_{\mathcal{I}} \times \mu_{\mathcal{D}}$ *where* $\mu_{\mathcal{I}} \in Dist(S_{\mathcal{I}}), \mathbf{P}^l_{\mathcal{I}}(s,s') \leq \mu_{\mathcal{I}}(s') \leq \mathbf{P}^u_{\mathcal{I}}(s,s')$ *and* $\mu_{\mathcal{D}}(t') = \mathbf{P}_{\mathcal{D}}(t,t')$ *for all* $s' \in S_{\mathcal{I}}, t' \in S_{\mathcal{D}}\}$, *and the labelling function is* $L((s,t)) = L_{\mathcal{I}}(s) \cup L_{\mathcal{D}}(t)$.

The idea is that the available distributions between any two states in the product IDTMC $\mathcal{I}\|\mathcal{D}$ should be restricted as a product of distributions between the corresponding states in the component IDTMC $\mathcal{I}$ and DTMC $\mathcal{D}$. To resolve the nondeterminism of $\mathcal{I}\|\mathcal{D}$, the external environment picks a distribution μ at an execution step in state $(s,t) \in S_{\mathcal{I}} \times S_{\mathcal{D}}$ such that $\mu = \mu_{\mathcal{I}} \times \mu_{\mathcal{D}}$, where $\mu_{\mathcal{I}}$ is nondeterministically chosen from the available distributions in state s of IDTMC $\mathcal{I}$ and $\mu_{\mathcal{D}}$ is the distribution in state t of DTMC $\mathcal{D}$. Note that, in a system where the two components $\mathcal{I}$ and $\mathcal{D}$ are not independent from each other, the nondeterministic choice of $\mu_{\mathcal{I}}$ in $\mathcal{I}$ might be affected by $\mathcal{D}$, i.e. for two states $(s,t_1),(s,t_2)$ in $\mathcal{I}\|\mathcal{D}$ who share the same projected state s in $\mathcal{I}$, different $\mu_{\mathcal{I}}$ might be chosen due to the difference between states t_1 and t_2 in $\mathcal{D}$.

The IMDP $\lceil \mathcal{I}\|\mathcal{D} \rceil$ can be treated as a standard MDP (with action labels omitted); so, here we reuse the notions of paths, adversaries and probabilistic measures defined in Section 3.2.2. Given a finite path $\rho = (s_0,t_0) \xrightarrow{\mu_0} (s_1,t_1) \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_{n-1}} (s_n,t_n)$ through $\lceil \mathcal{I}\|\mathcal{D} \rceil$, we define the *projection* of ρ onto IMDP $\lceil \mathcal{I} \rceil$ as $\rho' = s_0 \xrightarrow{\mu'_0} s_1 \xrightarrow{\mu'_1} \cdots \xrightarrow{\mu'_{n-1}} s_n$ where, for all $i \in \mathbb{N}$, $s_i \in S_{\mathcal{I}}$, $t_i \in S_{\mathcal{D}}$, and $\mu_i = \mu'_i \times \mu''_i$ with μ''_i representing the distribution in state t_i of DTMC $\mathcal{D}$. We show by the following lemma that the IMDP semantics of IDTMCs is closed under the synchronous parallel operator defined above.

Lemma 6.11 *Given an IDTMC* $\mathcal{I}$ *and a DTMC* $\mathcal{D}$, *there is a one-to-one correspondence between adversary* σ *of IMDP* $\lceil \mathcal{I}\|\mathcal{D} \rceil$ *and adversary* $\sigma_{\mathcal{I}}$ *of IMDP* $\lceil \mathcal{I} \rceil$ *such that, for any finite path* $\rho \in FPaths_{\lceil \mathcal{I}\|\mathcal{D} \rceil}$ *ending in state* (s_n,t_n), *we have* $\sigma(\rho) = \sigma_{\mathcal{I}}(\rho') \times \mu$, *where* ρ' *is a projection of* ρ *onto* $\lceil \mathcal{I} \rceil$ *and* μ *is the distribution in state* t_n *of* $\mathcal{D}$.

Proof: Given a finite path $\rho = (s_0,t_0) \xrightarrow{\mu_0} (s_1,t_1) \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_{n-1}} (s_n,t_n)$ under adversary σ of $\lceil \mathcal{I}\|\mathcal{D} \rceil$, we have $\sigma(\rho) \in \delta((s_n,t_n))$ where δ is the transition relation of $\lceil \mathcal{I}\|\mathcal{D} \rceil$. Based

on Definition 6.10 and Definition 3.8, we have $\sigma(\rho) = \mu' \times \mu$, where $\mu' \in \delta'(s_n)$ with δ' representing the transition relation of $\lceil\mathcal{I}\rceil$, and μ the DTMC distribution in state t_n of $\mathcal{D}$. Let $\rho' = s_0 \xrightarrow{\mu'_0} s_1 \xrightarrow{\mu'_1} \cdots \xrightarrow{\mu'_{n-1}} s_n$ be the projection of path ρ on $\lceil\mathcal{I}\rceil$. We can always construct an adversary $\sigma_{\mathcal{I}}$ of $\lceil\mathcal{I}\rceil$ such that $\sigma_{\mathcal{I}}(\rho') = \mu'$. Thus, $\sigma(\rho) = \sigma_{\mathcal{I}}(\rho') \times \mu$. Similarly, we can construct a corresponding adversary σ of $\lceil\mathcal{I}\|\mathcal{D}\rceil$ given an adversary $\sigma_{\mathcal{I}}$ of $\lceil\mathcal{I}\rceil$. Therefore, there exists a one-to-one correspondence between adversaries of $\lceil\mathcal{I}\|\mathcal{D}\rceil$ and $\lceil\mathcal{I}\rceil$. ■

Example 6.12 *Figure 6.5 gives the synchronous parallel composition product $\mathcal{I}\|\mathcal{D}_2$ of the IDTMC $\mathcal{I}$ and the DTMC $\mathcal{D}_2$ shown in Figure 6.4 (the labelling of states is omitted for clarity). To illustrate how to resolve the nondeterminism of $\mathcal{I}\|\mathcal{D}_2$, we draw the transition between any two states (s,t) and (s',t') in $\mathcal{I}\|\mathcal{D}_2$ as two parts linked by a black node, where the first part is labelled with a probability interval $[\mathbf{P}^l_{\mathcal{I}}(s,s'), \mathbf{P}^u_{\mathcal{I}}(s,s')]$ and the second part is labelled with probability $\mathbf{P}_{\mathcal{D}_2}(t,t')$. For example, the transition from state s_0t_0 to state s_1t_2 is labelled with, firstly, $[\mathbf{P}^l_I(s_0,s_1), \mathbf{P}^u_I(s_0,s_1)] = [0,1]$, and secondly, $\mathbf{P}_D(t_0,t_2) = 0.8$. To resolve the nondeterminism in the initial state s_0t_0 of $\mathcal{I}\|\mathcal{D}_2$, the external environment picks a distribution $\mu = \mu_{\mathcal{I}} \times \mu_{\mathcal{D}_2}$ where*

$$\begin{cases} \mu_{\mathcal{I}}(s_0) = 0 \\ 0 \le \mu_{\mathcal{I}}(s_1) \le 1 \\ 0 \le \mu_{\mathcal{I}}(s_2) \le 1 \\ \sum_{0\le i\le 2} \mu_{\mathcal{I}}(s_i) = 1 \end{cases}$$

and $\mu_{\mathcal{D}_2}(t_0) = 0, \mu_{\mathcal{D}_2}(t_1) = 0.2$ and $\mu_{\mathcal{D}_2}(t_2) = 0.8$. For instance, by determining $\mu_{\mathcal{I}}$ as $\mu_{\mathcal{I}}(s_0) = \mu_{\mathcal{I}}(s_1) = 0$ and $\mu_{\mathcal{I}}(s_2) = 1$, we can obtain the maximum probability of reaching state s_2t_1 from the initial state s_0t_0 in $\mathcal{I}\|\mathcal{D}_2$ as 0.2.

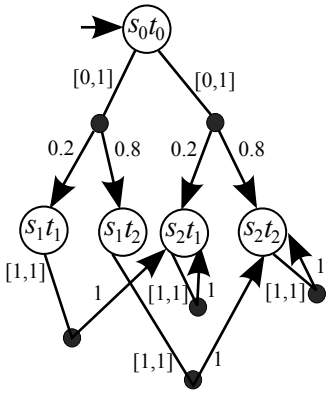


Figure 6.5: The synchronous parallel composition $\mathcal{I}\|\mathcal{D}_2$ (for $\mathcal{I}$ and $\mathcal{D}_2$ in Figure 6.4)

6.2.3 Assume-Guarantee Reasoning Rule (Asym-IDTMC)

The following lemma shows that the refinement relation between a DTMC and an IDTMC is preserved under the synchronous parallel composition.

Lemma 6.13 *If a DTMC $\mathcal{D}_1$ refines an IDTMC $\mathcal{I}$ (i.e. $\mathcal{D}_1 \preceq \mathcal{I}$), then $\mathcal{D}_1\|\mathcal{D}_2$ refines $\mathcal{I}\|\mathcal{D}_2$ (i.e. $\mathcal{D}_1\|\mathcal{D}_2 \preceq \mathcal{I}\|\mathcal{D}_2$), where $\mathcal{D}_2$ is a DTMC.*

Proof: Suppose that $\mathcal{D}_i = (S_i, \overline{s}_i, \mathbf{P}_i, L_i)$ for $i = 1, 2$. Since $\mathcal{D}_1 \preceq \mathcal{I}$, we have $\mathcal{I} = (S_1, \overline{s}_1, \mathbf{P}^l_{\mathcal{I}}, \mathbf{P}^u_{\mathcal{I}}, L_1)$ based on Definition 6.4, where $\mathbf{P}^l_{\mathcal{I}}(s_1, s'_1) \leq \mathbf{P}_1(s_1, s'_1) \leq \mathbf{P}^u_{\mathcal{I}}(s_1, s'_1)$ for any states $s_1, s'_1 \in S_1$. Based on Definition 6.7 and Definition 6.10, we know that $\mathcal{D}_1\|\mathcal{D}_2$ and $\mathcal{I}\|\mathcal{D}_2$ have the same state space $S_1 \times S_2$, initial state $(\overline{s}_1, \overline{s}_2)$ and labelling function L, where $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ for any $s_1 \in S_1, s_2 \in S_2$.

The transition matrix $\mathbf{P}$ of DTMC $\mathcal{D}_1\|\mathcal{D}_2$ is determined based on Definition 6.7 such that $\mathbf{P}((s_1, s_2), (s'_1, s'_2)) = \mathbf{P}_1(s_1, s'_1) \cdot \mathbf{P}_2(s_2, s'_2)$ for any states $s_1, s'_1 \in S_1$ and $s_2, s'_2 \in S_2$. Thus, we have

$$\mathbf{P}^l_{\mathcal{I}}(s_1, s'_1) \cdot \mathbf{P}_2(s_2, s'_2) \leq \mathbf{P}((s_1, s_2), (s'_1, s'_2)) \leq \mathbf{P}^u_{\mathcal{I}}(s_1, s'_1) \cdot \mathbf{P}_2(s_2, s'_2).$$

From Definition 6.10, the transition probability between two states (s_1, s_2) and (s'_1, s'_2) in $\mathcal{I}\|\mathcal{D}_2$ is restricted by the interval $[\mathbf{P}^l_{\mathcal{I}}(s_1, s'_1) \cdot \mathbf{P}_2(s_2, s'_2), \mathbf{P}^u_{\mathcal{I}}(s_1, s'_1) \cdot \mathbf{P}_2(s_2, s'_2)]$.

Therefore, by Definition 6.4, we have $\mathcal{D}_1 \| \mathcal{D}_2 \preceq \mathcal{I} \| \mathcal{D}_2$. ■

It is proved in [CSH08] that an IDTMC satisfies an LTL specification ψ if and only if its corresponding IMDP satisfies ψ. Therefore, to check a probabilistic safety property $\mathsf{P}_{\geq p}[\psi]$ on $\mathcal{I} \| \mathcal{D}$, we only need to check $\mathsf{P}_{\geq p}[\psi]$ on the IMDP $\lceil \mathcal{I} \| \mathcal{D} \rceil$, where standard model checking techniques for MDPs can be applied (see e.g. Section 3.3.3).

Lemma 6.14 *Given two DTMCs $\mathcal{D}_1, \mathcal{D}_2$ and an IDTMC $\mathcal{I}$, if $\mathcal{D}_1 \| \mathcal{D}_2 \preceq \mathcal{I} \| \mathcal{D}_2$ and $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$, where $\mathsf{P}_{\geq p}[\psi]$ is a probabilistic safety property, then $\mathcal{D}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$.*

Proof: Given that $\mathcal{D}_1 \| \mathcal{D}_2 \preceq \mathcal{I} \| \mathcal{D}_2$, based on Definition 6.4 and Definition 6.10, we have $\mu \in \delta((s,t))$ for all states (s,t) in $\mathcal{D}_1 \| \mathcal{D}_2$, where μ is the state distribution in $\mathcal{D}_1 \| \mathcal{D}_2$ and δ is the transition relation of $\lceil \mathcal{I} \| \mathcal{D}_2 \rceil$. Thus, we can construct an adversary σ' of $\lceil \mathcal{I} \| \mathcal{D}_2 \rceil$ such that it always chooses the corresponding state distribution in $\mathcal{D}_1 \| \mathcal{D}_2$. The behaviour of IMDP $\lceil \mathcal{I} \| \mathcal{D}_2 \rceil$ under adversary σ' is purely probabilistic and could be mimicked by the DTMC $\mathcal{D}_1 \| \mathcal{D}_2$, so that we have $Pr^{\sigma'}_{\lceil \mathcal{I} \| \mathcal{D}_2 \rceil}(\psi) = Pr_{\mathcal{D}_1 \| \mathcal{D}_2}(\psi)$.

Since $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$, the IMDP $\lceil \mathcal{I} \| \mathcal{D}_2 \rceil$ satisfies $\mathsf{P}_{\geq p}[\psi]$; that is, $Pr^{\sigma}_{\lceil \mathcal{I} \| \mathcal{D}_2 \rceil}(\psi) \geq p$ for all adversaries $\sigma \in Adv_{\lceil \mathcal{I} \| \mathcal{D}_2 \rceil}$. Therefore, $Pr_{\mathcal{D}_1 \| \mathcal{D}_2}(\psi) \geq p$ and $\mathcal{D}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$.

■

Theorem 6.15 *Given two DTMCs $\mathcal{D}_1$, $\mathcal{D}_2$, an IDTMC $\mathcal{I}$ and a probabilistic safety property $\mathsf{P}_{\geq p}[\psi]$, the following assume-guarantee reasoning rule holds:*

$$\frac{\mathcal{D}_1 \preceq \mathcal{I} \qquad \mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]}{\mathcal{D}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]} \qquad \text{(Asym-Idtmc)}$$

Proof: It follows easily from Lemma 6.13 and Lemma 6.14. ■

Thus, with an appropriate assumption $\mathcal{I}$, the verification of $\mathsf{P}_{\geq p}[\psi]$ on $\mathcal{D}_1 \| \mathcal{D}_2$ can be decomposed into two sub-problems: (1) checking if DTMC $\mathcal{D}_1$ refines IDTMC $\mathcal{I}$, which can be done via checking two implications of Boolean formulae with a SAT solver (Proposition 6.6); (2) checking if $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ holds, which reduces to the problem of computing reachability probabilities in IDTMCs and we will discuss it in Section 6.3. This compositional verification framework is *complete* in the sense that, if $\mathcal{D}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ is true, we can always find an assumption $\mathcal{I}$ to prove it with rule (Asym-Idtmc), because we can use $\mathcal{D}_1$ itself as the assumption.

Example 6.16 *Consider the DTMCs $\mathcal{D}_1$, $\mathcal{D}_2$ shown in Figure 6.3 and the IDTMC $\mathcal{I}$ shown in Figure 6.4. We want to verify whether $\mathcal{D}_1 \| \mathcal{D}_2$ satisfies the probabilistic safety property $\mathsf{P}_{\geq 0.75}[\Box\ (\neg s_2 t_1)]$, i.e. the probability of never reaching a bad state $s_2 t_1$ should be at least 0.75. We have shown that $\mathcal{D}_1 \preceq \mathcal{I}$ in Example 6.5, and, from Example 6.12, we know that the maximum probability of reaching state $s_2 t_1$ from the initial state $s_0 t_0$ in $\mathcal{I} \| \mathcal{D}_2$ is 0.2, so that $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq 0.75}[\Box\ (\neg s_2 t_1)]$ holds. Both premises of rule* (Asym-Idtmc) *are true, and thus $\mathcal{D}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq 0.75}[\Box\ (\neg s_2 t_1)]$.*

6.3 Reachability Analysis of IDTMCs

Recall from the previous section that, for model checking safety properties on IDTMCs, it is sufficient to verify its corresponding IMDPs by applying standard techniques for MDPs (see Section 3.3.3), where verifying probabilistic safety properties reduces to computing reachability probabilities. However, as IMDPs may have an exponential number of distributions in each state, the scalability of this method is quite limited. An alternative approach was proposed by [KKLW12], which, rather than unfolding all available distributions of IMDPs, resolves the nondeterminism of IDTMCs by analysing

Algorithm 8 Value iteration/adversary generation for $Pr^{\max}_{\mathcal{I}\|\mathcal{D},s}(reach_s(T))$

Input: IDTMC $\mathcal{I}$, DTMC $\mathcal{D}$, target states T and convergence criterion ε
Output: solution vector $\vec{v}$, optimal adversary σ

1: $\vec{v}(s) := 1$ if $s \in T$ and $\vec{v}(s) := 0$ otherwise, for each $s \in S_{\mathcal{I}} \times S_{\mathcal{D}}$
2: $\sigma[i][j] := 0$, for all $0 \leq i < |S_{\mathcal{I}} \times S_{\mathcal{D}}|$ and $0 \leq j < |S_{\mathcal{I}} \times S_{\mathcal{D}}|$
3: $\delta := 1$
4: **while** $\delta > \varepsilon$ **do**
5: A := **detAdv**$(\mathcal{I}, \mathcal{D}, \vec{v})$
6: $\vec{v}' := \mathrm{A} \cdot \vec{v}$
7: **for** all $0 \leq k < |S_{\mathcal{I}} \times S_{\mathcal{D}}|$ **do**
8: $\delta = \max_k(\vec{v}'(k) - \vec{v}(k))$
9: **if** $\vec{v}'(k) > \vec{v}(k)$ **then**
10: $\sigma[k][:] = \mathrm{A}[k][:]$
11: **end if**
12: **end for**
13: $\vec{v} = \vec{v}'$
14: **end while**
15: **return** $\vec{v}$, σ

the outgoing transition intervals of each state and determining an optimal distribution. This approach is a *value iteration* method: it approximates (unbounded) reachability probabilities by computing the probability of reaching target states within n steps.

This method cannot be applied directly on $\mathcal{I}\|\mathcal{D}$, which is a product IDTMC composed from an IDTMC $\mathcal{I}$ and a DTMC $\mathcal{D}$ using our newly defined synchronous parallel operator (see Definition 6.10), because the available distributions in $\mathcal{I}\|\mathcal{D}$ are restricted. In this section, we adapt the value iteration algorithm for computing the (maximum) reachability probabilities of product IDTMCs $\mathcal{I}\|\mathcal{D}$. We also develop a symbolic variant of our proposed algorithm based on MTBDDs.

6.3.1 The Value Iteration Algorithm

We present (Algorithm 8) a value iteration algorithm for computing (approximations of) maximum probabilities $Pr^{\max}_{\mathcal{I}\|\mathcal{D},s}(reach_s(T))$ of reaching a set of target states T from state s in $\mathcal{I}\|\mathcal{D}$. The algorithm requires as input the component IDTMC $\mathcal{I}$ and

DTMC $\mathcal{D}$, the target states set $T \subseteq S_{\mathcal{I}} \times S_{\mathcal{D}}$, and a convergence criterion ε. It outputs a solution vector $\vec{v}$ storing the values of $Pr^{\max}_{\mathcal{I}\|\mathcal{D},s}(reach_s(T))$, and an optimal adversary σ that achieves these maximum probabilities.

The solution vector is initialised such that, for each state $s \in S_{\mathcal{I}} \times S_{\mathcal{D}}$, $\vec{v}(s) = 1$ if $s \in T$ and $\vec{v}(s) = 0$ otherwise. The optimal adversary σ is initialised as an all-zero matrix of size $|S_{\mathcal{I}} \times S_{\mathcal{D}}| \times |S_{\mathcal{I}} \times S_{\mathcal{D}}|$. Line 3-14 of Algorithm 8 illustrates the procedure of value iteration. The iteration step n is not decided in advance; rather, it is determined on-the-fly by checking if the convergence of the solution vector drops below the specified criterion ε (see Line 4 and 8). In each iteration step, an adversary A over the state space $S_{\mathcal{I}} \times S_{\mathcal{D}}$ is determined by the function $\mathbf{detAdv}(\mathcal{I}, \mathcal{D}, \vec{v})$ as described in Algorithm 9. The solution vector is updated through a matrix-vector multiplication $\vec{v}' = \mathrm{A} \cdot \vec{v}$. The maximum difference of elements between $\vec{v}'$ and $\vec{v}$ is obtained in Line 8, which is compared with the convergence criterion ε in Line 4 to control the **while** loop. Line 9-11 considers the optimal adversary σ, which is updated only if the reachability probability of a particular state s_k is strictly improved; that is, if $\vec{v}'(k) > \vec{v}(k)$, then the matrix values of row $\sigma[k][:]$ will be replaced by the values of $\mathrm{A}[k][:]$, which represents the probabilistic distribution of state s_k in A.

Now we describe the details of function $\mathbf{detAdv}(\mathcal{I}, \mathcal{D}, \vec{v})$ as shown in Algorithm 9. Recall from Definition 6.10 that a distribution μ in state $(s_{\mathcal{I}}, s_{\mathcal{D}})$ of $\mathcal{I}\|\mathcal{D}$ should be restricted as $\mu = \mu_{\mathcal{I}} \times \mu_{\mathcal{D}}$, where $\mu_{\mathcal{I}}$ is an available distribution in state $s_{\mathcal{I}}$ of IDTMC $\mathcal{I}$ and $\mu_{\mathcal{D}}$ is the distribution in state $s_{\mathcal{D}}$ of DTMC $\mathcal{D}$. Accordingly, we take a two-step approach to determine an adversary A of $\mathcal{I}\|\mathcal{D}$ in each iteration step with function $\mathbf{detAdv}(\mathcal{I}, \mathcal{D}, \vec{v})$: firstly, for each state $(s_{\mathcal{I}}, s_{\mathcal{D}})$ of $\mathcal{I}\|\mathcal{D}$, a distribution $\mu_{\mathcal{I}}$ is chosen from the available distributions in state $s_{\mathcal{I}}$ of IDTMC $\mathcal{I}$ and stored in a matrix $\mathrm{A}_{\mathcal{I}}$ of size $|S_{\mathcal{I}} \times S_{\mathcal{D}}| \times |S_{\mathcal{I}}|$; secondly, the matrix entries of A are computed through the multiplication of corresponding entries in $\mathrm{A}_{\mathcal{I}}$ and the transition matrix $\mathbf{P}_{\mathcal{D}}$ of DTMC $\mathcal{D}$. Note that, if components $\mathcal{I}$ and $\mathcal{D}$ are acting independently (i.e. the choices of $\mu_{\mathcal{I}}$

Algorithm 9 Function **detAdv**$(\mathcal{I}, \mathcal{D}, \vec{v})$ used in Algorithm 8

Input: IDTMC $\mathcal{I}$, DTMC $\mathcal{D}$, and solution vector $\vec{v}$
Output: adversary A, to be used in Line 6 of Algorithm 8

```
1: A[i][j] := 0, for all 0 ≤ i < |S_I × S_D| and 0 ≤ j < |S_I × S_D|
2: A_I[i][j] := 0, dec[i][j] := false, for all 0 ≤ i < |S_I × S_D| and 0 ≤ j < |S_I|
3: put indices of elements of v⃗ in queue q with a descending order
4: while q is not empty do
5:    n = q.pophead()
6:    get the projecting indices n_I and n_D of n on S_I and S_D, respectively
7:    for all 0 ≤ m < |S_I × S_D| do
8:       get the projecting indices m_I and m_D of m on S_I and S_D, respectively
9:       if dec[m][n_I] == false then
10:         sum := Σ_i A_I[m][i], dec[m][n_I] = true
11:         if sum == 1 then
12:            A_I[m][n_I] = 0
13:         else
14:            low := 0
15:            for all 0 ≤ j < |S_I| do
16:               if dec[m][j] == false then
17:                  low = low + P^l_I[m_I][j]
18:               end if
19:            end for
20:            limit = 1 − sum − low
21:            A_I[m][n_I] = min{limit, P^u_I[m_I][n_I]}
22:         end if
23:      end if
24:      A[m][n] = A_I[m][n_I] · P_D[m_D][n_D]
25:   end for
26: end while
27: return A
```

are not influenced by $\mathcal{D}$), it would be sufficient to set the size of matrix $\mathrm{A}_{\mathcal{I}}$ as $|S_{\mathcal{I}}| \times |S_{\mathcal{I}}|$.

Initially, the adversaries A and $\mathrm{A}_{\mathcal{I}}$ are set as all-zero matrices. We use a Boolean matrix dec to record whether elements of $\mathrm{A}_{\mathcal{I}}$ have been decided, and the entries of dec are all initialised as false. To achieve the maximum reachability probabilities, when making the nondeterministic choices of probabilistic distributions we consider states $s = (s_{\mathcal{I}}, s_{\mathcal{D}})$ with higher solution vector values $\vec{v}(s)$ in priority order, and ensure that transition probabilities leading to such states are assigned values as high as possible.

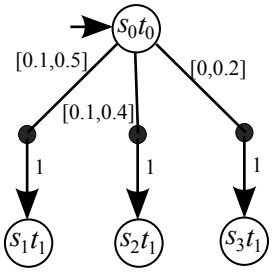


Figure 6.6: The behaviour of state s_0t_0 in an example IDTMC $\mathcal{I}\|\mathcal{D}$

This intuition is illustrated in Example 6.17. The detailed algorithm procedures are described in the following.

As shown in Line 3-5 of Algorithm 9, the elements of $\vec{v}$ are sorted in a descending order with their indices n stored in a queue q. The values of the nth column in A, representing the transition probabilities leading to a state $s_n \in S_{\mathcal{I}} \times S_{\mathcal{D}}$, would then be fixed following the sequence of queue q. In Line 6, a pair of projecting indices $n_{\mathcal{I}}$ and $n_{\mathcal{D}}$ are obtained by projecting the state s_n onto $S_{\mathcal{I}}$ and $S_{\mathcal{D}}$, respectively. Line 7-25 is a **for** loop over the row index m of matrix A. Similarly, a pair of projecting indices $m_{\mathcal{I}}$ and $m_{\mathcal{D}}$ are obtained. Line 9-23 describes how to determine the value of $\mathrm{A}_{\mathcal{I}}[m][n_{\mathcal{I}}]$ if it has not been decided. If a full distribution has already been assigned for state s_m, then $\mathrm{A}_{\mathcal{I}}[m][n_{\mathcal{I}}]$ would be assigned with zero (Line 11-12); otherwise, $\mathrm{A}_{\mathcal{I}}[m][n_{\mathcal{I}}]$ would either be a *limit* value that it could take in the probabilistic distribution, or the upper bound $\mathbf{P}^u_{\mathcal{I}}[m_{\mathcal{I}}][n_{\mathcal{I}}]$ specified in $\mathcal{I}$ (Line 21-22). Once $\mathrm{A}_{\mathcal{I}}[m][n_{\mathcal{I}}]$ is fixed, the value of $\mathrm{A}[m][n]$ is computed as the multiplication of $\mathrm{A}_{\mathcal{I}}[m][n_{\mathcal{I}}]$ and $\mathbf{P}_{\mathcal{D}}[m_{\mathcal{D}}][n_{\mathcal{D}}]$ (Line 25).

Example 6.17 *Suppose that, at iteration step n, we obtain the solution vector $\vec{v} = [0.2, 1, 0.6, 0.9]$ for states $\{s_0t_0, s_1t_1, s_2t_1, s_3t_1\}$ of the IDTMC $\mathcal{I}\|\mathcal{D}$ in Figure 6.6. To determine the distribution for state s_0t_0, we consider its successor states in the order of s_1t_1, s_3t_1, s_2t_1 based on the descending order of their corresponding values in the solution vector $\vec{v}$. Firstly, we take the maximal value 0.5 of the interval $[0.1, 0.5]$ leading to s_1t_1. The next successor state is s_3t_1, and we can still take its upper bound 0.2 from*

the interval $[0, 0.2]$*, since the distribution has total mass less than one. However, for state* s_2t_1*, we cannot take the the upper bound 0.4 of the interval* $[0.1, 0.4]$*, because the distribution has already assigned probability* $0.5 + 0.2 = 0.7$ *to other transitions, and thus the maximum probability we can assign here is* $1 - 0.7 = 0.3$*.*

Complexity. The complexity of function $\mathbf{detAdv}(\mathcal{I}, \mathcal{D}, \vec{v})$ as shown in Algorithm 9 is $\mathcal{O}(|S_{\mathcal{I}} \times S_{\mathcal{D}}|^2 \cdot |S_{\mathcal{I}}|)$, because the algorithm loops through every entry of the adversary matrix A which has a size of $|S_{\mathcal{I}} \times S_{\mathcal{D}}| \times |S_{\mathcal{I}} \times S_{\mathcal{D}}|$ and, when determining the value for each entry of A, corresponding values in $\mathrm{A}_{\mathcal{I}}$ are summed up over the column of size $|S_{\mathcal{I}}|$. In each iteration of Algorithm 8, the function $\mathbf{detAdv}(\mathcal{I}, \mathcal{D}, \vec{v})$ would be applied once only; therefore, the complexity of Algorithm 8 is $\mathcal{O}(n \cdot |S_{\mathcal{I}} \times S_{\mathcal{D}}|^2 \cdot |S_{\mathcal{I}}|)$ where n is the iteration step.

Correctness and termination. There are two key ideas behind this value iteration algorithm. Firstly, in each iteration step, the nondeterminism of $\mathcal{I} \| \mathcal{D}$ is resolved by an adversary A, which is determined by constructing an adversary $\mathrm{A}_{\mathcal{I}}$ based on $\mathcal{I}$; the correctness of this idea follows from Definition 6.10 and Lemma 6.11. Secondly, to ensure the maximum probability of reaching target states, we assign transition probabilities to states with higher solution vector values in priority order; the correctness of this idea has been proved in [KKLW12]. The algorithm termination is guaranteed, because an explicit convergence criterion ε is imposed to stop the value iteration.

6.3.2 The MTBDD-based Value Iteration Algorithm

We optimise Algorithm 8 and present its symbolic variant based on the data structure of MTBDDs (introduced in Section 6.1.1). As shown in Algorithm 10, the input of this algorithm includes a pair of MTBDDs low and up representing the matrices of lower and upper bounds, respectively, of transition probabilities of IDTMC $\mathcal{I}$, an MTBDD dtmc for the transition probability matrix of DTMC $\mathcal{D}$, a BDD target for the

Algorithm 10 Symbolic (MTBDD-based) variant of Algorithm 8

Input: low, up, dtmc, target, and convergence criterion ε
Output: sol, $\mathsf{adv}^{\mathsf{max}}$

1: sol := target
2: $\mathsf{adv}^{\mathsf{max}}$:= Const(0)
3: $\delta := 1$
4: **while** $\delta > \varepsilon$ **do**
5: $\quad \mathsf{adv}_{\mathcal{I}}$:= Const(0)
6: $\quad$ visited := Const(0)
7: $\quad$ tmp := Apply(+, sol, Const(1))
8: $\quad p_s$:= FindMax(tmp)
9: $\quad$ **while** $p_s \geq 1$ **do**
10: $\quad\quad$ max := Threshold(tmp, =, p_s)
11: $\quad\quad$ visited = Apply($\vee$, visited, max)
12: $\quad\quad$ sum := Abstract(+, y, $\mathsf{adv}_{\mathcal{I}}$)
13: $\quad\quad$ dist := Threshold(sum, <, 1)
14: $\quad\quad$ **if** dist $\neq$ Const(0) **then**
15: $\quad\quad\quad$ $\mathsf{low}_{\mathsf{no}}$:= Abstract(+, y, Apply($\wedge$, low, Not(visited)))
16: $\quad\quad\quad$ $\mathsf{limit}_{\mathsf{s}}$:= Apply($-$, Const(1), Apply(+, sum, $\mathsf{low}_{\mathsf{no}}$))
17: $\quad\quad\quad$ up_1 := Apply($\wedge$, up, max)
18: $\quad\quad\quad$ low_1 := Apply($\wedge$, low, max)
19: $\quad\quad\quad$ up_{s} := Abstract(+, y, up_1)
20: $\quad\quad\quad$ $\mathsf{low}_{\mathsf{s}}$:= Abstract(+, y, low_1)
21: $\quad\quad\quad$ diff := Apply($-$, up_1, low_1)
22: $\quad\quad\quad$ norm := Apply($\div$, Apply($-$, $\mathsf{limit}_{\mathsf{s}}$, $\mathsf{low}_{\mathsf{s}}$), Apply($-$, up_{s}, $\mathsf{low}_{\mathsf{s}}$))
23: $\quad\quad\quad$ limit := Apply(+, low_1, Apply($\times$, norm, diff))
24: $\quad\quad\quad$ sign := Threshold(Apply($-$, $\mathsf{limit}_{\mathsf{s}}$, up_{s}), $\leq$, 0)
25: $\quad\quad\quad$ value := Apply(+, Apply($\wedge$, limit, sign), Apply($\wedge$, up_1, Not(sign)))
26: $\quad\quad\quad$ new := Apply(+, Apply($\wedge$, value, dist), Apply($\wedge$, Const(0), Not(dist)))
27: $\quad\quad\quad$ $\mathsf{adv}_{\mathcal{I}}$ = Apply(+, $\mathsf{add}_{\mathcal{I}}$, new)
28: $\quad\quad$ **end if**
29: $\quad\quad$ tmp = Apply($-$, tmp, Apply($\times$, max, Const(p_s)))
30: $\quad\quad p_s$ = FindMax(tmp)
31: $\quad$ **end while**
32: $\quad$ adv := Apply($\times$, $\mathsf{adv}_{\mathcal{I}}$, dtmc)
33: $\quad$ sol' := Apply($\times$, adv, sol)
34: $\quad$ improve := Apply($-$, sol', sol)
35: $\quad \delta$ = FindMax(improve)
36: $\quad$ index := Threshold(improve, >, 0)
37: $\quad$ $\mathsf{adv}^{\mathsf{max}}$ = Apply(+, Apply($\wedge$, adv, index), Apply($\wedge$, $\mathsf{adv}^{\mathsf{max}}$, Not(index)))
38: $\quad$ sol = sol'
39: **end while**
40: **return** sol, $\mathsf{adv}^{\mathsf{max}}$

target states, and a convergence criterion ε. It outputs two MTBDDs: sol, the solution vector storing maximum probabilities of reaching target states from each state in $\mathcal{I}\|\mathcal{D}$, and $\mathsf{adv}^{\mathsf{max}}$, the optimal adversary. The meaning of basic BDD/MTBDD operations used in this algorithm, e.g. CONST, FINDMAX, THRESHOLD, APPLY and ABSTRACT, are summarised in Appendix B.

Algorithm 10 implements the value iteration and adversary generation procedures described in Algorithm 8 using MTBDD operations. Both algorithms determine the adversary for $\mathcal{I}\|\mathcal{D}$ following the descending order of solution vector values. The only difference is that, when there are multiple elements in the solution vector sharing the same value, Algorithm 8 would consider them one by one in arbitrary order, while Algorithm 10 processes them simultaneously to take advantage of the symbolic features of MTBDDs. Line 10-28 of Algorithm 10 illustrates how to assign transition probability values for such a group of successor states max at the same time. The basic idea is to treat such states as a set, with their upper and lower probability bounds summed into $\mathsf{up_s}$ and $\mathsf{low_s}$. The maximum value allocated to them is $\mathsf{limit_s}$ as computed in Line 16. If $\mathsf{up_s}$ is smaller than $\mathsf{limit_s}$, then the states would just take the upper probability bounds; otherwise, each state would take a limit value such that

$$\mathsf{limit} = \mathsf{low} + (\mathsf{up} - \mathsf{low}) \cdot \frac{\mathsf{limit_s} - \mathsf{low_s}}{\mathsf{up_s} - \mathsf{low_s}}$$

Actually, other schemes of assigning the limit value may also be feasible, as long as the values of limit sum up to $\mathsf{limit_s}$ over all states max, and $\mathsf{low} \leq \mathsf{limit} \leq \mathsf{up}$.

Example 6.18 *Let us again consider the IDTMC $\mathcal{I}\|\mathcal{D}$ shown in Figure 6.6. Suppose that the solution vector is $\vec{v} = [0.2, 1, 0.6, 0.6]$ in some iteration step, corresponding to states $\{s_0t_0, s_1t_1, s_2t_1, s_3t_1\}$. To resolve the nondeterministic behaviour of state s_0t_0, we first assign the maximal possible probability 0.5 to the outgoing transition to state s_1t_1, since it has the highest value in the solution vector. Then we consider states s_2t_1*

and s_3t_1 as a set and assign transition probabilities for them simultaneously, because they share the same solution vector value 0.6. The transition probabilities from state s_0t_0 to s_2t_1 and s_3t_1 can be any rational values satisfying the following conditions:

$$\begin{cases} 0.1 \leq x \leq 0.4 \\ 0 \leq y \leq 0.2 \\ x + y \leq 0.5 \end{cases}$$

where x and y are the transition probabilities to states s_2t_1 and s_3t_1, respectively. The scheme used in Algorithm 10 would assign $x = 0.1 + (0.4 - 0.1) \cdot \frac{0.5-0.1}{0.6-0.1} = 0.34$ and $y = 0 + (0.2 - 0) \cdot \frac{0.5-0.1}{0.6-0.1} = 0.16$, which apparently satisfy the above conditions.

6.4 Learning Assumptions for Rule (Asym-Idtmc)

In this section, we present a fully-automated approach for verifying probabilistic safety properties against systems composed from DTMCs, based on the assume-guarantee rule (Asym-Idtmc) proposed in Section 6.2 (Theorem 6.15). In particular, this approach uses the CDNF learning algorithm described in Section 3.5.3 to automatically learn assumptions as IDTMCs, which are represented symbolically as Boolean functions and translated to MTBDDs.

Figure 6.7 shows the overall structure of our approach. Given two DTMCs $\mathcal{D}_1, \mathcal{D}_2$ and a probabilistic safety property $\mathsf{P}_{\geq p}[\psi]$, we aim to verify whether $\mathcal{D}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ is true. If the property holds, then an IDTMC assumption $\mathcal{I}$ would be generated; otherwise, a counterexample would be provided to illustrate the violation. We use the CDNF learning algorithm to automatically learn assumptions. Recall that the Boolean representation of IDTMC $\mathcal{I}$ is $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$, containing two Boolean functions: $\iota(\mathbf{x})$, representing the initial predicate, and $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$, representing the transition relation (see Definition 6.2). Therefore, two instances of the CDNF learning

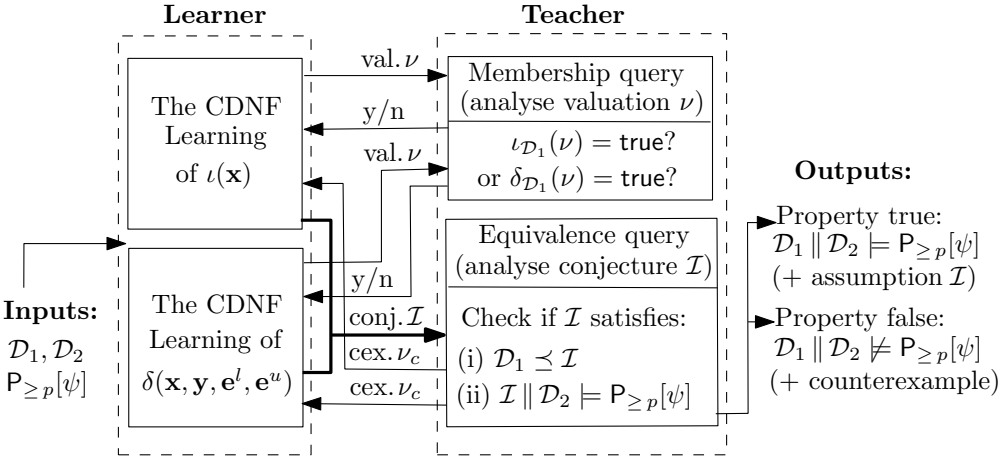


Figure 6.7: Learning probabilistic assumptions for rule (Asym-Idtmc)

algorithm are utilised in our approach, one for each Boolean function.

The membership queries (i.e. whether a valuation ν over a set of Boolean variables is true for the target function) are answered through checking the corresponding functions of DTMC $\mathcal{D}_1$. Based on Proposition 6.6, in order to satisfy $\mathcal{D}_1 \preceq \mathcal{I}$, i.e. the first premise of rule (Asym-Idtmc), any true valuation of the Boolean function $\iota_{\mathcal{D}_1}$ (resp. $\delta_{\mathcal{D}_1}$) should also be true for function ι (resp. δ). Therefore, the teacher answers *yes* to membership queries $\iota(\nu)$ (resp. $\delta(\nu)$) if valuation ν is true for the Boolean function $\iota_{\mathcal{D}_1}$ (resp. $\delta_{\mathcal{D}_1}$), and answers *no* otherwise. The teacher employs a SAT solver to check these queries.

When answering equivalence queries, the teacher considers conjectures proposed by the two CDNF learning instances for functions $\iota(\mathbf{x})$ and $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$ jointly as a Boolean representation for IDTMC $\mathcal{I}$ such that $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$. The teacher answers *yes* to an equivalence query, if both premises of rule (Asym-Idtmc), i.e. $\mathcal{D}_1 \preceq \mathcal{I}$ and $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$, are satisfied. Otherwise, the teacher either finds a counterexample illustrating the violation of property $\mathsf{P}_{\geq p}[\psi]$ on $\mathcal{D}_1 \| \mathcal{D}_2$, or obtains valuation ν_c from a spurious counterexample c to refine the conjectures. We explain the details in the following.

Recall from Proposition 6.6 that $\mathcal{D}_1 \preceq \mathcal{I}$ holds if $\forall \mathbf{x}.\ \iota_{\mathcal{D}_1}(\mathbf{x}) \iff \iota(\mathbf{x})$ and

$\forall \mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u.\ \delta_{\mathcal{D}_1}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) \implies \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$. Thus, the teacher examines these two implication formulae separately with a SAT solver, and returns any false valuation ν_c to the corresponding CDNF learning instance as a counterexample to refine the conjecture. If no false valuation can be found, then $\mathcal{D}_1 \preceq \mathcal{I}$ is true and the teacher proceeds to check the second premise $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$. Because the SAT-based methods for probabilistic model checking show poor performance (see e.g. [TF11]), the teacher checks $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ by first converting the Boolean functions of $\mathcal{I}$ to MTBDDs (see Section 6.1.3) and then applying the MTBDD-based model checking techniques for IDTMCs as described in Section 6.3.

Note that the CDNF learning algorithm learns arbitrary Boolean functions and the learnt function $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$ may not correspond to a valid IDTMC transition relation; that is, the transition probability interval between two states may break into fragments. For example, given a target transition relation function $\delta_{\mathcal{D}_1} = \mathbf{x} \wedge \mathbf{y} \wedge \mathbf{e}_1^l \wedge \mathbf{e}_1^u$, where $\mathbf{e}_1^l$ and $\mathbf{e}_1^u$ correspond to the predicates $p \geq 0.4$ and $p \leq 0.4$ respectively, a learnt function is $\delta = \mathbf{x} \wedge \mathbf{y} \wedge (\mathbf{e}_2^l \vee \mathbf{e}_1^u)$ with $\mathbf{e}_2^l \stackrel{\text{def}}{=} p \geq 0.5$ such that the condition $\delta_{\mathcal{D}_1} \implies \delta$ is satisfied; however, δ is not a valid transition between states $s(\mathbf{x})$ and $s(\mathbf{y})$ of an IDTMC $\mathcal{I}$, because its corresponding transition probability interval has a gap, i.e. $p \in [0, 0.4]$ or $p \in [0.5, 1]$. We solve this problem by applying a filter so that, for each transition between any two states of $\mathcal{I}$, we only keep the interval in which the corresponding transition probability of $\mathcal{D}_1$ lies and filter out other redundant intervals. For the above example, since the corresponding transition probability in $\mathcal{D}_1$ is 0.4, we would keep the interval $[0, 0.4]$ and filter out $[0.5, 1]$. Since the teacher guarantees that $\delta_{\mathcal{D}_1} \implies \delta$, for each transition, the learnt function δ must contain at least one interval in which the transition probability of $\mathcal{D}_1$ lies. Therefore, we can always apply the filter to the learnt Boolean functions and obtain MTBDDs for the valid IDTMC $\mathcal{I}$ so that $\mathcal{D}_1 \preceq \mathcal{I}$.

Recall from Section 6.2 that, to verify $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$, it is sufficient to check the probabilistic safety property $\mathsf{P}_{\geq p}[\psi]$ on the IMDP $\lceil \mathcal{I} \| \mathcal{D}_2 \rceil$ using standard model check-

ing techniques for MDPs (see Section 3.3.3), which reduces the problem to computing the maximum reachability probabilities and can be solved by using the MTBDD-based value iteration algorithm (e.g. Algorithm 10 in Section 6.3). If the model checking of $\mathcal{I}\|\mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ yields true, then the teacher can conclude that $\mathcal{D}_1\|\mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ is true because both premises of rule (Asym-Idtmc) have been satisfied. Otherwise, a probabilistic counterexample c illustrating the violation of $\mathsf{P}_{\geq p}[\psi]$ on $\mathcal{I}\|\mathcal{D}_2$ would be found based on the optimal adversary σ generated by Algorithm 10. The behaviour of $\mathcal{I}\|\mathcal{D}_2$ under the adversary σ is purely probabilistic and can be captured by a DTMC, therefore finding the probabilistic counterexample c reduces to generating counterexamples for the upper bound reachability problem of DTMCs, which has been described in Section 3.4 (here we implement the techniques based on MTBDDs for consistency).

By projecting the state space of c onto DTMC $\mathcal{D}_1$ and keeping only the transitions between projected states, we can obtain a (small) fragment $\mathcal{D}_1^c$ of $\mathcal{D}_1$. If $\mathsf{P}_{\geq p}[\psi]$ is violated on $\mathcal{D}_1^c\|\mathcal{D}_2$, then the teacher can conclude that $\mathcal{D}_1\|\mathcal{D}_2 \not\models \mathsf{P}_{\geq p}[\psi]$ and the learning terminates; otherwise, c is a spurious counterexample and the learnt conjecture needs to be updated. Recall from Section 3.5.3 that the CDNF learning algorithm only accepts a valuation over a set of Boolean variables of the target function as a counterexample. For the transition relation function $\delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$, a valuation corresponds to a single transition between two states in the conjectured IDTMC.

There are two heuristics to obtain such a transition from the conjecture and translate it into a valuation ν_c to refine the learning. One heuristic is to project the optimal adversary σ onto the state space of $\mathcal{D}_1$, and then find a projected transition which has the maximum (positive) difference of transition probability compared to its counterpart in $\mathcal{D}_1$. The other heuristic is to analyse the individual paths of the probabilistic counterexample c following the descending order of path probabilities: given a path of c, we project it onto the state space of $\mathcal{D}_1$ and compare the probabilities of projected transitions with the corresponding transitions in $\mathcal{D}_1$, until a transition with the maximum

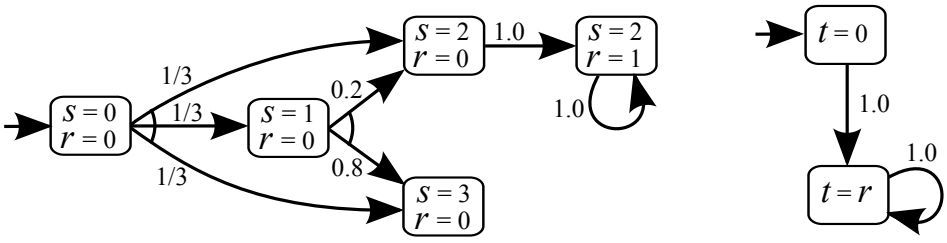


Figure 6.8: Two DTMCs $\mathcal{D}_1$ and $\mathcal{D}_2$ for Example 6.19

Predicate	Encoding	Predicate	Encoding	Predicate	Encoding
$s = 0$	$\neg\mathbf{x}_1\neg\mathbf{x}_2$ or $\neg\mathbf{y}_1\neg\mathbf{y}_2$	$p \geq 0$	$\mathbf{e}_1^l$	$p \leq 0$	$\mathbf{e}_1^u$
$s = 1$	$\neg\mathbf{x}_1\mathbf{x}_2$ or $\neg\mathbf{y}_1\mathbf{y}_2$	$p \geq 0.2$	$\mathbf{e}_2^l$	$p \leq 0.2$	$\mathbf{e}_2^u$
$s = 2$	$\mathbf{x}_1\neg\mathbf{x}_2$ or $\mathbf{y}_1\neg\mathbf{y}_2$	$p \geq 1/3$	$\mathbf{e}_3^l$	$p \leq 1/3$	$\mathbf{e}_3^u$
$s = 3$	$\mathbf{x}_1\mathbf{x}_2$ or $\mathbf{y}_1\mathbf{y}_2$	$p \geq 0.8$	$\mathbf{e}_4^l$	$p \leq 0.8$	$\mathbf{e}_4^u$
$r = 0$	$\neg\mathbf{x}_3$ or $\neg\mathbf{y}_3$	$p \geq 1$	$\mathbf{e}_5^l$	$p \leq 1$	$\mathbf{e}_5^u$
$r = 1$	$\mathbf{x}_3$ or $\mathbf{y}_3$	-	-	-	-

Figure 6.9: Boolean Encoding Scheme for DTMC $\mathcal{D}_1$ in Figure 6.8

positive probability difference over the path is found; and if no transition probability in the projected path is greater than its counterpart in $\mathcal{D}_1$, the search continues to the next path.

Example 6.19 *Figure 6.8 shows two DTMCs $\mathcal{D}_1$ and $\mathcal{D}_2$ whose states are annotated by the values of integer variables s, r and t. Note that the behaviour of $\mathcal{D}_2$ is influenced by $\mathcal{D}_1$, since $t = r$ and the value of r changes alongside the moves in $\mathcal{D}_1$. We want to verify whether the system $\mathcal{D}_1 \| \mathcal{D}_2$ satisfies the safety property $\psi = \Box\neg(t = 1)$, i.e. the value of t should never be 1, with probability at least 0.6.*

We follow the approach illustrated in Figure 6.7 and start two instances of the CDNF learning algorithm to learn the initial predicate and transition relation of assumption $\mathbb{B}(\mathcal{I}) = (\iota(\mathbf{x}), \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u))$. The target is the Boolean representation of $\mathcal{D}_1$ encoded with the scheme shown in Figure 6.9, where the Boolean variables $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$ (resp. $\mathbf{y} = \{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3\}$) encode the values of s and r of the start (resp. end) states of transitions, and the Boolean variable sets $\mathbf{e}^l, \mathbf{e}^u$ encode lower and upper probability bounds, respectively. The initial predicate of $\mathcal{D}_1$ is $\iota_{\mathcal{D}_1}(\mathbf{x}) = \neg\mathbf{x}_1 \wedge \neg\mathbf{x}_2 \wedge \neg\mathbf{x}_3$, cor-

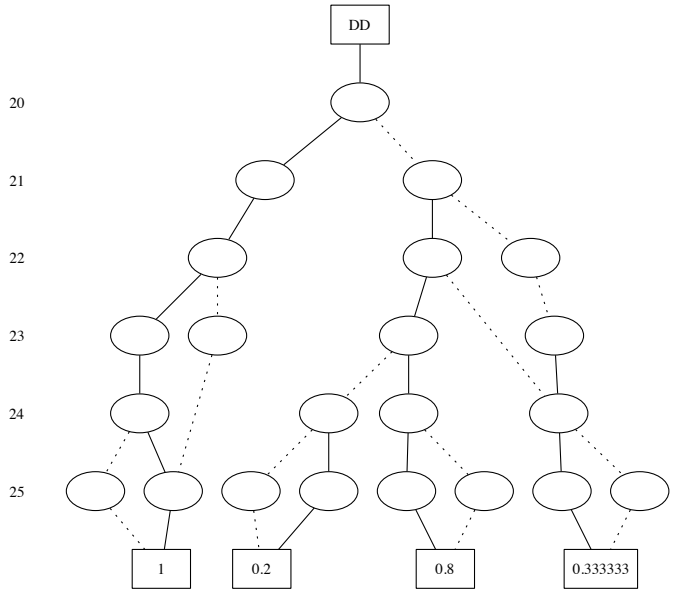


Figure 6.10: The MTBDD of DTMC $\mathcal{D}_1$ for Example 6.19

responding to the state where $s = 0$ *and* $r = 0$*. The translation relation of* $\mathcal{D}_1$ *is a Boolean function* $\delta_{\mathcal{D}_1}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$ *converted from the MTBDD shown in Figure 6.10 (the conversion method is described in Section 6.1.3), where the MTBDD variable indices* $\{20, \ldots, 25\}$ *correspond to the set of Boolean variables* $\{\mathbf{x}_1, \mathbf{y}_1, \mathbf{x}_2, \mathbf{y}_2, \mathbf{x}_3, \mathbf{y}_3\}$*.*

By asking membership queries and checking if any conjectured function $\iota(\mathbf{x})$ *satisfies* $\forall \mathbf{x}.\ \iota_{\mathcal{D}_1}(\mathbf{x}) \iff \iota(\mathbf{x})$*, the CDNF learning instance for the initial predicate determines that* $\iota(\mathbf{x}) = \iota_{\mathcal{D}_1}(\mathbf{x}) = \neg\mathbf{x}_1 \wedge \neg\mathbf{x}_2 \wedge \neg\mathbf{x}_3$*. The CDNF learning instance for the transition relation proposes a first hypothesised function as* $\delta_1(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) = \mathsf{true}$*. It is obvious that* $\delta_{\mathcal{D}_1}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) \implies \mathsf{true}$ *holds for any valuation over Boolean variables* $\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u$*. Thus, we have* $\mathcal{D}_1 \preceq \mathcal{I}_1$*, where* $\mathcal{I}_1$ *is an IDTMC with* $\mathbb{B}(\mathcal{I}_1) = (\iota(\mathbf{x}), \mathsf{true})$ *corresponding to a full graph over the state space of* $\mathcal{D}_1$ *(i.e. there is a transition with probability interval* $[0, 1]$ *between any two states). Model checking* $\mathcal{I}_1 \| \mathcal{D}_2 \models \mathsf{P}_{\geq 0.6}[\Box\neg(t = 1)]$ *yields a counterexample path:* $(s = 0, r = 0, t = 0) \xrightarrow{[0,1]} (s = 0, r = 1, t = 1)$*, which*

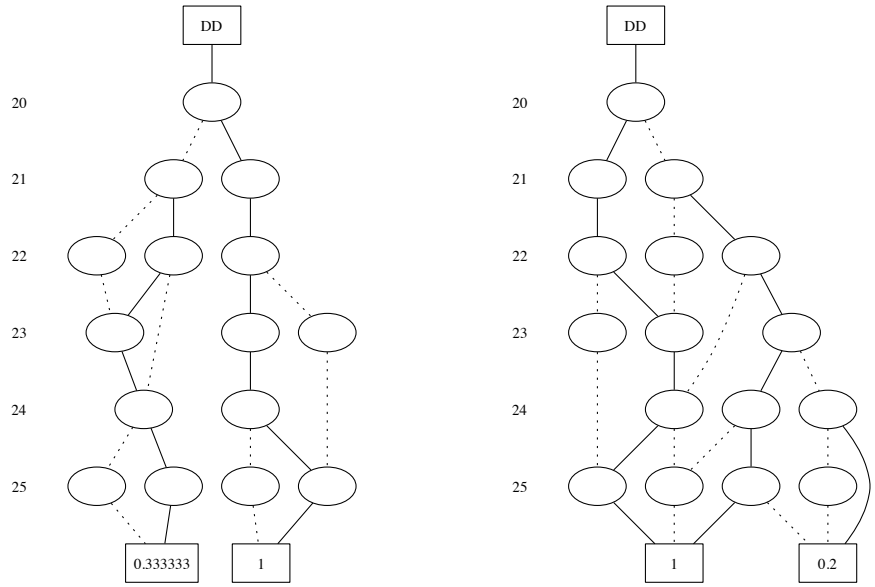


Figure 6.11: The lower/upper MTBDDs of IDTMC $\mathcal{I}$ for Example 6.19

shows that an error state annotated with $t = 1$ *can be reached from the initial state with maximum probability* 1. *By projecting this path onto the state space of* $\mathcal{D}_1$ *and encoding the projected transition over Boolean variables* $\{\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u\}$, *we can obtain a counterexample valuation* ν_c *to refine the CNDF learning of the transition relation.*

After a few iterations of checking membership and equivalence queries, and refining conjectures with counterexample valuations, eventually the transition relation is learnt:

$$\begin{aligned}\delta(\mathbf{x},\mathbf{y},\mathbf{e}^l,\mathbf{e}^u) = \quad & (((\mathbf{e}_2^u \wedge \neg\mathbf{e}_2^l \wedge \mathbf{e}_1^u) \vee (\mathbf{e}_3^l \wedge \neg\mathbf{y}_3 \wedge \mathbf{y}_2) \vee (\mathbf{e}_3^l \wedge \mathbf{x}_3 \wedge \mathbf{y}_2) \\ & \vee(\mathbf{e}_2^u \wedge \neg\mathbf{y}_3 \wedge \mathbf{x}_2 \wedge \mathbf{y}_1) \vee (\mathbf{e}_2^u \wedge \mathbf{x}_3 \wedge \mathbf{x}_2 \wedge \mathbf{y}_1) \vee (\mathbf{e}_3^l \wedge \neg\mathbf{y}_3 \wedge \mathbf{y}_1) \\ & \vee(\mathbf{e}_3^l \wedge \mathbf{x}_3 \wedge \mathbf{y}_1) \vee (\mathbf{e}_5^l \wedge \neg\mathbf{e}_4^u \wedge \mathbf{e}_4^l \wedge \neg\mathbf{e}_3^u \wedge \mathbf{e}_3^l \wedge \mathbf{y}_1 \wedge \mathbf{x}_1)) \\ & \wedge((\neg\mathbf{e}_2^l \wedge \mathbf{e}_1^u) \vee (\neg\mathbf{e}_2^u \wedge \mathbf{e}_3^l \wedge \neg\mathbf{x}_2 \wedge \neg\mathbf{x}_1) \vee (\neg\mathbf{x}_1) \\ & \vee(\mathbf{e}_5^l \wedge \neg\mathbf{e}_4^u \wedge \mathbf{e}_4^l \wedge \neg\mathbf{e}_2^u \wedge \neg\mathbf{e}_3^u \wedge \mathbf{e}_3^l \wedge \mathbf{y}_3 \wedge \neg\mathbf{x}_2) \\ & \vee(\mathbf{e}_5^l \wedge \neg\mathbf{e}_4^u \wedge \mathbf{e}_4^l \wedge \neg\mathbf{e}_2^u \wedge \neg\mathbf{e}_3^u \wedge \mathbf{e}_3^l \wedge \mathbf{y}_2))),\end{aligned}$$

*which can be converted into a pair of MTBDDs as shown in Figure 6.11, representing the lower and upper transition probability bound matrices of an IDTMC $\mathcal{I}$. The teacher checks that $\mathcal{D}_1 \preceq \mathcal{I}$, because $\forall \mathbf{x}.\ \iota_{\mathcal{D}_1}(\mathbf{x}) \iff \iota(\mathbf{x})$ and $\forall \mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u.\ \delta_{\mathcal{D}_1}(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u) \implies \delta(\mathbf{x}, \mathbf{y}, \mathbf{e}^l, \mathbf{e}^u)$. Also $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq 0.6}[\Box \neg (t = 1)]$, since any state annotated with $t = 1$ can be reached from the initial state of $\mathcal{I} \| \mathcal{D}_2$ with probability at most 0.3999. Thus, we can conclude that $\mathcal{D}_1 \| \mathcal{D}_2$ satisfies the probabilistic safety property $\mathsf{P}_{\geq 0.6}[\Box \neg (t = 1)]$ by using the IDTMC $\mathcal{I}$ as an assumption in the assume-guarantee rule (*ASYM-IDTMC*).*

We observe that, in this example, the learnt assumption $\mathcal{I}$ has a more compact MTBDD representation (Figure 6.11) than the target component $\mathcal{D}_1$ (Figure 6.10). Indeed, the number of nodes for the lower (resp. upper) MTBDD of $\mathcal{I}$ is 18 (resp. 19), while the number of MTBDD nodes of $\mathcal{D}_1$ is 27. The assumption $\mathcal{I}$ also has a more succinct Boolean representation than $\mathcal{D}_1$, in terms of the number of CNF clauses of the transition relation function (57 vs. 211). It takes overall 74 membership queries and 16 equivalence queries (13 times of checking $\mathcal{D}_1 \preceq \mathcal{I}$ and thrice checking $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq 0.6}[\Box \neg (t = 1)]$) to learn the correct assumption $\mathcal{I}$.

6.5 Implementation and Case Studies

We have built a prototype tool that implements our fully-automated learning-based compositional verification approach described in Section 6.4. Our prototype uses the Boolean fUnction Learning Library (http://code.google.com/p/project-bull) for the implementation of CDNF learning algorithm. It also uses the SAT solver SAT4J (http://www.sat4j.org) to check the satisfaction of Boolean formulae for membership queries and part of the equivalence queries (i.e. checking the refinement relation $\mathcal{D}_1 \preceq \mathcal{I}$). The other part of equivalence queries (i.e. verifying whether $\mathcal{I} \| \mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ holds) are answered using our own MTBDD-based implementation of Algorithm 10 in Section 6.3, which is built on top of the JDD library in the PRISM package [KNP11].

Our prototype also uses PRISM to export MTBDD representation of models as the input of our eager encoding implementation. We have applied the prototype tool to several benchmark case studies, which are described briefly below (also see Appendix E for detailed models and properties in the PRISM modelling language). The experiments were run on a 2.80GHz PC with 32GB RAM running 64-bit Fedora.

Contract Signing (egl). This case study is based on the contract signing protocol of Even, Goldreich and Lempel [EGL85], where two parties are exchanging N pairs of secrets and each secret contains L bits. We have used the same case study in Chapter 5, where the components are modelled as PIOSs and composed in a synchronous fashion. We adapt the models such that components are communicated through shared variables rather than synchronising actions, because we do not consider the encoding of actions in this chapter. Similarly to Chapter 5, we are interested in verifying the maximum probability that "party A is committed while party B is not". We decompose the system into two DTMCs: $\mathcal{D}_1$ for the counter and $\mathcal{D}_2$ for both parties. Assumptions are learnt about $\mathcal{D}_1$.

Client-Server Model. This case study is modified from the client-server model used in Chapter 4, where N clients request resources from a server and one client may behave wrongly with certain probability, causing the violation of a mutual exclusion property. We adapt the server model as a DTMC $\mathcal{D}_1$, and model the clients who are organised via a round robin schedule as another DTMC $\mathcal{D}_2$. Both DTMCs communicate via shared variables instead of actions. We verify the maximum probability that the mutual exclusion property is violated.

Results

Figure 6.12 shows the experimental results. For each model, we first report the number of Boolean variables (Var), the number of membership queries (MQ), and the number of

equivalence queries (EQ) used in the CDNF learning algorithm. Then, we compare the MTBDD size (number of nodes) between the learnt assumption $\mathcal{I}$ and the component DTMC $\mathcal{D}_1$, with $\mathcal{I}^l$, $\mathcal{I}^u$ representing the lower/upper MTBDDs for $\mathcal{I}$. We also compare the Boolean function size of $\mathcal{I}$ and $\mathcal{D}_1$ in terms of their CNF clause numbers (converted through the Tseitin transformation [Tse83]). Finally, we report the results and the total run-time of our learning-based compositional verification approach. For comparison, we also include the result and run-time of the non-compositional verification using PRISM (MTBDD engine).

We observe that, in all cases, our approach can learn assumptions that are significantly smaller than the component in terms of the Boolean formulae size (CNF clauses.) The succinct Boolean representation of assumptions may suggest more efficiency in model checking. However, since the SAT-based model checking for probabilistic safety properties is currently not feasible in practice, this potential gain is not reflected in our experiments. We would hope to achieve such improvement in future, with the advancement of SAT-based techniques for probabilistic model checking.

We also observe that the MTBDD representations of assumptions, which are converted from learnt Boolean formulae, are roughly the same size as components. Indeed, Figure 6.12 shows that the upper bound MTBDDs $\mathcal{I}^u$ are generally smaller than the MTBDD of the corresponding component $\mathcal{D}_1$, while the lower bound MTBDDs $\mathcal{I}^l$ are slightly larger. However, we should note that, when we perform symbolic model checking of $\mathcal{I}\|\mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$ using Algorithm 10, MTBDDs $\mathcal{I}^u, \mathcal{I}^l$ are not directly used for matrix-vector multiplication, which has a key impact on the efficiency; rather, the algorithm only uses $\mathcal{I}^u, \mathcal{I}^l$ as a reference to look up values when building the adversary for each iteration. By contrast, the MTBDD of $\mathcal{D}_1$ is used for the matrix-vector multiplication in the non-compositional verification $\mathcal{D}_1\|\mathcal{D}_2 \models \mathsf{P}_{\geq p}[\psi]$. Therefore, the comparison between MTBDD sizes of $\mathcal{I}^u$, $\mathcal{I}^l$ and $\mathcal{D}_1$ does not have a direct implication on model checking efficiency.

Recall from Section 6.4 that there are two heuristics for finding a single transition from a probabilistic counterexample to refine the CDNF learning. One is by searching the entire optimal adversary (adv), and the other is by analysing the individual paths in the probabilistic counterexample (path). We have implemented both heuristics and compare their performance in Figure 6.12. It seems that the (adv) heuristic performs better than the (path) heuristic in all egl cases, with fewer queries used in the learning, smaller size of learnt assumptions (in terms of CNF clauses) and better total run-time. But in the client-server cases the results are the other way around. Thus, it is not conclusive which heuristic is better in general.

Finally, we should note that the approximate model checking results produced by our approach are very close to those of the non-compositional verification. In all the egl cases, the results are exactly the same; and in the client-server cases, the results are close enough considering the precision of floating-numbers. We do not compare the run-time of our prototype tool with the highly-optimised PRISM, since we are more interested to investigate the feasibility of learning good quality assumptions. The performance of our tool may be improved by optimisations such as using the policy iteration algorithm rather than the value iteration method for model checking.

Case study [parameters]			Learning related			MTBDD nodes		CNF clauses		Compositional		Non-compositional	
			Var	MQ	EQ	$\mathcal{I}^l$	$\mathcal{I}^u$	$\mathcal{I}$	$\mathcal{D}_1$	Result	Time (s)	Result	Time (s)
egl (adv) $[N\ L]$	5	2	30	207	24	243	**231**	**202**	25,637	0.5156	7.6	0.5156	0.1
	5	6	32	159	19	259	**247**	**154**	60,017	0.5156	19.7	0.5156	0.4
	10	4	34	502	46	271	**259**	**497**	85,181	0.5	295.4	0.5	1.7
	10	8	36	725	63	282	**270**	**720**	153,449	0.5	997.9	0.5	7.9
	15	2	32	319	33	263	**251**	**314**	76,349	0.5	1,761.9	0.5	1.6
egl (path) $[N\ L]$	5	2	30	675	69	285	**231**	**656**	25,637	0.5156	15.1	0.5156	0.1
	5	6	32	780	71	301	**247**	**761**	60,017	0.5156	45.9	0.5156	0.4
	10	4	34	1,211	109	313	**259**	**1,192**	85,181	0.5	342.5	0.5	1.7
	10	8	36	797	71	324	**270**	**778**	153,449	0.5	1000.2	0.5	7.9
	15	2	32	1,020	92	305	**251**	**1,001**	76,349	0.5	1,802.4	0.5	1.6
client-server (adv) $[N]$	4		36	955	75	516	**370**	**804**	16,939	0.0037	9.3	0.0038	0.02
	5		39	1,155	88	647	**447**	**958**	29,035	0.0017	19.5	0.0017	0.04
	6		41	1,223	88	738	**506**	**964**	35,977	8.18E-4	25.2	8.19E-4	0.05
	7		43	1,127	84	824	**560**	**920**	48,361	3.91E-4	33.2	3.99E-4	0.07
	8		47	1,237	96	953	**665**	**1,047**	50,179	1.95E04	40.1	1.97E-4	0.12
client-server (path) $[N]$	4		36	691	60	485	**370**	**624**	16,939	0.0037	7.1	0.0038	0.02
	5		39	730	62	595	**447**	**661**	29,035	0.0017	12.9	0.0017	0.04
	6		41	697	62	678	**506**	**661**	35,977	8.18E-4	15.8	8.19E-4	0.05
	7		43	821	68	750	**560**	**757**	48,361	3.91E-4	25.2	3.99E-4	0.07
	8		47	1,041	87	880	**665**	**995**	50,179	1.95E-4	33.9	1.97E-4	0.12

Figure 6.12: Performance of the learning-based compositional verification using rule (Asym-Idtmc)

6.6 Summary and Discussion

In this chapter, we presented a novel (complete) assume-guarantee rule (Asym-Idtmc) for the compositional verification of DTMCs, where the assumptions are captured by IDTMCs. This rule is based on a new parallel operator that we defined for composing a DTMC and an IDTMC, which preserves the compositionality of IMDP semantics for IDTMCs. We also proposed a new value iteration algorithm for computing the maximum reachability probabilities on IDTMCs that are composed using our new parallel operator. A symbolic variant of this algorithm is also developed based on the data structure of MTBDDs. We built a fully-automated implementation of the proposed compositional verification framework, which uses the CDNF learning algorithm to automatically learn assumptions encoded in Boolean formulae. Experimental results on large case studies show that our approach can learn succint assumptions that enable the compositional verification of DTMCs using rule (Asym-Idtmc).

A disadvantage of our approach is that we have to convert the learnt assumptions represented in Boolean formulae into MTBDDs for the purpose of probabilistic model checking, which affects the performance of our approach. Ideally, we should be able to perform the model checking with Boolean functions directly, as in [CCF$^+$10] for non-probabilistic models. However, as mentioned before, SAT-based probabilistic model checking is still in early stages of development and not feasible for our approach currently. We hope the techniques will be improved in future.

A potential extension of our approach is to consider probabilistic systems that exhibit nondeterministic behaviour, e.g. those modelled in PAs (or MDPs). Recall from Section 6.1.2 that we have only considered the Boolean encoding of system states and transition probabilities. It is straightforward to encode transition actions as Boolean variables. Thus, we can represent PAs are Boolean formulae. The assumptions about PAs can be captured by some interval variant of PAs, e.g. the abstract probabilistic

automata [KKLW12]. We would need to develop a new assume-guarantee rule, which is left for future work.

Chapter 7

Conclusions

In this thesis, we aimed to address the problem of automatically learning assumptions for compositional verification of probabilistic systems. Compositional verification via assume-guarantee reasoning is a very promising solution to improve the scalability of model checking techniques, and thus can be used to extend the applicability of these techniques to broader classes of real-life systems. However, the conventional assume-guarantee verification techniques require non-trivial human effort to find appropriate assumptions. The research presented in this thesis develops, for the first time, fully-automated approaches for generating assumptions and performing compositional verification for various probabilistic systems. The applicability of the proposed approaches is evaluated on a range of benchmark case studies with prototype implementations. The encouraging experimental results demonstrate that the work presented in this thesis is helpful for scaling up probabilistic model checking techniques. The major contributions of this thesis are novel approaches for learning assumptions for three different compositional verification frameworks, presented in Chapters 4, 5 and 6, respectively. We draw conclusions about each approach as follows.

The approach in Chapter 4 targets systems composed of PAs and builds on top of the compositional verification framework proposed in [KNPQ10]. It learns assumptions

represented as probabilistic safety properties using the L* (or NL*) learning algorithm and multi-objective model checking. It can handle three different assume-guarantee rules, (Asym), (Asym-N) and (Circ), which are probabilistic analogs of the most important rules in the non-probabilistic setting. Experimental results show that this approach has the advantage of learning assumptions that are sufficient for verification and still significantly smaller than the corresponding system components. For example, an assumption with 4 states is successfully learnt for a component that has about 17.5 millions states in one case. There are also two large cases where the non-compositional verification is not feasible due to the size of models, but the compositional verification using our approach still works. One disadvantage is due to the *incompleteness* of the underlying assume-guarantee rules, i.e. it is not always possible to find a suitable assumption to prove the assume-guarantee rules.

The approach in Chapter 5 targets systems whose components are modelled as PIOSs and, when composed in a synchronous fashion, result in a DTMC. A new (complete) assume-guarantee rule (Asym-Pios) is proposed in this chapter, for verifying such systems against probabilistic safety properties compositionally, using a richer class of assumptions represented as RPAs. To build a fully-automated implementation of this compositional verification framework, a (semi-)algorithm for checking the language inclusion of RPAs and an L*-style learning algorithm for RPAs are developed. A prototype tool is implemented for this approach and applied to a set of benchmark case studies. Experiments show that this approach can be used to learn small assumptions about large components. A weakness is that we cannot guarantee termination, because the problems of checking the language inclusion of RPAs and learning RPAs are undeciable.

The approach in Chapter 6 targets purely probabilistic systems modelled as DTMCs and learns implicit assumptions encoded as Boolean formulae using the CDNF learning algorithm. A novel (complete) assume-guarantee rule (Asym-Idtmc) is proposed, in

which the assumptions are represented as IDTMCs, based on a new parallel operator for composing DTMCs and IDTMCs. A fully-automated implementation of this approach is developed, which also includes a symbolic (MTBDD-based) value iteration algorithm for verifying the probabilistic reachability of IDTMCs. Experiments on large case studies show that this approach can successfully learn small assumptions for the compositional verification using rule (Asym-Idtmc). A drawback is the inconvenience of converting assumptions between Boolean formulae and MTBDDs, because it is not practical to verify probabilistic safety properties on IDTMCs encoded in Boolean formulae directly.

In summary, this thesis presents novel approaches that are pioneering work for automatically learning probabilistic assumptions of good quality (i.e. small and sufficient for verification), enabling the fully-automated compositional verification of probabilistic systems, and thus improving the scalability of probabilistic model checking.

Future Work

The research presented in this thesis can be further optimised and extended in many directions. First, we can address the weaknesses of current approaches and optimise their performance. For instance, we can adapt the alphabet refinement techniques [PGB+08] and use the BDD-based symbolic variant of the L* algorithm [NMA08] to optimise the assumption learning approach proposed in Chapter 4; we can characterise the termination conditions for the language inclusion and RPA learning algorithms in Chapter 5; and we can explore the use of policy iteration instead of value iteration for the model checking algorithms in Chapter 6.

Secondly, we can investigate the application of our approaches to more case studies. For example, [CKJ12] presents a case study of verifying probabilistic safety properties in cloud computing environment using the compositional verification framework of [KNPQ10], but the assumptions in this work are derived manually. It might be possible

to apply the approach proposed in Chapter 4 to this case study.

Moreover, we can generalise the current approaches and develop further extensions. We briefly discuss a few potential extensions below:

- The work in this thesis only considers the compositional verification of probabilistic safety properties. In future, we may generalise the assumption learning approach for the compositional verification of ω-regular properties and expected reward properties. A good starting point is to consider the assume-guarantee rules proposed in [FKN$^+$11] and the learning approaches in [FCC$^+$08, CG10].

- The models considered in this thesis are all discrete time. It may be possible to extend our approaches for continuous-time models such as continuous-time Markov chains and probabilistic timed automata. Some recent work [LAD$^+$11, LLS$^+$12] has been devoted to studying the automated assumption generation for timed systems in the non-probabilistic setting. It might be possible to adapt them for probabilistic timed systems.

- We may extend the approach in Chapter 6 of learning implicit assumptions for the compositional verification of nondeterministic systems modelled as PAs. This would require consideration of the Boolean encoding of transition actions. We can also replace the CDNF learning algorithm used in Chapter 6 with alternative algorithms [CW12] that learn Boolean formulae incrementally (i.e. the number of Boolean variables is not fixed).

Appendix A

Proofs for Chapter 5

The following proofs first appeared in the jointly authored technical report [FHKP11b], contributed by Tingting Han. Notations are modified here for consistency of the thesis.

Proposition 5.14. *The triple* $\langle \mathcal{A} \rangle\ \mathcal{M}\ \langle G \rangle_{\geq p}$ *given in Definition 5.13 holds if and only if* $pios(\mathcal{A}) \| \mathcal{M} \models \langle G \rangle_{\geq p}$.

The proof of Proposition 5.14 follows directly from Lemma A.1 and Lemma A.2.

Lemma A.1 *Suppose* $\mathcal{M}_1, \mathcal{M}_2$ *are a pair of PIOSs and* $\mathcal{A}$ *is an assumption about* $\mathcal{M}_1$ *such that* $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A}$*, then* $\mathcal{M}_1 \| \mathcal{M}_2 \sqsubseteq_{\mathrm{w}} pios(\mathcal{A}) \| \mathcal{M}_2$.

Proof: Given any finite word w in $\mathcal{M}_1 \| \mathcal{M}_2$, w can be uniquely mapped back onto $\mathcal{M}_1$ and $\mathcal{M}_2$, as $w_1 = w \lceil_{\mathcal{M}_1}$ and $w_2 = w \lceil_{\mathcal{M}_2}$, by sequentially taking the synchronous actions and their respective hidden actions (if not $\perp$). We write $w = w_1 \| w_2$ for simplicity. For a word in a composed model, since only input/output actions are visible and all the other actions are τ actions (recall that all hidden actions are renamed to τ), it is easy to see that given a τ-free word $\bar{w} \in \alpha^*$, we have $\bar{w} = \bar{w}_1 \| \bar{w}_2$, where $\bar{w}_1$ and $\bar{w}_2$ are τ-free words in $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively.

Given a (finite) path ρ in $\mathcal{M}_1 \| \mathcal{M}_2$ with $act(\rho) = w$, then ρ can be decomposed into ρ_1 in $\mathcal{M}_1$ with $act(\rho_1) = w_1$, and ρ_2 in $\mathcal{M}_2$ with $act(\rho_2) = w_2$. We write $\rho = \rho_1 \| \rho_2$ for simplicity. The probability of the composed path is $Pr_{\mathcal{M}_1 \| \mathcal{M}_2}(\rho_1 \| \rho_2) = Pr_{\mathcal{M}_1}(\rho_1) \cdot Pr_{\mathcal{M}_2}(\rho_2)$. Since $\mathcal{M}_1 \sqsubseteq_{\mathrm{w}} \mathcal{A}$, we have that $Pr_{\mathcal{M}_1}(st(w_1)) \leq Pr_{\mathcal{A}}(st(w_1))$, which means that

$$\sum_{act(\rho_1) = st(w_1)} Pr_{\mathcal{M}_1}(\rho_1) \leq \sum_{act(\rho_1') = st(w_1)} Pr_{\mathcal{A}}(\rho_1'). \qquad (\dagger)$$

where ρ_1' is a finite path in $\mathcal{A}$. For any τ-free word $\bar{w}$ in $\mathcal{M}_1 \| \mathcal{M}_2$, we have

$$
\begin{aligned}
& Pr_{\mathcal{M}_1\|\mathcal{M}_2}(\bar{w}) \\
= \;& Pr_{\mathcal{M}_1\|\mathcal{M}_2}(\bar{w}_1\|\bar{w}_2) \\
= \;& \sum_{st(w')=\bar{w}_1\|\bar{w}_2} \sum_{act(\rho_1\|\rho_2)=w'} Pr_{\mathcal{M}_1\|\mathcal{M}_2}(\rho_1\|\rho_2) \\
= \;& \sum_{st(w')=\bar{w}_1\|\bar{w}_2} \sum_{act(\rho_1\|\rho_2)=w'} Pr_{\mathcal{M}_1}(\rho_1) \cdot Pr_{\mathcal{M}_2}(\rho_2) \\
= \;& \sum_{st(w')=\bar{w}_1\|\bar{w}_2, w'=\rho_1\|\rho_2} \sum_{act(\rho_2)=\bar{w}_2} \sum_{act(\rho_1)=\bar{w}_1} Pr_{\mathcal{M}_1}(\rho_1) \cdot Pr_{\mathcal{M}_2}(\rho_2) \\
= \;& \sum_{st(w')=\bar{w}_1\|\bar{w}_2, w'=\rho_1\|\rho_2} \sum_{act(\rho_2)=\bar{w}_2} Pr_{\mathcal{M}_1}(\bar{w}_1) \cdot Pr_{\mathcal{M}_2}(\rho_2) \\
\overset{(\dagger)}{\leq} \;& \sum_{st(w')=\bar{w}_1\|\bar{w}_2, w'=\rho_1\|\rho_2} \sum_{act(\rho_2)=\bar{w}_2} Pr_{pios(\mathcal{A})}(\bar{w}_1) \cdot Pr_{\mathcal{M}_2}(\rho_2) \\
= \;& \sum_{st(w')=\bar{w}_1\|\bar{w}_2, w'=\rho_1\|\rho_2} \sum_{act(\rho_2)=st(w_2)} \sum_{act(\rho_1')=st(w_1)} Pr_{pios(\mathcal{A})}(\rho_1') \cdot Pr_{\mathcal{M}_2}(\rho_2) \\
= \;& Pr_{pios(\mathcal{A})\|\mathcal{M}_2}(\bar{w}_1\|\bar{w}_2) \\
= \;& Pr_{pios(\mathcal{A})\|\mathcal{M}_2}(\bar{w})
\end{aligned}
$$

Therefore, $\mathcal{M}_1\|\mathcal{M}_2 \sqsubseteq_{\mathrm{w}} pios(\mathcal{A})\|\mathcal{M}_2$. ■

Lemma A.2 *Suppose that* $\mathcal{M}_1\|\mathcal{M}_2 \sqsubseteq_{\mathrm{w}} pios(\mathcal{A})\|\mathcal{M}_2$ *and* $pios(\mathcal{A})\|\mathcal{M}_2 \models \langle G \rangle_{\geq p}$*, then* $\mathcal{M}_1\|\mathcal{M}_2 \models \langle G \rangle_{\geq p}$.

Proof: Based on Lemma 5.4, $\mathcal{M}_1\|\mathcal{M}_2$ and $pios(\mathcal{A})\|\mathcal{M}_2$ are both DTMCs. Let $\mathcal{D}$=$\mathcal{M}_1\|\mathcal{M}_2$ and $\mathcal{D}'$=$pios(\mathcal{A})\|\mathcal{M}_2$. Since $\mathcal{D}' \models \langle G \rangle_{\geq p}$, recall from Section 3.3.2 that $Pr_{\mathcal{D}'}(G) = 1 - Pr_{\mathcal{D}'\otimes G^{err}}(\Diamond err') \geq p$, where err' represents the set of accepting paths in the product DTMC of $\mathcal{D}'$ and DFA G^{err}. Since $\mathcal{D} \sqsubseteq_{\mathrm{w}} \mathcal{D}'$ and there is no probability in DFA G^{err}, we have $\mathcal{D} \otimes G^{err} \sqsubseteq_{\mathrm{w}} \mathcal{D}' \otimes G^{err}$. Thus

$$Pr_{\mathcal{D}}(G) = 1 - Pr_{\mathcal{D}\otimes G^{err}}(\Diamond err) \geq 1 - Pr_{\mathcal{D}'\otimes G^{err}}(\Diamond err') \geq p.$$

Therefore, $\mathcal{M}_1\|\mathcal{M}_2 \models \langle G \rangle_{\geq p}$. ■

Appendix B

Basic MTBDD Operations

The followings are all of the basic BDD and MTBDD operations as described in [Par02]. We treat BDDs simply as a special case of MTBDDs. In the following, we assume that M, M_1 and M_2 are MTBDDs over the set of variables $\underline{\mathsf{x}} = (\mathsf{x}_1, \ldots, \mathsf{x}_n)$.

- Const(c), where $c \in \mathbb{R}$, creates a new MTBDD with the constant value c, i.e. a single terminal m, with $val(\mathsf{m}) = c$.
- Apply($op, \mathsf{M}_1, \mathsf{M}_2$), where op is a binary operation over the reals (e.g. $+$, $-$, $\times$, $\div$, min, max, etc.), returns the MTBDD representing the function f_{M_1} op f_{M_2}. If M_1 and M_2 are BDDs, op can also be a Boolean operation ($\wedge$, $\vee$, $\Longrightarrow$, etc.). For clarity, we allow Apply operations to be expressed in infix notation, e.g. $\mathsf{M}_1 \times \mathsf{M}_2 = \textsc{Apply}(\times, \mathsf{M}_1, \mathsf{M}_2)$ and $\mathsf{M}_1 \wedge \mathsf{M}_2 = \textsc{Apply}(\wedge, \mathsf{M}_1, \mathsf{M}_2)$.
- Not(M), where M is a BDD, returns the BDD representing the function $\neg f_{\mathsf{M}}$. As above, we may abbreviate Not(M) to $\neg \mathsf{M}$.
- Abs(M) returns the MTBDD representing the function $|f_{\mathsf{M}}|$, giving the absolute value of the original one.
- Threshold($\mathsf{M}, \bowtie, c$), where $\bowtie$ is a relational operator ($<$, $>$, $\leq$, $\geq$, etc.) and $c \in \mathbb{R}$, returns the BDD M' with $f_{\mathsf{M}'}$ equal to 1 if $f_{\mathsf{M}} \bowtie c$ and 0 otherwise.
- FindMin(M) returns the real constant equal to the minimum value of f_{M}.
- FindMax(M) returns the real constant equal to the maximum value of f_{M}.
- Abstract($op, \underline{\mathsf{x}}, \mathsf{M}$), where op is a commutative and associative binary operation over the reals, returns the result of abstracting all the variables in $\underline{\mathsf{x}}$ from

M by applying *op* over all possible values taken by the variables. For example, $\textsc{Abstract}(+, (\mathsf{x}_1), \mathsf{M})$ would give the MTBDD representing the function $f_{\mathsf{M}|_{\mathsf{x}_1=0}} + f_{\mathsf{M}|_{\mathsf{x}_1=1}}$ and $\textsc{Abstract}(\times, (\mathsf{x}_1, \mathsf{x}_2), \mathsf{M})$ would give the MTBDD representing the function $f_{\mathsf{M}|_{\mathsf{x}_1=0,\mathsf{x}_2=0}} \times f_{\mathsf{M}|_{\mathsf{x}_1=0,\mathsf{x}_2=1}} \times f_{\mathsf{M}|_{\mathsf{x}_1=1,\mathsf{x}_2=0}} \times f_{\mathsf{M}|_{\mathsf{x}_1=1,\mathsf{x}_2=1}}$. In the latter, $\mathsf{M}|_{\mathsf{x}_1=b_1,\mathsf{x}_2=b_2}$ is equivalent to $(\mathsf{M}|_{\mathsf{x}_1=b_1})|_{\mathsf{x}_2=b_2}$.

- $\textsc{ThereExists}(\underline{\mathsf{x}}, \mathsf{M})$, where M is a BDD, is equivalent to $\textsc{Abstract}(\vee, \underline{\mathsf{x}}, \mathsf{M})$.
- $\textsc{ForAll}(\underline{\mathsf{x}}, \mathsf{M})$, where M is a BDD, is equivalent to $\textsc{Abstract}(\wedge, \underline{\mathsf{x}}, \mathsf{M})$.
- $\textsc{ReplaceVars}(\mathsf{M}, \underline{\mathsf{x}}, \underline{\mathsf{y}})$, where $\underline{\mathsf{y}} = (\mathsf{y}_1, \ldots, \mathsf{y}_n)$, returns the MTBDD M' over variables $\underline{\mathsf{y}}$ with $f_{\mathsf{M}'}(b_1, \ldots, b_n) = f_{\mathsf{M}}(b_1, \ldots, b_n)$ for all $(b_1, \ldots, b_n) \in \mathbb{B}^n$.

Appendix C

Case Studies for Chapter 4

The followings are detailed models of case studies in Section 4.5, described in the PRISM modelling language, which are adapted from the benchmark case studies on the PRISM website (http://www.prismmodelchecker.org/). We only show the largest example of each case study. All models are verified against property: $1 - \mathsf{P}_{\max} =?[\diamond \text{“err”}]$.

Client-Server (1 failure) with $N = 7$

---

mdp

module server
s:[0..3];
i:[0..7];
j:[0..7];

// initial cancel loops
[client1–cancel] s=0 → true;
[client2–cancel] s=0 → true;
[client3–cancel] s=0 → true;
[client4–cancel] s=0 → true;
[client5–cancel] s=0 → true;
[client6–cancel] s=0 → true;
[client7–cancel] s=0 → true;

// client i request/grant/cancel
[client1–request] s=0 → (s'=1) & (i'=1);
[client1–grant] s=1 & i=1 → (s'=2);
[client1–cancel] s=2 & i=1 → (s'=0) & (i'=0);

```
    [client2–request] s=0 → (s'=1) & (i'=2);
    [client2–grant] s=1 & i=2 → (s'=2);
    [client2–cancel] s=2 & i=2 → (s'=0) & (i'=0);
    [client3–request] s=0 → (s'=1) & (i'=3);
    [client3–grant] s=1 & i=3 → (s'=2);
    [client3–cancel] s=2 & i=3 → (s'=0) & (i'=0);
    [client4–request] s=0 → (s'=1) & (i'=4);
    [client4–grant] s=1 & i=4 → (s'=2);
    [client4–cancel] s=2 & i=4 → (s'=0) & (i'=0);
    [client5–request] s=0 → (s'=1) & (i'=5);
    [client5–grant] s=1 & i=5 → (s'=2);
    [client5–cancel] s=2 & i=5 → (s'=0) & (i'=0);
    [client6–request] s=0 → (s'=1) & (i'=6);
    [client6–grant] s=1 & i=6 → (s'=2);
    [client6–cancel] s=2 & i=6 → (s'=0) & (i'=0);
    [client7–request] s=0 → (s'=1) & (i'=7);
    [client7–grant] s=1 & i=7 → (s'=2);
    [client7–cancel] s=2 & i=7 → (s'=0) & (i'=0);

    // deny other requests when serving
    [client1–request] s=2 → (s'=3) & (j'=1);
    [client1–deny] s=3 & j=1 → (s'=2) & (j'=0);
    [client2–request] s=2 → (s'=3) & (j'=2);
    [client2–deny] s=3 & j=2 → (s'=2) & (j'=0);
    [client3–request] s=2 → (s'=3) & (j'=3);
    [client3–deny] s=3 & j=3 → (s'=2) & (j'=0);
    [client4–request] s=2 → (s'=3) & (j'=4);
    [client4–deny] s=3 & j=4 → (s'=2) & (j'=0);
    [client5–request] s=2 → (s'=3) & (j'=5);
    [client5–deny] s=3 & j=5 → (s'=2) & (j'=0);
    [client6–request] s=2 → (s'=3) & (j'=6);
    [client6–deny] s=3 & j=6 → (s'=2) & (j'=0);
    [client7–request] s=2 → (s'=3) & (j'=7);
    [client7–deny] s=3 & j=7 → (s'=2) & (j'=0);

    // cancel loops when serving
    [client1–cancel] s=2 & i!=1 → true;
    [client2–cancel] s=2 & i!=2 → true;
    [client3–cancel] s=2 & i!=3 → true;
    [client4–cancel] s=2 & i!=4 → true;
    [client5–cancel] s=2 & i!=5 → true;
    [client6–cancel] s=2 & i!=6 → true;
    [client7–cancel] s=2 & i!=7 → true;
endmodule
```

```
// 1-p is probability of initial error occuring
const double p=0.9;

module client1
    c1:[-1..4];

    [] c1=-1 → p:(c1'=0)+(1-p):(c1'=3); [client1–request] c1=0 → (c1'=1);
    [client1–deny] c1=1 → (c1'=0);
    [client1–grant] c1=1 → (c1'=2);
    [client1–useResource] c1=2 → (c1'=2);
    [client1–cancel] c1=2 → (c1'=0);
    [client1–cancel] c1=3 → (c1'=1);
endmodule

module client2
    c2:[0..2];

    [client2–request] c2=0 → (c2'=1);
    [client2–deny] c2=1 → (c2'=0);
    [client2–grant] c2=1 → (c2'=2);
    [client2–useResource] c2=2 → (c2'=2);
    [client2–cancel] c2=2 → (c2'=0);
endmodule

module client3= client2[ c2=c3, client2–request=client3–request, client2–deny=client3–deny, client2–grant=client3–grant, client2–useResource=client3–useResource, client2–cancel=client3–cancel]
endmodule

module client4= client2[ c2=c4, client2–request=client4–request, client2–deny=client4–deny, client2–grant=client4–grant, client2–useResource=client4–useResource, client2–cancel=client4–cancel]
endmodule

module client5= client2[ c2=c5, client2–request=client5–request, client2–deny=client5–deny, client2–grant=client5–grant, client2–useResource=client5–useResource, client2–cancel=client5–cancel]
endmodule

module client6= client2[ c2=c6, client2–request=client6–request, client2–deny=client6–deny, client2–grant=client6–grant, client2–useResource=client6–useResource, client2–cancel=client6–cancel]
endmodule

module client7= client2[ c2=c7, client2–request=client7–request, client2–deny=client7–deny, client2–grant=client7–grant, client2–useResource=client7–useResource, client2–cancel=client7–cancel]
endmodule

module exclusion
    e:[0..3];
```

k:[0..7];
[client1–grant] e=0 → (e'=1) & (k'=1);
[client1–cancel] e=1 & k=1 → (e'=0) & (k'=0);
[client2–grant] e=0 → (e'=1) & (k'=2);
[client2–cancel] e=1 & k=2 → (e'=0) & (k'=0);
[client3–grant] e=0 → (e'=1) & (k'=3);
[client3–cancel] e=1 & k=3 → (e'=0) & (k'=0);
[client4–grant] e=0 → (e'=1) & (k'=4);
[client4–cancel] e=1 & k=4 → (e'=0) & (k'=0);
[client5–grant] e=0 → (e'=1) & (k'=5);
[client5–cancel] e=1 & k=5 → (e'=0) & (k'=0);
[client6–grant] e=0 → (e'=1) & (k'=6);
[client6–cancel] e=1 & k=6 → (e'=0) & (k'=0);
[client7–grant] e=0 → (e'=1) & (k'=7);
[client7–cancel] e=1 & k=7 → (e'=0) & (k'=0);

[client1–cancel] e=0 → (e'=2) & (k'=0);
[client2–cancel] e=0 → (e'=2) & (k'=0);
[client3–cancel] e=0 → (e'=2) & (k'=0);
[client4–cancel] e=0 → (e'=2) & (k'=0);
[client5–cancel] e=0 → (e'=2) & (k'=0);
[client6–cancel] e=0 → (e'=2) & (k'=0);
[client7–cancel] e=0 → (e'=2) & (k'=0);

[client1–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=2 → (e'=2) & (k'=0);

[client1–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=5 → (e'=2) & (k'=0);

```
    [client7–cancel] e=1 & k=5 → (e'=2) & (k'=0);
    [client1–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client2–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client3–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client4–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client5–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client6–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client7–grant] e=1 & k=6 → (e'=2) & (k'=0);
    [client1–cancel] e=1 & k=6 → (e'=2) & (k'=0);
    [client2–cancel] e=1 & k=6 → (e'=2) & (k'=0);
    [client3–cancel] e=1 & k=6 → (e'=2) & (k'=0);
    [client4–cancel] e=1 & k=6 → (e'=2) & (k'=0);
    [client5–cancel] e=1 & k=6 → (e'=2) & (k'=0);
    [client7–cancel] e=1 & k=6 → (e'=2) & (k'=0);
    [client1–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client2–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client3–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client4–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client5–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client6–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client7–grant] e=1 & k=7 → (e'=2) & (k'=0);
    [client1–cancel] e=1 & k=7 → (e'=2) & (k'=0);
    [client2–cancel] e=1 & k=7 → (e'=2) & (k'=0);
    [client3–cancel] e=1 & k=7 → (e'=2) & (k'=0);
    [client4–cancel] e=1 & k=7 → (e'=2) & (k'=0);
    [client5–cancel] e=1 & k=7 → (e'=2) & (k'=0);
    [client6–cancel] e=1 & k=7 → (e'=2) & (k'=0);

    [client1–grant] e=2 → true;
    [client1–cancel] e=2 → true;
    [client2–grant] e=2 → true;
    [client2–cancel] e=2 → true;
    [client3–grant] e=2 → true;
    [client3–cancel] e=2 → true;
    [client4–grant] e=2 → true;
    [client4–cancel] e=2 → true;
    [client5–grant] e=2 → true;
    [client5–cancel] e=2 → true;
    [client6–grant] e=2 → true;
    [client6–cancel] e=2 → true;
    [client7–grant] e=2 → true;
    [client7–cancel] e=2 → true;
endmodule
```

label "err" = e=2;

Client-Server (N failures) with $N = 7$

---

mdp

module server
s:[0..3];
i:[0..7];
j:[0..7];

// initial cancel loops
[client1–cancel] s=0 → true;
[client2–cancel] s=0 → true;
[client3–cancel] s=0 → true;
[client4–cancel] s=0 → true;
[client5–cancel] s=0 → true;
[client6–cancel] s=0 → true;
[client7–cancel] s=0 → true;

// client i request/grant/cancel
[client1–request] s=0 → (s'=1) & (i'=1);
[client1–grant] s=1 & i=1 → (s'=2);
[client1–cancel] s=2 & i=1 → (s'=0) & (i'=0);
[client2–request] s=0 → (s'=1) & (i'=2);
[client2–grant] s=1 & i=2 → (s'=2);
[client2–cancel] s=2 & i=2 → (s'=0) & (i'=0);
[client3–request] s=0 → (s'=1) & (i'=3);
[client3–grant] s=1 & i=3 → (s'=2);
[client3–cancel] s=2 & i=3 → (s'=0) & (i'=0);
[client4–request] s=0 → (s'=1) & (i'=4);
[client4–grant] s=1 & i=4 → (s'=2);
[client4–cancel] s=2 & i=4 → (s'=0) & (i'=0);
[client5–request] s=0 → (s'=1) & (i'=5);
[client5–grant] s=1 & i=5 → (s'=2);
[client5–cancel] s=2 & i=5 → (s'=0) & (i'=0);
[client6–request] s=0 → (s'=1) & (i'=6);
[client6–grant] s=1 & i=6 → (s'=2);
[client6–cancel] s=2 & i=6 → (s'=0) & (i'=0);
[client7–request] s=0 → (s'=1) & (i'=7);
[client7–grant] s=1 & i=7 → (s'=2);

```
    [client7–cancel] s=2 & i=7 → (s'=0) & (i'=0);

    // deny other requests when serving
    [client1–request] s=2 → (s'=3) & (j'=1);
    [client1–deny] s=3 & j=1 → (s'=2) & (j'=0);
    [client2–request] s=2 → (s'=3) & (j'=2);
    [client2–deny] s=3 & j=2 → (s'=2) & (j'=0);
    [client3–request] s=2 → (s'=3) & (j'=3);
    [client3–deny] s=3 & j=3 → (s'=2) & (j'=0);
    [client4–request] s=2 → (s'=3) & (j'=4);
    [client4–deny] s=3 & j=4 → (s'=2) & (j'=0);
    [client5–request] s=2 → (s'=3) & (j'=5);
    [client5–deny] s=3 & j=5 → (s'=2) & (j'=0);
    [client6–request] s=2 → (s'=3) & (j'=6);
    [client6–deny] s=3 & j=6 → (s'=2) & (j'=0);
    [client7–request] s=2 → (s'=3) & (j'=7);
    [client7–deny] s=3 & j=7 → (s'=2) & (j'=0);

    // cancel loops when serving
    [client1–cancel] s=2 & i!=1 → true;
    [client2–cancel] s=2 & i!=2 → true;
    [client3–cancel] s=2 & i!=3 → true;
    [client4–cancel] s=2 & i!=4 → true;
    [client5–cancel] s=2 & i!=5 → true;
    [client6–cancel] s=2 & i!=6 → true;
    [client7–cancel] s=2 & i!=7 → true;
endmodule

// 1-p is probability of initial error occuring
const double p=0.9;

module client1
    c1:[-1..4];

    [] c1=-1 → p:(c1'=0)+(1-p):(c1'=3); [client1–request] c1=0 → (c1'=1);
    [client1–deny] c1=1 → (c1'=0);
    [client1–grant] c1=1 → (c1'=2);
    [client1–useResource] c1=2 → (c1'=2);
    [client1–cancel] c1=2 → (c1'=0);
    [client1–cancel] c1=3 → (c1'=1);
endmodule

module client2= client1[ c1=c2, client1–request=client2–request, client1–deny=client2–deny, client1–
grant=client2–grant, client1–useResource=client2–useResource, client1–cancel=client2–cancel]
endmodule
```

module client3= client1[c1=c3, client1–request=client3–request, client1–deny=client3–deny, client1–grant=client3–grant, client1–useResource=client3–useResource, client1–cancel=client3–cancel]
endmodule

module client4= client1[c1=c4, client1–request=client4–request, client1–deny=client4–deny, client1–grant=client4–grant, client1–useResource=client4–useResource, client1–cancel=client4–cancel]
endmodule

module client5= client1[c1=c5, client1–request=client5–request, client1–deny=client5–deny, client1–grant=client5–grant, client1–useResource=client5–useResource, client1–cancel=client5–cancel]
endmodule

module client6= client1[c1=c6, client1–request=client6–request, client1–deny=client6–deny, client1–grant=client6–grant, client1–useResource=client6–useResource, client1–cancel=client6–cancel]
endmodule

module client7= client1[c1=c7, client1–request=client7–request, client1–deny=client7–deny, client1–grant=client7–grant, client1–useResource=client7–useResource, client1–cancel=client7–cancel]
endmodule

module exclusion
e:[0..3];
k:[0..7];
[client1–grant] e=0 → (e'=1) & (k'=1);
[client1–cancel] e=1 & k=1 → (e'=0) & (k'=0);
[client2–grant] e=0 → (e'=1) & (k'=2);
[client2–cancel] e=1 & k=2 → (e'=0) & (k'=0);
[client3–grant] e=0 → (e'=1) & (k'=3);
[client3–cancel] e=1 & k=3 → (e'=0) & (k'=0);
[client4–grant] e=0 → (e'=1) & (k'=4);
[client4–cancel] e=1 & k=4 → (e'=0) & (k'=0);
[client5–grant] e=0 → (e'=1) & (k'=5);
[client5–cancel] e=1 & k=5 → (e'=0) & (k'=0);
[client6–grant] e=0 → (e'=1) & (k'=6);
[client6–cancel] e=1 & k=6 → (e'=0) & (k'=0);
[client7–grant] e=0 → (e'=1) & (k'=7);
[client7–cancel] e=1 & k=7 → (e'=0) & (k'=0);

[client1–cancel] e=0 → (e'=2) & (k'=0);
[client2–cancel] e=0 → (e'=2) & (k'=0);
[client3–cancel] e=0 → (e'=2) & (k'=0);
[client4–cancel] e=0 → (e'=2) & (k'=0);
[client5–cancel] e=0 → (e'=2) & (k'=0);
[client6–cancel] e=0 → (e'=2) & (k'=0);
[client7–cancel] e=0 → (e'=2) & (k'=0);

[client1–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=4 → (e'=2) & (k'=0);

[client5–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=5 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=5 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=6 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=6 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=6 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=6 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=6 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=6 → (e'=2) & (k'=0);
[client7–cancel] e=1 & k=6 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client5–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client6–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client7–grant] e=1 & k=7 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=7 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=7 → (e'=2) & (k'=0);

[client3–cancel] e=1 & k=7 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=7 → (e'=2) & (k'=0);
[client5–cancel] e=1 & k=7 → (e'=2) & (k'=0);
[client6–cancel] e=1 & k=7 → (e'=2) & (k'=0);

[client1–grant] e=2 → true;
[client1–cancel] e=2 → true;
[client2–grant] e=2 → true;
[client2–cancel] e=2 → true;
[client3–grant] e=2 → true;
[client3–cancel] e=2 → true;
[client4–grant] e=2 → true;
[client4–cancel] e=2 → true;
[client5–grant] e=2 → true;
[client5–cancel] e=2 → true;
[client6–grant] e=2 → true;
[client6–cancel] e=2 → true;
[client7–grant] e=2 → true;
[client7–cancel] e=2 → true;

endmodule

label "err"= e=2;

Randomised Consensus with $N = 3, R = 3, K = 20$

---

mdp

// constants for the shared coin
const int N=3;
const int K=20;
const int range = 2*(K+1)*N;
const int counter–init = (K+1)*N;
const int left = N;
const int right= 2*(K+1)*N -N;

// shared coins for round 1 and 2
global counter1 : [0..range] init counter–init;
global counter2 : [0..range] init counter–init;

module r1–coin1
r1–start1 : bool; *// when protocol is initialised*
r1–pc1 : [0..3]; *// program counter: 0 - flip, 1 - write, 2 - check, 3 - finished*

```
    r1–coin1 : [0..1]; // local coin

    // start coin protocol for process 1
    [coin1–s1–start] !r1–start1 → (r1–start1'=true);
    // flip coin
    [] r1–start1 & (r1–pc1=0) → 0.5 : (r1–coin1'=0) & (r1–pc1'=1)
            + 0.5 : (r1–coin1'=1) & (r1–pc1'=1);
    // write tails -1 (reset coin to add regularity)
    [] r1–start1 & (r1–pc1=1) & (r1–coin1=0) & (counter1>0) → (counter1'=counter1-1)
            & (r1–pc1'=2) & (r1–coin1'=0);
    // write heads +1 (reset coin to add regularity)
    [] r1–start1 & (r1–pc1=1) & (r1–coin1=1) & (counter1< range) → (counter1'=counter1+1)
            & (r1–pc1'=2) & (r1–coin1'=0);
    // decide tails
    [coin1–s1–p1] r1–start1 & (r1–pc1=2) & (counter1<=left) → (r1–pc1'=3) & (r1–coin1'=0);
    // decide heads
    [coin1–s1–p2] r1–start1 & (r1–pc1=2) & (counter1>=right) → (r1–pc1'=3) & (r1–coin1'=1);
    // flip again
    [] r1–start1 & (r1–pc1=2) & (counter1¿left) & (counter1¡right) → (r1–pc1'=0);
endmodule

module r1–coin2 = r1–coin1[r1–start1=r1–start2,r1–pc1=r1–pc2,r1–coin1=r1–coin2,coin1–s1–
start=coin1–s2–start,coin1–s1–p1=coin1–s2–p1,coin1–s1–p2=coin1–s2–p2,counter1=counter1]
endmodule

module r1–coin3 = r1–coin1[r1–start1=r1–start3,r1–pc1=r1–pc3,r1–coin1=r1–coin3,coin1–s1–
start=coin1–s3–start,coin1–s1–p1=coin1–s3–p1,coin1–s1–p2=coin1–s3–p2,counter1=counter1]
endmodule

const int MAX=3;
formula leaders–agree1 = (p1=1 | r1<max(r1,r2,r3)) & (p2=1 | r2<max(r1,r2,r3)) & (p3=1 |
r3<max(r1,r2,r3));
formula leaders–agree2 = (p1=2 | r1<max(r1,r2,r3)) & (p2=2 | r2<max(r1,r2,r3)) & (p3=2 |
r3<max(r1,r2,r3));
formula decide1 = leaders–agree1 & (p1=1 | r1<max(r1,r2,r3)-1) & (p2=1 | r2<max(r1,r2,r3)-1) &
(p3=1 | r3<max(r1,r2,r3)-1);
formula decide2 = leaders–agree2 & (p1=2 | r1<max(r1,r2,r3)-1) & (p2=2 | r2<max(r1,r2,r3)-1) &
(p3=2 | r3<max(r1,r2,r3)-1);

module process1
    s1 : [0..5]; // local state
    // 0 initialise/read registers
    // 1 finish reading registers (make a decision)
    // 1 warn of change
    // 2 enter shared coin protocol
```

```
    // 4 finished
    // 5 error (reached max round and cannot decide)
    r1 : [0..MAX]; // round of the process
    p1 : [0..2]; // preference (0 corresponds to null)

    // nondeterministic choice as to initial preference
    [] s1=0 & r1=0 → (p1'=1) & (r1'=1);
    [] s1=0 & r1=0 → (p1'=2) & (r1'=1);
    // read registers (currently does nothing because read vs from other processes
    [] s1=0 & r1>0 & r1<=MAX → (s1'=1);
    // make a decision
    [] s1=1 & decide1 → (s1'=4) & (p1'=1);
    [] s1=1 & decide2 → (s1'=4) & (p1'=2);
    [] s1=1 & r1<MAX & leaders–agree1 & !decide1 → (s1'=0) & (p1'=1) & (r1'=r1+1);
    [] s1=1 & r1<MAX & leaders–agree2 & !decide2 → (s1'=0) & (p1'=2) & (r1'=r1+1);
    [] s1=1 & r1<MAX & !(leaders–agree1 — leaders–agree2) → (s1'=2) & (p1'=0);
    [] s1=1 & r1=MAX & !(decide1 — decide2) → (s1'=5);
    // run out of rounds so error // enter the coin procotol for the current round
    [coin1–s1–start] s1=2 & r1=1 → (s1'=3);
    [coin2–s1–start] s1=2 & r1=2 → (s1'=3);
    // get response from the coin protocol
    [coin1–s1–p1] s1=3 & r1=1 → (s1'=0) & (p1'=1) & (r1'=r1+1);
    [coin1–s1–p2] s1=3 & r1=1 → (s1'=0) & (p1'=2) & (r1'=r1+1);
    [coin2–s1–p1] s1=3 & r1=2 → (s1'=0) & (p1'=1) & (r1'=r1+1);
    [coin2–s1–p2] s1=3 & r1=2 → (s1'=0) & (p1'=2) & (r1'=r1+1);
    // done so loop
    [done] s1=4 → true;
    [fail] s1=5 → true;
endmodule

module process2 = process1[ s1=s2,p1=p2,p2=p3,p3=p1,r1=r2,r2=r3,r3=r1,coin1–s1–start=coin1–
s2–start,coin2–s1–start=coin2–s2–start,coin1–s1–p1=coin1–s2–p1,coin2–s1–p1=coin2–s2–p1,coin1–s1–
p2=coin1–s2–p2,coin2–s1–p2=coin2–s2–p2 ]
endmodule

module process3 = process1[ s1=s3,p1=p3,p2=p1,p3=p2,r1=r3,r2=r1,r3=r2,coin1–s1–start=coin1–
s3–start,coin2–s1–start=coin2–s3–start,coin1–s1–p1=coin1–s3–p1,coin2–s1–p1=coin2–s3–p1,coin1–s1–
p2=coin1–s3–p2,coin2–s1–p2=coin2–s3–p2 ]
endmodule

module failure
    z : [0..1] init 0;
    [fail] z=0 → (z'=1);
    [fail] z=1 → true;
```

endmodule

label "err" = z=1;

Sensor Network with $N = 3$

---

mdp

// probability of successful send (from channel to processor)
const double p–succ = 0.8;
// probability of successful receive of noack (from channel to sensor)
const double p–succ1 = 1;

module sensor1
y1 : [0..6] init 0; *// local state*
msg1 : [1..3] init 1; *// message code: 1=data, 2=alert, 3=stop*

// send data packets (msg=1)
[s1–to–c] y1=0 → (y1'=1);
[ack1] y1=1 → (y1'=0);
[noack1] y1=1 → (y1'=0);
// detect a problem and start to send warning/error messages
[detect1] y1=0 → (y1'=2) & (msg1'=2);
// send warning message
[s1–to–c] y1=2 → (y1'=3);
[ack1] y1=3 → (y1'=4) & (msg1'=3);
[noack1] y1=3 → p–succ1:(y1'=2) + (1-p–succ1):(y1'=4) & (msg1'=3);
// send error message
[s1–to–c] y1=4 → (y1'=5);
[ack1] y1=5 → (y1'=6);
[noack1] y1=5 → p–succ1:(y1'=4) + (1-p–succ1):(y1'=6);
// stop (after error)
[] y1=6 → true;
endmodule

module sensor2 = sensor1 [y1=y2, msg1=msg2, detect1=detect2, s1–to–c=s2–to–c, ack1=ack2, noack1=noack2]
endmodule

module sensor3 = sensor1 [y1=y3, msg1=msg3, detect1=detect3, s1–to–c=s3–to–c, ack1=ack3, noack1=noack3]
endmodule

```
module channel
    // local state: 0=IDLE, 1=STORE, 2=SEND
    c : [0..3] init 0;
    // who is currently sending to the channel
    who: [1..3];
    // what message is currently being sent onwards
    what: [0..3];
    // buffer of messages to be sent
    // (see sensor for message codes)
    cb1 : [0..3];
    cb2 : [0..3];
    // buffer size
    cbs : [0..2];

    // IDLE: either...
    // receive incoming message
    [s1–to–c] c=0 → (c'=1) & (who'=1);
    [s2–to–c] c=0 → (c'=1) & (who'=2);
    [s3–to–c] c=0 → (c'=1) & (who'=3);
    // or choose to process next message in buffer
    [process] c=0 & cbs¿0 → (c'=2);

    // STORE: send ack and then store in buffer
    [ack1] c=1 & who=1 & cbs=0 → (c'=0)&(cb1'=msg1)&(cbs'=1);
    [ack1] c=1 & who=1 & cbs=1 → (c'=0)&(cb2'=msg1)&(cbs'=2);
    [noack1] c=1 & who=1 & cbs=2 → (c'=0);
    [ack2] c=1 & who=2 & cbs=0 → (c'=0)&(cb1'=msg2)&(cbs'=1);
    [ack2] c=1 & who=2 & cbs=1 → (c'=0)&(cb2'=msg2)&(cbs'=2);
    [noack2] c=1 & who=2 & cbs=2 → (c'=0);
    [ack3] c=1 & who=3 & cbs=0 → (c'=0)&(cb1'=msg3)&(cbs'=1);
    [ack3] c=1 & who=3 & cbs=1 → (c'=0)&(cb2'=msg3)&(cbs'=2);
    [noack3] c=1 & who=3 & cbs=2 → (c'=0);

    // SEND: send next message in buffer
    // first step of send (extraction from buffer, possible fail)
    [extract] c=2 & cbs=1 → p–succ:(c'=3)&(what'=cb1)&(cbs'=0)&(cb1'=0)
            + (1-p–succ):(c'=0)&(cbs'=0)&(cb1'=0);
    [extract] c=2 & cbs=2 → p–succ:(c'=3)&(what'=cb1)&(cbs'=1)&(cb1'=cb2)&(cb2'=0)
            + (1-p–succ):(c'=0)&(cbs'=1)&(cb1'=cb2)&(cb2'=0);
    // second step of send (actual communication)
    [data] c=3 & what=1 → (c'=0);
    [alert] c=3 & what=2 → (c'=0);
    [stop] c=3 & what=3 → (c'=0);
endmodule
```

```
const double p–chunk = 0.01;
const int K = 10; // number of chunks of data

// processor receiving data from sensors
module proc

    x : [0..2] init 0; // local state
    count : [0..10] init 0; // number of chunks to process

    // receive (K chunks of) data
    [data] x=0 → (count'=K);
    // process one chunk of data
    [work] x=0 & count>0 → (count'=count-1);
    // receive alert, discard unprocessed data
    [alert] x=0 → (count'=0);
    // receive stop signal (probability of crash depends on number of remaining chunks)
    [stop] x=0 → (1-count*p–chunk):(x'=1) + (count*p–chunk):(x'=2);
    // stopped
    [done] x=1 → true;
    // crashed
    [crash] x=2 → true;
endmodule

module failure
    z : [0..1] init 0;
    [crash] z=0 → (z'=1);
    [crash] z=1 → true;
endmodule
label "err" = z=1;
```

Mars Exploration Rovers with $N = 5, R = 5$

```
mdp

const double p=0.9;
const int n=1;


module arbiter
    s:[-1..5] init 0;
    r:[0..5] init 0;
```

```
k:[0..n] init 0;

commUser:[0..5] init 0;
driveUser:[0..5] init 0;
panCamUser:[0..5] init 0;
armUser:[0..5] init 0;
ratUser:[0..5] init 0;

gc: bool init false; // grant comm
dc: bool init false; // deny comm
gd: bool init false; // grant drive
dd: bool init false; // deny drive
ga: bool init false; // grant arm
da: bool init false; // deny arm
gr: bool init false; // grant rat
dr: bool init false; // deny rat
gp: bool init false; // grant PanCam
dp: bool init false; // deny PanCam

// request comm
[u1–request–comm] s=0 & commUser=0 & driveUser=0 → (s'=1) & (gc'=true);
[u1–request–comm] s=0 & commUser! =0 → (s'=1) & (dc'=true);
[u1–request–comm] s=0 & commUser=0 & driveUser! =0 & k<n → p:(s'=1)
        & (r'=driveUser) & (k'=k+1)+(1-p):(s'=-1) & (gc'=true) & (k'=k+1);
[u1–request–comm] s=0 & commUser=0 & driveUser! =0 & k=n → (s'=1) & (r'=driveUser);
[u1–grant–comm] s=1 & commUser=0 & driveUser=0 & gc & !gd → (s'=0)
        & (commUser'=1) & (gc'=false);
[u1–grant–comm] s=-1 & gc → (s'=0) & (commUser'=1) & (gc'=false);
[u1–deny–comm] s=1 & dc → (s'=0) & (dc'=false);
[] s=1 → true;

[u2–request–comm] s=0 & commUser=0 & driveUser=0 → (s'=2) & (gc'=true);
[u2–request–comm] s=0 & commUser! =0 → (s'=2) & (dc'=true);
[u2–request–comm] s=0 & commUser=0 & driveUser! =0 → (s'=2) & (r'=driveUser);
[u2–grant–comm] s=2 & commUser=0 & driveUser=0 & gc & !gd → (s'=0)
        & (commUser'=2) & (gc'=false);
[u2–deny–comm] s=2 & dc → (s'=0) & (dc'=false);
[] s=2 → true;

[u3–request–comm] s=0 & commUser=0 & driveUser=0 → (s'=3) & (gc'=true);
[u3–request–comm] s=0 & commUser! =0 → (s'=3) & (dc'=true);
[u3–request–comm] s=0 & commUser=0 & driveUser! =0 → (s'=3) & (r'=driveUser);
[u3–grant–comm] s=3 & commUser=0 & driveUser=0 & gc & !gd → (s'=0)
        & (commUser'=3) & (gc'=false);
```

```
[u3–deny–comm] s=3 & dc → (s'=0) & (dc'=false);
[] s=3 → true;

[u4–request–comm] s=0 & commUser=0 & driveUser=0 → (s'=4) & (gc'=true);
[u4–request–comm] s=0 & commUser! =0 → (s'=4) & (dc'=true);
[u4–request–comm] s=0 & commUser=0 & driveUser! =0 → (s'=4) & (r'=driveUser);
[u4–grant–comm] s=4 & commUser=0 & driveUser=0 & gc & !gd → (s'=0)
        & (commUser'=4) & (gc'=false);
[u4–deny–comm] s=4 & dc → (s'=0) & (dc'=false);
[] s=4 → true;

[u5–request–comm] s=0 & commUser=0 & driveUser=0 → (s'=5) & (gc'=true);
[u5–request–comm] s=0 & commUser! =0 → (s'=5) & (dc'=true);
[u5–request–comm] s=0 & commUser=0 & driveUser! =0 → (s'=5) & (r'=driveUser);
[u5–grant–comm] s=5 & commUser=0 & driveUser=0 & gc & !gd → (s'=0)
        & (commUser'=5) & (gc'=false);
[u5–deny–comm] s=5 & dc → (s'=0) & (dc'=false);
[] s=5 → true;

// rescind drive
[u1–rescind–drive] r=1 & driveUser=1 → (r'=0) & (gc'=true);
[u2–rescind–drive] r=2 & driveUser=2 → (r'=0) & (gc'=true);
[u3–rescind–drive] r=3 & driveUser=3 → (r'=0) & (gc'=true);
[u4–rescind–drive] r=4 & driveUser=4 → (r'=0) & (gc'=true);
[u5–rescind–drive] r=5 & driveUser=5 → (r'=0) & (gc'=true);
[] r! =0 & driveUser=0 → (r'=0) & (gc'=true);

// request drive
[u1–request–drive] s=0 & commUser=0 & driveUser=0 → (s'=1) & (gd'=true);
[u1–grant–drive] s=1 & driveUser=0 & gd → (s'=0) & (driveUser'=1) & (gd'=false);
[u1–request–drive] s=0 & (commUser! =0 — driveUser! =0) → (s'=1) & (dd'=true);
[u1–deny–drive] s=1 & dd → (s'=0) & (dd'=false);

[u2–request–drive] s=0 & commUser=0 & driveUser=0 → (s'=2) & (gd'=true);
[u2–grant–drive] s=2 & driveUser=0 & gd → (s'=0) & (driveUser'=2) & (gd'=false);
[u2–request–drive] s=0 & (commUser! =0 — driveUser! =0) → (s'=2) & (dd'=true);
[u2–deny–drive] s=2 & dd → (s'=0) & (dd'=false);

[u3–request–drive] s=0 & commUser=0 & driveUser=0 → (s'=3) & (gd'=true);
[u3–grant–drive] s=3 & driveUser=0 & gd → (s'=0) & (driveUser'=3) & (gd'=false);
[u3–request–drive] s=0 & (commUser! =0 — driveUser! =0) → (s'=3) & (dd'=true);
[u3–deny–drive] s=3 & dd → (s'=0) & (dd'=false);

[u4–request–drive] s=0 & commUser=0 & driveUser=0 → (s'=4) & (gd'=true);
```

[u4–grant–drive] s=4 & driveUser=0 & gd → (s'=0) & (driveUser'=4) & (gd'=false);
[u4–request–drive] s=0 & (commUser! =0 — driveUser! =0) → (s'=4) & (dd'=true);
[u4–deny–drive] s=4 & dd → (s'=0) & (dd'=false);

[u5–request–drive] s=0 & commUser=0 & driveUser=0 → (s'=5) & (gd'=true);
[u5–grant–drive] s=5 & driveUser=0 & gd → (s'=0) & (driveUser'=5) & (gd'=false);
[u5–request–drive] s=0 & (commUser! =0 — driveUser! =0) → (s'=5) & (dd'=true);
[u5–deny–drive] s=5 & dd → (s'=0) & (dd'=false);

// request arm
[u1–request–arm] s=0 & armUser=0 & ratUser=0 & panCamUser=0 → (s'=1) & (ga'=true);
[u1–grant–arm] s=1 & armUser=0 & ga → (s'=0) & (armUser'=1);
[u1–request–arm] s=0 & (armUser! =0 — ratUser! =0 — panCamUser! =0) → (s'=1) & (da'=true);
[u1–deny–arm] s=1 & da → (s'=0) & (da'=false);

[u2–request–arm] s=0 & armUser=0 & ratUser=0 & panCamUser=0 → (s'=2) & (ga'=true);
[u2–grant–arm] s=2 & armUser=0 & ga → (s'=0) & (armUser'=2);
[u2–request–arm] s=0 & (armUser! =0 — ratUser! =0 — panCamUser! =0) → (s'=2) & (da'=true);
[u2–deny–arm] s=2 & da → (s'=0) & (da'=false);

[u3–request–arm] s=0 & armUser=0 & ratUser=0 & panCamUser=0 → (s'=3) & (ga'=true);
[u3–grant–arm] s=3 & armUser=0 & ga → (s'=0) & (armUser'=3);
[u3–request–arm] s=0 & (armUser! =0 — ratUser! =0 — panCamUser! =0) → (s'=3) & (da'=true);
[u3–deny–arm] s=3 & da → (s'=0) & (da'=false);

[u4–request–arm] s=0 & armUser=0 & ratUser=0 & panCamUser=0 → (s'=4) & (ga'=true);
[u4–grant–arm] s=4 & armUser=0 & ga → (s'=0) & (armUser'=4);
[u4–request–arm] s=0 & (armUser! =0 — ratUser! =0 — panCamUser! =0) → (s'=4) & (da'=true);
[u4–deny–arm] s=4 & da → (s'=0) & (da'=false);

[u5–request–arm] s=0 & armUser=0 & ratUser=0 & panCamUser=0 → (s'=5) & (ga'=true);
[u5–grant–arm] s=5 & armUser=0 & ga → (s'=0) & (armUser'=5);
[u5–request–arm] s=0 & (armUser! =0 — ratUser! =0 — panCamUser! =0) → (s'=5) & (da'=true);
[u5–deny–arm] s=5 & da → (s'=0) & (da'=false);

// request rat
[u1–request–rat] s=0 & commUser! =0 & armUser=0 & ratUser=0 → (s'=1) & (gr'=true);
[u1–grant–rat] s=1 & ratUser=0 & gr → (s'=0) & (ratUser'=1);
[u1–request–rat] s=0 & (commUser=0 — armUser! =0 — ratUser! =0) → (s'=1) & (dr'=true);
[u1–deny–rat] s=1 & dr → (s'=0) & (dr'=false);

[u2–request–rat] s=0 & commUser! =0 & armUser=0 & ratUser=0 → (s'=2) & (gr'=true);
[u2–grant–rat] s=2 & ratUser=0 & gr → (s'=0) & (ratUser'=2);
[u2–request–rat] s=0 & (commUser=0 — armUser! =0 — ratUser! =0) → (s'=2) & (dr'=true);

[u2–deny–rat] s=2 & dr → (s'=0) & (dr'=false);

[u3–request–rat] s=0 & commUser! =0 & armUser=0 & ratUser=0 → (s'=3) & (gr'=true);
[u3–grant–rat] s=3 & ratUser=0 & gr → (s'=0) & (ratUser'=3);
[u3–request–rat] s=0 & (commUser=0 — armUser! =0 — ratUser! =0) → (s'=3) & (dr'=true);
[u3–deny–rat] s=3 & dr → (s'=0) & (dr'=false);

[u4–request–rat] s=0 & commUser! =0 & armUser=0 & ratUser=0 → (s'=4) & (gr'=true);
[u4–grant–rat] s=4 & ratUser=0 & gr → (s'=0) & (ratUser'=4);
[u4–request–rat] s=0 & (commUser=0 — armUser! =0 — ratUser! =0) → (s'=4) & (dr'=true);
[u4–deny–rat] s=4 & dr → (s'=0) & (dr'=false);

[u5–request–rat] s=0 & commUser! =0 & armUser=0 & ratUser=0 → (s'=5) & (gr'=true);
[u5–grant–rat] s=5 & ratUser=0 & gr → (s'=0) & (ratUser'=5);
[u5–request–rat] s=0 & (commUser=0 — armUser! =0 — ratUser! =0) → (s'=5) & (dr'=true);
[u5–deny–rat] s=5 & dr → (s'=0) & (dr'=false);

// request PamCam
[u1–request–pancam] s=0 & panCamUser=0 & armUser=0 → (s'=1) & (gp'=true);
[u1–grant–pancam] s=1 & panCamUser=0 & gp → (s'=0) & (panCamUser'=1) & (gp'=false);
[u1–request–pancam] s=0 & (panCamUser! =0 — armUser! =0) → (s'=1) & (dp'=true);
[u1–deny–pancam] s=1 & dp → (s'=0) & (dp'=false);

[u2–request–pancam] s=0 & panCamUser=0 & armUser=0 → (s'=2) & (gp'=true);
[u2–grant–pancam] s=2 & panCamUser=0 & gp → (s'=0) & (panCamUser'=2) & (gp'=false);
[u2–request–pancam] s=0 & (panCamUser! =0 — armUser! =0) → (s'=2) & (dp'=true);
[u2–deny–pancam] s=2 & dp → (s'=0) & (dp'=false);

[u3–request–pancam] s=0 & panCamUser=0 & armUser=0 → (s'=3) & (gp'=true);
[u3–grant–pancam] s=3 & panCamUser=0 & gp → (s'=0) & (panCamUser'=3) & (gp'=false);
[u3–request–pancam] s=0 & (panCamUser! =0 — armUser! =0) → (s'=3) & (dp'=true);
[u3–deny–pancam] s=3 & dp → (s'=0) & (dp'=false);

[u4–request–pancam] s=0 & panCamUser=0 & armUser=0 → (s'=4) & (gp'=true);
[u4–grant–pancam] s=4 & panCamUser=0 & gp → (s'=0) & (panCamUser'=4) & (gp'=false);
[u4–request–pancam] s=0 & (panCamUser! =0 — armUser! =0) → (s'=4) & (dp'=true);
[u4–deny–pancam] s=4 & dp → (s'=0) & (dp'=false);

[u5–request–pancam] s=0 & panCamUser=0 & armUser=0 → (s'=5) & (gp'=true);
[u5–grant–pancam] s=5 & panCamUser=0 & gp → (s'=0) & (panCamUser'=5) & (gp'=false);
[u5–request–pancam] s=0 & (panCamUser! =0 — armUser! =0) → (s'=5) & (dp'=true);
[u5–deny–pancam] s=5 & dp → (s'=0) & (dp'=false);

// cancel comm

[u1–cancel–comm] commUser=1 → (commUser'=0);
[u2–cancel–comm] commUser=2 → (commUser'=0);
[u3–cancel–comm] commUser=3 → (commUser'=0);
[u4–cancel–comm] commUser=4 → (commUser'=0);
[u5–cancel–comm] commUser=5 → (commUser'=0);

// cancel drive
[u1–cancel–drive] driveUser=1 → (driveUser'=0);
[u2–cancel–drive] driveUser=2 → (driveUser'=0);
[u3–cancel–drive] driveUser=3 → (driveUser'=0);
[u4–cancel–drive] driveUser=4 → (driveUser'=0);
[u5–cancel–drive] driveUser=5 → (driveUser'=0);

// cancel arm
[u1–cancel–arm] armUser=1 → (armUser'=0);
[u2–cancel–arm] armUser=2 → (armUser'=0);
[u3–cancel–arm] armUser=3 → (armUser'=0);
[u4–cancel–arm] armUser=4 → (armUser'=0);
[u5–cancel–arm] armUser=5 → (armUser'=0);

// cancel rat
[u1–cancel–rat] ratUser=1 → (ratUser'=0);
[u2–cancel–rat] ratUser=2 → (ratUser'=0);
[u3–cancel–rat] ratUser=3 → (ratUser'=0);
[u4–cancel–rat] ratUser=4 → (ratUser'=0);
[u5–cancel–rat] ratUser=5 → (ratUser'=0);

// cancel PanCam
[u1–cancel–pancam] panCamUser=1 → (panCamUser'=0);
[u2–cancel–pancam] panCamUser=2 → (panCamUser'=0);
[u3–cancel–pancam] panCamUser=3 → (panCamUser'=0);
[u4–cancel–pancam] panCamUser=4 → (panCamUser'=0);
[u5–cancel–pancam] panCamUser=5 → (panCamUser'=0);
endmodule

module user2 = user1 [s1=s2, r1=r2, rescind1=rescind2, u1–request–comm=u2–request–comm, u1–request–drive=u2–request–drive, u1–request–pancam=u2–request–pancam, u1–request–arm=u2–request–arm, u1–request–rat=u2–request–rat, u1–grant–comm=u2–grant–comm, u1–grant–drive=u2–grant–drive, u1–grant–pancam=u2–grant–pancam, u1–grant–arm=u2–grant–arm, u1–grant–rat=u2–grant–rat, u1–deny–comm=u2–deny–comm, u1–deny–drive=u2–deny–drive, u1–deny–pancam=u2–deny–pancam, u1–deny–arm=u2–deny–arm, u1–deny–rat=u2–deny–rat, u1–rescind–comm=u2–rescind–comm, u1–rescind–drive=u2–rescind–drive, u1–rescind–pancam=u2–rescind–pancam, u1–rescind–arm=u2–rescind–arm, u1–rescind–rat=u2–rescind–rat, u1–cancel–comm=u2–cancel–comm, u1–cancel–drive=u2–cancel–drive, u1–cancel–pancam=u2–cancel–pancam, u1–cancel–arm=u2–cancel–

arm, u1–cancel–rat=u2–cancel–rat]
endmodule

module user3 = user1 [s1=s3, r1=r3, rescind1=rescind3, u1–request–comm=u3–request–comm, u1–request–drive=u3–request–drive, u1–request–pancam=u3–request–pancam, u1–request–arm=u3–request–arm, u1–request–rat=u3–request–rat, u1–grant–comm=u3–grant–comm, u1–grant–drive=u3–grant–drive, u1–grant–pancam=u3–grant–pancam, u1–grant–arm=u3–grant–arm, u1–grant–rat=u3–grant–rat, u1–deny–comm=u3–deny–comm, u1–deny–drive=u3–deny–drive, u1–deny–pancam=u3–deny–pancam, u1–deny–arm=u3–deny–arm, u1–deny–rat=u3–deny–rat, u1–rescind–comm=u3–rescind–comm, u1–rescind–drive=u3–rescind–drive, u1–rescind–pancam=u3–rescind–pancam, u1–rescind–arm=u3–rescind–arm, u1–rescind–rat=u3–rescind–rat, u1–cancel–comm=u3–cancel–comm, u1–cancel–drive=u3–cancel–drive, u1–cancel–pancam=u3–cancel–pancam, u1–cancel–arm=u3–cancel–arm, u1–cancel–rat=u3–cancel–rat]
endmodule

module user4 = user1 [s1=s4, r1=r4, rescind1=rescind4, u1–request–comm=u4–request–comm, u1–request–drive=u4–request–drive, u1–request–pancam=u4–request–pancam, u1–request–arm=u4–request–arm, u1–request–rat=u4–request–rat, u1–grant–comm=u4–grant–comm, u1–grant–drive=u4–grant–drive, u1–grant–pancam=u4–grant–pancam, u1–grant–arm=u4–grant–arm, u1–grant–rat=u4–grant–rat, u1–deny–comm=u4–deny–comm, u1–deny–drive=u4–deny–drive, u1–deny–pancam=u4–deny–pancam, u1–deny–arm=u4–deny–arm, u1–deny–rat=u4–deny–rat, u1–rescind–comm=u4–rescind–comm, u1–rescind–drive=u4–rescind–drive, u1–rescind–pancam=u4–rescind–pancam, u1–rescind–arm=u4–rescind–arm, u1–rescind–rat=u4–rescind–rat, u1–cancel–comm=u4–cancel–comm, u1–cancel–drive=u4–cancel–drive, u1–cancel–pancam=u4–cancel–pancam, u1–cancel–arm=u4–cancel–arm, u1–cancel–rat=u4–cancel–rat]
endmodule

module user5 = user1 [s1=s5, r1=r5, rescind1=rescind5, u1–request–comm=u5–request–comm, u1–request–drive=u5–request–drive, u1–request–pancam=u5–request–pancam, u1–request–arm=u5–request–arm, u1–request–rat=u5–request–rat, u1–grant–comm=u5–grant–comm, u1–grant–drive=u5–grant–drive, u1–grant–pancam=u5–grant–pancam, u1–grant–arm=u5–grant–arm, u1–grant–rat=u5–grant–rat, u1–deny–comm=u5–deny–comm, u1–deny–drive=u5–deny–drive, u1–deny–pancam=u5–deny–pancam, u1–deny–arm=u5–deny–arm, u1–deny–rat=u5–deny–rat, u1–rescind–comm=u5–rescind–comm, u1–rescind–drive=u5–rescind–drive, u1–rescind–pancam=u5–rescind–pancam, u1–rescind–arm=u5–rescind–arm, u1–rescind–rat=u5–rescind–rat, u1–cancel–comm=u5–cancel–comm, u1–cancel–drive=u5–cancel–drive, u1–cancel–pancam=u5–cancel–pancam, u1–cancel–arm=u5–cancel–arm, u1–cancel–rat=u5–cancel–rat]
endmodule

module property
e:[0..3] init 0;
u:[0..5] init 0;

[u1–grant–comm] e=0 → (e'=1) & (u'=1);
[u1–grant–drive] e=0 → (e'=2) & (u'=1);

[u1–cancel–comm] e=0 → true;
[u1–cancel–drive] e=0 → true;
[u2–grant–comm] e=0 → (e'=1) & (u'=2);
[u2–grant–drive] e=0 → (e'=2) & (u'=2);
[u2–cancel–comm] e=0 → true;
[u2–cancel–drive] e=0 → true;
[u3–grant–comm] e=0 → (e'=1) & (u'=3);
[u3–grant–drive] e=0 → (e'=2) & (u'=3);
[u3–cancel–comm] e=0 → true;
[u3–cancel–drive] e=0 → true;
[u4–grant–comm] e=0 → (e'=1) & (u'=4);
[u4–grant–drive] e=0 → (e'=2) & (u'=4);
[u4–cancel–comm] e=0 → true;
[u4–cancel–drive] e=0 → true;
[u5–grant–comm] e=0 → (e'=1) & (u'=5);
[u5–grant–drive] e=0 → (e'=2) & (u'=5);
[u5–cancel–comm] e=0 → true;
[u5–cancel–drive] e=0 → true;

[u1–cancel–comm] e=1 & u=1 → (e'=0);
[u1–grant–drive] e=1 → (e'=3);
[u1–cancel–drive] e=1 → true;
[u1–grant–comm] e=1 → true;
[u2–cancel–comm] e=1 & u=2 → (e'=0);
[u2–grant–drive] e=1 → (e'=3);
[u2–cancel–drive] e=1 → true;
[u2–grant–comm] e=1 → true;
[u3–cancel–comm] e=1 & u=3 → (e'=0);
[u3–grant–drive] e=1 → (e'=3);
[u3–cancel–drive] e=1 → true;
[u3–grant–comm] e=1 → true;
[u4–cancel–comm] e=1 & u=4 → (e'=0);
[u4–grant–drive] e=1 → (e'=3);
[u4–cancel–drive] e=1 → true;
[u4–grant–comm] e=1 → true;
[u5–cancel–comm] e=1 & u=5 → (e'=0);
[u5–grant–drive] e=1 → (e'=3);
[u5–cancel–drive] e=1 → true;
[u5–grant–comm] e=1 → true;

[u1–cancel–drive] e=2 & u=1 → (e'=0);
[u1–grant–comm] e=2 → (e'=3);
[u1–cancel–comm] e=2 → true;
[u1–grant–drive] e=2 → true;

```
    [u2–cancel–drive] e=2 & u=2 → (e'=0);
    [u2–grant–comm] e=2 → (e'=3);
    [u2–cancel–comm] e=2 → true;
    [u2–grant–drive] e=2 → true;
    [u3–cancel–drive] e=2 & u=3 → (e'=0);
    [u3–grant–comm] e=2 → (e'=3);
    [u3–cancel–comm] e=2 → true;
    [u3–grant–drive] e=2 → true;
    [u4–cancel–drive] e=2 & u=4 → (e'=0);
    [u4–grant–comm] e=2 → (e'=3);
    [u4–cancel–comm] e=2 → true;
    [u4–grant–drive] e=2 → true;
    [u5–cancel–drive] e=2 & u=5 → (e'=0);
    [u5–grant–comm] e=2 → (e'=3);
    [u5–cancel–comm] e=2 → true;
    [u5–grant–drive] e=2 → true;

    [u1–grant–comm] e=3 → true;
    [u1–grant–drive] e=3 → true;
    [u1–cancel–comm] e=3 → true;
    [u1–cancel–drive] e=3 → true;
    [u2–grant–comm] e=3 → true;
    [u2–grant–drive] e=3 → true;
    [u2–cancel–comm] e=3 → true;
    [u2–cancel–drive] e=3 → true;
    [u3–grant–comm] e=3 → true;
    [u3–grant–drive] e=3 → true;
    [u3–cancel–comm] e=3 → true;
    [u3–cancel–drive] e=3 → true;
    [u4–grant–comm] e=3 → true;
    [u4–grant–drive] e=3 → true;
    [u4–cancel–comm] e=3 → true;
    [u4–cancel–drive] e=3 → true;
    [u5–grant–comm] e=3 → true;
    [u5–grant–drive] e=3 → true;
    [u5–cancel–comm] e=3 → true;
    [u5–cancel–drive] e=3 → true;
endmodule
label "err" = z=3;
```

Appendix D

Case Studies for Chapter 5

The followings are detailed models of case studies in Section 5.5, described in the PRISM modelling language, which are adapted from the benchmark case studies on the PRISM website (http://www.prismmodelchecker.org/). We only show the largest example of each case study. All models are verified against property: P =?[◇"err"].

Contract Signing (EGL) with $N = 7, L = 8$

---

dtmc
const int N=7; // *number of pairs of secrets the party sends*
const int L=8; // *number of bits in each secret*

module counter
b : [1..L]; // *counter for current bit to be send (used in phases 2 and 3)*
n : [0..max(N-1,1)]; // *counter as parties send N messages in a row*
phase : [0..5];// *phase of the protocol*
party : [1..2]; // *which party moves*
coin : [0..1];
// *1 first phase of the protocol (sending messages of the form OT(.,.,.,.)*
// *2 and 3 - second phase of the protocol (sending secretes 1..n and n+1..2n respectively)*
// *4 finished the protocol*

// // *FIRST PHASE*
[] phase=0 → 0.5:(coin'=0)&(phase'=1) + 0.5:(coin'=1)&(phase'=1);
[receiveB0] phase=1 & party=1 & coin=0 → (party'=2)&(phase'=0);
[receiveB1] phase=1 & party=1 & coin=1 → (party'=2)&(phase'=0);
[receiveA0] phase=1 & party=2 & coin=0 & n<N-1 → (party'=1)&(phase'=0) & (n'=n+1);
[receiveA1] phase=1 & party=2 & coin=1 & n<N-1 → (party'=1)&(phase'=0) & (n'=n+1);

```
    [receiveA0] phase=1 & party=2 & coin=0 & n=N-1 → (party'=1) & (phase'=2) & (n'=0);
    [receiveA1] phase=1 & party=2 & coin=1 & n=N-1 → (party'=1) & (phase'=2) & (n'=0);
    // SECOND AND THIRD PHASES
    [receiveB] ((phase)¿=(2)&(phase)<=(3))& party=1 & n=0→ (party'=2);
    [receiveA] ((phase)¿=(2)&(phase)<=(3))& party=2 & n<N-1→ (n'=n+1);
    [receiveA] ((phase)¿=(2)&(phase)<=(3))& party=2 & n=N-1 → (party'=1) & (n'=1);
    [receiveB] ((phase)¿=(2)&(phase)<=(3))& party=1 & n<N-1 & n¿0 → (n'=n+1);
    [receiveB] ((phase)¿=(2)&(phase)<=(3))& party=1 & n=N-1 & b<L → (party'=1)
            & (n'=0) & (b'=b+1);
    [receiveB] phase=2 & party=1 & n=N-1 & b=L → (phase'=3) & (party'=1) & (n'=0) & (b'=1);
    [receiveB] phase=3 & party=1 & n=N-1 & b=L → (phase'=4);

    // FINISHED
    [] phase=4 → (phase'=4);
endmodule

module partyA
    // bi the number of bits of B's ith secret A knows
    // (keep pairs of secrets together to give a more structured model)

    b0 : [0..L]; b20 : [0..L];
    b1 : [0..L]; b21 : [0..L];
    b2 : [0..L]; b22 : [0..L];
    b3 : [0..L]; b23 : [0..L];
    b4 : [0..L]; b24 : [0..L];
    b5 : [0..L]; b25 : [0..L];
    b6 : [0..L]; b26 : [0..L];

    nA : [0..max(N-1,1)]; // counter as parties send N messages in a row
    phaseA : [1..5]; // phase of the protocol

    // first step (get either secret i or (N-1)+i with equal probability)
    [receiveA0] phaseA=1 & nA=0 → (b0'=L) & (nA'=nA+1);
    [receiveA1] phaseA=1 & nA=0 → (b20'=L) & (nA'=nA+1);
    [receiveA0] phaseA=1 & nA=1 → (b1'=L) & (nA'=nA+1);
    [receiveA1] phaseA=1 & nA=1 → (b21'=L) & (nA'=nA+1);
    [receiveA0] phaseA=1 & nA=2 → (b2'=L) & (nA'=nA+1);
    [receiveA1] phaseA=1 & nA=2 → (b22'=L) & (nA'=nA+1);
    [receiveA0] phaseA=1 & nA=3 → (b3'=L) & (nA'=nA+1);
    [receiveA1] phaseA=1 & nA=3 → (b23'=L) & (nA'=nA+1);
    [receiveA0] phaseA=1 & nA=4 → (b4'=L) & (nA'=nA+1);
    [receiveA1] phaseA=1 & nA=4 → (b24'=L) & (nA'=nA+1);
    [receiveA0] phaseA=1 & nA=5 → (b5'=L) & (nA'=nA+1);
    [receiveA1] phaseA=1 & nA=5 → (b25'=L) & (nA'=nA+1);
```

[receiveA0] phaseA=1 & nA=6 → (b6'=L) & (phaseA'=2) & (nA'=0);
[receiveA1] phaseA=1 & nA=6 → (b26'=L) & (phaseA'=2) & (nA'=0);

// second step (secrets 0,...,N-1)
[receiveA] phaseA=2 & nA=0 → (b0'=min(b0+1,L)) & (nA'=nA+1);
[receiveA] phaseA=2 & nA=1 → (b1'=min(b1+1,L)) & (nA'=nA+1);
[receiveA] phaseA=2 & nA=2 → (b2'=min(b2+1,L)) & (nA'=nA+1);
[receiveA] phaseA=2 & nA=3 → (b3'=min(b3+1,L)) & (nA'=nA+1);
[receiveA] phaseA=2 & nA=4 → (b4'=min(b4+1,L)) & (nA'=nA+1);
[receiveA] phaseA=2 & nA=5 → (b5'=min(b5+1,L)) & (nA'=nA+1);
[receiveA] phaseA=2 & nA=6 → (b6'=min(b6+1,L)) & (nA'=1);

// second step (secrets N,...,2N-1)
[receiveA] phaseA=3 & nA=0 → (b20'=min(b20+1,L)) & (nA'=nA+1);
[receiveA] phaseA=3 & nA=1 → (b21'=min(b21+1,L)) & (nA'=nA+1);
[receiveA] phaseA=3 & nA=2 → (b22'=min(b22+1,L)) & (nA'=nA+1);
[receiveA] phaseA=3 & nA=3 → (b23'=min(b23+1,L)) & (nA'=nA+1);
[receiveA] phaseA=3 & nA=4 → (b24'=min(b24+1,L)) & (nA'=nA+1);
[receiveA] phaseA=3 & nA=5 → (b25'=min(b25+1,L)) & (nA'=nA+1);
[receiveA] phaseA=3 & nA=6 → (b26'=min(b26+1,L)) & (nA'=1);
endmodule

module partyB=partyA [b0=a0, b1=a1, b2=a2, b3=a3, b4=a4, b5=a5, b6=a6, b20=a20, b21=a21, b22=a22, b23=a23, b24=a24, b25=a25, b26=a26, receiveA=receiveB, receiveA0=receiveB0, receiveA1=receiveB1, phaseA=phaseB, nA=nB]
endmodule

formula kB = ((a0=L & a20=L) | (a1=L & a21=L) | (a2=L & a22=L) | (a3=L & a23=L) | (a4=L & a24=L) | (a5=L & a25=L)x | (a6=L & a26=L));
formula kA = ((b0=L & b20=L) | (b1=L & b21=L) | (b2=L & b22=L) | (b3=L & b23=L) | (b4=L & b24=L) | (b5=L & b25=L) | (b6=L & b26=L));
label "err"= !kA&kB;

Contract Signing (EGL) with $N = 7, L = 8$

---

dtmc

const int N=64; *// number of chunks*
const int MAX=5; *// maximum number of retransmissions*
const double p = 0.98;
const double q = 0.99;

```
module receiver
    r : [0..5]; // 0: new file, 1: fst safe, 2: frame received, 3: frame reported, 4: idle, 5: resync
    rrep : [0..4]; // 0: bottom, 1: fst, 2: inc, 3: ok, 4: nok
    fr : bool;
    lr : bool;
    br : bool;
    r–ab : bool;
    l : [0..2];
    recv : bool;

    // new file
    [SyncWait1] (r=0) → (r'=0);
    [SyncWait0] (r=0) → (r'=0);
    [aG000] (r=0) → (r'=1) & (fr'=false) & (lr'=false) & (br'=false);
    [aG001] (r=0) → (r'=1) & (fr'=false) & (lr'=false) & (br'=true);
    [aG011] (r=0) → (r'=1) & (fr'=false) & (lr'=true) & (br'=true);
    [aG100] (r=0) → (r'=1) & (fr'=true) & (lr'=false) & (br'=false);
    // fst safe frame
    [] (r=1) → (r'=2) & (r–ab'=br);
    // frame received
    [] (r=2) & (r–ab=br) & (fr=true) & (lr=false) → (r'=3) & (rrep'=1);
    [] (r=2) & (r–ab=br) & (fr=false) & (lr=false) → (r'=3) & (rrep'=2);
    [] (r=2) & (r–ab=br) & (fr=false) & (lr=true) → (r'=3) & (rrep'=3);
    [aA] (r=2) & (l=0) & !(r–ab=br) → q : (r'=4) & (l'=1) + (1-q) : (r'=4) & (l'=2);
    // frame reported
    [aA] (r=3) & (l=0) → q : (r'=4) & (r–ab'=!r–ab) & (l'=1)
            + (1-q) : (r'=4) & (r–ab'=!r–ab) & (l'=2);
    // sending
    [aB] (r=4) & (l=1) → (l'=0);
    // lost
    [TO–Ack] (r=4) & (l=2) → (l'=0);
    // idle
    [aG000] (r=4) & (l=0) → (r'=2) & (fr'=false) & (lr'=false) & (br'=false);
    [aG001] (r=4) & (l=0) → (r'=2) & (fr'=false) & (lr'=false) & (br'=true);
    [aG011] (r=4) & (l=0) → (r'=2) & (fr'=false) & (lr'=true) & (br'=true);
    [aG100] (r=4) & (l=0) → (r'=2) & (fr'=true) & (lr'=false) & (br'=false);

    [SyncWait1] (r=4) & (l=0) → (r'=5);
    [SyncWait0] (r=4) & (l=0) → (r'=5) & (rrep'=4);
    // resync
    [SyncWait1] (r=5) → (r'=0) & (rrep'=0);
    [SyncWait0] (r=5) → (r'=0) & (rrep'=0);
```

endmodule

module sender

s : [0..7];

// 0: idle,1: next frame, 2: wait ack, 3: retransmit, 4: success, 5: error, 6: wait sync

srep : [0..3];

// 0: bottom, 1: not ok (nok), 2: do not know (dk), 3: ok (ok)

nrtr : [0..MAX];

i : [0..N];

bs : bool;

s–ab : bool;

fs : bool;

ls : bool;

k : [0..2];

// idle

[NewFile] (s=0) → (s'=1) & (i'=1) & (srep'=0);

// next frame

[aF] (s=1) & (k=0) → p : (s'=2) & (k'=1) & (fs'=(i=1)) & (ls'=(i=N)) & (bs'=s–ab) & (nrtr'=0)
+(1-p) : (s'=2) & (k'=2) & (fs'=(i=1)) & (ls'=(i=N)) & (bs'=s–ab) & (nrtr'=0);

// sending

[aG000] (s=2) & (k=1) & (fs=false) & (ls=false) & (bs=false) → (k'=0);

[aG001] (s=2) & (k=1) & (fs=false) & (ls=false) & (bs=true) → (k'=0);

[aG010] (s=2) & (k=1) & (fs=false) & (ls=true) & (bs=false) → (k'=0);

[aG011] (s=2) & (k=1) & (fs=false) & (ls=true) & (bs=true) → (k'=0);

[aG100] (s=2) & (k=1) & (fs=true) & (ls=false) & (bs=false) → (k'=0);

[aG101] (s=2) & (k=1) & (fs=true) & (ls=false) & (bs=true) → (k'=0);

[aG110] (s=2) & (k=1) & (fs=true) & (ls=true) & (bs=false) → (k'=0);

[aG111] (s=2) & (k=1) & (fs=true) & (ls=true) & (bs=true) → (k'=0);

// lost

[TO–Msg] (s=2) & (k=2) → (s'=3) & (k'=0);

// wait ack

[aB] (s=2) & (k=0) → (s'=4) & (s–ab'=!s–ab);

[TO–Ack] (s=2) & (k=0) → (s'=3);

// retransmit

[aF] (s=3) & (nrtr¡MAX) & (k=0) → p : (s'=2) & (k'=1) & (fs'=(i=1)) & (ls'=(i=N)) & (bs'=s–ab) & (nrtr'=nrtr+1)
+(1-p) : (s'=2) & (k'=2) & (fs'=(i=1)) & (ls'=(i=N)) & (bs'=s–ab) & (nrtr'=nrtr+1);

[] (s=3) & (nrtr=MAX) & (i¡N) → (s'=5) & (srep'=1);

[] (s=3) & (nrtr=MAX) & (i=N) → (s'=5) & (srep'=2);

// success

[] (s=4) & (i¡N) → (s'=1) & (i'=i+1);

[] (s=4) & (i=N) → (s'=0) & (srep'=3);

// error

```
    [SyncWait1] (s=5) & (ls=true) → (s'=6);
    [SyncWait0] (s=5) & (ls=false) → (s'=6);
    [error] s=6 → (s'=7);
    // wait sync
    [SyncWait1] (s=7) & (ls=true) → (s'=0) & (s–ab'=false);
    [SyncWait0] (s=7) & (ls=false) → (s'=0) & (s–ab'=false);
endmodule

module error
    T : bool;
    g :[0..1];

    [NewFile] (T=false) → (T'=true);
    [error] g=0 → (g'=1);
endmodule
label "err"= g=1;
```

Client-Server with $N = 4$

```
dtmc

const int N=4;
const int maxr=128;
const double p=0.8;
const double q=0.9;

module server
    s:[-1..3] init -1;
    i:[0..4];
    j:[0..4];
    n:[0..N*maxr];
    counter : bool init false;
    broken1 : bool init true;
    broken2 : bool init false;
    broken3 : bool init false;
    broken4 : bool init false;

    [] s=-1&!counter → p:(s'=0) + (1-p):(counter'=true);
    [idle] counter & (n<N*maxr-1) → (n'=n+1);
    [reset] counter & (n=N*maxr) → (n'=0);
```

```
    // initial cancel loops
    [client1–cancel] s=0 → true;
    [client2–cancel] s=0 → true;
    [client3–cancel] s=0 → true;
    [client4–cancel] s=0 → true;

    // client i request/grant/cancel
    [client1–request] s=0 → (s'=1) & (i'=1);
    [client1–grant] s=1 & i=1 → (s'=2);
    [client1–cancel] s=2 & i=1 → (s'=0) & (i'=0);
    [client2–request] s=0 → (s'=1) & (i'=2);
    [client2–grant] s=1 & i=2 → (s'=2);
    [client2–cancel] s=2 & i=2 → (s'=0) & (i'=0);
    [client3–request] s=0 → (s'=1) & (i'=3);
    [client3–grant] s=1 & i=3 → (s'=2);
    [client3–cancel] s=2 & i=3 → (s'=0) & (i'=0);
    [client4–request] s=0 → (s'=1) & (i'=4);
    [client4–grant] s=1 & i=4 → (s'=2);
    [client4–cancel] s=2 & i=4 → (s'=0) & (i'=0);

    // deny other requests when serving
    [client1–request] s=2&broken1 → q:(s'=3) & (j'=1)&(broken1'=false)
            + (1-q):(s'=1)&(i'=1)&(broken1'=false);
    [client1–request] s=2&!broken1 → (s'=3) & (j'=1);
    [client1–deny] s=3 & j=1 → (s'=2) & (j'=0);
    [client2–request] s=2&broken2 → q:(s'=3) & (j'=2)&(broken2'=false)
            + (1-q):(s'=1)&(i'=2)&(broken2'=false);
    [client2–request] s=2&!broken2 → (s'=3) & (j'=2);
    [client2–deny] s=3 & j=2 → (s'=2) & (j'=0);
    [client3–request] s=2&broken3 → q:(s'=3) & (j'=3)&(broken3'=false)
            + (1-q):(s'=1)&(i'=3)&(broken3'=false);
    [client3–request] s=2&!broken3 → (s'=3) & (j'=3);
    [client3–deny] s=3 & j=3 → (s'=2) & (j'=0);
    [client4–request] s=2&broken4 → q:(s'=3) & (j'=4)&(broken4'=false)
            + (1-q):(s'=1)&(i'=4)&(broken4'=false);
    [client4–request] s=2&!broken4 → (s'=3) & (j'=4);
    [client4–deny] s=3 & j=4 → (s'=2) & (j'=0);

    // cancel loops when serving
    [client1–cancel] s=2 & i!=1 → true;
    [client2–cancel] s=2 & i!=2 → true;
    [client3–cancel] s=2 & i!=3 → true;
    [client4–cancel] s=2 & i!=4 → true;
endmodule
```

```
module sched
    turn : [1..4] init 1;

    // outputs
    [client1–request] turn=1 → (turn'=1);
    [client1–wait] turn=1 → (turn'=2);
    [client1–deny] turn=1 → (turn'=2);
    [client1–grant] turn=1 → (turn'=2);
    [client1–useResource] turn=1 → (turn'=2);
    [client1–cancel] turn=1 → (turn'=2);

    // outputs
    [client2–request] turn=2 → (turn'=2);
    [client2–wait] turn=2 → (turn'=3);
    [client2–deny] turn=2 → (turn'=3);
    [client2–grant] turn=2 → (turn'=3);
    [client2–useResource] turn=2 → (turn'=3);
    [client2–cancel] turn=2 → (turn'=3);

    // outputs
    [client3–request] turn=3 → (turn'=3);
    [client3–wait] turn=3 → (turn'=4);
    [client3–deny] turn=3 → (turn'=4);
    [client3–grant] turn=3 → (turn'=4);
    [client3–useResource] turn=3 → (turn'=4);
    [client3–cancel] turn=3 → (turn'=4);

    [client4–request] turn=4 → (turn'=4);
    [client4–wait] turn=4 → (turn'=1);
    [client4–deny] turn=4 → (turn'=1);
    [client4–grant] turn=4 → (turn'=1);
    [client4–useResource] turn=4 → (turn'=1);
    [client4–cancel] turn=4 → (turn'=1);
endmodule

module client1
    c1:[-1..4] init -1;

    [client1–wait] c1=-1 → 0.5:(c1'=-1) + 0.5:(c1'=0);
    [client1–request] c1=0 → (c1'=1);
    [client1–deny] c1=1 → (c1'=-1);
    [client1–grant] c1=1 → (c1'=2);
    [client1–useResource] c1=2 → 0.5:(c1'=2) + 0.5:(c1'=3);
    [client1–cancel] c1=3 → (c1'=-1);
```

endmodule

module client2 = client1[c1=c2, client1–wait=client2–wait, client1–request=client2–request, client1–deny=client2–deny, client1–grant=client2–grant, client1–useResource=client2–useResource, client1–cancel=client2–cancel]
endmodule

module client3 = client1[c1=c3, client1–wait=client3–wait, client1–request=client3–request, client1–deny=client3–deny, client1–grant=client3–grant, client1–useResource=client3–useResource, client1–cancel=client3–cancel]
endmodule

module client4 = client1[c1=c4, client1–wait=client4–wait, client1–request=client4–request, client1–deny=client4–deny, client1–grant=client4–grant, client1–useResource=client4–useResource, client1–cancel=client4–cancel]
endmodule

module exclusion
e:[0..3];
k:[0..4];

[client1–grant] e=0 → (e'=1) & (k'=1);
[client1–cancel] e=1 & k=1 → (e'=0) & (k'=0);
[client2–grant] e=0 → (e'=1) & (k'=2);
[client2–cancel] e=1 & k=2 → (e'=0) & (k'=0);
[client3–grant] e=0 → (e'=1) & (k'=3);
[client3–cancel] e=1 & k=3 → (e'=0) & (k'=0);
[client4–grant] e=0 → (e'=1) & (k'=4);
[client4–cancel] e=1 & k=4 → (e'=0) & (k'=0);

[client1–cancel] e=0 → (e'=2) & (k'=0);
[client2–cancel] e=0 → (e'=2) & (k'=0);
[client3–cancel] e=0 → (e'=2) & (k'=0);
[client4–cancel] e=0 → (e'=2) & (k'=0);

[client1–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=1 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=1 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=2 → (e'=2) & (k'=0);

[client4–grant] e=1 & k=2 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=2 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=3 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client4–cancel] e=1 & k=3 → (e'=2) & (k'=0);
[client1–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client2–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client3–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client4–grant] e=1 & k=4 → (e'=2) & (k'=0);
[client1–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client2–cancel] e=1 & k=4 → (e'=2) & (k'=0);
[client3–cancel] e=1 & k=4 → (e'=2) & (k'=0);

[client1–grant] e=2 → true;
[client1–cancel] e=2 → true;
[client2–grant] e=2 → true;
[client2–cancel] e=2 → true;
[client3–grant] e=2 → true;
[client3–cancel] e=2 → true;
[client4–grant] e=2 → true;
[client4–cancel] e=2 → true;

endmodule

label "err"= e=2;

Appendix E

Case Studies for Chapter 6

The followings are detailed models of case studies in Section 6.5, described in the PRISM modelling language, which are adapted from the benchmark case studies on the PRISM website (http://www.prismmodelchecker.org/). We only show the largest example of each case study. All models are verified against property: P =?[◇"err"].

Contract Signing (EGL) with $N = 10, L = 8$

---

dtmc
const int N=10; // *number of pairs of secrets the party sends*
const int L=8; // *number of bits in each secret*

const int receiveA0=1;
const int receiveA1=2;
const int receiveB0=3;
const int receiveB1=4;
const int receiveA=5;
const int receiveB=6;

init
b=1 & n=0 & phase=0 & party=1 & action=0 & b0=0 & b1=0 & b2=0 & b3=0 & b4=0 & b5=0 & b6=0 & b7=0 & b8=0 & b9=0 & b20=0 & b21=0 & b22=0 & b23=0 & b24=0 & b25=0 & b26=0 & b27=0 & b28=0 & b29=0 & a0=0 & a1=0 & a2=0 & a3=0 & a4=0 & a5=0 & a6=0 & a7=0 & a8=0 & a9=0 & a20=0 & a21=0 & a22=0 & a23=0 & a24=0 & a25=0 & a26=0 & a27=0 & a28=0 & a29=0
endinit

module counter
    b : [1..L+1]; // *counter for current bit to be send (used in phases 2 and 3)*

```
    n : [0..max(N-1,1)]; // counter as parties send N messages in a row
    phase : [0..4]; // phase of the protocol
    // 1 first phase of the protocol (sending messages of the form OT(.,.,.,.)
    // 2 and 3 - second phase of the protocol (sending secretes 1..n and n+1..2n respectively)
    // 4 finished the protocol
    party : [1..2]; // which party moves
    action : [0..6];

    // FIRST PHASE
    [sync] phase=0 & party=1 → 0.5:(phase'=1)&(action'=receiveB0)
            + 0.5:(phase'=1)&(action'=receiveB1);
    [sync] phase=0 & party=2 → 0.5:(phase'=1)&(action'=receiveA0)
            + 0.5:(phase'=1)&(action'=receiveA1);
    [sync] action=receiveB0 & phase=1 & party=1 → (phase'=0) & (party'=2);
    [sync] action=receiveB1 & phase=1 & party=1 → (phase'=0) & (party'=2);
    [sync] action=receiveA0 & phase=1 & party=2 & n<N-1 → (phase'=0) & (party'=1) & (n'=n+1);
    [sync] action=receiveA1 & phase=1 & party=2 & n<N-1 → (phase'=0) & (party'=1) & (n'=n+1);
    [sync] action=receiveA0 & phase=1 & party=2 & n=N-1 → (phase'=2) & (party'=1)
            & (n'=0) & (action'=receiveB);
    [sync] action=receiveA1 & phase=1 & party=2 & n=N-1 → (phase'=2) & (party'=1)
            & (n'=0) & (action'=receiveB);

    // SECOND AND THIRD PHASES
    [sync] action=receiveB & ((phase)>=(2)&(phase)<=(3))& party=1 & n=0 → (party'=2)
            & (action'=receiveA);
    [sync] action=receiveA & ((phase)>=(2)&(phase)<=(3))& party=2 & n<N-1 → (n'=n+1);
    [sync] action=receiveA & ((phase)>=(2)&(phase)<=(3))& party=2 & n=N-1 → (party'=1)
            & (n'=1) & (action'=receiveB);
    [sync] action=receiveB & ((phase)>=(2)&(phase)<=(3))& party=1
            & n>0&n<N-1 → (n'=n+1);
    [sync] action=receiveB & ((phase)>=(2)&(phase)<=(3))& party=1
            & n=N-1 & b<L → (party'=1) & (n'=0) & (b'=b+1);
    [sync] action=receiveB & phase=2 & party=1 & n=N-1 & b=L → (phase'=3)
            & (party'=1) & (n'=0) & (b'=1);
    [sync] action=receiveB & phase=3 & party=1 & n=N-1 & b=L → (phase'=4);

    // FINISHED
    [sync] phase=4 → (phase'=4);
endmodule

module partyAB
    // bi the number of bits of B's ith secret A knows
    // (keep pairs of secrets together to give a more structured model)
    b0 : [0..L]; b20 : [0..L];
```

b1 : [0..L]; b21 : [0..L];
b2 : [0..L]; b22 : [0..L];
b3 : [0..L]; b23 : [0..L];
b4 : [0..L]; b24 : [0..L];
b5 : [0..L]; b25 : [0..L];
b6 : [0..L]; b26 : [0..L];
b7 : [0..L]; b27 : [0..L];
b8 : [0..L]; b28 : [0..L];
b9 : [0..L]; b29 : [0..L];
// *bi the number of bits of A's ith secret B knows*
// *(keep pairs of secrets together to give a more structured model)*
a0 : [0..L]; a20 : [0..L];
a1 : [0..L]; a21 : [0..L];
a2 : [0..L]; a22 : [0..L];
a3 : [0..L]; a23 : [0..L];
a4 : [0..L]; a24 : [0..L];
a5 : [0..L]; a25 : [0..L];
a6 : [0..L]; a26 : [0..L];
a7 : [0..L]; a27 : [0..L];
a8 : [0..L]; a28 : [0..L];
a9 : [0..L]; a29 : [0..L];

[sync] phase=0 → true;

// *FIRST PHASE*
[sync] action=receiveA0 & phase=1 & n=0 → (b0'=L);
[sync] action=receiveA0 & phase=1 & n=1 → (b1'=L);
[sync] action=receiveA0 & phase=1 & n=2 → (b2'=L);
[sync] action=receiveA0 & phase=1 & n=3 → (b3'=L);
[sync] action=receiveA0 & phase=1 & n=4 → (b4'=L);
[sync] action=receiveA0 & phase=1 & n=5 → (b5'=L);
[sync] action=receiveA0 & phase=1 & n=6 → (b6'=L);
[sync] action=receiveA0 & phase=1 & n=7 → (b7'=L);
[sync] action=receiveA0 & phase=1 & n=8 → (b8'=L);
[sync] action=receiveA0 & phase=1 & n=9 → (b9'=L);

[sync] action=receiveA1 & phase=1 & n=0 → (b20'=L);
[sync] action=receiveA1 & phase=1 & n=1 → (b21'=L);
[sync] action=receiveA1 & phase=1 & n=2 → (b22'=L);
[sync] action=receiveA1 & phase=1 & n=3 → (b23'=L);
[sync] action=receiveA1 & phase=1 & n=4 → (b24'=L);
[sync] action=receiveA1 & phase=1 & n=5 → (b25'=L);
[sync] action=receiveA1 & phase=1 & n=6 → (b26'=L);
[sync] action=receiveA1 & phase=1 & n=7 → (b27'=L);

[sync] action=receiveA1 & phase=1 & n=8 → (b28'=L);
[sync] action=receiveA1 & phase=1 & n=9 → (b29'=L);

[sync] action=receiveB0 & phase=1 & n=0 → (a0'=L);
[sync] action=receiveB0 & phase=1 & n=1 → (a1'=L);
[sync] action=receiveB0 & phase=1 & n=2 → (a2'=L);
[sync] action=receiveB0 & phase=1 & n=3 → (a3'=L);
[sync] action=receiveB0 & phase=1 & n=4 → (a4'=L);
[sync] action=receiveB0 & phase=1 & n=5 → (a5'=L);
[sync] action=receiveB0 & phase=1 & n=6 → (a6'=L);
[sync] action=receiveB0 & phase=1 & n=7 → (a7'=L);
[sync] action=receiveB0 & phase=1 & n=8 → (a8'=L);
[sync] action=receiveB0 & phase=1 & n=9 → (a9'=L);

[sync] action=receiveB1 & phase=1 & n=0 → (a20'=L);
[sync] action=receiveB1 & phase=1 & n=1 → (a21'=L);
[sync] action=receiveB1 & phase=1 & n=2 → (a22'=L);
[sync] action=receiveB1 & phase=1 & n=3 → (a23'=L);
[sync] action=receiveB1 & phase=1 & n=4 → (a24'=L);
[sync] action=receiveB1 & phase=1 & n=5 → (a25'=L);
[sync] action=receiveB1 & phase=1 & n=6 → (a26'=L);
[sync] action=receiveB1 & phase=1 & n=7 → (a27'=L);
[sync] action=receiveB1 & phase=1 & n=8 → (a28'=L);
[sync] action=receiveB1 & phase=1 & n=9 → (a29'=L);

// SECOND AND THIRD PHASE
[sync] action=receiveA & phase=2 & n=0 → (b0'=min(b0+1,L));
[sync] action=receiveA & phase=2 & n=1 → (b1'=min(b1+1,L));
[sync] action=receiveA & phase=2 & n=2 → (b2'=min(b2+1,L));
[sync] action=receiveA & phase=2 & n=3 → (b3'=min(b3+1,L));
[sync] action=receiveA & phase=2 & n=4 → (b4'=min(b4+1,L));
[sync] action=receiveA & phase=2 & n=5 → (b5'=min(b5+1,L));
[sync] action=receiveA & phase=2 & n=6 → (b6'=min(b6+1,L));
[sync] action=receiveA & phase=2 & n=7 → (b7'=min(b7+1,L));
[sync] action=receiveA & phase=2 & n=8 → (b8'=min(b8+1,L));
[sync] action=receiveA & phase=2 & n=9 → (b9'=min(b9+1,L));

[sync] action=receiveA & phase=3 & n=0 → (b20'=min(b20+1,L));
[sync] action=receiveA & phase=3 & n=1 → (b21'=min(b21+1,L));
[sync] action=receiveA & phase=3 & n=2 → (b22'=min(b22+1,L));
[sync] action=receiveA & phase=3 & n=3 → (b23'=min(b23+1,L));
[sync] action=receiveA & phase=3 & n=4 → (b24'=min(b24+1,L));
[sync] action=receiveA & phase=3 & n=5 → (b25'=min(b25+1,L));
[sync] action=receiveA & phase=3 & n=6 → (b26'=min(b26+1,L));

[sync] action=receiveA & phase=3 & n=7 → (b27'=min(b27+1,L));
[sync] action=receiveA & phase=3 & n=8 → (b28'=min(b28+1,L));
[sync] action=receiveA & phase=3 & n=9 → (b29'=min(b29+1,L));

[sync] action=receiveB & phase=2 & n=0 → (a0'=min(a0+1,L));
[sync] action=receiveB & phase=2 & n=1 → (a1'=min(a1+1,L));
[sync] action=receiveB & phase=2 & n=2 → (a2'=min(a2+1,L));
[sync] action=receiveB & phase=2 & n=3 → (a3'=min(a3+1,L));
[sync] action=receiveB & phase=2 & n=4 → (a4'=min(a4+1,L));
[sync] action=receiveB & phase=2 & n=5 → (a5'=min(a5+1,L));
[sync] action=receiveB & phase=2 & n=6 → (a6'=min(a6+1,L));
[sync] action=receiveB & phase=2 & n=7 → (a7'=min(a7+1,L));
[sync] action=receiveB & phase=2 & n=8 → (a8'=min(a8+1,L));
[sync] action=receiveB & phase=2 & n=9 → (a9'=min(a9+1,L));

[sync] action=receiveB & phase=3 & n=0 → (a20'=min(a20+1,L));
[sync] action=receiveB & phase=3 & n=1 → (a21'=min(a21+1,L));
[sync] action=receiveB & phase=3 & n=2 → (a22'=min(a22+1,L));
[sync] action=receiveB & phase=3 & n=3 → (a23'=min(a23+1,L));
[sync] action=receiveB & phase=3 & n=4 → (a24'=min(a24+1,L));
[sync] action=receiveB & phase=3 & n=5 → (a25'=min(a25+1,L));
[sync] action=receiveB & phase=3 & n=6 → (a26'=min(a26+1,L));
[sync] action=receiveB & phase=3 & n=7 → (a27'=min(a27+1,L));
[sync] action=receiveB & phase=3 & n=8 → (a28'=min(a28+1,L));
[sync] action=receiveB & phase=3 & n=9 → (a29'=min(a29+1,L));

[sync] phase=4 → true;

endmodule

formula kB = ((a0=L & a20=L) | (a1=L & a21=L) | (a2=L & a22=L) | (a3=L & a23=L) | (a4=L & a24=L) | (a5=L & a25=L) | (a6=L & a26=L) | (a7=L & a27=L) | (a8=L & a28=L) | (a9=L & a29=L));

formula kA = ((b0=L & b20=L) | (b1=L & b21=L) | (b2=L & b22=L) | (b3=L & b23=L) | (b4=L & b24=L) | (b5=L & b25=L) | (b6=L & b26=L) | (b7=L & b27=L) | (b8=L & b28=L) | (b9=L & b29=L));

label "err"= !kA&kB;

Client-Server with $N = 8$

---

dtmc

// client actions
const int wait=0;
const int request=1;
const int cancel=2;
const int useResource=3;

//server feedbacks
const int client1–deny=1;
const int client1–grant=2;
const int client2–deny=3;
const int client2–grant=4;
const int client3–deny=5;
const int client3–grant=6;
const int client4–deny=7;
const int client4–grant=8;
const int client5–deny=9;
const int client5–grant=10;
const int client6–deny=11;
const int client6–grant=12;
const int client7–deny=13;
const int client7–grant=14;
const int client8–deny=15;
const int client8–grant=16;

init
s=0 & feedback=0 & broken=true & error=false & turn=1 & c1=-1 & action1=0 & c2=-1 & action2=0 & c3=-1 & action3=0 & c4=-1 & action4=0 & c5=-1 & action5=0 & c6=-1 & action6=0 & c7=-1 & action7=0 & c8=-1 & action8=0
endinit

module server
s:[0..3];
feedback:[0..16];
broken: bool;
error: bool;

[sync] (action1=wait|action1=useResource) & turn=1 → true;
[sync] (action2=wait|action2=useResource) & turn=2 → true;
[sync] (action3=wait|action3=useResource) & turn=3 → true;
[sync] (action4=wait|action4=useResource) & turn=4 → true;
[sync] (action5=wait|action5=useResource) & turn=5 → true;
[sync] (action6=wait|action6=useResource) & turn=6 → true;
[sync] (action7=wait|action7=useResource) & turn=7 → true;
[sync] (action8=wait|action8=useResource) & turn=8 → true;

// client i request/grant/cancel
[sync] action1=request & turn=1 & s=0 → (s'=1) & (feedback'=client1–grant);
[sync] feedback=client1–grant & turn=1 & s=1 → (s'=2);
[sync] action1=cancel & turn=1 & s=2 → (s'=0);
[sync] action2=request & turn=2 & s=0 → (s'=1) & (feedback'=client2–grant);
[sync] feedback=client2–grant & turn=2 & s=1 → (s'=2);
[sync] action2=cancel & turn=2 & s=2 → (s'=0);
[sync] action3=request & turn=3 & s=0 → (s'=1) & (feedback'=client3–grant);
[sync] feedback=client3–grant & turn=3 & s=1 → (s'=2);
[sync] action3=cancel & turn=3 & s=2 → (s'=0);
[sync] action4=request & turn=4 & s=0 → (s'=1) & (feedback'=client4–grant);
[sync] feedback=client4–grant & turn=4 & s=1 → (s'=2);
[sync] action4=cancel & turn=4 & s=2 → (s'=0);
[sync] action5=request & turn=5 & s=0 → (s'=1) & (feedback'=client5–grant);
[sync] feedback=client5–grant & turn=5 & s=1 → (s'=2);
[sync] action5=cancel & turn=5 & s=2 → (s'=0);
[sync] action6=request & turn=6 & s=0 → (s'=1) & (feedback'=client6–grant);
[sync] feedback=client6–grant & turn=6 & s=1 → (s'=2);
[sync] action6=cancel & turn=6 & s=2 → (s'=0);
[sync] action7=request & turn=7 & s=0 → (s'=1) & (feedback'=client7–grant);
[sync] feedback=client7–grant & turn=7 & s=1 → (s'=2);
[sync] action7=cancel & turn=7 & s=2 → (s'=0);
[sync] action8=request & turn=8 & s=0 → (s'=1) & (feedback'=client8–grant);
[sync] feedback=client8–grant & turn=8 & s=1 → (s'=2);
[sync] action8=cancel & turn=8 & s=2 → (s'=0);

// deny other requests when serving
[sync] action1=request & turn=1 & s=2 & broken → 0.9:(s'=3) & (broken'=false) & (feedback'=client1–deny) + 0.1:(s'=1) & (broken'=false) & (feedback'=client1–grant) & (error'=true);
[sync] action1=request & turn=1 & s=2 & !broken → (s'=3) & (feedback'=client1–deny);
[sync] feedback=client1–deny & turn=1 & s=3 → (s'=2);
[sync] action2=request & turn=2 & s=2 → (s'=3) & (feedback'=client2–deny);
[sync] feedback=client2–deny & turn=2 & s=3 → (s'=2);
[sync] action3=request & turn=3 & s=2 → (s'=3) & (feedback'=client3–deny);
[sync] feedback=client3–deny & turn=3 & s=3 → (s'=2);
[sync] action4=request & turn=4 & s=2 → (s'=3) & (feedback'=client4–deny);
[sync] feedback=client4–deny & turn=4 & s=3 → (s'=2);
[sync] action5=request & turn=5 & s=2 → (s'=3) & (feedback'=client5–deny);
[sync] feedback=client5–deny & turn=5 & s=3 → (s'=2);
[sync] action6=request & turn=6 & s=2 → (s'=3) & (feedback'=client6–deny);
[sync] feedback=client6–deny & turn=6 & s=3 → (s'=2);
[sync] action7=request & turn=7 & s=2 → (s'=3) & (feedback'=client7–deny);
[sync] feedback=client7–deny & turn=7 & s=3 → (s'=2);

```
    [sync] action8=request & turn=8 & s=2 → (s'=3) & (feedback'=client8–deny);
    [sync] feedback=client8–deny & turn=8 & s=3 → (s'=2);
endmodule

module sched–client
    turn: [1..8];
    c1:[-1..4];
    action1: [0..3];
    c2:[-1..4];
    action2: [0..3];
    c3:[-1..4];
    action3: [0..3];
    c4:[-1..4];
    action4: [0..3];
    c5:[-1..4];
    action5: [0..3];
    c6:[-1..4];
    action6: [0..3];
    c7:[-1..4];
    action7: [0..3];
    c8:[-1..4];
    action8: [0..3];

    [sync] action1=wait & c1=-1 & turn=1 → 0.5:(c1'=-1) & (turn'=2)
            + 0.5:(c1'=0) & (turn'=2) & (action1'=request);
    [sync] action1=request & c1=0 & turn=1 → (c1'=1);
    [sync] feedback=client1–deny & c1=1 & turn=1 → (c1'=-1) & (turn'=2) & (action1'=wait);
    [sync] feedback=client1–grant & c1=1 & turn=1 → (c1'=2) & (turn'=2) & (action1'=useResource);
    [sync] action1=useResource & c1=2 & turn=1 → 0.5:(c1'=2) & (turn'=2)
            + 0.5:(c1'=3) & (turn'=2) & (action1'=cancel);
    [sync] action1=cancel & c1=3 & turn=1 → (c1'=-1) & (turn'=2) & (action1'=wait);
    [sync] action2=wait & c2=-1 & turn=2 → 0.5:(c2'=-1) & (turn'=3)
            + 0.5:(c2'=0) & (turn'=3) & (action2'=request);
    [sync] action2=request & c2=0 & turn=2 → (c1'=1);
    [sync] feedback=client2–deny & c2=1 & turn=2 → (c2'=-1) & (turn'=3) & (action2'=wait);
    [sync] feedback=client2–grant & c2=1 & turn=2 → (c2'=2) & (turn'=3) & (action2'=useResource);
    [sync] action2=useResource & c2=2 & turn=2 → 0.5:(c2'=2) & (turn'=3)
            + 0.5:(c2'=3) & (turn'=3) & (action2'=cancel);
    [sync] action2=cancel & c2=3 & turn=2 → (c2'=-1) & (turn'=3) & (action2'=wait);
    [sync] action3=wait & c3=-1 & turn=3 → 0.5:(c3'=-1) & (turn'=4)
            + 0.5:(c3'=0) & (turn'=4) & (action3'=request);
    [sync] action3=request & c3=0 & turn=3 → (c1'=1);
    [sync] feedback=client3–deny & c3=1 & turn=3 → (c3'=-1) & (turn'=4) & (action3'=wait);
    [sync] feedback=client3–grant & c3=1 & turn=3 → (c3'=2) & (turn'=4) & (action3'=useResource);
```

```
    [sync] action3=useResource & c3=2 & turn=3 → 0.5:(c3'=2) & (turn'=4)
            + 0.5:(c3'=3) & (turn'=4) & (action3'=cancel);
    [sync] action3=cancel & c3=3 & turn=3 → (c3'=-1) & (turn'=4) & (action3'=wait);
    [sync] action4=wait & c4=-1 & turn=4 → 0.5:(c4'=-1) & (turn'=5)
            + 0.5:(c4'=0) & (turn'=5) & (action4'=request);
    [sync] action4=request & c4=0 & turn=4 → (c1'=1);
    [sync] feedback=client4–deny & c4=1 & turn=4 → (c4'=-1) & (turn'=5) & (action4'=wait);
    [sync] feedback=client4–grant & c4=1 & turn=4 → (c4'=2) & (turn'=5) & (action4'=useResource);
    [sync] action4=useResource & c4=2 & turn=4 → 0.5:(c4'=2) & (turn'=5)
            + 0.5:(c4'=3) & (turn'=5) & (action4'=cancel);
    [sync] action4=cancel & c4=3 & turn=4 → (c4'=-1) & (turn'=5) & (action4'=wait);
    [sync] action5=wait & c5=-1 & turn=5 → 0.5:(c5'=-1) & (turn'=6)
            + 0.5:(c5'=0) & (turn'=6) & (action5'=request);
    [sync] action5=request & c5=0 & turn=5 → (c1'=1);
    [sync] feedback=client5–deny & c5=1 & turn=5 → (c5'=-1) & (turn'=6) & (action5'=wait);
    [sync] feedback=client5–grant & c5=1 & turn=5 → (c5'=2) & (turn'=6) & (action5'=useResource);
    [sync] action5=useResource & c5=2 & turn=5 → 0.5:(c5'=2) & (turn'=6)
            + 0.5:(c5'=3) & (turn'=6) & (action5'=cancel);
    [sync] action5=cancel & c5=3 & turn=5 → (c5'=-1) & (turn'=6) & (action5'=wait);
    [sync] action6=wait & c6=-1 & turn=6 → 0.5:(c6'=-1) & (turn'=7)
            + 0.5:(c6'=0) & (turn'=7) & (action6'=request);
    [sync] action6=request & c6=0 & turn=6 → (c1'=1);
    [sync] feedback=client6–deny & c6=1 & turn=6 → (c6'=-1) & (turn'=7) & (action6'=wait);
    [sync] feedback=client6–grant & c6=1 & turn=6 → (c6'=2) & (turn'=7) & (action6'=useResource);
    [sync] action6=useResource & c6=2 & turn=6 → 0.5:(c6'=2) & (turn'=7)
            + 0.5:(c6'=3) & (turn'=7) & (action6'=cancel);
    [sync] action6=cancel & c6=3 & turn=6 → (c6'=-1) & (turn'=7) & (action6'=wait);
    [sync] action7=wait & c7=-1 & turn=7 → 0.5:(c7'=-1) & (turn'=8)
            + 0.5:(c7'=0) & (turn'=8) & (action7'=request);
    [sync] action7=request & c7=0 & turn=7 → (c1'=1);
    [sync] feedback=client7–deny & c7=1 & turn=7 → (c7'=-1) & (turn'=8) & (action7'=wait);
    [sync] feedback=client7–grant & c7=1 & turn=7 → (c7'=2) & (turn'=8) & (action7'=useResource);
    [sync] action7=useResource & c7=2 & turn=7 → 0.5:(c7'=2) & (turn'=8)
            + 0.5:(c7'=3) & (turn'=8) & (action7'=cancel);
    [sync] action7=cancel & c7=3 & turn=7 → (c7'=-1) & (turn'=8) & (action7'=wait);
    [sync] action8=wait & c8=-1 & turn=8 → 0.5:(c8'=-1) & (turn'=1)
            + 0.5:(c8'=0) & (turn'=1) & (action8'=request);
    [sync] action8=request & c8=0 & turn=8 → (c8'=1);
    [sync] feedback=client8–deny & c8=1 & turn=8 → (c8'=-1) & (turn'=1) & (action8'=wait);
    [sync] feedback=client8–grant & c8=1 & turn=8 → (c8'=2) & (turn'=1) & (action8'=useResource);
    [sync] action8=useResource & c8=2 & turn=8 → 0.5:(c8'=2) & (turn'=1)
            + 0.5:(c8'=3) & (turn'=1) & (action8'=cancel);
    [sync] action8=cancel & c8=3 & turn=8 → (c8'=-1) & (turn'=1) & (action8'=wait);
endmodule
```

label “err”= error;

Bibliography

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002. 19

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 15(1):441–460, 1990. 97

[AHP92] H. Aizenstein, L. Hellerstein, and L. Pitt. Read-thrice dnf is hard to learn with membership and equivalence queries. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:523–532, 1992. 16

[AK91] Dana Angluin and Michael Kharitonov. When won't membership queries help? In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, STOC '91, pages 444–454, New York, NY, USA, 1991. ACM. 16

[AL10] Husain Aljazzar and Stefan Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. Software Eng.*, 36(1):37–60, 2010. 13

[ALFLS11] Husain Aljazzar, Florian Leitner-Fischer, Stefan Leue, and Dimitar Simeonov. Dipro - a tool for probabilistic counterexample generation. In *SPIN*, pages 183–187, 2011. 14

[Ang87a] Dana Angluin. Learning regular sets from queries and counterexamples. *Inform. and Comput.*, 75(2):87–106, 1987. vii, 4, 15, 46, 47, 48, 77, 130

[Ang87b] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987. 16

[AP92] Howard Aizenstein and Leonard Pitt. Exact learning of read-k disjoint dnf and not-so-disjoint dnf. In *COLT*, pages 71–76, 1992. 16

[AvMN05] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 98–109, New York, NY, USA, 2005. ACM. 19

[BC01] Vincent Blondel and Vincent Canterini. Undecidable problems for probabilistic automata of fixed dimension. *Theory of Computing Systems*, 36:231–245, 2001. 106, 112

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 428–439. IEEE Computer Society Press, 1990. 8

[BdA95] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *Proc. FSTTCS'95*, volume 1026 of *LNCS*. Springer, 1995. 9

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Computers*, C-21:592–597, 1972. 15

[BGP03] Howard Barringer, Dimitra Giannakopoulou, and Corina S. Pasareanu. Proof rules for automated compositional verification through learning. In *Proc. SAVCBS Workshop*, pages 14–21, 2003. 17

[BHKL09] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. Angluin-style learning of NFA. In *21st International Joint Conference on Artifical Intelligence (IJCAI'09)*, July 2009. vii, 4, 15, 18, 51, 53, 54

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008. 21, 30, 35, 36, 37, 39

[BKK+10] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. Piegdon. libalf: The automata learning framework. In *Proc. CAV'10*, LNCS. Springer, 2010. To appear. 96

[BLW13] Michael Benedikt, Rastislav Lenhardt, and James Worrell. Ltl model checking of interval markov chains. In *TACAS*, 2013. To appear. 10

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, January 2003. 13

[BP95] Bard Bloom and Robert Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comput. Program.*, 24(3):189–220, 1995. 12

[BR92] Avrim Blum and Steven Rudich. Fast learning of k-term dnf formulas with queries. In *STOC*, pages 382–389, 1992. 16

[Bru04] Stefan D. Bruda. Preorder relations. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*, pages 117–149. Springer, 2004. 11

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. 11, 143, 144

[Bsh95] Nader H. Bshouty. Exact learning boolean functions via the monotone theory. *INFORMATION AND COMPUTATION*, 123:146–153, 1995. vii, 5, 16, 18, 55, 56, 58, 141

[BV96] Francesco Bergadano and Stefano Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM J. Comput.*, 25(6):1268–1280, 1996. 16, 125, 127, 130, 138

[BWB+11] Bettina Braitling, Ralf Wimmer, Bernd Becker, Nils Jansen, and Erika Ábrahám. Counterexample generation for markov chains using smt-based bounded model checking. In *Proceedings of the joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 international conference on Formal techniques for distributed systems*, FMOODS'11/FORTE'11, pages 75–89, Berlin, Heidelberg, 2011. Springer-Verlag. 14

[CB06] F. Ciesinski and C. Baier. Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Proc. 3rd International Con-*

ference on Quantitative Evaluation of Systems (QEST'06), pages 131–132. IEEE CS Press, 2006. 10

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, page 2001. Kluwer Academic Publishers, 2001. 8

[CCF+10] Yu-Fang Chen, Edmund M. Clarke, Azadeh Farzan, Ming-Hsien Tsai, Yih-Kuen Tsay, and Bow-Yaw Wang. Automated assume-guarantee reasoning through implicit learning. In *Proc. Computer Aided Verification (CAV'10)*, pages 511–526, 2010. 2, 18, 59, 62, 63, 141, 142, 146, 177

[CDL+11] Benoît Caillaud, Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wasowski. Constraint markov chains. *Theor. Comput. Sci.*, 412(34):4373–4404, 2011. 10

[CE81] E. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *LNCS*. Springer, 1981. 8

[CFC+09] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating dfa's for compositional verification. In *TACAS*, pages 31–45, 2009. 18

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, pages 1–15, 1993. Also available in *Formal Methods in System Design*, 10(2/3):149–169, 1997. 143, 144

[CG10] S. Chaki and A. Gurfinkel. Automated assume-guarantee reasoning for omega-regular systems and specifications. In *Proc. NFM'10*, pages 57–66, 2010. 182

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000. 8

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003. 2, 17, 18, 59, 60, 66, 101

[CHK13] Taolue Chen, Tingting Han, and Marta Kwiatkowska. On the complexity of model checking interval-valued discrete time markov chains. *Information Processing Letters*, 2013. accepted. 10, 29

[CJR11] Radu Calinescu, Kenneth Johnson, and Yasmin Rafiq. Using observation ageing to improve markovian model learning in qos engineering. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, pages 505–510, New York, NY, USA, 2011. ACM. 19

[CKJ12] Radu Calinescu, Shinji Kikuchi, and Kenneth Johnson. Compositional reverification of probabilistic safety properties for large-scale complex it systems. In *Monterey Workshop*, pages 303–329, 2012. 1, 181

[CLSV04] L. Cheung, N. Lynch, R. Segala, and F. Vaandrager. Switched probabilistic I/O automata. In *Proc. ICTAC'04*, volume 3407 of *LNCS*. Springer, 2004. 13

[CMJ+12] Yingke Chen, Hua Mao, Manfred Jaeger, Thomas Dyhre Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning markov models for stationary system behaviors. In *NASA Formal Methods*, pages 216–230, 2012. 19

[CO94] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *ICGI '94: Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 139–152, London, UK, 1994. Springer-Verlag. 16

[CS07] Sagar Chaki and Ofer Strichman. Optimized l*-based assume-guarantee reasoning. In *TACAS*, pages 276–291, 2007. 18

[CSH08] Krishnendu Chatterjee, Koushik Sen, and Thomas A. Henzinger. Model-checking omega-regular properties of interval markov chains. In *FoSSaCS*, pages 302–317, 2008. 10, 156

[CW12] Yu-Fang Chen and Bow-Yaw Wang. Learning boolean functions incrementally. In *CAV*, pages 55–70, 2012. 17, 182

[CY88] C. Courcoubetis and M. Yannakakis. Verifying temporal properties of finite state probabilistic programs. In *Proc. 29th Annual Symposium on Foun-*

dations of Computer Science (FOCS'88), pages 338–345. IEEE Computer Society Press, 1988. 9, 35

[CY90] C. Courcoubetis and M. Yannakakis. Markov decision processes and regular events. In *Proc. ICALP'90*, volume 443 of *LNCS*. Springer, 1990. 9, 38

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995. 9

[dA97] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997. 38, 88

[dAHJ01] L. de Alfaro, T. Henzinger, and R. Jhala. Compositional methods for probabilistic systems. In *Proc. CONCUR'01*, volume 2154 of *LNCS*. Springer, 2001. 13

[dAHM00] Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Detecting errors before reaching them. In *CAV*, pages 186–201, 2000. 13

[DHR08] Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Equivalence of labeled Markov chains. *Int. J. Found. Comput. Sci.*, 19(3):549–563, 2008. 112

[Dij59] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 44

[DKN+10] M. Duflot, M. Kwiatkowska, G. Norman, D. Parker, S. Peyronnet, C. Picaronny, and J. Sproston. *FMICS Handbook on Industrial Critical Systems*, chapter Practical Applications of Probabilistic Model Checking to Communication Protocols, pages 133–150. IEEE Computer Society Press, 2010. 1

[dlH97] Colin de la Higuera. Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27(2):125–138, 1997. 14

[dlH10] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. 14

[dlHO04] Colin de la Higuera and José Oncina. Learning stochastic finite automata. In *ICGI*, pages 175–186, 2004. 16

[DLL+11] Benoît Delahaye, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, and Andrzej Wasowski. Decision problems for interval markov chains. In *LATA*, pages 274–285, 2011. 12

[DLT02] François Denis, Aurélien Lemay, and Alain Terlutte. Residual finite state automata. *Fundam. Inform.*, 51(4):339–368, 2002. 51

[DLT04] François Denis, Aurélien Lemay, and Alain Terlutte. Learning regular languages using rfsas. *Theor. Comput. Sci.*, 313(2):267–294, 2004. 15, 51

[EDK89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional Model Checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 353–361, Washington D.C., 1989. IEEE Computer Society Press. 12

[EGL85] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985. 134, 173

[EKVY07] K. Etessami, M. Kwiatkowska, M. Vardi, and M. Yannakakis. Multi-objective model checking of Markov decision processes. In *Proc. TACAS'07*, volume 4424 of *LNCS*. Springer, 2007. 9, 13, 42

[Epp98] David Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28(2):652–673, 1998. 44, 96

[FCC+08] A. Farzan, Y.-F. Chen, E. Clarke, Y.-K. Tsay, and B.-Y. Wang. Extending automated compositional verification to the full class of omega-regular languages. In *Proc. TACAS'08*, volume 4963 of *LNCS*, pages 2–17. Springer, 2008. 182

[FHKP11a] L. Feng, T. Han, M. Kwiatkowska, and D. Parker. Learning-based compositional verification for synchronous probabilistic systems. In *Proc. 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, volume 6996 of *LNCS*, pages 511–521. Springer, 2011. vi, 6, 16, 18, 106, 109, 116

[FHKP11b] L. Feng, T. Han, M. Kwiatkowska, and D. Parker. Learning-based compositional verification for synchronous probabilistic systems. Technical Report RR-11-05, Department of Computer Science, University of Oxford, 2011. 6, 183

[FHPW10] Harald Fecher, Michael Huth, Nir Piterman, and Daniel Wagner. Pctl model checking of markov chains: Truth and falsity as winning strategies in games. *Perform. Eval.*, 67(9):858–872, 2010. 14

[FKN+11] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Quantitative multi-objective verification for probabilistic systems. In *TACAS*, pages 112–127, 2011. 13, 42, 103, 182

[FKNP11] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011. 21, 30, 41

[FKP10] L. Feng, M. Kwiatkowska, and D. Parker. Compositional verification of probabilistic systems using learning. In *Proc. 7th International Conference on Quantitative Evaluation of SysTems (QEST'10)*, pages 133–142. IEEE CS Press, 2010. v, 2, 5, 6, 18, 19, 66, 74, 75, 76, 82, 83, 84, 97

[FKP11] L. Feng, M. Kwiatkowska, and D. Parker. Automated learning of probabilistic assumptions for compositional reasoning. In D. Giannakopoulou and F. Orejas, editors, *Proc. 14th International Conference on Fundamental Approaches to Software Engineering (FASE'11)*, volume 6603 of *LNCS*, pages 2–17. Springer, 2011. 5, 6, 18, 19, 66

[GDK+12] Khalil Ghorbal, Parasara Sridhar Duggirala, Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. Efficient probabilistic model checking of systems with ranged probabilities. In *6th International Workshop on Reachability Problems (RP)*, Bordeaux, France, 2012. 10

[GGP07] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, pages 292–307, 2007. 17

[GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994. 12

[Gol67] E. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, May 1967. 14

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978. 14

[GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *TACAS*, pages 357–370, 2002. 19

[GS08] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 51–60, New York, NY, USA, 2008. ACM. 19

[GSS10] Michael Günther, Johann Schuster, and Markus Siegle. Symbolic calculation of k-shortest paths and related measures with the stochastic process algebra tool caspa. In *Proceedings of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems*, DYADEM-FTS '10, pages 13–18, New York, NY, USA, 2010. ACM. 44

[Har98] D.J. Hartfiel. *Markov Set-Chains.* Springer-Verlag, Berlin, 1998. 10

[HHK95] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462. IEEE Computer Society Press, 1995. 12

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6(5):512–535, 1994. 9

[HKD09] Tingting Han, Joost-Pieter Katoen, and Berteun Damman. Counterexample generation in probabilistic model checking. *IEEE Transactions on Software Engineering*, 35(2):241–257, 2009. 13, 43

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. 11

[HP06] Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In *CSL*, pages 395–410, 2006. 88

[HS94] D.J. Hartfiel and E. Seneta. On the theory of markov set-chains. *Advances in Applied Probability*, 26(4):947964, 1994. 10

[HSV94] L. Helmink, M. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proc. International Workshop on Types for Proofs and Programs (TYPES'93)*, volume 806 of *LNCS*, pages 127–165. Springer, 1994. 135

[JJ99] J.Magee and J.Kramer. *Concurrency: State Models and Jave Programs.* John Wiley and Sons, 1999. 61

[JKWY10] Yungbum Jung, Soonho Kong, Bow-Yaw Wang, and Kwangkeun Yi. Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In *VMCAI*, pages 180–196, 2010. 20

[JL91] Bengt Jonsson and Kim Guldstrand Larsen. Specification and refinement of probabilistic processes. In *LICS*, pages 266–277, 1991. 10, 12, 28

[JLWY11] Yungbum Jung, Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. Predicate generation for learning-based quantifier-free loop invariant inference. In *TACAS*, pages 205–219, 2011. 20

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983. 12

[JRS04] HoonSang Jin, Kavita Ravi, and Fabio Somenzi. Fate and free will in error traces. *Int. J. Softw. Tools Technol. Transf.*, 6(2):102–116, August 2004. 13

[KKLW12] Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf. Three-valued abstraction for probabilistic systems. *J. Log. Algebr. Program.*, 81(4):356–389, 2012. 10, 157, 162, 178

[KMO+11] Stefan Kiefer, Andrzej S. Murawski, Joel Ouaknine, Björn Wachter, and James Worrell. Language equivalence for probabilistic automata. In *Proc. CAV'11*, 2011. To appear. 112

[KNP04] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004. 11, 143

[KNP10] M. Kwiatkowska, G. Norman, and D. Parker. *Symbolic Systems Biology*, chapter Probabilistic Model Checking for Systems Biology, pages 31–59. Jones and Bartlett, 2010. 1

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011. 10, 96, 134, 172

[KNPQ10] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-guarantee verification for probabilistic systems. In J. Esparza and R. Majumdar, editors, *Proc. 16th International Conference on Tools and Algorithms for*

the Construction and Analysis of Systems (TACAS'10), volume 6105 of *LNCS*, pages 23–37. Springer, 2010. v, 2, 4, 5, 13, 28, 42, 65, 66, 67, 70, 72, 73, 102, 179, 181

[KNPQ12] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Compositional probabilistic verification through multi-objective model checking. *Information and Computation*, 2012. submitted. v, 68, 82, 84

[KNS01] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 194–206. Springer, 2001. 97

[KPC12a] Anvesh Komuravelli, Corina S. Pasareanu, and Edmund M. Clarke. Assume-guarantee abstraction refinement for probabilistic systems. In *CAV*, pages 310–326, 2012. 18, 19

[KPC12b] Anvesh Komuravelli, Corina S. Pasareanu, and Edmund M. Clarke. Learning probabilistic systems from tree samples. In *LICS*, pages 441–450, 2012. 18

[KSK76] John G. Kemeny, J. Laurie Snell, and Anthony W. Knapp. *Denumerable Markov chains*. Springer-Verlag, New York :, 2d. ed. edition, 1976. 25

[KU02] Igor Kozine and Lev V. Utkin. Interval-valued finite markov chains. *Reliable Computing*, 8(2):97–113, 2002. 10, 28

[KZH+09] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker mrmc. In *QEST*, pages 167–176, 2009. 10

[LAD+11] Shang-Wei Lin, Étienne André, Jin Song Dong, Jun Sun, and Yang Liu. An efficient algorithm for learning event-recording automata. In *ATVA*, pages 463–472, 2011. 18, 182

[LLS+12] Shang-Wei Lin, Yang Liu, Jun Sun, Jin Song Dong, and Étienne André. Automatic compositional verification of timed systems. In *FM*, pages 272–276, 2012. 18, 182

[LPC+12] M. Lakin, D. Parker, L. Cardelli, M. Kwiatkowska, and A. Phillips. Design and analysis of DNA strand displacement devices using probabilistic model checking. *Journal of the Royal Society Interface*, 9(72):1470–1485, 2012. 1

[LWY12] Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. Termination analysis with algorithmic learning. In *CAV*, pages 88–104, 2012. 20

[MCJ+11] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning probabilistic automata for model checking. In *QEST*, pages 111–120, 2011. 16, 19

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In *Proc. Computer Aided Verification (CAV'03)*, pages 1–13, 2003. 63, 146

[Mil71] Robin Milner. An algebraic definition of simulation between programs. In *IJCAI*, pages 481–489, 1971. 12

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 12

[MJ12] Hua Mao and Manfred Jaeger. Learning and model-checking networks of i/o automata. *Journal of Machine Learning Research - Proceedings Track*, 25:285–300, 2012. 19

[MO05] Andrzej Murawski and Joël Ouaknine. On probabilistic program equivalence and refinement. In *Proc. CONCUR'05, volume 3653 of LNCS*, volume 3653 of *LNCS*. Springer, 2005. 106, 112

[NMA08] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Formal Methods in System Design*, 32(3):207–234, 2008. 18, 102, 181

[OG92] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, pages 49–61, 1992. 15

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002. 9, 11, 143, 144, 145, 146, 185

[Paz71] Azaria Paz. *Introduction to probabilistic automata (Computer science and applied mathematics)*. Academic Press, Inc., Orlando, FL, USA, 1971. 112

[PG06] Corina S. Pasareanu and Dimitra Giannakopoulou. Towards a compositional spin. In *SPIN*, pages 234–251, 2006. 17, 97

[PGB+08] C. Pasareanu, D. Giannakopoulou, M. Bobaru, J. Cobleigh, and H. Barringer. Learning to divide and conquer: Applying the L* algorithm to automate assume-guarantee reasoning. *FMSD*, 32(3):175–205, 2008. 2, 17, 61, 66, 90, 96, 101, 102, 135, 181

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 46–57, 1977. 31

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. *Logics and models of concurrent systems*, pages 123–144, 1985. 12

[Put94] M. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994. 9

[PVY99] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *FORTE*, pages 225–240, 1999. 19

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proc. 5th International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982. 8

[Rab63] M. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963. 16, 26, 111

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959. 23

[RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993. vii, 15, 17, 46, 49, 50, 77

[Sch61] Marcel Paul Schützenberger. On the definition of a family of automata. *Information and Control*, 4(2-3):245–270, 1961. 112

[Seg95] R. Segala. *Modelling and Verification of Randomized Distributed Real Time Systems*. PhD thesis, Massachusetts Institute of Technology, 1995. 9, 12, 26, 67, 105, 111

[SL95] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2), 1995. 12

[SSB02] Ofer Strichman, Sanjit A. Seshia, and Randal E. Bryant. Deciding separation formulas with sat. In *Proc. Computer Aided Verification (CAV'02)*, pages 209–222, 2002. 147

[SVA04] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning continuous time markov chains from sample executions. In *QEST*, pages 146–155, 2004. 19

[SVA06] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Model-checking markov chains in the presence of uncertainties. In *TACAS*, pages 394–410, 2006. 10

[TF11] Tino Teige and Martin Fränzle. Generalized craig interpolation for stochastic boolean satisfiability problems. In *TACAS*, pages 158–172, 2011. 142, 167

[Tse83] G. S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 466–483. Springer, Berlin, Heidelberg, 1983. 174

[Tze92a] Wen-Guey Tzeng. Learning probabilistic automata and markov chains via queries. *Machine Learning*, 8(2):151–166, 1992. 16

[Tze92b] Wen-Guey Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM J. Comput.*, 21(2):216–227, 1992. 112, 119, 121

[Val84] Leslie G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984. 14

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 327–338, Washington, DC, USA, 1985. IEEE Computer Society. 8

[vG90] Rob J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR*, pages 278–297, 1990. 12

[vG93] Rob J. van Glabbeek. The linear time - branching time spectrum ii. In *CONCUR*, pages 66–81, 1993. 12

[Zha09] Lijun Zhang. *Decision Algorithms for Probabilistic Simulation*. PhD thesis, Saarland University, 2009. 12