Lehrstuhl für Informatik 2
Software Modeling and Verification

RWTH AACHEN UNIVERSITY

Master Thesis

Accelerating Predicate Abstraction for Probabilistic Automata

Dimitri Bohlender

September 2, 2014

First Referee:
Prof. Dr. Ir. Joost-Pieter Katoen

Second Referee:
Apl. Prof. Dr. Thomas Noll

Acknowledgements.
First of all, I want to thank Prof. Katoen, Prof. Noll and Prof. Ábrahám for their inspirational lectures on different approaches to modelling and verification, which got me interested in formal methods in the first place.
Also, I want to thank Christian Dehnert for his advice and support during the numerous ups and downs of this thesis.
Furthermore, I owe my thanks to my girlfriend Julia for bearing with me during this busy time of working from dawn till dusk.
Lastly, and most importantly, I want to express my gratitude towards my family for nurturing my skills and always having faith in me and my decisions.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 2. September 2014

Dimitri Bohlender

Zusammenfassung

Probabilistisches Model Checking ist eine Technik, die beweisen oder widerlegen kann, ob ein probabilistisches System sich konform zu einer Spezifikation verhält. Um Systeme mit besonders vielen Zuständen analysieren zu können, greift man auf speichereffiziente symbolische Datenstrukturen, wie etwa binäre Entscheidungsdiagramme (BED) zurück oder fasst ähnliche Zustände zusammen, um die Spezifikation potentiell bereits auf einem kleineren, überapproximierenden Zustandsraum verifizieren zu können. Menübasierte Abstraktion ist eine solche Überapproximationstechnik, welche einen spieltheoretischen Ansatz mit Predicate Abstraction kombiniert. Wir zeigen erstmals ein komplett symbolisches Verfahren zur Bestimmung der Menü-basierten Abstraktion und Verifikation des Systems. Insbesondere werden Optimierungen vorgestellt, die notwendig sind, um das Verfahren vergleichbar schnell zur, teils expliziten, Referenzimplementierung Pass zu machen. Diese vergleichen wir mit unserer prototypischen Implementierung und evaluieren die Effektivität verschiedener Optimierungen anhand von Fallstudien. Zudem gehen wir darauf ein, wie das Verfahren auf modulare Systeme erweiterbar ist.

Abstract

Probabilistic model checking is a technique to verify the compliance of systems which exhibit probabilistic behaviour with respect to a specification. Treatment of systems with a large state space is enabled through the employment of memory-efficient, symbolic data structures, such as Binary Decision Diagrams (BDDs), or lumping of similar states to potentially allow for a verification of the specification on a smaller, over-approximating state space. Menu-based abstraction is such an over-approximation which combines a game theoretic approach with predicate abstraction. We firstly present a fully symbolic procedure to both compute the menu-based abstraction and verify the system. In particular, we present the optimisations necessary to make the procedure comparable fast to the partially explicit reference implementation Pass. We compare the latter to our prototypical implementation and evaluate the efficiency of different optimisations on the basis of case studies. Additionally, we elaborate on how the procedure can be extended to work with modular systems.

Contents

List of Figures

1. Introduction

This chapter introduces the concept of (probabilistic) model checking as a formal method to verify a system's compliance with respect to a specification. We motivate the need for such a technique, elaborate on the fundamental problem of model checking and sketch already proposed techniques for its treatment before we end this chapter with an outline of this thesis.

1.1. Motivation

The steady increase of system design complexity is accompanied by an increasing difficulty of their analysis and often renders manual analysis even impossible. Yet, these systems pervade all areas of life and people place reliance on them. Especially network protocols, like the IEEE 802.11 wireless LAN protocol[1], are usually probabilistic in nature due to the use of randomised algorithms and the need of modelling probabilistic phenomena like message loss. While the repair of previously undetected errors in standardised network protocols may merely be expensive, the compliance of safety-critical systems with respect to a safety-specification is of utter importance as errors may cost lives, e.g. [Lacan et al., 1998]. Clearly, the need for rigorous, automated verification procedures arises. Testing, however, cannot fill this gap, as it only uncovers bugs, but does not proof their absence (unless every possible execution is covered).

Model checking, pioneered by [Clarke and Emerson, 1982, Queille and Sifakis, 1982], was one formal method that emerged from this urge, capable to prove or disprove a system's compliance with respect to some specification by an exhaustive examination of all possible executions of the system and checking whether any of them violate the specification.

Figure 1.1 [Baier and Katoen, 2008] illustrates the structure of the general model checking approach. The initial (informal) artefacts are the description of the system and the specification it is expected to respect. Both must be formalised first to get precise descriptions of *what* the system should do, and what it should not do, as well as *how* the system behaves. The specification is usually formalised as a set of temporal logic formulae, e.g. CTL [Clarke et al., 1986], which are suitable to specify *qualitative* properties like "eventually a collision free transmission occurs" or "there can never be more than one process in the critical section". The system's formal representation is usually a digraph structure, e.g. LTS [Baier and Katoen, 2008], where vertices represent the systems's states while edges represent the possibilities to transfer the system from one state to another. Both these formal artefacts act as input to the fully automated

[1] `http://standards.ieee.org/about/get/802/802.11.html`

model checking tool, the model checker, which analyses whether the properties are satisfied by the model or there is some violating execution – the counterexample.

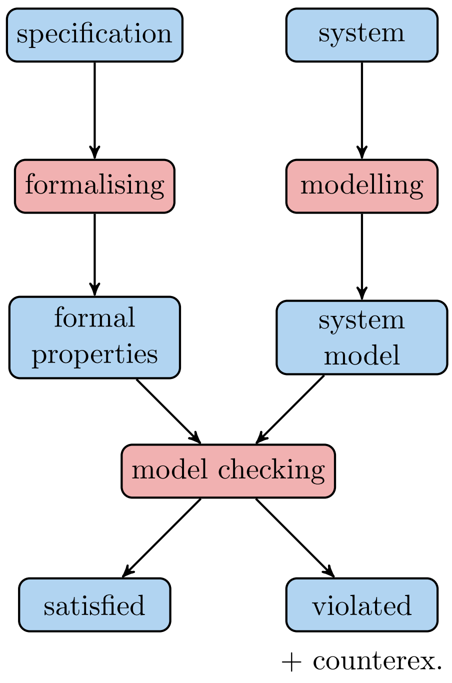


Figure 1.1.: General structure of the model checking approach

Whilst qualitative properties are suitable in the non-probabilistic setting, they are rarely of interest for probabilistic systems. For instance, the wireless LAN protocol clearly violates the property "no collision will ever occur". This, however, does not mean that the protocol is bad. It actually suffices if the protocol is guaranteed to satisfy properties like "the probability for a collision to occur twice in a row is always below 5%" and rather indicates that, in the probabilistic setting, *quantitative* measures like performance and quality-of-service guarantees become vital concerns. Probabilistic model checking builds upon conventional model checking by extending both the properties, e.g. PCTL [Hansson and Jonsson, 1994], and the models with quantifiable features like probabilities, such that properties like the aforementioned one can be verified. In fact, this technique has been successfully applied to the wireless LAN protocol to reason about similar properties [Kwiatkowska et al., 2002].

1.2. State Space Explosion Problem

Based on the tempting advantages model checking exhibits, one may wonder why this technique is rarely used on real-world models but mostly subject to academic usage. The fundamental problem with this approach is known as the "State Space Explosion

Problem", which refers to the number of states in the system model which rapidly increases with model complexity and does often not fit into memory. Consider, for example, a system that is modelled by N variables from a domain of k possible values. Such a system may have up to k^N states. Albeit, not all of them may be reachable, the growth of the state space is asymptotically exponential in the number of variables. Using *explicit* state-space enumeration, state-of-the-art model checkers can handle state spaces of about 10^8 to 10^9 states. One strategy to alleviate this shortcoming is to use *symbolic* data structures, as their size decreases the more similarities the data they represent contains, and practical system models often feature symmetries in their structure. Another approach is to abstract from the concrete state space by lumping states which are "similar" with respect to some measure. Of course, such a measure must be defined in a way so that the property of interest can still be checked, i.e. the *abstraction* must preserve it. [Baier and Katoen, 2008]

However, even if a model fits into memory, its analysis may still be too computationally expensive to be carried out. Many techniques have been proposed to enable probabilistic model checking of real-world systems by avoiding building the whole state space. Since properties are typically quite specific, a large portion of the state space is not relevant to verify them. This motivates the use of iterative abstraction *refinement* schemes, like [Hermanns et al., 2008], which begin with a very coarse abstraction lumping many states together, and refine it iteratively if it turns out too coarse to prove or disprove the satisfiability of the respective property. Other techniques are the game-based abstraction-refinement [Kattenbelt et al., 2010, Wachter, 2011] which reinterprets the probabilistic model as a game and the assume-guarantee verification [Kwiatkowska et al., 2010] which exploits the modular structure of system models. This thesis develops a fully symbolic version of the menu-based abstraction proposed in [Wachter, 2011], presents necessary optimisations to make the procedure comparable fast to the partially explicit reference implementation PASS, and elaborates on how the procedure can be employed for assume-guarantee style verification.

1.3. Outline

The thesis consists of seven chapters. In chapter 2 we give an introduction into the theoretical background, i.e. the models and concepts this thesis deals with. The subsequent chapter 3 goes into detail about computing and analysing the menu-based abstraction in a fully symbolical way. Chapter 4 presents several optimisations for both the abstraction and the analysis procedure. We evaluate the optimisations and compare our prototypical implementation to PASS in chapter 5. Finally, we elaborate on a compositionality-exploiting assume-guarantee style extension in chapter 6 and summarise our findings in chapter 7.

2. Preliminaries

This chapter builds the theoretical foundation of this thesis and presents the knowledge needed to understand subsequent chapters. We start with a gentle introduction of the probabilistic models, in particular Probabilistic Automata (PA), as these will become the formal models for our probabilistic systems. It continues with the concept of abstraction, with focus on the game-based abstraction as a form of predicate abstraction. Subsequently, we introduce Multi-Terminal Binary Decision Diagrams (MTBDDs) as the symbolic data structure that will be used to store our data in a memory-efficient way. The chapter concludes with a short insight into SMT-solving.

2.1. Probabilistic Models

In a nutshell, Probabilistic Automata [Segala, 1995, Rabin, 1963] have a set of states and, for every state, may define probability distributions to other states, such that the transition probability to any other state is determined by the current state and chosen distribution only. Given a strategy to resolve a PA's non-determinism we obtain a Discrete-Time Markov Chain (DTMC) for which the unique probability of reaching a specific set of states can be computed. However, modelling real-world PA by hand is infeasible due to their size. Therefore, probabilistic automata are usually specified in terms of probabilistic programs. A wide-spread modelling formalism for probabilistic programs is the language of the probabilistic model checker PRISM [Kwiatkowska et al., 2011]. Finally, we define stochastic games, whose relation to PA will become apparent later on.

2.1.1. Probability Distribution

To reason about probabilistic behaviour we, first of all, need the notion of a probability distribution. Besides conventional probability distributions, we introduce the concept of labelled probability distributions which extend probability distributions with unique labels for their branches [Wachter, 2011].

Definition 2.1 (Probability Distribution). A (discrete) probability distribution μ over a set S is a function $\mu : S \to [0,1]$ such that $\mu(S) := \sum_{s \in S} \mu(s) = 1$. □

We denote the the domain of (discrete) probability distributions over a set S by $Dist(S)$.

Definition 2.2 (Labelled Probability Distribution). Let U be a finite set of labels. A labelled probability distribution μ_U over a set $U \times S$ is a function $\mu_U : U \times S \to [0,1]$ such that $\mu_U(S) := \sum_{(u,s) \in U \times S} \mu(u,s) = 1$, where μ_U is right-unique, i.e. for all $s, t \in S$ and $u \in U$, $\mu_U(u,s) > 0$ and $\mu_U(u,t) > 0$ implies $s = t$. □

We denote the the domain of (discrete) labelled probability distributions over a set $U \times S$ by $\mathit{Dist}_U(S)$.

Note that a labelled probability distribution $\mu_U \in \mathit{Dist}_U(S)$ induces a (unlabelled) probability distribution $\widehat{\mu_U} \in \mathit{Dist}(S)$:

$$\widehat{\mu_U}(s) := \sum_{u \in U} \mu(u, s).$$

To simplify the construction of a labelled distribution based on a set of states $S = \{s_0, \ldots, s_n\}$, probabilities $p_0, \ldots, p_n \in [0, 1]$ summing up to 1, and pairwise labels $U = \{u_0, \ldots, u_n\}$, we employ the auxiliary operator $\oplus$, where

$$\left(\bigoplus_{i=0}^{n} p_i : (u_i, s_i) \right) (u, s) := \begin{cases} p_i & \text{if } (u, s) = (u_i, s_i) \text{ for some } i \in [0, n] \\ 0 & \text{otherwise} \end{cases}.$$

Example 2.3. Consider modelling a fair die by an uniform labelled distribution over its numbers $S = \{1, 2, 3, 4, 5, 6\}$. Using the auxiliary operator, we can express it as

$$\frac{1}{6} : (u_0, 1) \oplus \frac{1}{6} : (u_1, 2) \oplus \frac{1}{6} : (u_2, 3) \oplus \frac{1}{6} : (u_3, 4) \oplus \frac{1}{6} : (u_4, 5) \oplus \frac{1}{6} : (u_5, 6).$$

□

2.1.2. Discrete-Time Markov Chain

A Discrete-Time Markov Chain (DTMC) is a modelling formalism for probabilistic behaviour. A DTMC is a digraph-like structure to the effect that it consist of *states* and *transitions*, too. However, unlike for conventional graphs, the transitions represent probability distributions over the set of states and every state is equipped with exactly one of them. That is, reaching a specific successor of a state is governed by chance. The fact that the probability distributions are static in the sense that the likelihood of reaching a specific state from the current one does not depend on the system's evolution up to this point, but merely on the current state, is known as the *memoryless property* or *Markov property* [Baier and Katoen, 2008].

Definition 2.4 (Discrete-Time Markov Chain (DTMC))**.** A Discrete-Time Markov Chain is a tuple

$$\mathcal{D} = (S, U, \mathbf{P}, s_{init})$$

where

- S is a countable, nonempty set of states,
- U is a finite update alphabet,
- $\mathbf{P} : S \to \mathit{Dist}_U(S)$ is the transition function that assigns a labelled distribution $\mathbf{P}(s)$ to every state $s \in S$,

- $s_{init} \in S$ is the initial state,

$\square$

In literature, the notion of DTMCs with an initial distribution $\iota_{init} \in \mathit{Dist}_U(S)$ is common but does not yield a more expressive formalism than the presented one. Consider such a DTMC $\mathcal{D} = (S, U, \mathbf{P}, \iota_{init})$, then $\mathcal{D}' = (S \uplus \{s_{init}\}, U, \mathbf{P}', s_{init})$ with

$$\mathbf{P}'(s) := \begin{cases} \iota_{init} & \text{if } s = s_{init} \\ \mathbf{P}(s) & \text{otherwise} \end{cases}$$

is an equivalent DTMC using the initial state notation.

A *path* in $\mathcal{D}$ is an infinite sequence of states $\pi = s_0 s_1 s_2 \cdots \in S^\omega$ such that every s_i has a transition to s_{i+1}, i.e. $\widehat{\mathbf{P}(s_i)}(s_{i+1}) > 0$ for all s_i. Let $\mathit{Paths}(\mathcal{D})$ denote the set of paths in $\mathcal{D}$, and $\mathit{Paths}_{\mathit{fin}}(\mathcal{D})$ denote the set of finite *path fragments* $s_0 s_1 \ldots s_n$ where $n \geq 0$ and $\widehat{\mathbf{P}(s_i)}(s_{i+1}) > 0$ for all $0 \leq i < n$. We employ the notation $\mathit{Paths}(s)$ and $\mathit{Paths}_{\mathit{fin}}(s)$ to denote the set of all paths and path fragments that start in state s.

Example 2.5. A DTMC can be used to model the simulation of a fair die by a coin as proposed by [Knuth and Yao, 1976] and illustrated below.

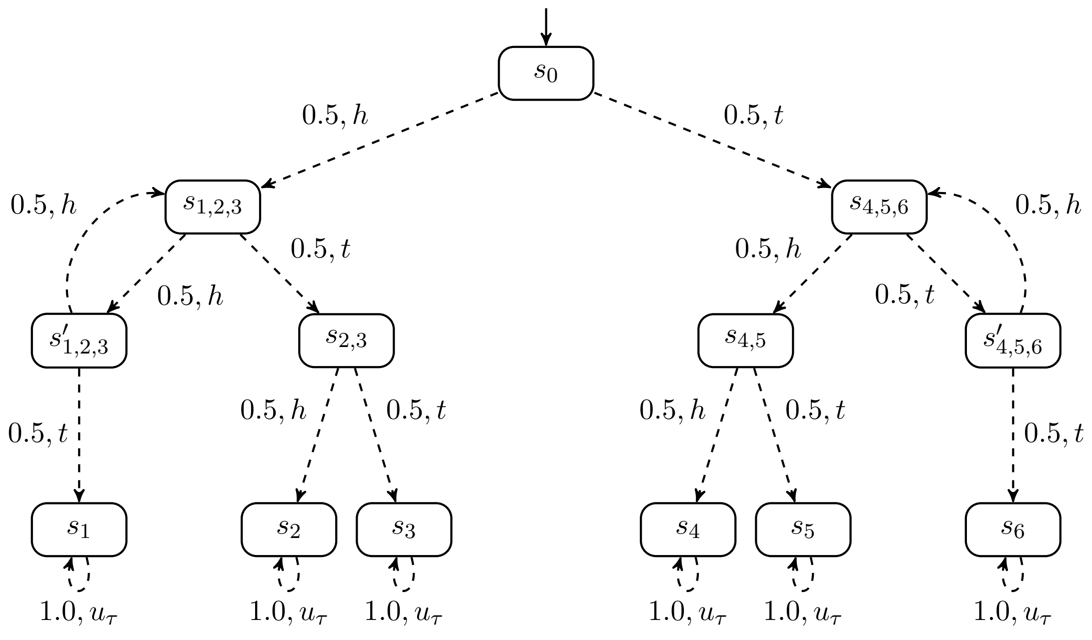


Figure 2.1.: Simulating a fair coin by a die

The states $\{s_1, s_2, \ldots, s_6\}$ stand for the possible die outcomes $\{1, 2, 3, 4, 5, 6\}$. While the simulation is in progress the outgoing transitions represent the two possible outcomes of a coin toss – *heads* and *tails*. Accordingly, we identify the updates h and t but use update u_τ, which models neither *heads* nor *tails*, at the end of the coin tossing. If the state is in the initial state s_0 and coin-tossing yields *heads* the system will transfer into state $s_{1,2,3}$ with probability 0.5. At this point it may not be clear why this DTMC properly models a fair die. This will be established later with an argument on the reachability probabilities for the states at the bottom of Figure 2.1. [Baier and Katoen, 2008] □

2.1.3. Probability Measure for DTMCs

Although DTMCs yield a rather intuitive probabilistic model, it still remains to establish a concept of associating probabilities with sets of paths to enable reasoning about probabilistic properties. To this end, we need the notion of both *σ-algebra* and *probability measure*. This section follows the characterisation of [Baier and Katoen, 2008].

Definition 2.6 (σ-algebra)**.** A σ-algebra is a pair $(\mathit{Outc}, \mathfrak{E})$, where Outc is a nonempty set of possible *outcomes* and $\mathfrak{E} \subseteq 2^{\mathit{Outc}}$ a set of subsets of Outc that contains the empty set and is closed under complementation and countable unions. We refer to the elements of $\mathfrak{E}$ as *events*. □

Definition 2.7 (Probability Measure)**.** A probability measure on a σ-algebra is a function $\mathit{Pr} : \mathfrak{E} \to [0,1]$ such that

$$\mathit{Pr}(\mathit{Outc}) = 1$$

and if $(E_n)_{n\geq 1}$ is a family of pairwise disjoint events $E_n \in \mathfrak{E}$, then

$$\mathit{Pr}\left(\bigcup_{n\geq 1} E_n\right) = \sum_{n\geq 1} \mathit{Pr}(E_n).$$

□

For countable Outc, a probability measure can trivially be obtained by fixing a distribution $\mu \in \mathit{Dist}(\mathit{Outc})$. Any such distribution μ induces a probability measure on $(\mathit{Outc}, \mathfrak{E})$ in the following way

$$\mathit{Pr}(\mathit{Event}) = \sum_{\mathit{outc} \in \mathit{Event}} \mu(\mathit{outc}),$$

where $\mathit{Event} \in \mathfrak{E}$.

Example 2.8. In the context of a fair die an adequate σ-algebra would be $(\mathit{Outc}, \mathfrak{E})$, where

$$\mathit{Outc} = \{1,2,3,4,5,6\}$$

and

$$\begin{aligned} \mathfrak{E} &= \{\{1,3,5\}\} \\ &\cup \{\emptyset\} \\ &\cup \{\{2,4,6\}, \mathit{Outc}\}, \end{aligned}$$

assuming that the event of interest is the die-cast resulting in an odd number. Note that although we are only interested in the event $\{1, 3, 5\}$ the empty set and both complements and unions of all events must be added to meet requirements of the definition.

The definition of a fair die requires all outcomes to be uniformly distributed, i.e. $\mu(outc) = \frac{1}{6}$ for each $outc \in Outc$. As a result, the probability for a die-cast to result in an odd outcome is given by

$$Pr(\{1,3,5\}) = \sum_{outc \in \{1,3,5\}} \mu(outc) = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$$

□

Definition 2.9 (Cylinder Set)**.** The cylinder set of a finite path $\hat{\pi} = s_0 \dots s_n \in Paths_{fin}(\mathcal{D})$ is the set of all infinite paths starting with $\hat{\pi}$ and is defined as

$$Cyl(\hat{\pi}) = \{\pi \in Paths(\mathcal{D}) \mid \hat{\pi} \text{ is prefix of } \pi\}\,.$$

□

In order to enable associating probabilities to events in DTMCs, it remains to construct a respective σ-algebra and probability measure. Let $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$ be a DTMC. It has been proven [Baier and Katoen, 2008] that there exists a unique probability measure $Pr_{\mathcal{D}}$ on the σ-algebra $\mathfrak{E}^{\mathcal{D}}$ where $Outc := Paths(\mathcal{D})$ and the probabilities for the cylinder sets are given by

$$Pr_{\mathcal{D}}(Cyl(s_0 \dots s_n)) = \mathbf{P}(s_0 \dots s_n)$$

where

$$\mathbf{P}(s_0 \dots s_n) := \prod_{0 \le i < n} \widehat{\mathbf{P}(s_i)}(s_{i+1}).$$

Intuitively speaking, infinite paths take the place of the possible outcomes while certain subsets of $Paths(\mathcal{D})$ act as events. This allows for assignment of probabilities to finite paths via the cylinder set construction.

Example 2.10. Consider the DTMC for simulation of a fair die with a coin from Figure 2.1. Let $\hat{\pi} = s_0\ s_{1,2,3}\ s_{1,2,3}\ s_{2,3} \in Paths_{fin}(s_{init})$ be a finite path. The probability of its cylinder set is given by

$$Pr_{\mathcal{D}}(Cyl(\hat{\pi})) = \widehat{\mathbf{P}(s_0)}(s_{1,2,3}) \cdot \widehat{\mathbf{P}(s_{1,2,3})}(s_{2,3}) = 0.5^2 = \frac{1}{4},$$

where

$$\begin{aligned} Cyl(\hat{\pi}) &= \{s_0\ s_{1,2,3}\ s_{2,3}\ s_2^\omega\} \\ &\cup \{s_0\ s_{1,2,3}\ s_{2,3}\ s_3^\omega\}\,. \end{aligned}$$

□

2.1.4. Reachability Probability

Specifications for probabilistic systems usually demand the probability for something "bad" happening to stay below a certain value, and dually, the probability for something "good" happening to be above a certain value. Checking of both *reachability properties* boils down to computing the probability of reaching a set of designated *goal states*.

Let $\mathcal{D} = (S, U, \mathbf{P}, s_{init})$ be a DTMC and $G \subseteq S$ the set of goal states. To determine the probability of reaching the goal states, we need to characterise the respective event, denoted by $\Diamond G$, as a measurable set of paths. Basically, we are interested in the set of all paths eventually reaching a state from G. However, due to the fact that the probability measure is based on cylinder sets, we will equivalently use

$$\Diamond G := \bigcup_{\substack{\hat{\pi} = s_0 \ldots s_n \in \mathit{Paths}_{\mathit{fin}}(s_{init}),\\ s_0, \ldots, s_{n-1} \notin G \text{ but } s_n \in G}} \mathit{Cyl}(s_0 \ldots s_n),$$

which intuitively is the union of cylinder sets of initial finite path fragments which end in goal states [Baier and Katoen, 2008]. Taking account of Definition 2.7, the actual reachability probability is given by

$$\begin{aligned} \mathit{Pr}_{\mathcal{D}}(\Diamond G) &= \mathit{Pr}\left(\bigcup_{\hat{\pi} = s_0 \ldots s_n \in \mathit{Paths}_{\mathit{fin}}(s_{init})} \mathit{Cyl}(s_0 \ldots s_n) \right) \\ &= \sum_{\hat{\pi} = s_0 \ldots s_n \in \mathit{Paths}_{\mathit{fin}}(s_{init})} \mathit{Pr}(\mathit{Cyl}(s_0 \ldots s_n)) \\ &= \sum_{\hat{\pi} = s_0 \ldots s_n \in \mathit{Paths}_{\mathit{fin}}(s_{init})} \mathbf{P}(s_0 \ldots s_n) \end{aligned}$$

where

$$s_0, \ldots, s_{n-1} \notin G \text{ but } s_n \in G.$$

We employ the notation $\mathit{Pr}_{D,s}(\Diamond G)$ to denote the reachability probability of G from state s instead of s_{init}, which is technically obtained by declaring s as the initial state.

Example 2.11. Let us now reconsider the simulation of a fair die by a coin, see Figure 2.1, and verify that it indeed is a proper simulation of a fair die, i.e. $\mathit{Pr}_{\mathcal{D}}(\Diamond\{s_i\}) = \frac{1}{6}$ for $i \in \{1, 2, 3, 4, 5, 6\}$. The initial finite paths leading to $G = \{s_1\}$ are of the form

$$\hat{\pi}_n = s_0 \ s_{1,2,3} \ s'_{1,2,3} \ (s_{1,2,3} \ s'_{1,2,3})^n \ s_1$$

where $n \in \mathbb{N}$. Computing the associated probability as follows

$$\mathit{Pr}_{\mathcal{D}}(\Diamond\{s_1\}) = \sum_{n=0}^{\infty} \left(\frac{1}{2}\right)^3 \cdot \left(\left(\frac{1}{2}\right)^2\right)^n = \frac{1}{8} \cdot \sum_{n=0}^{\infty} \left(\frac{1}{4}\right)^n = \frac{1}{8} \cdot \frac{1}{1 - \frac{1}{4}} = \frac{1}{6}$$

proves that the simulation of casting the die by a coin toss is adequate. Bear in mind that we omitted the proof of $\mathit{Pr}_{\mathcal{D}}(\Diamond\{s_i\}) = \frac{1}{6}$, for $i \in \{2, 3, 4, 5, 6\}$, as it is analogous. □

Value Iteration

Now, we have a formalisation of probabilistic reachability. However, as illustrated in Example 2.11, calculating the reachability probability by means of infinite sums is rather cumbersome – especially for large systems. This section presents an alternative approach, called *value iteration*, using a *fixed point characterisation* of probabilistic reachability such that the reachability probability, with respect to a set of goal states, can be computed in an iterative way.

Domain of Computation Since, we aim for a fixed point characterisation we have to reconsider the domain of computation such that both the states are associated with reachability probabilities and several variants, think of iterations, of state spaces associated with reachability probabilities can be compared and combined. To this end we need the notion of *lattice* and *valuation.*

Definition 2.12 (Lattice)**.** A (complete) lattice is a partial order $(L, \sqsubseteq_L)$, where all subsets of L have both *least upper bound* and *greatest lower bound* in L. □

We employ the operators $\bigsqcap : 2^L \to L$ and $\bigsqcup : 2^L \to L$, induced by $\sqsubseteq_L$, to denote the greatest lower bound and least upper bound of a subset of L.

Definition 2.13 (Valuations)**.** Let S be a set of of states. A valuation over S is a function $w : S \to [0,1]$ that maps states to probabilities $[0,1] \subseteq \mathbb{R}$. □

We denote the set of valuations over a state S by

$$[0,1]^S = \{w \mid w : S \to [0,1]\}\,.$$

Combining both formalisms yields the desired domain of computation. The pair $([0,1]^S, \leq)$ is a lattice [Wachter, 2011], where valuations are ordered according to $\leq$:

$$\forall_{w,w' \in [0,1]^S}\ w \leq w' \iff \forall_{s \in S}\ w(s) \leq w'(s).$$

Example 2.14. For a lattice $([0,1]^S, \leq)$ over a set S of four states, consider the valuations w_0, w_1 and w_2 from the Figure below, where labelling indicates the associated probability. Clearly, $w_1 \leq w_2$, since w_2 maps all states to values greater or equal to those of w_1. Bear in mind though, that $\leq$ may not be *total*, as illustrated by the valuation-pair w_0 and w_1.

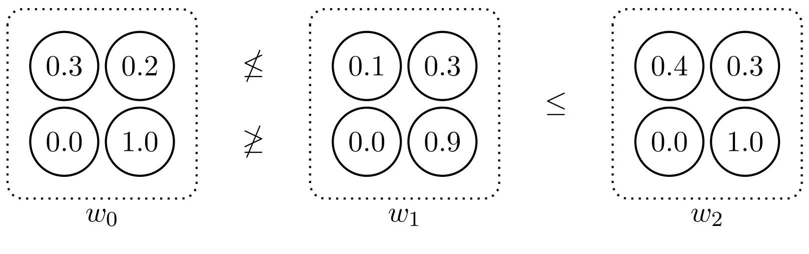


Figure 2.2.: Example usage of $\leq$ with respect to valuations

While Figure 2.2 illustrates the usage of the $\leq$-relation with respect to valuations, Figure 2.3 tries to convey the intuition behind the operators $\bigsqcup$ and $\bigsqcap$, which correspond to min and max for $\leq$.

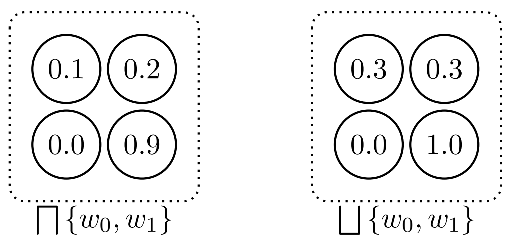


Figure 2.3.: Illustration of greatest lower bound and least upper bound concept

$\square$

Fixed Point Characterisation Intuitively, the reachability probability of a state is the weighted sum of the reachability probabilities of its successors. As a result, the reachability probability can be formulated by a recursive system of equations.

Example 2.15. For example, in the DTMC from Figure 2.1, the probability of reaching $G = \{s_1\}$ is the smallest solution of the following equations:

$$Pr_{D,s}(\Diamond G) := \begin{cases} 0.5 \cdot Pr_{D,s_{1,2,3}}(\Diamond G) + 0.5 \cdot Pr_{D,s_{4,5,6}}(\Diamond G) & \text{if } s = s_0 \\ 0.5 \cdot Pr_{D,s_{s'_{1,2,3}}}(\Diamond G) + 0.5 \cdot Pr_{D,s_{2,3}}(\Diamond G) & \text{if } s = s_{1,2,3} \\ 0.5 \cdot Pr_{D,s_{s_{1,2,3}}}(\Diamond G) + 0.5 \cdot Pr_{D,s_1}(\Diamond G) & \text{if } s = s'_{1,2,3} \\ 1 & \text{if } s = s_1 \\ 0 & \text{if } s \in S \setminus \left\{s_0, s_1, s_{1,2,3}, s'_{1,2,3}\right\} \end{cases}$$

where states which cannot reach the goal, are assigned the probability 0, while s_1, being part of the goal set, by definition has the reachability probability 1. $\square$

The recursive equation system which defines the reachability probabilities can be viewed as a monotone function over $[0,1]^S$ – a *valuation transformer* [Wachter, 2011].

Definition 2.16 (Monotone Function)**.** Let $(L, \subseteq_L)$ and $(M, \subseteq_M)$ be two lattices. A function $f : L \to M$ is monotone if for all $l, l' \in L$, $l \sqsubseteq_L l'$ implies $f(l) \sqsubseteq_M f(l')$. $\square$

Definition 2.17 (Valuation Transformer)**.** A monotone function $f : [0,1]^S \to [0,1]^S$ is a valuation transformer. $\square$

For a DTMC $\mathcal{D} = (S, U, \mathbf{P}, s_{init})$ and goal set $G \subseteq S$, the valuation transformer $pre_G : [0,1]^S \to [0,1]^S$ for probabilistic reachability is given by

$$pre_G(w)(s) := \begin{cases} 1 & \text{if } s \in G \\ 0 & \text{if } s \in G_0 \\ \displaystyle\sum_{u \in U, s' \in S} \mathbf{P}(s)(u, s') \cdot w(s') & \text{otherwise} \end{cases},$$

where $G_0 \subseteq S$ denotes the set of states, for which there exists no path to any of the states from G. Note that this is the very scheme that was applied in Example 2.15. Now, it only remains to assure ourselves of the existence of a fixed point of pre_G. This assurance is provided by the famous Fixed Point Theorem of Tarski.

Theorem 2.18 (Fixed Point Theorem [Tarski, 1955])**.** *Let $f : L \to L$ be a monotone function over a lattice $(L, \sqsubseteq_L)$. Then the set of fixed points $Fix(f) = \{x \in L \mid f(x) = x\}$ is a lattice with respect to $\sqsubseteq_L$, too.* $\square$

Accordingly, the least and greatest fixed points of such an f, $lfp_{\sqsubseteq_L}$ and $gfp_{\sqsubseteq_L}$, can be characterised as greatest lower bound of pre-fixed points and least upper bound of post-fixed points of f:

$$\begin{aligned} lfp_{\sqsubseteq_L}(f) &= \bigsqcap Fix(f) = \bigsqcap \{x \in L \mid f(x) \sqsubseteq_L x\} \\ gfp_{\sqsubseteq_L}(f) &= \bigsqcup Fix(f) = \bigsqcup \{x \in L \mid f(x) \sqsupseteq_L x\}\,. \end{aligned}$$

In our case, the valuation transformer pre_G is a monotone function over the lattice $([0,1]^S, \leq)$, such that the existence of a fixed point $lfp_{\leq}(pre_G)$ is guaranteed [Wachter, 2011]. As a result, the reachability probability of a state s with respect to the goal set G is given by the least fixed point

$$Pr_{\mathcal{D},s}(\Diamond G) = (lfp_{\leq}(pre_G))(s).$$

2.1.5. Probabilistic Automata

Probabilistic Automata (PA) extend the DMTC formalism with non-determinism by allowing choosing from several labelled distributions in a state. This extension does no harm to the Markov property of the formalism, though. Groups of distributions can also be distinguished by the actions they can be reached with. It is important to realise that the randomness, which comes along with using probability distributions, is not a form of non-determinism, since in contrast to likelihood information, non-determinism models the absence of any information with respect to the outcomes of a choice.

Definition 2.19 (Probabilistic Automaton (PA))**.** A Probabilistic Automaton is a tuple

$$\mathcal{A} = (S, Act, U, \mathbf{P}, s_{init})$$

where

- S, U and s_{init} are the same as for DTMCs,
- Act is a finite set of actions, and
- $\mathbf{P} : S \to 2^{Act \times Dist_U(S)}$ is the transition function that assigns non-empty set of action-distribution pairs $\mathbf{P}(s)$ to a state $s \in S$.

$\square$

For convenience, we denote the set of distributions enabled at a state s by

$$\mathbf{P}_{\mu}(s) := \{\mu \in \mathit{Dist}_U(S) \mid (a,\mu) \in \mathbf{P}(s)\}.$$

A *path* in $\mathcal{A}$ is an infinite, alternating sequence of states and action-distribution pairs $\pi = s_0\ (a_0,\mu_0)\ s_1\ (a_1,\mu_1)\ s_2 \cdots \in (S \times \mathit{Act} \times \mathit{Dist}_U(S))^{\omega}$ such that every s_i has a transition to s_{i+1}, i.e. $(a_i,\mu_i) \in \mathbf{P}(s_i)$ and $\widehat{\mu_i}(s_{i+1}) > 0$. As for DTMCs, we employ $\mathit{Paths}(\mathcal{A})$ to denote the set of paths in $\mathcal{A}$, and respectively, $\mathit{Paths}_{\mathit{fin}}(\mathcal{A})$ for finite path fragments. In the sequel, let $\mathit{En}(s) := \{a \in \mathit{Act} \mid \exists (a,\mu) \in \mathbf{P}(s)\}$ for a $s \in S$, be the set of *enabled* actions in a state s.

Similar as with DTMCs, a PA $\mathcal{A} = (S, \mathit{Act}, U, \mathbf{P}, I)$ with a set of initial states $I = \{\mathit{init}_1, \ldots, \mathit{init}_n\}$ can be transformed to a semantically equivalent PA

$$\mathcal{A}' = (S \uplus \{s_{\mathit{init}}\}, \mathit{Act} \uplus \mathit{Act}_{\mathit{choice}}, U \cup \{u_\tau\}, \mathbf{P}', s_{\mathit{init}}),$$

with a single initial state and an initial non-deterministic choice over the states from I, where $\mathit{Act}_{\mathit{choice}} := \{\mathit{choose}_i \mid \mathit{init}_i \in I\}$, and

$$\mathbf{P}'(s) := \begin{cases} \{(\mathit{choose}_i, 1.0 : (u_\tau, \mathit{init}_i)) \mid \mathit{init}_i \in I\} & \text{if } s = s_{\mathit{init}} \\ \mathbf{P}(s) & \text{otherwise} \end{cases}.$$

Example 2.20. Figure 2.4 shows a PA with the set of actions $\{a, b\}$ and four states s_0, s_1, s_2 and s_3. Only the states s_0 and s_3 feature non-determinism as none of the others have several distributions, illustrated by the black nodes, to choose from. Thus, if the system is in the initial state s_0, choosing the distribution reachable by action a, will transfer it into state s_1 (or s_2) with the respective probability 0.3 (or 0.7), while choosing the distribution associated with action b, will allow it to reach either s_0, s_2 or s_3. Note that, according to the definition of PA, it is perfectly fine for states, like s_3, to have several outgoing transitions labeled with the same action.

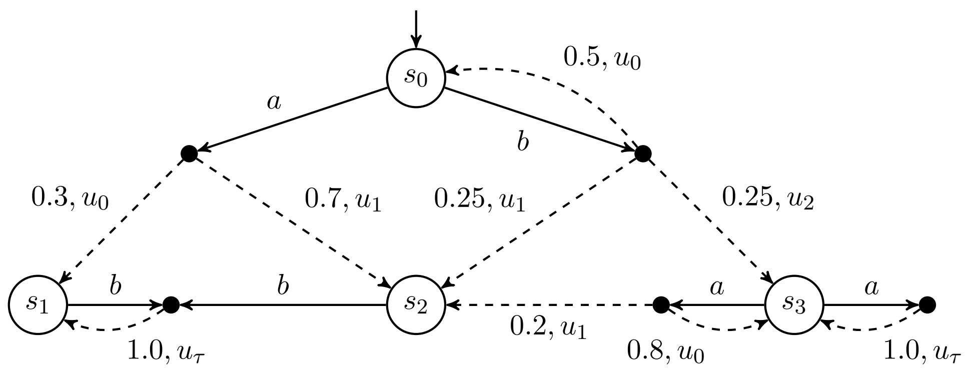


Figure 2.4.: Example of a PA

□

2.1.6. Minimal and Maximal Reachability Probabilities

Let $\mathcal{A} = (S, Act, U, \mathbf{P}, s_{init})$ be a PA and $G \subseteq S$ a set of goal states. If for all states of $\mathcal{A}$ there is exactly one distribution to choose from, i.e. $\forall_{s \in S} |\mathbf{P}(s)| = 1$, the PA doesn't feature non-determinism and can, ignoring the actions, as well be interpreted as a DTMC. Otherwise, the reachability probability is undefined as it may depend on the way non-determinism is resolved, e.g. a state might have a choice between either reaching itself or a goal state with probability 1. Hence, we generally don't have a unique reachability probability but rather a range of reachability probabilities, the most relevant ones being the minimal and maximal probabilities, respectively denoted by $Pr^{min}(\Diamond G)$ and $Pr^{max}(\Diamond G)$. These probabilities guarantee that no matter how the non-determinism is resolved, the probability of reaching G is bounded by them. To formalise how non-determinism is resolved we introduce the notion of *strategy*.

Definition 2.21 (Strategy for a PA)**.** Let $\mathcal{A} = (S, Act, U, \mathbf{P}, s_{init})$ be a PA. A (memoryless, deterministic) strategy, in literature also referred to as *scheduler* or *policy*, $\sigma : S \to Act \times Dist_U(S), s \mapsto c \in \mathbf{P}(s)$ for $\mathcal{A}$ is a function which resolves the non-determinism in a state s by choosing a unique element from $\mathbf{P}(s)$. □

There exist more general definitions for strategies but it has been shown that, due to PA having the Markov property, memoryless, deterministic strategies suffice when reasoning about minimal and maximal reachability probabilities [Baier and Katoen, 2008]. We denote the strategies minimising and maximising the reachability probability by σ_{min} and σ_{max}. The actual computation of both σ_{min} and σ_{max} is a bit more involved and deferred until chapter 3, where we derive strategies for even more general systems.

Since a strategy resolves all non-deterministic choices of a PA $\mathcal{A}$, the application of a strategy σ for $\mathcal{A}$ induces the DTMC $\mathcal{A}^\sigma$.

Definition 2.22 (Induced DTMC for PA)**.** Let $\mathcal{A} = (S, Act, U, \mathbf{P}, s_{init})$ be a PA and σ a strategy for $\mathcal{A}$. Then, the combination of both σ and $\mathcal{A}$ induces the DTMC

$$\mathcal{A}^\sigma = (S, U, \mathbf{P}_\sigma, s_{init}),$$

where $\mathbf{P}_\sigma(s) := \mu_s$ with $\sigma(s) = (a, \mu_s)$ for all $s \in S$. □

Note that $\mathcal{A}^\sigma$ in turn induces the probability measure $Pr_{\mathcal{A}^\sigma}$ on $Paths(\mathcal{A}^\sigma)$ which we conveniently denote by $Pr^\sigma_{\mathcal{A}}$. Using this notation we can formalise the minimal and maximal reachability probability as

$$\begin{aligned} Pr^{max}(\Diamond G) &:= \max_\sigma Pr^\sigma(\Diamond G) \\ Pr^{min}(\Diamond G) &:= \min_\sigma Pr^\sigma(\Diamond G), \end{aligned}$$

such that the inequality $Pr^{min}(\Diamond G) \leq Pr^\sigma_{\mathcal{A}}(\Diamond G) \leq Pr^{max}(\Diamond G)\,(\Diamond G)$ holds for all σ by definition. As for DTMCs we employ the subscript notation $Pr^\sigma_{\mathcal{A},s}$ when considering the reachability from any other state s than s_{init}.

Example 2.23. Reconsider the PA $\mathcal{A} = (S, Act, U, \mathbf{P}, s_{init})$ from Figure 2.4 and let $G = \{s_2\}$. Due to the fact that there exist only four different strategies for $\mathcal{A}$, it is easy to realise that the minimising strategy with respect to G is given by:

$$\sigma_{min}(s) := \begin{cases} (b, 0.5 : (u_0, s_0) \oplus 0.25 : (u_1, s_2) \oplus 0.25 : (u_2, s_3)) & \text{if } s = s_0 \\ (b, 1.0 : (u_\tau, s_1)) & \text{if } s = s_1 \\ (b, 1.0 : (u_\tau, s_1)) & \text{if } s = s_2 \\ (a, 1.0 : (u_\tau, s_3)) & \text{if } s = s_3 \end{cases}.$$

Employing this strategy induces the DTMC $\mathcal{A}^{\sigma_{min}}$ illustrated below.

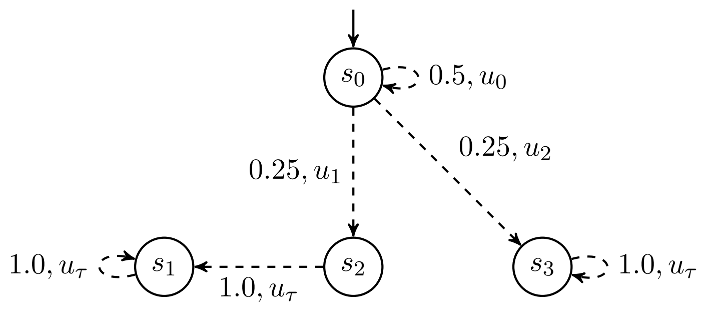


Figure 2.5.: Induced DTMC $\mathcal{A}^{\sigma_{min}}$

Referring to the induced DTMC, we can now conclude

$$Pr^{min}(\lozenge G) = Pr^{\sigma_{min}}_{\mathcal{A}}(\lozenge G) = Pr_{\mathcal{A}^{\sigma_{min}}}(\lozenge G) = \frac{1}{4} \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^n = \frac{1}{2}$$

□

Fixed Point Characterisation

As for DTMCs, probabilistic reachability for PA is expressible as a recursive system of equations [Baier and Katoen, 2008]. However, in contrast to the valuation transformer for DTMCs, the minimal reachability probability of a state is not simply a weighted sum of the reachability probabilities of the successors but the smallest weighted sum, respectively the biggest weighted sum for maximal reachability probability. Accordingly, for minimal probabilistic reachability the valuation transformer $pre_G^- : [0,1]^S \to [0,1]^S$ is given by

$$pre_G^-(w)(s) := \begin{cases} 1 & \text{if } s \in G \\ 0 & \text{if } s \in G_0 \\ \min\limits_{(a,\mu) \in \mathbf{P}(s)} \sum\limits_{s' \in S} \widehat{\mu}(s') \cdot w(s') & \text{otherwise} \end{cases},$$

where $G_0 \subseteq S$ denotes the set of states, for which there exists no path to any of the states from G. The minimal reachability probability for a state s is respectively given by the least fixed point

$$Pr_s^{min}(\Diamond G) = \left(\mathit{lfp}_{\leq} pre_G^{-}\right)(s),$$

see [Baier and Katoen, 2008]. Analogously, for the maximal probabilistic reachability the valuation transformer $pre_G^{+} : [0,1]^S \to [0,1]^S$ is given by

$$pre_G^{+}(w)(s) := \begin{cases} 1 & \text{if } s \in G \\ 0 & \text{if } s \in G_0 \\ \max\limits_{(a,\mu)\in\mathbf{P}(s)} \sum\limits_{s' \in S} \widehat{\mu}(s') \cdot w(s') & \text{otherwise} \end{cases},$$

and the maximal reachability probability for a state s is given by the least fixed point

$$Pr_s^{max}(\Diamond G) = \left(\mathit{lfp}_{\leq} pre_G^{-}\right)(s).$$

2.1.7. The Prism Modelling Language

The PRISM model checker [Kwiatkowska et al., 2011] is the most popular modelling and verification environment for finite probabilistic systems. Most importantly it features a widely spread modelling language, based on [Alur and Henzinger, 1999], suitable to describe probabilistic systems in terms of *probabilistic programs*. This allows formalising real-world systems in a more concise, structured and modular manner, in contrast to large and confusing PA.

Probabilistic programs consist of modules which in turn consist of variables and guarded commands over these variables. To facilitate modular modelling, the PRISM language defines CSP-style composition operators [Hoare, 1985, Roscoe et al., 1997], which synchronise equally labeled commands of different modules. However, for the sake of convenience we only consider single-module programs. This does not restrict the set of supported programs though since systems modelled by several modules can be flattened to single-module programs using the transformations presented in [Katoen et al., 2010]. In the following, we present a simplified version of the PRISM modelling language and define the corresponding semantics in terms of PA, complying with the description of [Wachter, 2011] and repeatedly referring to the simple probabilistic program from Listing 2.1 for better grasp of the matter.

The example program P_{simple} models a system which has four phases and needs to perform two runs when it is in its running phase. The different phases are modelled by a bounded integer variable *phase* which can take on four different values, while the number of remaining runs is modelled by an unbounded integer variable *run*. After the initialisation the system is put into the running phase, i.e. *phase* is set to one, and the number of remaining runs to perform is set to two. During a run there is a 3% chance of the system breaking, i.e. ending up in phase three. If the system does not break, the number of remaining runs will be decremented by one. This continues until no runs remain, i.e. $run \leq 0$.

```
 1  mdp
 2
 3  module simple
 4      phase : [0..3]; // 0=init, 1=running, 2=finished, 3=broken
 5      run   : int;    // -1 -> 2 -> 1 -> 0
 6      [a] phase=0            -> 1.0 :(run'=2) & (phase'=1);
 7      [b] phase=1 & run>0    -> 0.97:(run'=run-1) + 0.03:(phase'=3);
 8      [c] phase=1 & run<=0   -> 1.0 :(phase'=2);
 9  endmodule
10
11  init
12      phase=0 & run=-1;
13  endinit
```

Listing 2.1: Simple probabilistic program P_{simple}

Syntax

A single-module probabilistic program P consists of a set of typed variables *Var*, commands *Cmd* and a Boolean expression *init* which characterises the initial state.

Variables can only be of Boolean or integer type. Our example program does not make use of Boolean variables but employs both bounded and unbounded integers, see *phase* and *run* variables in lines 4 and 5. The set of variables is accordingly given by $\mathit{Var} = \{\mathit{phase}, \mathit{run}\}$. We require expressions over *Var* to be part of some quantifier-free fragment of first-order logic which comprises Boolean combinations of arithmetic expressions, and use $\mathit{Expr}_{\mathit{Var}}$ and $\mathit{BExpr}_{\mathit{Var}} \subseteq \mathit{Expr}_{\mathit{Var}}$ to denote expressions and Boolean expressions over *Var*. The *initial state expression*, refer to line 12, is such a Boolean expression.

State transitions are realised via the concept of commands. A command has a unique label called *action*, a *guard* from $\mathit{BExpr}_{\mathit{Var}}$ and several probabilistic *alternatives* separated by a plus sign. The latter consist of assignments $\mathit{Var} \to \mathit{Expr}_{\mathit{Var}}$, where variables which are not mentioned explicitly keep their values tacitly, and a probability, such that the probability weights of all alternatives sum up to one. Note that we can safely assume actions to be unique because we only consider flattened programs, where actions are not needed for synchronisation and can be renamed arbitrarily. We formalise the syntax of commands as follows.

Definition 2.24 (Command)**.** A command is a tuple $c = (a, g, ((p_1, E_1), \ldots, ((p_k, E_k)))$, denoted by

$$[a]\ g \to p_1 : \mathit{Var}' = E_1 + \cdots + p_k : \mathit{Var}' = E_k,$$

with

- a unique label a called *action*,
- a guard $g \in \mathit{BExpr}_{\mathit{Var}}$, and

- assignments $E_1, \ldots, E_k$ in linear arithmetic, where $\mathit{Var}' = E$ denotes the simultaneous update of variables Var according to E, weighted with probabilities $p_1, \ldots, p_k$ which sum up to one, i.e. $\sum_i p_i = 1$.

$\square$

We utilise a_c and g_c to denote the action and guard of a command c. Also, let $\mathit{deg}(c)$ denote the number of assignments k.

Definition 2.25 (Probabilistic Program)**.** A probabilistic program

$$P = (\mathit{Var}, \mathit{VarType}, \mathit{Act}, \mathit{Cmd}, \mathit{init})$$

consists of

- a finite set of variables Var,
- a mapping $\mathit{VarType} : \mathit{Var} \to \{\mathit{int}[a, b] \mid a, b \in \mathbb{N} \cup \{-\infty, \infty\}, a < b\} \cup \{\mathit{bool}\}$ of variables to types,
- a finite set of actions $\mathit{Act} := \bigcup_{c \in \mathit{Cmd}} a_c$, where
- Cmd is a finite set of commands over Var, and
- the initial state expression $\mathit{init} \in \mathit{BExpr}_{\mathit{Var}}$.

$\square$

Example 2.26. The probabilistic program P_{simple} from Listing 2.1 is formalised as follows.

$$P_{\mathit{simple}} = (\mathit{Var}, \mathit{VarType}, \mathit{Act}, \mathit{Cmd}, \mathit{init})$$

where

- $\mathit{Var} := \{\mathit{phase}, \mathit{run}\}$ and
- $VarType : \mathit{phase} \mapsto \mathit{int}[0, 3], \mathit{run} \mapsto \mathit{int}[-\infty, \infty]$ (lines 4-5),
- $\mathit{Act} := \{a, b, c\}$
- $\mathit{Cmd} := \{c_a, c_b, c_c\}$ (lines 6-8), where

$$\begin{aligned}
c_a &:= (a, \mathit{phase} = 0, (1.0, \{\mathit{phase} \mapsto 1, \mathit{run} \mapsto 2\})) \\
c_b &:= (b, \mathit{phase} = 1 \land \mathit{run} > 0, (0.97, E_1), (0.03, E_2)) \\
&\quad\; E_1 : \mathit{phase} \mapsto \mathit{phase}, \mathit{run} \mapsto \mathit{run} - 1 \\
&\quad\; E_2 : \mathit{phase} \mapsto 3, \mathit{run} \mapsto \mathit{run} \\
c_c &:= (c, \mathit{phase} = 1 \land \mathit{run} \le 0, (1.0, \{\mathit{phase} \mapsto 2, \mathit{run} \mapsto \mathit{run}\}))
\end{aligned}$$

- $\mathit{init} := \mathit{phase} = 0 \land \mathit{run} = -1$ (line 12).

$\square$

Semantics

Let $P = (\mathit{Var}, \mathit{VarType}, \mathit{Act}, \mathit{Cmd}, \mathit{init})$ be a probabilistic program, then its semantics is a PA $[\![P]\!]$. For illustration, Figure 2.6 shows the (reachable part of the) semantics of the example program from Figure 2.1.

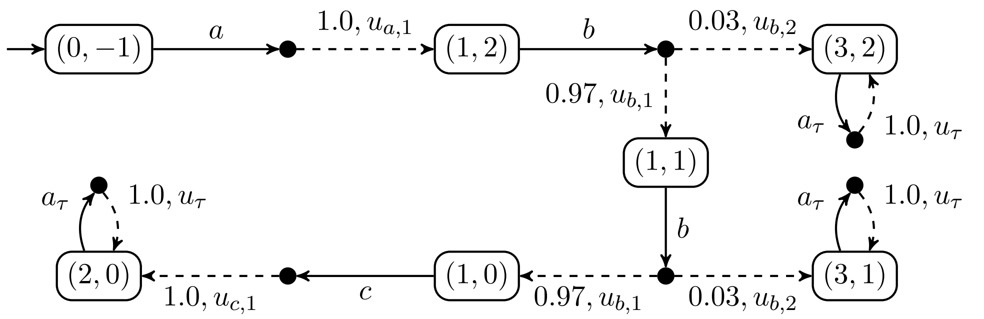


Figure 2.6.: Semantics $[\![P_{\mathit{simple}}]\!]$ of example probabilistic program P_{simple}

Let $\mathit{dom} : \mathit{Var} \to 2^{\mathbb{N}} \cup \{-\infty, \infty\}$ be the function mapping each variable to its domain, i.e.

$$\mathit{dom}(\mathit{var}) := \begin{cases} \{0,1\} & \text{if } \mathit{VarType}(\mathit{var}) = \mathit{bool} \\ [a,b] \cap \mathbb{N} & \text{if } \mathit{VarType}(\mathit{var}) = \mathit{int}[a,b] \end{cases}.$$

Notice, that we treat the Boolean values $\{\mathit{false}, \mathit{true}\}$ as $\{0, 1\}$ with 1 being *true*, and that $|\mathit{dom}(\mathit{var})|$ may be infinite, since we allow for integers to be unbounded. A state of P is a valuation function $\nu : \mathit{Var} \to \mathbb{N}$ assigning a variable $\mathit{var} \in \mathit{Var}$ an element from its semantic domain, i.e. $\nu(\mathit{var}) \in \mathit{dom}(\mathit{var})$. For example, a tuple (p, r) in Figure 2.6 indicates the valuation $\{\mathit{phase} \mapsto p, \mathit{run} \mapsto r\}$, i.e. the initial state is $s_{\mathit{init}} = \{\mathit{phase} \mapsto 0, \mathit{run} \mapsto -1\}$. The state space is accordingly given as the set of all states

$$S(P) := \{\nu \mid \nu \text{ is a state of } P\}.$$

Semantics of Expressions We denote the evaluation of an expression $e \in \mathit{Expr}_{\mathit{Var}}$ in a given state $s \in S(P)$ by $[\![e]\!]_s$. In particular, we say that a state s satisfies a Boolean expression $e \in \mathit{BExpr}_{\mathit{Var}}$, denoted by $s \models e$, if $[\![e]\!]_s = 1$, i.e. $\{\mathit{phase} \mapsto 1, \mathit{run} \mapsto 2\} \models \mathit{phase} > 0 \wedge \mathit{run} > 0$. Conversely, we denote the set of states which satisfy e by $[\![e]\!] := \{s \in S(P) \mid s \models e\}$, i.e. the set of initial states is given by $[\![\mathit{init}]\!]$.

Semantics of Commands The semantics of a command c is a set of action-distribution pairs viable in a state, i.e. $[\![c]\!] \subseteq S(P) \times \mathit{Act} \times \mathit{Dist}_U(S(P))$. A pair (a_c, μ) is viable in a state s, if s satisfies the guard of the command labelled with a_c, i.e. $s \models g_c$, with $\mu := \bigoplus_{i=1}^{\mathit{deg}(c)} p_i : (u_i, s_i)$ being derived from the command's assignments. The successor state s_i is determined by updating the state variables according to assignment E_i, i.e.

$s_i = \lambda_{var \in Var} \llbracket E_i(var) \rrbracket_s$, with lambda indicating that s_i is again a mapping to semantic values. Formally, the semantics are defined as follows [Wachter, 2011]:

Definition 2.27 (Command Semantics)**.** Let

$$c = (a, g, ((p_1, E_1), \ldots, ((p_k, E_k)))$$

be a command of program P. Its semantics $\llbracket c \rrbracket$ is given by:

$$\left\{ \left(s_0, a, \bigoplus_{i=1}^{k} p_i : (u_{a,i}, s_i) \right) \middle| \begin{array}{ll} s_0, \ldots, s_k \in S(P) & \\ s_0 \models g & \text{“guard”} \\ \forall_{i \in \{1,\ldots,k\}}\ s_i = \lambda_{var \in Var} \llbracket E_i(var) \rrbracket_{s_0} & \text{“updates”} \end{array} \right\}.$$

□

As a result, a tuple $(s, a_c, \mu) \in \llbracket c \rrbracket$ induces a transition from the state s to a distribution μ, which is reachable by an action a_c.

Example 2.28. Let P_{simple} be the program which we defined in example 2.26 and whose semantics we illustrated in Figure 2.6. Consider its command

$$\begin{array}{rcl} c_b & := & (b, phase = 1 \wedge run > 0, (0.97, E_1), (0.03, E_2)) \\ & & E_1 : phase \mapsto phase, run \mapsto run - 1 \\ & & E_2 : phase \mapsto 3, run \mapsto run \end{array}$$

Its semantics $\llbracket c_b \rrbracket$ is

$$\left\{ \left(s_0, b, \bigoplus_{i=1}^{k} p_i : (u_{b,i}, s_i) \right) \middle| \begin{array}{ll} s_0, \ldots, s_2 \in S(P) & \\ s_0 \models phase = 1 \wedge run > 0 & \text{“guard”} \\ s_1(phase) = s_0(phase), s_1(run) = s_0(run) - 1 & \text{“update 1”} \\ s_2(phase) = 3, s_2(run) = s_0(run) & \text{“update 2”} \end{array} \right\},$$

with $p_1 = 0.97$ and $p_2 = 0.03$.

Consider $s = \{phase \mapsto 1, run \mapsto 2\}$. Since s clearly satisfies the guard $phase = 1 \wedge run > 0$, there is a $(s, b, \mu) \in \llbracket c_b \rrbracket$ with $\mu : 0.97 : (u_{b,1}, s_1) \oplus 0.03 : (u_{b,2}, s_2)$, where

$$s_1 = \begin{cases} phase \mapsto s(phase) = 1 \\ run \mapsto s(run) - 1 = 1 \end{cases}$$

and

$$s_2 = \begin{cases} phase \mapsto 3 \\ run \mapsto s(run) = 2 \end{cases}.$$

The illustration of the action-distribution pairs of state s is part of Figure 2.6. From the figure it is easy to see that there exists a tuple $(s', b, \mu') \in \llbracket c_b \rrbracket$ for the state $s' = \{phase \mapsto 1, run \mapsto 1\}$, too. □

Semantics of Programs The semantics of a program is a PA whose states are the derived from the program variables' domains. The subset of states which satisfies the initial state expression becomes the set of initial states. Remember that we can transform the PA to have only one initial state, though. Both actions and updates can be derived from the commands, with the actions being the union of the commands' labels and the set of updates being determined by every command's number of assignments. Note that there might be states which don't have any action enabled, i.e. do not satisfy any command's guard. We have to introduce an auxiliary action $a_\mathcal{T}$, unrelated to any command, to add self-loops to such *deadlocks* since the transition function must be defined for all states. Overall, this yields the following semantics:

Definition 2.29 (Program Semantics)**.** The semantics of a program

$$P = (\mathit{Var}, \mathit{VarType}, \mathit{Act}_P, \mathit{Cmd}, \mathit{init})$$

is a PA

$$[\![P]\!] = (S, \mathit{Act}, U, \mathbf{P}, I),$$

with

- the set of states $S := S(P)$,
- the set of actions $\mathit{Act} := \mathit{Act}_P \uplus \{a_\mathcal{T}\}$,
- the set of updates $U := \{u_{a_c,i} \mid c \in \mathit{Cmd}, i \in \{1, 2, \dots, \mathit{deg}(c)\}\} \cup \{u_\mathcal{T}\}$, where the indices indicate from which command and assignment they originate,
- the transition function

  $$\mathbf{P}(s) := \begin{cases} \{(a_c, \mu)\} & \text{if } (s, a_c, \mu) \in [\![c]\!] \\ \{(a_\mathcal{T}, 1.0 : (u_\mathcal{T}, s))\} & \text{if } s \not\models g_{\mathit{Cmd}} \end{cases},$$

  where $g_{\mathit{Cmd}} := \bigvee_{c \in \mathit{Cmd}} g_c$, and
- the set of initial states $I := [\![\mathit{init}]\!]$.

□

From this definition it is easy to see that, for any state s, there are no two action-distribution pairs in $\mathbf{P}(s)$ with the same action, i.e. an action uniquely specifies a distribution. Thus for a program P and the respective PA $[\![P]\!] = (S, \mathit{Act}, U, \mathbf{P}, I)$ we employ the more convenient notation

$$\mathbf{P}(s, a) = \mu,$$

where $(a, \mu) \in \mathbf{P}(s)$.

2.1.8. Stochastic Games

Now that we have formalised PA and have a suitable modelling language to express complex PA, what do we need another formalism for? In fact, we will not use Stochastic Games for modelling purposes, instead they will become the *abstract domain* of computation, as we will see in section 2.2.

Similar as PA extend DTMCs with a level of non-determinism, Stochastic Games [Condon, 1992] extend PA with another level of non-determinism. Accordingly, besides stochastic choice, they feature two *players*, each of which resolves a level of non-determinism. Such games are therefore often referred to as $2\frac{1}{2}$-player games. Most importantly, the players may act adversarial to the effect that, if one player tries to maximise (or minimise) the probability of reaching a certain vertex, the other player seeks to minimise (or maximise) this probability.

Stochastic Games are digraph-like structures, where the vertices are partitioned into three sets – the player 1-, player 2- and stochastic choice-vertices. Stochastic choice vertices, like states of DTMCs, are distributions over the set of player 1 vertices. Their predecessors are player 2 vertices which, like states of PA, resolve non-determinism by choosing a successor, in our case a distributions over player 1 vertices. Player 1 vertices, in turn, are predecessors of player 2 vertices, and choose from their successors. Keep in mind that this formalism, too, has the Markov property as both the choices and the probability distributions are independent of the system's execution history.

Definition 2.30 (Stochastic Game (SG))**.** A Stochastic Game is a tuple

$$\mathcal{G} = ((V, E), (V_1, V_2, V_p), U, v_{init}),$$

where

- (V, E) is a digraph with edges $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_p) \cup (V_p \times V_1)$,
- (V_1, V_2, V_p) is a partition of V, i.e. $V_1 \uplus V_2 \uplus V_p = V$, where
  - V_1 is the set of player 1 vertices,
  - V_2 is the set of player 2 vertices,
  - $V_p \subseteq \mathit{Dist}_U(V_1)$ is the set of probabilistic vertices, where for all $\mu \in V_p$ $\mu(u, v') > 0$ implies $(\mu, v') \in E$,
- U is, as for PA, the set of updates,
- and the initial vertex v_{init}.

$\square$

For convenience, we denote the successors of a vertex by $E(v) := \{v' \in V \mid (v, v') \in E\}$. A *path* π in a SG $\mathcal{G}$ is an infinite sequence of tuples of vertices

$$\pi = (v_{0,1}\ v_{0,2}\ v_{0,p})\ \ (v_{1,1}\ v_{1,2}\ v_{1,p}) \cdots \in (V_1 \times V_2 \times V_p)^{\omega},$$

such that $(v_{i,1}, v_{i,2}), (v_{i,2}, v_{i,p}) \in E$ and $\widehat{v_{i,p}}(v_{i+1,1}) > 0$. As for PA, we employ $\mathit{Paths}(\mathcal{G})$ to denote the set of paths in $\mathcal{G}$.

Similar as with PA, a SG $\mathcal{G} = ((V, E), (V_1, V_2, V_p), U, I)$ with a set of initial vertices $I = \{\mathit{init}_1, \ldots, \mathit{init}_n\}$ can be transformed to a semantically equivalent SG

$$\mathcal{G}' = \big((V', E'), (V_1', V_2', V_p'), U \cup \{u_\tau\}, v_{\mathit{init}}\big),$$

where

- $V_1' := V_1 \uplus \{v_{\mathit{init}}\}$,
- $V_2' := V_2 \uplus \{\mathit{choose}_{i,2} \mid \mathit{init}_i \in I\}$,
- $V_p' := V_p \uplus \{\mathit{choose}_{i,p} \mid \mathit{init}_i \in I\}$, with $\mathit{choose}_{i,p} := 1.0 : (u_\tau, \mathit{init}_i)$
- $E' := E \uplus \{(v_{\mathit{init}}, \mathit{choose}_{i,2}) \mid \mathit{init}_i \in I\}$
  $\uplus \{(\mathit{choose}_{i,2}, \mathit{choose}_{i,p}) \mid \mathit{init}_i \in I\}$
  $\uplus \{(\mathit{choose}_{i,p}, \mathit{init}_i) \mid \mathit{init}_i \in I\}$

with a single initial vertex and an initial non-deterministic choice over vertices from I.

Example 2.31. Figure 2.7 shows a SG with 16 vertices, where the black circles represent probabilistic vertices, the grey squares are player 2 vertices, and the remaining ones belong to player 1. Intuitively, the semantics of the visualisation hardly differ from the semantics of PA.

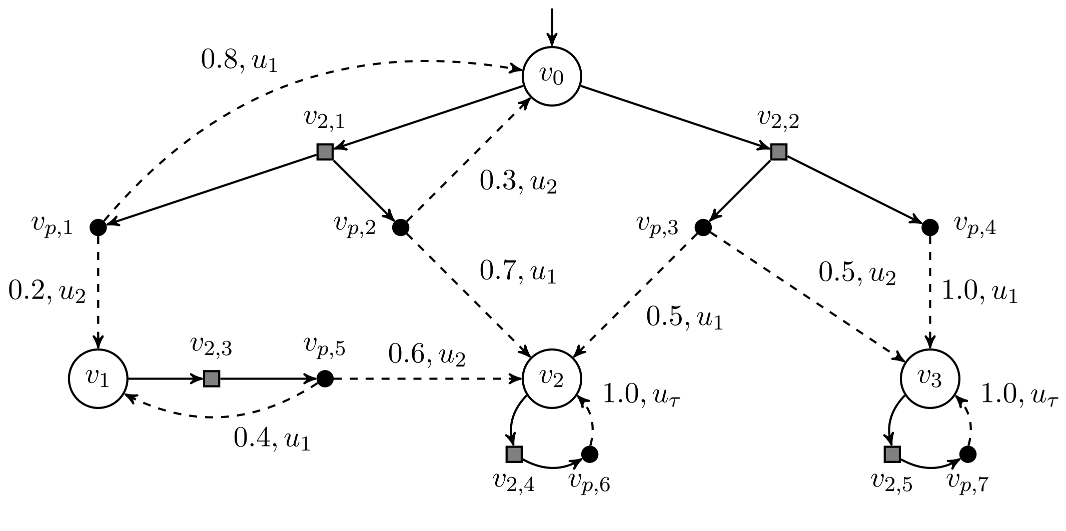


Figure 2.7.: Example of a SG

For example, in the initial vertex v_0, player 1 may choose between its successors $v_{2,1}$ and $v_{2,2}$. Choosing $v_{2,1}$ will transfer the system into a player 2 vertex, where player 2 may choose from its probabilistic successor vertices $E(v_{2,1}) = \{v_{p,1}, v_{p,2}\}$. If player 2 chooses $v_{p,2}$ the system will either return to vertex v_0 with probability 0.3 or transfer into v_2 with probability 0.7, according to the distribution $v_{p,2}$. $\square$

Reachability Probability for Stochastic Games

Let $\mathcal{G} = ((V, E), (V_1, V_2, V_p), U, v_{init})$ be a SG and $G \subseteq V$ a set of goal vertices. If for every player 2 vertex there is exactly one distribution to choose from, i.e. $\forall_{v \in V} |E(v)| = 1$, the SG features at most one level of non-determinism and can as well be interpreted as a PA. Otherwise, as for PA, the reachability probability is undefined if non-determinism is present, since it depends on how the non-determinism is resolved in each vertex. However, in contrast to PA, we now have two possibly adversarial players such that it does not suffice to distinguish between a minimising and maximising strategy anymore. Instead, to define a probability measure on paths, two strategies are needed to resolve both levels of non-determinism separately.

Definition 2.32 (Strategy Pair for SG)**.** Let $\mathcal{G} = ((V, E), (V_1, V_2, V_p), U, v_{init})$ be a SG. A (memoryless, deterministic) strategy pair is a pair (σ_1, σ_2) of player 1 and player 2 strategies, where

$$\sigma_i : V_i \to V, \sigma_i(v) \mapsto v' \in E(v), i \in \{1, 2\}\,,$$

such that each player resolves non-determinism in his vertices by choosing to which vertex the system transfers to next. $\square$

Analogous to strategies for PA, a pair of strategies σ_1 and σ_2 induces a DTMC, denoted by $\mathcal{G}^{\sigma_1,\sigma_2}$.

Definition 2.33 (Induced DTMC for SG)**.** Let $\mathcal{G} = ((V, E), (V_1, V_2, V_p), U, v_{init})$ be a SG and (σ_1, σ_2) be a strategy pair for $\mathcal{G}$. Then, the combination of both (σ_1, σ_2) and $\mathcal{G}$ induces the DTMC

$$\mathcal{G}^{\sigma_1,\sigma_2} = (S, U, \mathbf{P}_{\sigma_1,\sigma_2}, v_{init}),$$

where $S := V$, and for all $s \in S$, $\mathbf{P}_{\sigma_1,\sigma_2}(s) := \sigma_2(\sigma_1(s))$. $\square$

Note that $\mathcal{G}^{\sigma_1,\sigma_2}$ in turn induces the probability measure $Pr_{\mathcal{G}^{\sigma_1,\sigma_2}}$ on $Paths(\mathcal{G}^{\sigma_1,\sigma_2})$ which we conveniently denote by $Pr_{\mathcal{G}}^{\sigma_1,\sigma_2}$. As for PA we employ the subscript notation $Pr_{\mathcal{G},v}^{\sigma_1,\sigma_2}$ when considering the reachability from any other vertex v than v_{init}.

Example 2.34. Reconsider the SG $\mathcal{G}$ from Figure 2.7 and let (σ_1, σ_2) be a strategy pair for $\mathcal{G}$, where the strategies are defined as

$$\begin{aligned} \sigma_1 &:= \{v_0 \mapsto v_{2,1}, v_1 \mapsto v_{2,3}, v_2 \mapsto v_{2,4}, v_3 \mapsto v_{2,5}\} \\ \sigma_2 &:= \{v_{2,1} \mapsto v_{p,2}, v_{2,2} \mapsto v_{p,4}, v_{2,3} \mapsto v_{p,5}, v_{2,4} \mapsto v_{p,6}, v_{2,5} \mapsto v_{p,7}\}\,. \end{aligned}$$

Employing this strategy pair induces the DTMC $\mathcal{G}^{\sigma_1,\sigma_2}$ illustrated below.

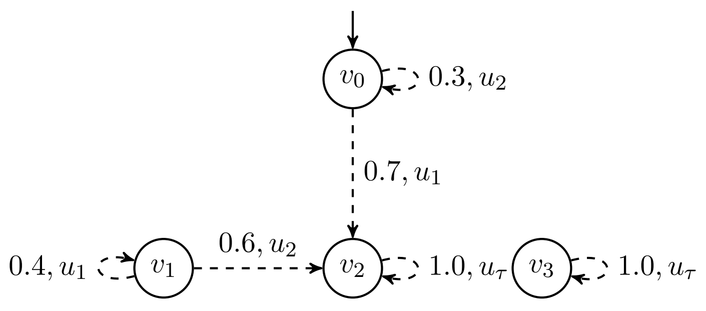


Figure 2.8.: Induced DTMC $\mathcal{G}^{\sigma_1,\sigma_2}$

□

As for PA, the most relevant strategy pairs are the extremal ones which enclose a whole range of reachability probabilities with respect to a set of goal vertices. Note that two of the extremal reachability probabilities result from the players' collaboration:

$$\sup_{\sigma_1}\sup_{\sigma_2} Pr_{\mathcal{G}}^{\sigma_1,\sigma_2}(\Diamond G)$$
$$\inf_{\sigma_1}\inf_{\sigma_2} Pr_{\mathcal{G}}^{\sigma_1,\sigma_2}(\Diamond G).$$

As a result, for these cases, we can compute probabilistic reachability by treating both levels of non-determinism like a single level and employing the methods for PA. However, for the extremal probabilities where both players act adversarially

$$\sup_{\sigma_1}\inf_{\sigma_2} Pr_{\mathcal{G}}^{\sigma_1,\sigma_2}(\Diamond G)$$
$$\inf_{\sigma_1}\sup_{\sigma_2} Pr_{\mathcal{G}}^{\sigma_1,\sigma_2}(\Diamond G),$$

we have to either determine the respective strategies or use another approach, e.g. a fixed point characterisation of the probabilities. As for PA, the actual computation of the extremal strategies is a bit more involved and deferred until chapter 3.

Fixed Point Characterisation

Similar to PA, probabilistic reachability for SG is expressible as a recursive system of equations. When constructing the system of equations for PA, we had to consider the maximal (or minimal) weighted sums of the reachability probabilities of the successors to handle a single level of non-determinism. For two levels of non-determinism, we have to apply this scheme twice, i.e. let player 1 minimise (or maximise) over the reachability probabilities of its successor player 2 vertices, while player 2 minimises (or maximises) over the weighted sum of probabilities of successor player 1 vertices.

Accordingly, in the case of *adversary* players, the minimal probabilistic reachability is given by the least fixed point of the valuation transformer $pre_G^{-+} : [0,1]^{V_1} \to [0,1]^{V_1}$:

$$pre_G^{-+}(w)(v) := \begin{cases} 1 & \text{if } v \in G \\ 0 & \text{if } v \in G_0 \\ \min\limits_{v_2 \in E(v)} \max\limits_{v_p \in E(v_2)} \sum\limits_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v') & \text{otherwise} \end{cases},$$

where $G_0 \subseteq S$ denotes the set of vertices, for which there exists no path to any of the vertices from G. Analogously, the maximal probabilistic reachability for adversary players is given by the least fixed point of the valuation transformer $pre_G^{+-} : [0,1]^{V_1} \to [0,1]^{V_1}$:

$$pre_G^{+-}(w)(v) := \begin{cases} 1 & \text{if } v \in G \\ 0 & \text{if } v \in G_0 \\ \max\limits_{v_2 \in E(v)} \min\limits_{v_p \in E(v_2)} \sum\limits_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v') & \text{otherwise} \end{cases}.$$

For clarity, we also illustrate the valuation transformers for both minimal and maximal probabilistic reachability in case of *collaborating* players:

$$pre_G^{--}(w)(v) := \begin{cases} 1 & \text{if } v \in G \\ 0 & \text{if } v \in G_0 \\ \min\limits_{v_2 \in E(v), v_p \in E(v_1)} \sum\limits_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v') & \text{otherwise} \end{cases}$$

and

$$pre_G^{++}(w)(v) := \begin{cases} 1 & \text{if } v \in G \\ 0 & \text{if } v \in G_0 \\ \max\limits_{v_2 \in E(v), v_p \in E(v_2)} \sum\limits_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v') & \text{otherwise} \end{cases}.$$

As mentioned before, in the case of collaborating players, we essentially employ the same methods as for PA. This is especially easy to see considering the structural resemblance of the respective valuation transformers. Overall, we get

$$\begin{aligned} \inf_{\sigma_1} \sup_{\sigma_2} Pr_{\mathcal{G},v}^{\sigma_1,\sigma_2}(\Diamond G) &= \left(lfp_{\leq}\left(pre_G^{-+}\right)\right)(v) \\ \sup_{\sigma_1} \inf_{\sigma_2} Pr_{\mathcal{G},v}^{\sigma_1,\sigma_2}(\Diamond G) &= \left(lfp_{\leq}\left(pre_G^{+-}\right)\right)(v) \\ \inf_{\sigma_1} \inf_{\sigma_2} Pr_{\mathcal{G},v}^{\sigma_1,\sigma_2}(\Diamond G) &= \left(lfp_{\leq}\left(pre_G^{--}\right)\right)(v) \\ \sup_{\sigma_1} \sup_{\sigma_2} Pr_{\mathcal{G},v}^{\sigma_1,\sigma_2}(\Diamond G) &= \left(lfp_{\leq}\left(pre_G^{++}\right)\right)(v), \end{aligned}$$

to compute probabilistic reachability for SG. [Condon, 1992, Wachter, 2011]

2.2. Abstraction

We are interested in computing lower and upper bounds on reachability probabilities but cannot use the straightforward approach of analysing an actual PA due to its usually large state space. Ideally, we would like to perform the analysis on an over-approximating *abstraction* of the system, which due to lumping of "similar" states, has a smaller state space but preserves enough information to derive valuable information. This section illustrates how the previously introduced formalisms fit into the *abstract interpretation* framework [Cousot and Cousot, 1977, Cousot and Cousot, 1979] by establishing a relation between valuations for PA as the *concrete domain* and valuations for SG as the *abstract domain.* To this end we recapitulate the formalisms of [Wachter, 2011].

2.2.1. Relating Concrete and Abstract Domain

To begin with, let S be a set of states and Q be a partition of S, which indicates a lumping of states, i.e.

$$S = \biguplus_{B \in Q} B,$$

where a *block* $B \subseteq S$ aggregates a set of states $s \in S$. We write $\overline{s}$ to denote the unique block B subsuming a state s, i.e. $s \in B$. Analogously, distributions over states are lifted to distributions over blocks, i.e. for a $\pi \in \mathit{Dist}_U(S)$ the lifted distribution is given by $\overline{\pi} \in \mathit{Dist}_U(Q)$ with $\overline{\pi}(u, \overline{s}) = \pi(u, s)$.

For the computation of probabilistic reachability we employed a fixed point iteration over the lattice $([0,1]^S, \leq)$ – our concrete domain of computation. However, lumping states according to a partition, we cannot assign probabilities to single states anymore but merely to blocks of states. This naturally gives rise to use the lattice $([0,1]^Q, \leq)$ as the abstract domain for computation. Given a concrete valuation $w \in [0,1]^S$, an abstract valuation $w^\# \in [0,1]^Q$ should preserve the property of interest with respect to the partitioning. For example, if we are interested in the minimal probabilities, the *abstraction* must yield an over-approximation of w, i.e. ideally the valuation of a block should correspond to the smallest valuation of the block's encompassed states. Furthermore, to minimise loss of precision, the abstraction after *concretisation* of an abstract valuation $w^\#$ should then again yield $w^\#$. The concept of a *Galois connection* is known to formalise this very requirements.

Definition 2.35 (Galois Connection)**.** Let $(L, \sqsubseteq_L)$ and $(M, \sqsubseteq_M)$ be complete lattices. A pair (α, γ) of monotonic functions

$$\alpha : L \to M \text{ and } \gamma : M \to L$$

is a Galois connection if

$$\forall_{l \in L}\ l \sqsubseteq_L \gamma(\alpha(l)) \text{ and } \forall_{m \in M}\ \alpha(\gamma(m)) \sqsubseteq_M m$$

□

It has been shown [Wachter, 2011], that for the lattices $([0,1]^S, \leq)$ and $([0,1]^Q, \leq)$ both (α^l, γ) and (α^u, γ), with the lower and upper bound *abstraction* functions

$$\alpha^l : [0,1]^S \to [0,1]^Q, \quad \alpha^l(w)(B) := \inf_{s \in B} w(s) \text{ for all } B \in Q$$
$$\alpha^u : [0,1]^S \to [0,1]^Q, \quad \alpha^u(w)(B) := \sup_{s \in B} w(s) \text{ for all } B \in Q,$$

and the shared *concretisation* function

$$\gamma : [0,1]^Q \to [0,1]^S, \ \gamma(w^{\#})(s) = w^{\#}(b) \text{ for all } s \in S$$

are Galois connections, which have the desired behaviour.

Example 2.36. The figure below illustrates the application of abstraction and concretisation on a small scale. The concrete domain has a blue tint while the abstract domain is tinted red.

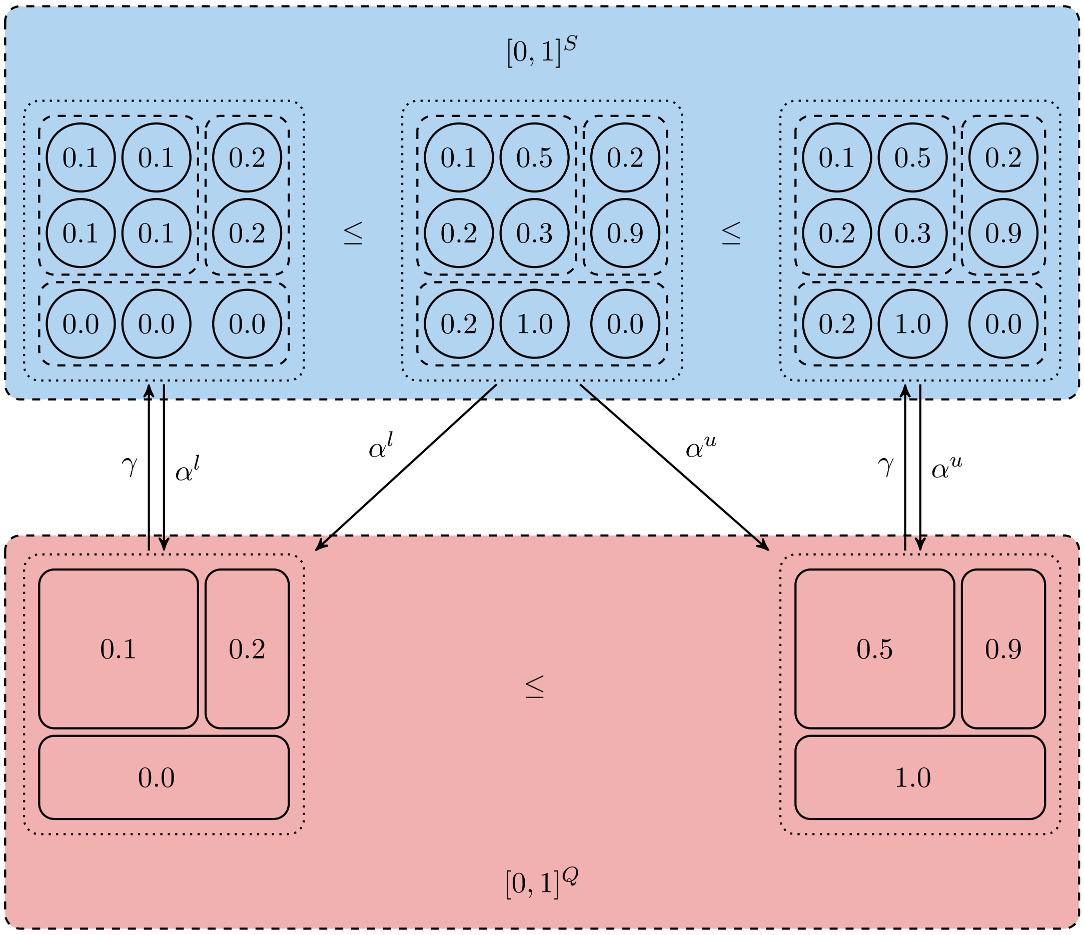


Figure 2.9.: Example of abstraction and concretisation functions usage

The dashed groups of states indicate the partitioning of the concrete state space. Note how the number of states shrinks from nine to three in the process of abstraction and the abstract valuation properly over-approximates the concrete valuation of subsumed states. Also, bear in mind that no precision is lost, and the ordering with respect to $\leq$ is kept, when concretising and abstracting back and forth. □

2.2.2. Relating Concrete and Abstract Valuation Transformers

The Galois connections (α^l, γ) and (α^u, γ) for lower and upper bound abstraction establish relations between concrete and abstract valuations. In the next step, we establish a relation between concrete and abstract valuation transformers, such that we can employ an over-approximating abstract valuation transformer in the abstract domain.

Consider a concrete valuation transformer f. A *valid* abstraction $f^\#$ must be over-approximating in the sense that, applying $f^\#$ to an abstract valuation $w^\#$ may not yield finer results than concretising $w^\#$ and applying f, i.e. $(f \circ \gamma)(w^\#) \leq (\gamma \circ f^\#)(w^\#)$ must hold.

Since there exist many valid abstractions differing in precision, it is desirable to choose the most precise one. Ideally, no precision should be lost due to the application of $f^\#$, however, losing precision due to abstraction is fine and even intended. That is, concretising a $w^\#$, applying f and subsequent abstraction should yield the same as computing $f^\#(w^\#)$, i.e. $\alpha \circ f \circ \gamma = f^\#$ instead of just $\alpha \circ f \circ \gamma \leq f^\#$. Figure 2.10 visualises the idea.

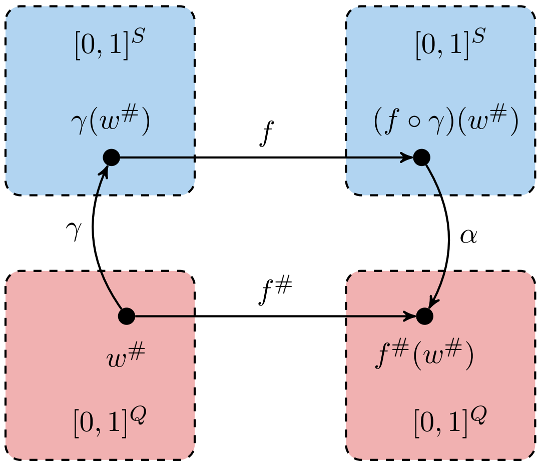


Figure 2.10.: Best transformer $f^\# = \alpha \circ f \circ \gamma$

It has been proven [Cousot and Cousot, 1992] that an abstract transformer $f^\#$ satisfying $f^\# = \alpha \circ f \circ \gamma$ is the most precise, valid abstract transformer. Such a transformer is accordingly called *best transformer*.

Now, let us look at how this transfers to fixed points. [Wachter, 2011] reused a result of [Cousot and Cousot, 1992] to show that for a valuation transformer f and the valid lower and upper bound abstractions $f_l^\#, f_u^\#$, i.e. $\gamma \circ f_l^\# \leq f \circ \gamma \leq \gamma \circ f_u^\#$, the following inequality holds for their fixed points:

$$\gamma\left(\mathit{gfp}_{\geq}(f_l^\#)\right) \leq \mathit{lfp}_{\leq}(f) \leq \gamma\left(\mathit{lfp}_{\leq}(f_u^\#)\right).$$

Employing the best transformers $f_l^\# = \alpha^l \circ f \circ \gamma$ and $f_u^\# = \alpha^u \circ f \circ \gamma$, the latter is equivalent to

$$\gamma\left(\mathit{gfp}_{\geq}(\alpha^l \circ f \circ \gamma)\right) \leq \mathit{lfp}_{\leq}(f) \leq \gamma\left(\mathit{lfp}_{\leq}(\alpha^u \circ f \circ \gamma)\right).$$

Summing up, this result gives us a declarative way of computing the most precise bounds, with respect to the employed abstraction and concretisation functions, for the least fixed point of a valuation transformer f. Exploiting that $\mathit{Pr}_s^{\mathit{min}}(\Diamond G) = (\mathit{lfp}_{\leq}\mathit{pre}_G^-)(s)$ and $\mathit{Pr}_s^{\mathit{max}}(\Diamond G) = (\mathit{lfp}_{\leq}\mathit{pre}_G^+)(s)$, establishes the connection to probabilistic reachability for PA and yields bounds for both the minimal and maximal reachability probabilities:

$$\left(\gamma\left(\mathit{gfp}_{\geq}(\alpha^l \circ \mathit{pre}_G^- \circ \gamma)\right)\right)(s) \leq \mathit{Pr}_s^{\mathit{min}}(\Diamond G) \leq \left(\gamma\left(\mathit{lfp}_{\leq}(\alpha^u \circ \mathit{pre}_G^- \circ \gamma)\right)\right)(s)$$
$$\left(\gamma\left(\mathit{gfp}_{\geq}(\alpha^l \circ \mathit{pre}_G^+ \circ \gamma)\right)\right)(s) \leq \mathit{Pr}_s^{\mathit{max}}(\Diamond G) \leq \left(\gamma\left(\mathit{lfp}_{\leq}(\alpha^u \circ \mathit{pre}_G^+ \circ \gamma)\right)\right)(s)$$

2.2.3. Game-based Abstraction

The previous section presented a declarative result to compute bounds for probabilistic reachability. To enable the actual computation we still need to deduct an imperative approach to computing the best transformer. This section revises the findings of [Kwiatkowska et al., 2006], which convey that SG are suitable best transformers.

Let $\mathcal{A} = (S, \mathit{Act}, U, \mathbf{P}, s_{\mathit{init}})$ be a PA, Q a partition of S, and consider we are interested in computing the upper bound of the minimal reachability probability with respect to some goal set $G \subseteq S$. According to the previous result, the best transformer for an abstract valuation $w^\# \in [0,1]^Q$ is given by

$$\begin{aligned}\left(\alpha^u \circ \mathit{pre}_G^- \circ \gamma(w^\#)\right)(B) &= \sup_{s \in B} \mathit{pre}_G^-(\gamma(w^\#))(s) \\ &= \sup_{s \in B} \min_{(a,\mu) \in \mathbf{P}(s)} \sum_{s' \in S} \widehat{\mu}(s') \cdot (\gamma(w^\#))(s'),\end{aligned}$$

where $B \in Q$ is a block of the partition. At the top level, the expanded formula maximises over the different states a block B subsumes. For each of these states, the formula minimises over the weighted sum of valuations of successor states. This formula looks quite similar to the value iteration for SG, where the first player respectively maximised over successor player 2 vertices, while the second one minimised the weighted sum of reachability probabilities of its successors. In fact, it has been shown [Kwiatkowska et al., 2006] that a SG, where player 1 states correspond to the blocks $B \in Q$ and player 2 states correspond to sets of distributions over Q, provides an implementation of best transformers.

Definition 2.37 (Game-based Abstraction)**.** Let $\mathcal{A} = (S, Act, U, \mathbf{P}, s_{init})$ be a PA and Q a partition of S. Then the game-based abstraction of $\mathcal{A}$ with respect to Q, is defined as the SG

$$\mathcal{G}_{\mathcal{A},Q} = ((V, E), (V_1, V_2, V_p), U, v_{init})$$

where

- the player 1 vertices $V_1 = Q$ are the blocks of Q,
- probabilistic vertices $V_p = \bigcup_{s \in S} \overline{P_\mu(s)}$ are the distributions occurring in $\mathcal{A}$, lifted to distributions over blocks,
- the player 2 vertices $V_2 = \left\{ \overline{P_\mu(s)} \subseteq Dist_U(Q) \;\middle|\; s \in S \right\}$, are sets of probabilistic vertices, grouped by the state they are available at,
- the initial vertex $v_{init} = \overline{s_{init}}$ is the block subsuming s_{init},

such that

- probabilistic vertices have edges to vertices which can be reached with probability greater zero,
- player 2 vertices have edges to the probabilistic vertices they contain,
- and a player 1 vertex v_1 has edges to those player 2 vertices, which correspond to behaviours of states in v_1, i.e. vertices corresponding to lifted distributions available at a state s contained in v_1,

i.e.

$$\begin{array}{rcl} E & := & \{(v_p, v_1) \in V_p \times V_1 \mid \widehat{v_p}(v_1) > 0\} \\ & \cup & \{(v_2, v_p) \in V_2 \times V_3 \mid v_p \in v_2\} \\ & \cup & \left\{(v_1, v_2) \in V_1 \times V_2 \mid \exists_{s \in v_1}\ v_2 = \overline{\mathbf{P}_\mu(s)}\right\}. \end{array}$$

□

Theorem 2.38 (Game-based Abstraction as Transformer [Kwiatkowska et al., 2006])**.** *Let $\mathcal{A}$ be a PA with states S, $G \subseteq S$ a set of goal states and Q a finite partition of S such that G is exactly representable, i.e. $\exists_{G\# \subseteq Q}\ G = \bigcup_{B \in G\#} B$. Then, for a game-based abstraction $\mathcal{G}_{\mathcal{A},Q}$, the the valuation transformers for $\mathcal{G}_{\mathcal{A},Q}$ correspond to the best transformers for the valuation transformers for $\mathcal{A}$, i.e.*

$$\begin{array}{ll} pre^{--}_{G\#} = \alpha^l \circ pre^{-}_{G} \circ \gamma & pre^{+-}_{G\#} = \alpha^u \circ pre^{-}_{G} \circ \gamma \\ pre^{-+}_{G\#} = \alpha^l \circ pre^{+}_{G} \circ \gamma & pre^{++}_{G\#} = \alpha^u \circ pre^{+}_{G} \circ \gamma. \end{array}$$

□

Example 2.39. Figure 2.11 contrasts the PA $\llbracket P_{simple} \rrbracket$ of our example program P_{simple} from Listing 2.1 with its game-based abstraction $\mathcal{G}_{\llbracket P_{simple} \rrbracket, Q}$ given the partition

$$Q := \{\underbrace{\{s_0\}}_{B_0}, \underbrace{\{s_1, s_3, s_5\}}_{B_1}, \underbrace{\{s_2, s_4\}}_{B_2}, \underbrace{\{s_6\}}_{B_3}\},$$

also indicated by the nodes' tint, which groups states with equal valuation of *phase*.

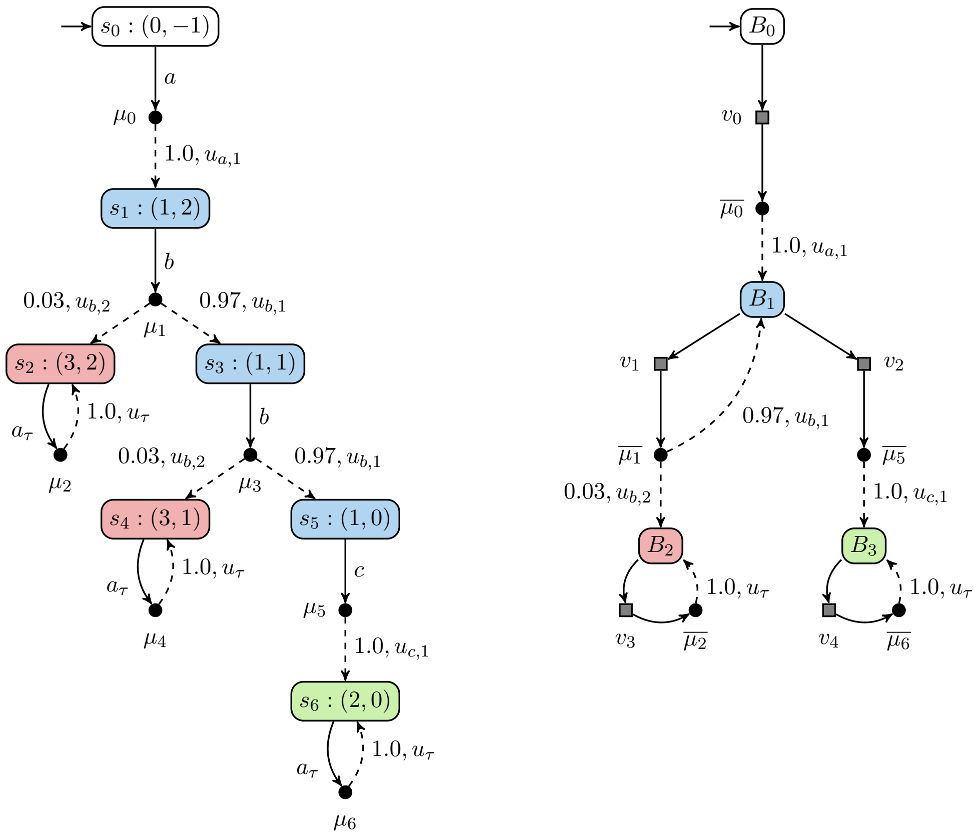


Figure 2.11.: PA $\llbracket P_{simple} \rrbracket$ and its game-based abstraction $\mathcal{G}_{\llbracket P_{simple} \rrbracket, Q}$

Let us take a look at why the game-based abstraction turns out like this. To begin with, consider the initial vertex B_0. Since B_0 subsumes only s_0, its only successor is

$$v_0 = \overline{\mathbf{P}_\mu(s_0)} = \{\overline{\mu_0}\},$$

where $\overline{\mu_0}$ is the distribution μ_0, lifted from states to blocks. The successors of a player 2 vertex are the distributions it contains and may choose from. Thus, the only successor of v_0 is $\overline{\mu_0}$. The same argument applies for B_3 which, too, contains only one state. The most interesting block is B_1 which subsumes three states. However, since $\overline{\mathbf{P}_\mu(s_1)} = \{\overline{\mu_1}\} = \{\overline{\mu_3}\} = \overline{\mathbf{P}_\mu(s_3)}$, B_1 has only two successors

$$\left\{\overline{\mathbf{P}_\mu(s_1)}, \overline{\mathbf{P}_\mu(s_3)}, \overline{\mathbf{P}_\mu(s_5)}\right\} = \left\{\overline{\mathbf{P}_\mu(s_1)}, \overline{\mathbf{P}_\mu(s_5)}\right\} = \{\underbrace{\{\overline{\mu_1}\}}_{v_1}, \underbrace{\{\overline{\mu_5}\}}_{v_2}\}.$$

Accordingly, both v_1 and v_2 have only one successor to choose from. A similar argument applies to B_2, as $\overline{\mathbf{P}_\mu(s_2)} = \overline{\mathbf{P}_\mu(s_4)}$.

Assume we are interested in the maximal probability of the system breaking, i.e. reaching a state where *phase* $= 3$. This corresponds to letting $G = \{s_2, s_4\}$ be the set of goal states and determining $Pr^{max}(\Diamond G) = \left(\mathit{lfp}_{\leq} \mathit{pre}_G^{+}\right)(s_0)$. For the given example it is easy to see that

$$\begin{aligned} Pr^{max}(\Diamond G) &= \mu_0(s_1) \cdot \max\{\mu_1(s_2) \cdot 1 + \mu_1(s_3) \cdot \max\{\mu_3(s_4) \cdot 1 + \mu_3(s_5) \cdot 0\}\} \\ &= 0.0591. \end{aligned}$$

Here, the computation is simple. However, as models become larger, the computation becomes more and more complex, rendering this approach infeasible for real-world examples. Let us compute bounds

$$\gamma\left(\mathit{gfp}_{\geq} \mathit{pre}_{G\#}^{-+}\right) \leq Pr^{max}(\Diamond G) \leq \gamma\left(\mathit{lfp}_{\leq} \mathit{pre}_{G\#}^{++}\right)$$

using the game-based abstraction, where $G^{\#} = B_2$. Note that for every player 1 and player 2 vertex except of B_1 there is only one successor. Thus, when computing the lower bound, player 1 will pick v_2 to minimise, and for the upper bound, pick v_1 to maximise the reachability probability, which inevitably will yield the bounds 0 and 1.

To give an example of determining bounds by performing the fixed point iteration, we also compute both the lower and upper bound in the tables below. In iteration 0 we start with the valuation which maps all blocks to 0, and apply the respective value transformers until a fixed point is reached. Note that due to the numerical nature of the computation, the fixed point may not be reachable in a finite number of steps. Thus, a termination condition, like consecutive valuations hardly differing, i.e. difference being smaller than some δ, is used to enforce termination in a finite number of steps.

Iteration	B_0	B_1	B_2	B_3
#0	0.0	0.0	0.0	0.0
#1	0.0	0.0	1.0	0.0
#2	0.0	0.0	1.0	0.0

(a) Fixed point iteration for $\mathit{pre}_{G\#}^{-+}$

Iteration	B_0	B_1	B_2	B_3
#0	0.0	0.0	0.0	0.0
#1	0.0	0.3	1.0	0.0
#2	0.3	0.0591	1.0	0.0
⋮	⋮	⋮	⋮	⋮
#120	0.973	0.974	1.0	0.0
#121	0.974	0.975	1.0	0.0
⋮	⋮	⋮	⋮	⋮
∞	1.0	1.0	1.0	0.0

(b) Fixed point iteration for $\mathit{pre}_{G\#}^{++}$

Table 2.1.: Computing bounds for $Pr^{max}(\Diamond G)$

Although the resulting bounds $[0, 1]$ are correct, i.e. $0 \leq 0.0591 \leq 1$, they are not informative. This is due to the fact that our partition Q was too coarse. We would have gotten a better upper bound if there was no loop at B_1. This can be achieved by *refining* Q so that s_1 and s_3 are in different blocks. □

2.2.4. Predicate Abstraction

In Section 2.2.1 we introduced the theoretical background to relate the concrete domain $([0,1]^S, \leq)$ with the abstract domain $([0,1]^Q, \leq)$, where states are lumped together into blocks, according to a partition Q – an approach known as *partition abstraction*.

As we have seen in Example 2.39 the precision of the abstraction is governed by the choice of Q, i.e. $Q := S$ does not even introduce loss of precision. This raises the question of how to choose Q, i.e. when states are to be considered "similar".

Predicate abstraction [Graf and Saïdi, 1997] is a special kind of partition abstraction where Q is induced by a set of *predicates*. A predicate p for a set S is an indicator function $p : S \to \mathbb{B}$ partitioning S into elements s for which $p(s) = \mathit{true}$ and this for which $p(s) = \mathit{false}$. Accordingly, n predicates suffice to partition S in up to 2^n blocks.

In the context of a PA $[\![P]\!]$ given by a probabilistic program P, S corresponds to states and a predicate $p \in \mathit{BExpr}_{\mathit{var}}$ is a Boolean expression over the program's variables. Let $\mathcal{P}$ be a set of n predicates. The induced partition Q is then a set of blocks $B_{b_1,\dots,b_n}$, where

$$s \in B_{b_1,\dots,b_n} \iff p_1(s) = b_1, \dots, p_n(s) = b_n.$$

Example 2.40. Consider the partition Q from Example 2.39. It was derived by grouping states with equal valuation of *phase*, and thereby was already an example of predicate abstraction as Q is induced by the predicates

$$\mathcal{P} = \{\underbrace{\mathit{phase} = 0}_{p_1}, \underbrace{\mathit{phase} = 1}_{p_2}, \underbrace{\mathit{phase} = 2}_{p_3}, \underbrace{\mathit{phase} = 3}_{p_4}\},$$

where $B_0 := B_{1,0,0,0}$, $B_1 := B_{0,1,0,0}$, $B_2 := B_{0,0,0,1}$ and $B_3 := B_{0,0,1,0}$. □

2.3. Multi-Terminal Binary Decision Diagrams

The approaches presented so far, assumed that the models are given in an *explicit* representation, e.g. single states, labels and functions over them. However, as stated earlier, such *enumerative* representations suffer to the state space explosion problem since, even for the simplest real-world systems, their representation may be too large to fit into memory.

Besides sparse representations, e.g. [Katoen et al., 2009], one of the most wide-spread approaches to approach this problem is using *symbolic* representations like Binary Decision Diagrams (BDDs) [Bryant, 1986] and their extension Multi-Terminal Decision Diagrams (MTBDDs) [Fujita et al., 1997, Bahar et al., 1997], which allows to represent any function with a finite range of values, instead of merely 0 and 1. The lower memory usage of such a representation is attributed to the storage scheme of the data structure, which exploits regularities in the function.

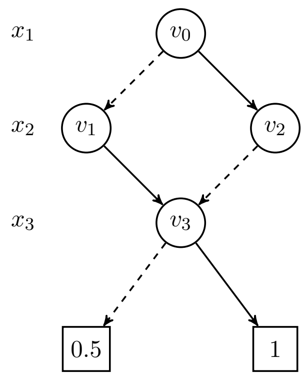
2.3.1. Concept

Let $Var = \{x_1, \dots, x_n\}$ be a set of Boolean variables and $x_1 \prec x_2 \prec \dots \prec x_n$ their total order. An MTBDD $\mathfrak{D}$ over $\underline{x} = (x_1, \dots, x_n)$ is a rooted, acyclic digraph, whose semantics is a function $f_{\mathfrak{D}} : \mathbb{B}^n \to D$. Since we employ MTBDDs to represent probabilistic models and perform quantitative analyses we adopt $D := \mathbb{R}$.

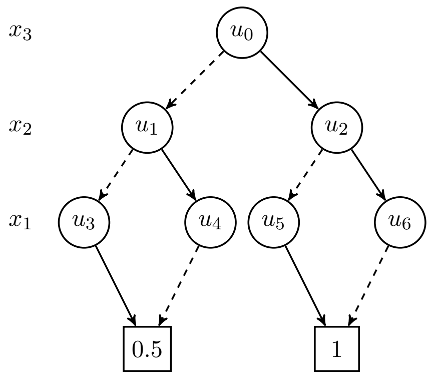
The vertices of $\mathfrak{D}$ are referred to as *nodes* and partitioned into *terminal* nodes, which are the leaves of the graph, and the remaining *non-terminal* nodes. While a terminal node v is labelled with a real number $val(v) \in \mathbb{R}$ and has no successors, a non-terminal v node is labelled with a variable $x_i = var(v) \in Var$ and has exactly two successors – denoted by $then(v)$ and $else(v)$. Additionally, the successor relation respects the variable order, i.e. for any two non-terminal nodes v_i and v_j, where v_j is a successor of v_i, $var(v_i) \prec var(v_j)$ holds.

The value of $f_{\mathfrak{D}}(x_1, \dots, x_n)$ is determined by tracing a path from the root to a terminal node, where for each non-terminal node v the successor $then(v)$ is taken if $var(v)$ is set to 1, or $else(v)$ if $var(v)$ is set to 0. The value $val(v)$ of a terminal node v at the end of the path, which is induced by the variables' valuations, is $f_{\mathfrak{D}}(x_1, \dots, x_n)$. For convenience we use the notation $f_{\mathfrak{D}}[x_1 = 0, x_3 = 1, x_2 = 0]$ to denote $f_{\mathfrak{D}}(0, 0, 1)$, since it is independent of the actual variable order.

Example 2.41. Figure 2.12 illustrates the storage scheme and the impact of the variables' ordering, opposing two MTBDDs which represent the same function.



(a) Ordering $x_1 \prec x_2 \prec x_3$



(b) Ordering $x_3 \prec x_2 \prec x_1$

x_1	x_2	x_3	$f_{\mathfrak{D}}$
0	0	0	0
0	0	1	0
0	1	0	0.5
0	1	1	1
1	0	0	0.5
1	0	1	1
1	1	0	0
1	1	1	0

(c) Represented function

Figure 2.12.: Two MTBDDs representing the same function

Let us first consider the MTBDD from Figure 2.12a. The labelling $val(v)$ of a terminal node v is illustrated by the value it contains. In contrast, the labelling $var(v)$ of a non-terminal node v corresponds to the variable annotated at the same level, e.g. $x_2 = var(v_1)$. The edges visualise the successor relation, i.e. the solid edges correspond to the *then*-successors while the dashed ones represent the *else*-successors. Note that it is common to omit illustrating the terminal node with value 0 and its incoming edges for the purpose of clarity, e.g. $val(else(v_1)) = 0$ and $then(v_1) = v_3$.

Both the MTBDD from Figure 2.12a and 2.12b represent the function defined in 2.12c. For example consider the valuation $\{x_1 = 1, x_2 = 0, x_3 = 0\}$. In the left MTBDD, this induces a path $v_0 v_2 v_3 \mathit{else}(v_3)$ where $\mathit{val}(\mathit{else}(v_3)) = 0.5$ while in the right one, the respective path is $u_0 u_1 u_3 \mathit{then}(u_3)$ with $\mathit{val}(\mathit{then}(u_3)) = 0.5$.

The most striking observation though, is that the variable order has an significant impact on the size of a MTBDD. In our case, the flipping of the ordering increased the number of non-terminal nodes by 75%. While the absolute number corresponding to the 75% increase is rather small for this simple example, it may significantly affect the necessary memory for real-world MTBDDs. □

When talking about MTBDDs, we assume that they are fully reduced, i.e. contain no redundant nodes. This implies that no nodes have identical *then*- and *else*-successors and shared nodes are merged. For an in-depth description, see [Baier and Katoen, 2008].

Definition 2.42 (Cofactor)**.** Let $\mathfrak{D}$ be a MTBDD over $\underline{x} = (x_1, \dots, x_n)$. Then, its cofactor $\mathfrak{D}|_{x_i=b}, b \in \mathbb{B}$ is an MTBDD over $(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ representing the function $f_{\mathfrak{D}}(x_1, \dots, x_{i-1}, b, x_{i+1}, \dots, x_n)$. □

Using the notion of cofactors, the function $f_{\mathfrak{D}}$ can be expressed recursively as

$$f_{\mathfrak{D}} = x_1 \cdot f_{M|_{x_1=1}} + (1 - x_1) \cdot f_{M|_{x_1=0}}.$$

2.3.2. Operations

Since we want to employ MTBDDs not only for storage but for all our computations on the system, we need some operations on MTBDDs. In the following, we sketch the operations, which will be needed later, without going into detail about their implementation. We assume that $\mathfrak{D}$, $\mathfrak{D}_1$ and $\mathfrak{D}_2$ are MTBDDs over $\underline{x} = (x_1, \dots, x_n)$. Bear in mind, that we treat BDDs as MTBDDs.

- $\textsc{Const}(c), c \in \mathbb{R}$ creates a MTBDD which represents the constant function c. It consists of a single terminal node v, such that $\mathit{val}(v) = c$. We conveniently denote $\textsc{Const}(c)$ by $c_{\mathfrak{D}}$.
- $\textsc{Set}_{\underline{z}}(b_1, \dots, b_m), |\underline{z}| = m$ creates a BDD over $\underline{z}$, which represents the function

$$f(\underline{z}) = \begin{cases} 1 & \text{if } \underline{z} = (b_1, \dots, b_m) \\ 0 & \text{otherwise} \end{cases}$$

- $\text{ITE}(\mathfrak{D}, \mathfrak{D}_{\mathit{then}}, \mathfrak{D}_{\mathit{else}})$, creates a MTBDD over $(\underline{x})$ with "if-then-else" semantics, i.e.

$$f_{\mathfrak{D}}(\underline{x}) = \begin{cases} f_{D_{\mathit{then}}}(\underline{x}) & \text{if } f_{\mathfrak{D}}(\underline{x}) = 1 \\ f_{D_{\mathit{else}}}(\underline{x}) & \text{otherwise} \end{cases}.$$

- We lift operations over the reals to the MTBDD domain, such that $\mathfrak{D}_1 \oplus \mathfrak{D}_2$, where $\oplus \in \{+, -, *, /\}$, yields the MTBDD representing the function $f_{\mathfrak{D}_1} \oplus f_{\mathfrak{D}_2}$. Note that the unary minus operation is lifted too.

- Analogously, we lift both binary and unary Boolean operations to BDDs, e.g. $\neg\mathfrak{D}$ yields the BDD representing the function $\neg f_{\mathfrak{D}}$.
- Furthermore, we lift relational operators ($\leq, \neq$ etc.) to MTBDDs, such that for example $\mathfrak{D} \neq 0_{\mathfrak{D}}$ returns the BDD representing the function $f_{\mathfrak{D}} \neq 0$.
- Abstract($\oplus, \underline{z}, \mathfrak{D}$), abstracts from the variables in $\underline{z} \subseteq \underline{x}$ by combining all cofactors for these variables by $\oplus$. For this to work, $\oplus$ must be a commutative and associative binary operator, depending on the type of $\mathfrak{D}$ either over the reals or Booleans. For example, Abstract($\vee, (x_1, x_2), \mathfrak{D}$) returns the BDD representing the function

$$f_{M|_{x_1=0,x_2=0}} \vee f_{M|_{x_1=0,x_2=1}} \vee f_{M|_{x_1=1,x_2=0}} \vee f_{M|_{x_1=1,x_2=1}}.$$

- ExistsAbstract($\underline{z}, \mathfrak{D}$) := Abstract($\vee, \underline{z}, \mathfrak{D}$), roughly speaking, yields the function $\exists_{\text{valuation of } \underline{z}}\ f_{\mathfrak{D}}(\underline{x}) = 1$, over the restricted variable set $\underline{x} \setminus \underline{z}$.
- UniversalAbstract($\underline{z}, \mathfrak{D}$) := Abstract($\wedge, \underline{z}, \mathfrak{D}$), analogously returns the function $\forall_{\text{valuation of } \underline{z}}\ f_{\mathfrak{D}}(\underline{x}) = 1$, over the restricted variable set $\underline{x} \setminus \underline{z}$.
- MinAbstract($\underline{z}, \mathfrak{D}$) := Abstract($\min, \underline{z}, \mathfrak{D}$), minimises the function over $\underline{z}$, i.e. returns the function $\min_{\text{valuation of } \underline{z}} f_{\mathfrak{D}}(\underline{x})$, over the restricted variable set $\underline{x} \setminus \underline{z}$.
- MaxAbstract($\underline{z}, \mathfrak{D}$) := Abstract($\max, \underline{z}, \mathfrak{D}$) is dual to MinAbstract.
- ReplaceVar($x_i, z, \mathfrak{D}$), returns the MTBDD $\mathfrak{D}'$ over the modified variables $\underline{x}' = (x_1, \ldots, x_{i-1}, z, x_{i+1}, \ldots, x_n)$ such that $f_{\mathfrak{D}'}(b_1, \ldots, b_n) = f_{\mathfrak{D}}(b_1, \ldots, b_n)$ holds for all $(b_1, \ldots, b_n) \in \mathbb{B}^n$. This operation can be lifted to allow replacing sets of variables.

2.4. Satisfiability Modulo Theories

Satisfiability Modulo Theories (Smt) [Ganzinger et al., 2004] is an extension of the well-known Boolean satisfiability (Sat) problem [Cook, 1971]. While the Sat problem is, to determine whether there exists a satisfying valuation for a Boolean formula in conjunctive normal form, the Smt problem allows the utilisation of more expressive theories, e.g. real and linear integer arithmetic (RIA and LIA), arrays and uninterpreted functions.

Although even the plain Sat problem is known to be NP-complete, state-of-the-art Smt-solvers like Z3 [de Moura and Bjørner, 2008] solve LIA instances with several thousand constraints in mere seconds and have successfully been applied in verification [Biere et al., 1999, Clarke et al., 2001, Ball and Rajamani, 2002].

In our case, we will employ Smt-solving to determine the abstraction of a PA $[\![P]\!]$ for a program P directly from P, without computing $[\![P]\!]$ in the first place. This will be achieved by encoding the programs semantics in terms of a LIA formula, such that its solutions correspond to valid system behaviour. This is similar to the approach for "conventional" predicate abstraction [Lahiri et al., 2007].

3. Symbolical Model Checking with Menu-games

This chapter builds upon the preliminaries and introduces the menu-based abstraction as an abstraction superior to game-based abstraction in terms of size and computational cost. We elaborate on the symbolical representation of menu-based abstraction in terms of MTBDDs and illustrate how it can be derived from a probabilistic program, without constructing its semantics in the first place, by taking advantage of SMT-solving. Subsequently, we present a novel approach to solving and refining the menu-based abstraction in a fully symbolical way.

3.1. Menu-based Abstraction

In the previous chapter, we have seen that game-based abstraction provides an implementation of best transformers for probabilistic reachability. However, computing the game-based abstraction has two major drawbacks, which are closely related.

Firstly, in the worst case, the number of player 2 vertices grows exponentially in the number of probabilistic vertices. This is due to the fact, that the number of player 2 vertices is equal to the number of distinguishable behaviours. According to Definition 2.37, the number of player 2 vertices is

$$\begin{aligned} |V_2| &= \left|\left\{\overline{\mathbf{P}_\mu(s)} \subseteq Dist_U(Q) \mid s \in S\right\}\right| \\ &= \sum_{v_1 \in V_1} \left|\left\{\overline{\mathbf{P}_\mu(s)} \subseteq Dist_U(Q) \mid s \in v_1\right\}\right|. \end{aligned}$$

However in the worst case, since player 2 vertices are subsets of probabilistic vertices, a player 1 vertex may have up to $2^{|V_p|}$ successors. Combined with the number of player 1 vertices this amounts to

$$|V_2| \in \mathcal{O}\left(|V_1| \cdot 2^{|V_p|}\right).$$

Secondly, experiments have shown that the computation of player 2 vertices becomes increasingly expensive with increasing number of commands in a program. This is tied to the fact that compounds of commands have to be considered, since a player 2 vertex is determined by a concrete valuation of the respective player 1 vertex and the set of enabled concrete commands for this valuation [Kattenbelt et al., 2008].

Altogether, it is desirable for a "superior" abstraction to not exhibit exponential growth in player 2 vertex numbers and to allow for isolated abstraction of commands of a probabilistic program. This gives rise to the menu-based abstraction [Wachter, 2011] which attends to these issues at the expense of not being a best transformer implementation, i.e. yielding at most as precise bounds as game-based abstraction.

3.1.1. Concept

Just as game-based abstraction, menu-based abstraction is a SG where the player 1 vertices correspond to a partition Q. The main difference and key is the flipping of the semantics of the players' choices.

In game-based abstraction, player 2 successor vertices of a player 1 vertex v_1 corresponded to the distinguishable behaviours of states subsumed by v_1. Respectively the player 1 choice resolved the non-determinism introduced by abstraction. In turn, player 2 chose between distributions over Q for a group of states which behave "similar", thereby resolving the non-determinism introduced by the menu of commands enabled for this group of states. Menu-based abstraction swaps the meanings of both choices, such that player 1 chooses from a menu of commands and player 2 resolves the non-determinism introduced by abstraction. To avoid player 2 choosing a behaviour, where the previously chosen command is not enabled a fix will have to be applied, which will cause the abstraction to be generally less precise than the game-based one. With this in mind we introduce the respective valuation transformers for PA.

Maximal Menu-based Abstraction

To begin with, we focus on the concrete transformer pre_G^+ for maximal probabilistic reachability for a PA $[\![P]\!] = (S, \mathit{Act}, U, \mathbf{P}, s_{\mathit{init}})$, where $[\![P]\!]$ is given by a program P and $G \subseteq S$ is the set of goal states.

In contrast to the value transformer for game-based abstraction, we do not maximise over the successor states but instead the commands, uniquely identified by their labelling, leading to them. Thus, to determine the maximal reachability probability in a state $s \in S$ a concrete transformer must choose an maximising action $a \in \mathit{Act}$ enabled in s, i.e.

$$\mathit{pre}_G^+(w)(s) = \max_{a \in \mathit{En}(s)} \mathit{pre}[a]_G^+(w)(s),$$

where $\mathit{pre}[a]_G^+$ is a sub-transformer returning the reachability probability given action a was chosen. Similar to conventional value transformers for SG, for a valuation $w \in [0,1]^S$, $\mathit{pre}[a]_G^+(w)(s)$ returns 1 if $s \in S$ is a goal state, 0 if no path from s to any goal state exists and the weighted sum of the distribution reachable by a otherwise. However, if a is not enabled in s, i.e. $a \notin \mathit{En}(s)$, naturally 0 must be returned:

$$\mathit{pre}[a]_G^+(w)(s) := \begin{cases} 1 & \text{if } s \in G \text{ and } s \in \mathit{En}(a) \\ 0 & \text{if } s \in G_0 \text{ or } s \notin \mathit{En}(a) \\ \max\limits_{(a,\mu) \in \mathbf{P}(s)} \sum\limits_{s' \in S} \widehat{\mu}(s') \cdot w(s') & \text{otherwise} \end{cases},$$

where $G_0 \subseteq S$ denotes the set of states, for which there exists no path to any of the states from G. The respective valid *maximal menu-based abstraction* transformers are obtained by the best transformer construction. As a result the lower and upper bounds for maximal probabilistic reachability with respect to G are given by

$$\begin{aligned} pre^{l+}_{G^\#}(w^\#)(B) &:= \max_{a\in En(B)} \left(\alpha^l \circ pre[a]^+_G \circ \gamma\right)(w^\#)(B) \\ pre^{u+}_{G^\#}(w^\#)(B) &:= \max_{a\in En(B)} \left(\alpha^u \circ pre[a]^+_G \circ \gamma\right)(w^\#)(B), \end{aligned}$$

for $w^\# \in [0,1]^Q$ and $B \in Q$. As for game-based abstraction $G^\#$ is an exact representation of G, i.e. $\exists_{G^\# \subseteq Q}\ G = \bigcup_{B\in G^\#} B$. [Wachter, 2011]

Minimal Menu-based Abstraction

The concrete value transformer $pre[a]^-_G$ for minimal probabilistic reachability is constructed analogously. The main difference to its counterpart, besides minimising over actions, being that it has to return 1 if the respective action a is not enabled in s, i.e. $a \notin En(s)$ to ensure that disabled actions have no effect:

$$pre[a]^-_G(w)(s) := \begin{cases} 1 & \text{if } s \in G \text{ or } s \notin En(a) \\ 0 & \text{if } s \in G_0 \text{ and } s \in En(a) \\ \min\limits_{(a,\mu)\in \mathbf{P}(s)} \sum\limits_{s'\in S} \widehat{\mu}(s') \cdot w(s') & \text{otherwise} \end{cases} .$$

Accordingly, using the best transformer construction, the *minimal menu-based abstraction* yields the lower and upper bounds

$$\begin{aligned} pre^{l-}_{G^\#}(w^\#)(B) &:= \max_{a\in En(B)} \left(\alpha^l \circ pre[a]^-_G \circ \gamma\right)(w^\#)(B) \\ pre^{u-}_{G^\#}(w^\#)(B) &:= \max_{a\in En(B)} \left(\alpha^u \circ pre[a]^-_G \circ \gamma\right)(w^\#)(B), \end{aligned}$$

for $w^\# \in [0,1]^Q$ and $B \in Q$, which are valid transformers, too. [Wachter, 2011]

3.1.2. Menu-game as Implementation of Menu-based Abstraction

Note that the constructed transformers only give us a declarative way of computing abstract transformers. We still need a SG which implements them. The menu-based abstraction has been been shown to represent such an implementation.

Definition 3.1 (Menu-game)**.** Let P be a probabilistic program, $[\![P]\!] = (S, Act, U, \mathbf{P}, s_{init})$ its semantics and Q a partition of S. The Menu-game is a SG

$$\widehat{\mathcal{G}}_{[\![P]\!],Q} = ((V,E),(V_1,V_2,V_p),U,v_{init}),$$

where

- player 1 vertices $V_1 := Q \uplus \{v_1^\perp\}$, correspond to the partition and an auxiliary *trap* vertex,

- player 2 vertices $V_2 := \{(v_1, a) \in V_1 \times Act \mid a \in En(v_1)\} \uplus \left\{v_2^\perp\right\}$ are induced by the actions enabled in player 1 vertices,
- probabilistic vertices $V_p := \left\{\overline{\mathbf{P}(s,a)} \in Dist_U(Q) \mid s \in S, a \in En(v_1)\right\} \uplus \left\{v_p^\perp\right\}$, where $v_p^\perp = 1.0 : (u_\tau, v_1^\perp)$, are essentially the lifted distributions occurring in $[\![P]\!]$,
- and the initial vertex $v_{init} = \overline{s_{init}}$ corresponds to the block containing s_{init}.

The edges are defined by

$$\begin{aligned}
E &:= \{(v_1, v_2) \in V_1 \times V_2 \mid v_2 = (v_1, a), a \in En(v_1)\} \\
&\cup \left\{(v_2, v_p) \in V_2 \times V_p \mid v_2 = (v_1, a), \exists_{s \in v_1}\ v_p = \overline{\mathbf{P}(s,a)}\right\} \\
&\cup \left\{(v_2, v_p^\perp) \in V_2 \times V_p \mid v_2 = (v_1, a), \exists_{s \in v_1}\ a \notin En(s)\right\} \\
&\cup \left\{(v_1^\perp, v_2^\perp), (v_2^\perp, v_p^\perp)\right\} \\
&\cup \left\{(v_p, v') \in V_p \times V_1 \mid \widehat{v_p}(v') > 0\right\}.
\end{aligned}$$

□

Note the (artificial) introduction of a trap state $v_1^\perp$, which once reached cannot be left due to the fact that it only has the self-loop $v_1^\perp v_2^\perp v_p^\perp$. This state is added, to implement the value transformers' $pre[a]_G^+$ and $pre[a]_G^-$ behaviour of returning 0 and 1 when a disabled action is chosen. For this purpose, $v_1^\perp$ can only reached when, for vertex a $v_2 \in V_2$, player 2 chooses an action which is not enabled in v_2. Additionally, we have to ensure that the probabilistic reachability of some $G^\#$ from $v_1^\perp$ is 0 for maximal and 1 for minimal reachability. This is realised by considering $v_1^\#$ as part of the set of goal states G, when computing minimal reachability. For maximal reachability nothing has to be adapted since the trap state cannot reach G by definition.

The main result of [Wachter, 2011] is that with these considerations our value transformers for SG are equivalent to the transformers of menu-based abstraction:

$$pre^{--}_{G\# \cup \{v_1^\perp\}} = pre^{l-}_{G\#} \qquad pre^{-+}_{G\# \cup \{v_1^\perp\}} = pre^{u-}_{G\#}$$
$$pre^{+-}_{G\#} = pre^{l+}_{G\#} \qquad pre^{++}_{G\#} = pre^{u+}_{G\#}$$

The proof is similar to the one for game-based abstraction, as it argues that the expressions gained from the abstract valuation transformers correspond to the structure of the Menu-game.

Example 3.2. Figure 3.1 compares our running example PA $[\![P_{simple}]\!]$ with its Menu-game $\widehat{\mathcal{G}}_{[\![P_{simple}]\!],Q}$ where Q is again induced by the predicates

$$\mathcal{P} = \{\underbrace{phase = 0}_{p_1}, \underbrace{phase = 1}_{p_2}, \underbrace{phase = 2}_{p_3}, \underbrace{phase = 3}_{p_4}\},$$

i.e.

$$Q := \{\underbrace{\{s_0\}}_{B_0}, \underbrace{\{s_1, s_3, s_5\}}_{B_1}, \underbrace{\{s_2, s_4\}}_{B_2}, \underbrace{\{s_6\}}_{B_3}\},$$

and for clarity also indicated by the vertices' colours. Note that the trap vertex is not part of Q and accordingly has inverted colours. Bear in mind that the edges between player 1 and player 2 vertices are actually not labelled. However, since they represent commands of a PA we add the command's labelling, such that it is easier to see the relation.

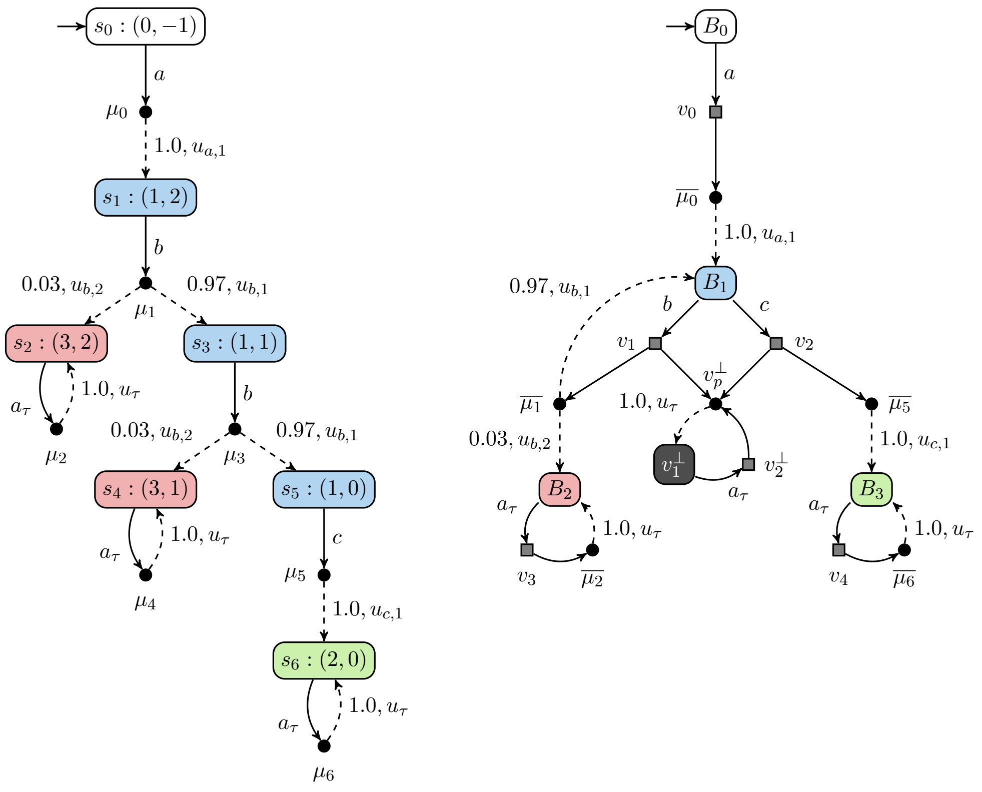


Figure 3.1.: PA $[\![P_{simple}]\!]$ and its menu-based abstraction $\widehat{\mathcal{G}}_{[\![P_{simple}]\!],Q}$

Let us walk through the derivation of this Menu-game. First of all, consider the initial vertex B_0 which contains only the state s_0. Since a is enabled in s_0, so it is in B_0 and leads to the player 2 state $v_0 = (B_0, a)$. v_0 has only one successor, since there is only one distinguishable behaviour in B_0 – the distribution $\mu_0 = \mathbf{P}(s_0, a)$. The thought process for B_3 is identical and thus omitted.

B_1 contains three states, of which s_1 and s_3 have b enabled while in s_5 there is only c enabled. Accordingly, B_1 has one successor for each of these actions. Let us focus on the player 2 state $v_1 = (B_1, b)$. In v_1 player 2 may choose from the distinguishable behaviours of states in B_1 – the behaviours of s_1, s_3 and s_5. For the states s_1 and s_3, where b is enabled, there is only one distinguishable behaviour though, since $\overline{\mu_1} = \overline{\mu_3}$. This corresponds to the $\overline{\mu_1}$ successor. However, choosing a state where c is not enabled, i.e. s_5, leads to the trap-cycle. Dually, v_2 has the successor $\overline{\mu_5}$ which corresponds to the behaviour of s_5, but also an edge to the trap-cycle, since there are some states in B_1 which do not have c enabled.

Assume we are interested in the maximal probability of the system breaking, i.e. reaching a state where $\mathit{phase} = 3$. In a previous example we have seen that $Pr^{max}(\Diamond G) = 0.0591$, where $G = \{s_2, s_4\}$. Let us compute respective bounds

$$\gamma\left(\mathit{gfp}_{\geq}\mathit{pre}^{+-}_{G\#}\right) \leq Pr^{max}(\Diamond G) \leq \gamma\left(\mathit{lfp}_{\leq}\mathit{pre}^{++}_{G\#}\right)$$

using the Menu-game from Figure 3.1, where $G^{\#} = B_2$.

For the lower bound, player 1 will try to maximise the probability, while player 2 aims to minimise it. Accordingly, player 2 will always choose the $v_p^{\perp}$ successor for the vertices v_1 and v_2 such that it does not even matter which action player 1 chooses for B_1. The lower bound is over-approximated by 0.

For the upper bound, both players try to maximise the probability. As a result, both players will collaboratively choose the path-fragment $v_1\overline{\mu_1}$ for B_1 and loop in B_1 until $\overline{\mu_1}$ forwards to B_2 eventually. The upper bound is respectively given by 1.

As in Example 2.39, it turns out that the partition is too coarse to allow for more precise bounds. That was actually to be expected since the menu-based abstraction does not correspond to the best transformer and thus is at most as precise as game-based abstraction, given the same partition Q. □

3.2. Representing Menu-games via MTBDDs

Let P be a probabilistic program, $[\![P]\!] = (S, \mathit{Act}, U, \mathbf{P}, s_{init})$ its semantics and

$$\widehat{\mathcal{G}}_{[\![P]\!],Q} = ((V, E), (V_1, V_2, V_p), U, v_{init})$$

its Menu-game with respect to the partition Q, which is induced by a set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$. The general idea is to interpret the Menu-game as a transition function

$$\delta : V_1 \times \mathit{Act} \times \mathit{Opt} \times \mathit{Upd} \times V_1 \to \mathbb{R},$$

where

- the set of options Opt is the player 2 equivalent to actions, identifying player 2 choices, i.e. there exists an injective function $h : V_2 \times \mathit{Opt} \to V_p$ such that

$$\forall_{v_2 \in V_2}\ \forall_{v_p \in E(v_2)}\ \exists_{o \in \mathit{Opt}}\ h(v_2, o) = v_p,$$

  e.g. $\mathit{Opt} = V_p$ and $h(v_2, v_p) = v_p$,

- the set of updates $\mathit{Upd} := \{u_i \mid u_{a,i} \in U\} \uplus \{u_\tau\}$, which, in contrast to the updates U in $[\![P]\!]$, are not associated with actions anymore, such that

$$\delta(v,a,o,u,v') := \begin{cases} p & \text{if } \exists_{v,v' \in V_1} \exists_{a \in En(v)} \exists_{o \in Opt} \exists_{u \in U} \; \mu = h\left((v,a),o\right), \mu(u,v') = p, p > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that we split the original set of updates U into Act and Upd since an update $u_{a,i} \in U$ is uniquely defined by an action $a \in Act$ and an update index $u_i \in Upd$. In practice, such a splitting requires less space, since most of the time $|Act| + |Upd|$ is less than $|U|$.

However, we cannot represent δ in terms of a MTBDD yet, as its signature is not of the required form, i.e. not $\delta : \mathbb{B}^k \to \mathbb{R}$. To this end we need a suitable encoding. A finite set F can be encoded in terms of $m = \lceil \log_2 |F| \rceil$ Boolean variables using an any injective function $enc : F \to \mathbb{B}^m$. We denote the respective encoding functions for actions, options and updates by enc_{Act}, enc_{Opt} and enc_{Upd}.

For vertices in V_1 we employ a more natural encoding. In Section 2.2.4, we have seen that every block is identified by a vector $(b_1, \ldots, b_n) \in \mathbb{B}^n$ of predicates. Introducing a Boolean variable for every predicate, the encoding enc_B of a block $B_{b_1,\ldots,b_n}$ is defined as

$$enc_B : Q \to \mathbb{B}^n, enc_B(B_{b_1,\ldots,b_n}) \mapsto (b_1, \ldots, b_n).$$

Considering that the trap vertex $v_1^\perp$ is part of V_1 too, but has no relation to the predicates, we define the encoding enc_V for a $v \in V_1$ as

$$enc_V : V_1 \to \mathbb{B}^{n+1}, enc_V(v) = \begin{cases} (0, enc_B(v)) & \text{if } v \neq v_1^\perp \\ (1, 0^n) & \text{otherwise} \end{cases}.$$

Overall, an MTBDD $\mathfrak{D}$ encodes the Menu-game $\widehat{\mathcal{G}}_{[\![P]\!],Q}$ if it represents the function δ, i.e. for $f_{\mathfrak{D}}(\underline{src}, \underline{act}, \underline{opt}, \underline{upd}, \underline{dst})$ it holds

$$f_{\mathfrak{D}}\left(enc_v(v), enc_{Act}(a), enc_{Opt}(o), enc_{Upd}(u), enc_v(v')\right) = \delta\left(v,a,o,u,v'\right).$$

Example 3.3. Figure 3.2 illustrates a Menu-game $\widehat{\mathcal{G}}_{[\![P]\!],Q}$ similar to $\widehat{\mathcal{G}}_{[\![P_{simple}]\!],Q}$ but simpler, where Q is induced by the set of predicates $\mathcal{P} = \{p_1\}$, and its symbolical encoding in terms of a MTBDD $\mathfrak{D}$.

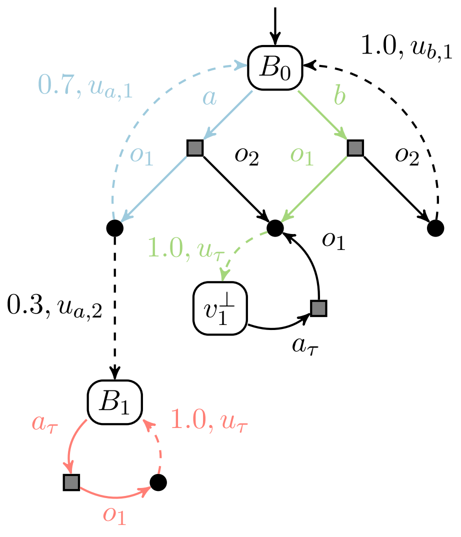
In the first place, let us determine which sets have to be encoded. The actions Act correspond to the player 1 choices and are given by $Act = \{a_\tau, a, b\}$. Upd is derived from U by dropping the actions from all updates $u_{a,i}$ and adding u_τ, i.e. $Upd = \{u_\tau, u_1, u_2\}$. Note that the options o_1 and o_2 suffice to encode the Menu-game, i.e. $Opt = \{o_1, o_2\}$. We indicate a suitable mapping $h : V_2 \times Opt \to V_p$ by the options at the player 2 edges.

Now, we can infer the the number of variables needed. Since Act contains three values, we need two Boolean variables $\underline{act} = (act_1, act_0)$ to encode them. The same reasoning applies to Upd, such that we get $\underline{upd} = (upd_1, upd_0)$. To encode both values in Opt, a single Boolean variable opt_0 suffices. Since the partition Q is induced by a single predicate p_1, we need two variables $\underline{src} = (src_\perp, src_{p_1})$ to encode a source vertex. Destination vertices are dually encoded by $\underline{dst} = (dst_\perp, dat_{p_1})$.

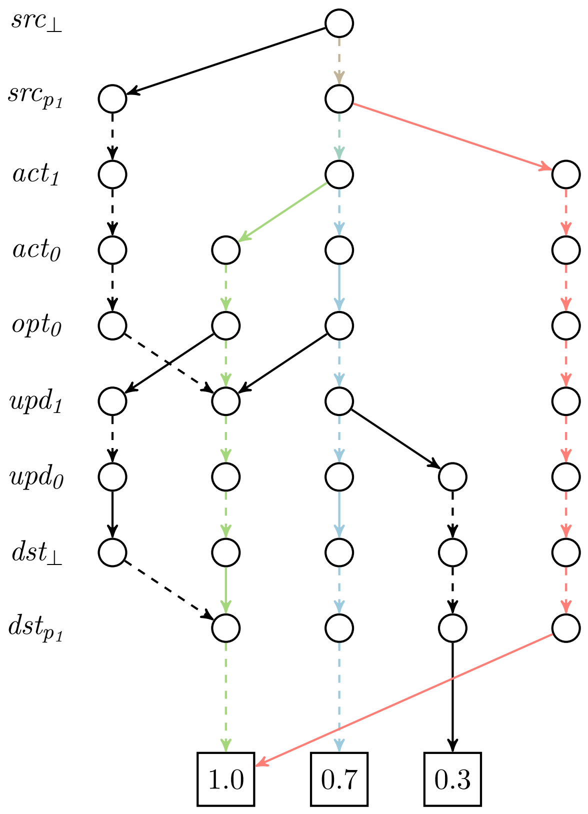
Finally, we define encodings for *Act*, *Opt* and *Upd*:

$$\begin{aligned}
enc_{Act} &:= \{a_\tau \mapsto (0,0)\,, a \mapsto (0,1)\,, v \mapsto (1,0)\} \\
enc_{Opt} &:= \{o_1 \mapsto (0)\,, o_2 \mapsto (1)\} \\
enc_{Upd} &:= \{u_\tau \mapsto (0,0)\,, u_1 \mapsto (0,1)\,, u_2 \mapsto (1,0)\}\,.
\end{aligned}$$

Bear in mind, that the encoding for player 1 vertices is induced by the predicates, e.g. $enc_v(B_1) = (0,1)$.



(a) Menu-game $\widehat{\mathcal{G}}_{[\![P]\!],Q}$



(b) MTBDD $\mathfrak{D}$ encoding $\widehat{\mathcal{G}}_{[\![P]\!],Q}$

Figure 3.2.: A Menu-game and its MTBDD

To get a better idea of which valuations of the MTBDD correspond to which path fragments, we coloured some of them equally. For example consider the green valuation of the MTBDD. It corresponds to the green path fragment of $\widehat{\mathcal{G}}_{[\![P]\!],Q}$, because

$$f_{\mathfrak{D}}[\underline{src} = \underbrace{(0,0)}_{enc_v(B_0)}, \underline{act} = \underbrace{(1,0)}_{enc_{Act}(b)}, \underline{opt} = \underbrace{(0)}_{enc_{Opt}(o_1)}, \underline{upd} = \underbrace{(0,0)}_{enc_{Upd}(u_\tau)}, \underline{dst} = \underbrace{(1,0)}_{enc_v(v_1^\perp)}] = 1.0$$

and the game's induced transition function δ returns $\delta(B_0, b, o_1, u_\tau, v_1^\perp) = 1.0$, too.

Note that the variable order used in Figure 3.2b is not the one which yields the smallest MTBDD but rather the one which yields the most human-readable MTBDDs. The variable ordering corresponds to the order in which we read the game's transitions. □

3.3. Construction of Menu-games from Probabilistic Programs

Formally, menu-based abstraction is defined on the PA semantics of a probabilistic program. Thus, until now, to determine the Menu-game of a probabilistic program P with respect to a set of predicates $\mathcal{P}$, we had to take the detour of computing the program's explicit semantics $[\![P]\!]$. In this section, based on [Wachter, 2011], we elaborate on how Smt solvers can be utilised to directly obtain the abstract semantics without constructing the explicit state space in the first place.

3.3.1. Logical Characterisation of Concrete Semantics

To begin with, let us reformulate the explicit semantics $[\![P]\!]$ of a probabilistic program $P = (\mathit{Var}, \mathit{VarType}, \mathit{Act}_P, \mathit{Cmd}, \mathit{init})$ in terms of linear integer arithmetic formulae, such that the computation of $[\![P]\!]$ can be carried out by Smt solvers.

The program's behaviour is governed by the semantics of its commands. Definition 2.27 already gives us an explicit method to compute the semantics $[\![c]\!]$ of a command c:

$$[a]\ g \to p_1 : \mathit{Var}' = E_1 + \cdots + p_k : \mathit{Var}' = E_k.$$

However, aiming for a symbolical description, we reformulate

$$\left\{ \left(s_0, a, \bigoplus_{i=1}^{k} p_i : (u_{a,i}, s_i) \right) \middle| \begin{array}{ll} s_0, \ldots, s_k \in S(P) & \\ s_0 \models g & \text{“guard”} \\ \forall_{i \in \{1,\ldots,k\}}\ s_i = \lambda_{var \in \mathit{Var}}\ [\![E_i(var)]\!]_{s_0} & \text{“updates”} \end{array} \right\}$$

in terms of transition constraints, which relate source- and destination-state instances of the programs variables.

It is a wide-spread convention to use primed variables for source- and unprimed variables for destination-state variables instances. For example an assignment incrementing x can be characterised by the expression $x' = x + 1$. Generally, commands may have several destinations though, such that a corresponding expression must contain several instances of destination-state variables.

Knowing that the source-states of a command are those which satisfy its guard g, we can characterise them by $g\,[\mathit{Var}/\,\mathit{Var}_0]$, which denotes the simultaneous substitution of variables Var by the source-state variables instance Var_0 in the expression g. Accordingly, the effect of an assignment E_i is expressed in terms of a conjunction of equalities $\mathit{Var}_i = E_i\,[\mathit{Var}/\,\mathit{Var}_0]$, which corresponds to evaluating the expression in the source state and binding the resulting valuation to the i-th destination-state variables instance.

Definition 3.4 (Transition Constraint [Wachter, 2011])**.** The concrete transitions of a command c, given by

$$[a]\ g \to p_1 : Var' = E_1 + \cdots + p_k : Var' = E_k,$$

are expressed by the transition constraint

$$\mathcal{R}_c = g\,[Var/Var_0] \wedge \bigwedge_{i=1}^{deg(c)} (Var_i = E_i\,[Var/Var_i])\,.$$

□

Using this result, the computation of $[\![c]\!]$ amounts to finding all solutions of $\mathcal{R}_c$:

$$[\![c]\!] = \left\{ \left(s_0, a, \bigoplus_{i=1}^{k} p_i : (u_{a,i}, s_i) \right) \middle| (s_0, \ldots, s_k) \models \mathcal{R}_c \right\}.$$

Example 3.5. Let us have a look on how this can be applied to the command c_b

$$[b]\ phase = 1\ \&\ run > 0 \to 0.97 : (run' = run - 1) + 0.03 : (phase' = 3)$$

from Listing 2.1. A corresponding transition constraint must refer to three variable sets $Var_0 = \{phase_0, run_0\}$, $Var_1 = \{phase_1, run_1\}$ and $Var_2 = \{phase_2, run_2\}$, where Var_0 is the source-state variables instance and the others are the destination-state variables instances – one instance per assignment. The transition constraint for c_b is given by

$$\begin{array}{rcl} \mathcal{R}_{c_b} & = & g\,[Var/Var_0] \wedge Var_1 = E_1\,[var/Var_0] \wedge Var_2 = E_2\,[Var/Var_0] \\ & = & phase_0 = 1 \wedge run_0 > 0 \\ & \wedge & phase_1 = phase_0 \wedge run_1 = run_0 - 1 \\ & \wedge & phase_2 = 3 \wedge run_2 = run_0. \end{array}$$

This constraint symbolically characterises the semantics of c_b, since its solutions induce $[\![c]\!]$. For example, the solution

$$\begin{array}{rcl} Var_0 & = & \{phase_0 = 1, run = 2\} \\ Var_1 & = & \{phase_1 = 1, run = 1\} \\ Var_2 & = & \{phase_2 = 3, run = 2\} \end{array}$$

corresponds to the tuple $(s, b, \mu) \in [\![c_b]\!]$ which we computed (explicitly) in Example 2.28. □

3.3.2. Logical Characterisation of Abstract Semantics

Now that we have a logical characterisation of the concrete semantics $[\![c]\!]$ of a command c, we illustrate how this can be used to obtain a logical characterisation of its abstract semantics $[\![c]\!]^{\#}$.

When composing concrete states into blocks we lift distributions with respect to the partition Q. As a result, the partition abstracted transitions in $[\![c]\!]^{\#}$ consist exactly of the lifted concrete transitions

$$[\![c]\!]^{\#} := \{(B, a, \overline{\mu}) \mid \exists_{s \in B}\ (s, a, \mu) \in [\![c]\!]\}\,.$$

However, this is not a logical characterisation, yet.

We begin by formalising the notion of predicate abstraction in terms of LIA formulae. Section 2.2.4 illustrated that for a set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$

$$s \in B_{b_1,\ldots,b_n} \iff p_1(s) = b_1, \ldots, p_n(s) = b_n.$$

Identifying blocks with bit-vectors $(b_1, \ldots, b_n)$ we get the constraint

$$\mathcal{B} := \bigwedge_{p \in \mathcal{P}} (b_i \iff p_i)\,,$$

whose solutions induce the set of blocks. Accordingly, the set of blocks satisfying an expression e over Var is logically characterised by the expression $e \wedge \mathcal{B}$, whose solutions

$$\alpha([\![e]\!]) := \{(b_1, \ldots, b_n) \mid b_1, \ldots, b_n \models \exists_{\mathit{Var}}\ e \wedge \mathcal{B}\}$$

correspond to the blocks' identifiers. Applying the block characterisation to $\mathcal{R}_c$, i.e. making the block identifiers the single free variables and existentially quantifying over the rest, yields us the abstract transition constraint.

Definition 3.6 (Abstract Transition Constraint [Wachter, 2011])**.** Let c be a command with $k = \mathit{deg}(c)$, $\mathcal{R}_c$ its transition constraint and $\mathcal{P} = \{p_1, \ldots, p_n\}$ a set of predicates. The abstract transition constraint is then given by

$$\mathcal{R}_c^{\#} := \exists_{\mathit{Var}_0,\ldots,\mathit{Var}_k} \mathcal{R}_c \wedge \bigwedge_{j=0}^{k} \mathcal{B}_j$$

with

$$\mathcal{B}_j := \bigwedge_{p_i \in \mathcal{P}} \left(b_i^j \iff p_i\,[\mathit{Var}/\mathit{Var}_j]\right),$$

where variables $b_i^0, 1 \leq i \leq n$ identify the source-block and $b_i^j, 1 \leq i \leq n$ identify the j-th destination-block. Employing the notion of weakest precondition [Dijkstra, 1976] syntactically, i.e. simultaneously substituting all variables Var_j in $\mathcal{B}_j$ according to assignment E_j, we obtain the structurally simpler but equivalent constraint

$$\mathcal{R}_c^{\#} := \exists_{\mathit{Var}_0}\ g\,[\mathit{Var}/\mathit{Var}_0] \wedge \mathcal{B}_0 \wedge \bigwedge_{j=1}^{k} \mathcal{B}_j\,[\mathit{Var}_j/E_j]\,[\mathit{Var}/\mathit{Var}_0]\,.$$

□

Note that $\mathcal{R}_c^\#$ is a LIA formula, since assignments E_j are assumed to use linear arithmetic only (see Definition 2.24). Analogous to $[\![c]\!]$, the computation of $[\![c]\!]^\#$ amounts to finding all solutions of $\mathcal{R}_c^\#$, where blocks take the place of states:

$$[\![c]\!] = \left\{ \left(s_0, a, \bigoplus_{j=1}^{k} p_i : (u_{a,i}, B_j) \right) \middle| (B_0, \ldots, B_k) \models \mathcal{R}_c^\# \right\}.$$

Example 3.7. Let us come back to our running example command c_b

$$[b]\ \mathit{phase} = 1\ \&\ \mathit{run} > 0 \rightarrow 0.97 : (\mathit{run}' = \mathit{run} - 1) + 0.03 : (\mathit{phase}' = 3)$$

from Listing 2.1. For illustration, we use the set of predicates from Example 3.2

$$\mathcal{P} = \{\underbrace{\mathit{phase} = 0}_{p_1}, \underbrace{\mathit{phase} = 1}_{p_2}, \underbrace{\mathit{phase} = 2}_{p_3}, \underbrace{\mathit{phase} = 3}_{p_4}\},$$

and construct $\mathcal{R}_{c_b}^\#$:

$$\begin{array}{rcl}
\mathcal{R}_{c_b}^\# & = & \exists_{\mathit{phase}_0, \mathit{run}_0 \in \mathbb{N}} (\mathit{phase}_0 = 1 \wedge \mathit{run}_0 > 0) \\
& \wedge & \left(b_1^0 \Leftrightarrow \mathit{phase}_0 = 0\right) \wedge \left(b_2^0 \Leftrightarrow \mathit{phase}_0 = 1\right) \wedge \left(b_3^0 \Leftrightarrow \mathit{phase}_0 = 2\right) \wedge \left(b_4^0 \Leftrightarrow \mathit{phase}_0 = 3\right) \\
& \wedge & \left(b_1^1 \Leftrightarrow \mathit{phase}_0 = 0\right) \wedge \left(b_2^1 \Leftrightarrow \mathit{phase}_0 = 1\right) \wedge \left(b_3^1 \Leftrightarrow \mathit{phase}_0 = 2\right) \wedge \left(b_4^1 \Leftrightarrow \mathit{phase}_0 = 3\right) \\
& \wedge & \left(b_1^2 \Leftrightarrow 3 = 0\right) \wedge \left(b_2^2 \Leftrightarrow 3 = 1\right) \wedge \left(b_3^2 \Leftrightarrow 3 = 2\right) \wedge \left(b_4^2 \Leftrightarrow 3 = 3\right)
\end{array}$$

At first glance, the abstract transition constraint may seem wrong, because the second and third lines hardly differ and clearly $b_i^0 \Leftrightarrow b_i^1, 1 \le i \le 4$. This, however, is correct and attributed to the fact that the assignment $\mathit{run}' = \mathit{run} - 1$ does not modify any of the predicates in $\mathcal{P}$, as those only refer to the variable *phase*. For the given constraint it is easy to see that there is only one solution

$$\left(b_1^0, b_2^0, b_3^0, b_4^0\right) = \left(b_1^1, b_2^1, b_3^1, b_4^1\right) = (0,1,0,0)\,, \left(b_1^2, b_2^2, b_3^2, b_4^2\right) = (0,0,0,1)\,,$$

such that $[\![c_b]\!]^\# = \{(B_{0,1,0,0}, b, 0.97 : (u_{b,1}, B_{0,1,0,0}) \oplus 0.03 : (u_{b,2}, B_{0,0,0,1}))\}$, i.e. action b is enabled in block $B_{0,1,0,0}$ and choosing it leads to either $B_{0,1,0,0}$ with probability 0.97 or to $B_{0,0,0,1}$ in 3% of the cases.

We have seen this very semantics in the Menu-game from Figure 3.1, where $B_{0,1,0,0}$ was named B_1 and $B_{0,0,0,1}$ was known as B_2. The distribution $\overline{\mu_1}$ in that figure corresponds to the one we just determined by solving the abstract transition constraint. □

3.3.3. Logical Characterisation of Menu-games

In the following we adapt Definition 3.1 with respect to the logical characterisation of commands, extending the description of [Wachter, 2011] with the "tedious corner case" of blocks subsuming deadlock states.

Definition 3.8 (Symbolic Abstraction)**.** Let P be a probabilistic program with the initial state expression *init*, $[\![P]\!] = (S, Act, U, \mathbf{P}, s_{init})$ its semantics and Q a partition of S, induced by a set of predicates $\mathcal{P}$. The Menu-game

$$\widehat{\mathcal{G}}_{[\![P]\!],Q} = ((V, E), (V_1, V_2, V_p), U, v_{init}),$$

can be obtained from P with

- $V_1 := Q \uplus \{v_1^\perp\}$,
- $V_2 := \bigcup_{c \in Cmd} \left\{(v_1, a) \in V_1 \times Act \;\middle|\; (v_1, a, \mu) \in [\![c]\!]^\#\right\} \uplus \{v_2^\perp\} \uplus \{v_{2,v_1}^{dl} \mid v_1 \in V_{dl}\}$,
- $V_p := \bigcup_{c \in Cmd} \left\{\mu \in V_p \;\middle|\; (v_1, a, \mu) \in [\![c]\!]^\#\right\} \uplus \{v_p^\perp\} \uplus \{v_{p,v_1}^{dl} \mid v_1 \in V_{dl}\}$, with $v_{p,v_1}^{dl} := 1.0 : (u_\tau, v_1)$ and $v_p^\perp := 1.0 : (u_\tau, v_1^\perp)$,
- $v_{init} \in \alpha\left([\![init]\!]\right)$,

where $V_{dl} := Q \setminus \alpha\left([\![\bigvee_{c \in Cmd} g_c]\!]\right)$ is the set of blocks which contain a deadlock state, i.e. a state not satisfying any guard, and, without loss of generality, *init* specifies exactly one block, i.e. $|\alpha\left([\![init]\!]\right)| = 1$. The edges are given by

$$\begin{aligned} E \;:= & \bigcup_{c \in Cmd} \{(v_1, v_2) \in V_1 \times V_2 \mid v_1 \in \alpha([\![g_c]\!]), v_2 = (v_1, a_c)\} \\ \cup & \bigcup_{c \in Cmd} \left\{(v_2, v_p) \in V_2 \times V_p \mid v_2 = (v_1, a_c), (v_1, a_c, v_p) \in [\![c]\!]^\#\right\} \\ \cup & \bigcup_{c \in Cmd} \left\{(v_2, v_p^\perp) \in V_2 \times V_p \mid v_2 = (v_1, a_c), v_1 \in Q \setminus \alpha\left([\![g_c]\!]\right)\right\} \\ \cup & \left\{(v_1^\perp, v_2^\perp), (v_2^\perp, v_p^\perp)\right\} \\ \cup & \left\{(v_1, v_{2,v_1}^{dl}), (v_{2,v_1}^{dl}, v_{p,v_1}^{dl}) \;\middle|\; v_1 \in V_{dl}\right\} \\ \cup & \left\{(v_p, v') \in V_p \times V_1 \mid \widehat{v_p}(v') > 0\right\}. \end{aligned}$$

□

This definition makes it easy to realise the independence of commands in the abstraction since all the parts related to commands (except for deadlock states) are obtained by unions over command-dependent sets. Later, in Section 5.1, we will see how we can get rid of the exceptional handling of trap- and deadlock states.

3.3.4. Construction Algorithm

Let $P = (Var, VarType, Act, Cmd, init)$ be a probabilistic program and $\mathcal{P}$ a set of predicates. In the following, we describe the algorithms employed to obtain a Menu-game's MTBDD $\mathfrak{D}_{sys}$ in terms of pseudocode and begin with a walk-though of constructing the MTBDD $\mathfrak{D}_c$ for a command $c \in Cmd$, see Algorithm 1.

Essentially, the algorithm consists of two parts, separated by line 11. The first part solves the abstract transition constraint, i.e. computes $[\![c]\!]^\#$, and creates MTBDDs for both the source blocks and the available distributions. Hereafter, we know how many distributions are available at every block, i.e. the maximal number of options we have to encode and thus can define enc_{Opt}. The second part combines the MTBDDs for source blocks and distributions, where distributions are distinguished by options.

Algorithm 1 Command abstraction

```
1:  procedure AbstractCommand($c$)                                  ▷ Creates the MTBDD $\mathfrak{D}_c$
2:      $f \leftarrow \{\}$                                          ▷ Maps $\mathfrak{D}_{src}$ to set of $\mathfrak{D}_{\overline{\mu}}$
3:      for all $(\mathcal{B}_{src}, a_c, \overline{\mu}) \in [\![c]\!]^\#$ do
4:          $\mathfrak{D}_{src} \leftarrow \text{SET}_{\underline{src}}(enc_v(\mathcal{B}_{src}))$
5:          $\mathfrak{D}_{\overline{\mu}} = 0_{\mathfrak{D}}$
6:          for all $p : (u, \mathcal{B}_{dst}) \in \overline{\mu}$ do                ▷ Represent $\overline{\mu}$ as $\mathfrak{D}_{\overline{\mu}}$
7:              $\mathfrak{D}_{\overline{\mu}} \leftarrow \mathfrak{D}_{\overline{\mu}} + \text{SET}_{\underline{upd}}(enc_{Upd}(u)) \cdot \text{SET}_{\underline{dst}}(enc_v(\mathcal{B}_{dst})) \cdot \text{CONST}(p)$
8:          end for
9:          $f(\mathfrak{D}_{src}) \leftarrow f(\mathfrak{D}_{src}) \cup \{\mathfrak{D}_{\overline{\mu}}\}$
10:     end for
11:
12:     $\mathfrak{D}_c = 0_{\mathfrak{D}}$
13:     for all $\left(\mathfrak{D}_{src}, \{\mathfrak{D}_{\overline{\mu}_1}, \ldots, \mathfrak{D}_{\overline{\mu}_m}\}\right) \in f$ do   ▷ Combine sources with distributions
14:         $\mathfrak{D}_c \leftarrow \mathfrak{D}_c + \mathfrak{D}_{src} \cdot \left(\mathfrak{D}_{\overline{\mu}_1} \cdot \text{SET}_{\underline{opt}}(enc_{Opt}(o_1)) + \cdots + \mathfrak{D}_{\overline{\mu}_k} \cdot \text{SET}_{\underline{opt}}(enc_{Opt}(o_m))\right)$
15:     end for
16:     return $\mathfrak{D}_c \cdot \text{SET}_{\underline{act}}(enc_{Act}(a_c))$
17: end procedure
```

Command abstraction begins with the creation of a mapping f from MTBDDs to sets of MTBDDs, which we use to keep track of distributions available at each block. $[\![c]\!]^\#$ is obtained from enumerating the solutions of the respective abstract transition constraint $\mathcal{R}_c^\#$ with a SMT-solver. A single iteration of the loop starting in line 3 creates the MTBDDs for the source block $\mathcal{B}_{src}$ and available distribution $\overline{\mu}$ of $(\mathcal{B}_{src}, a, \overline{\mu}) \in \mathcal{R}_c^\#$. While the creation of $\mathfrak{D}_{src}$ is straightforward, the representation of the distribution amounts to summing up intermediate MTBDDs for each update (ref. line 3).

Once the MTBDDs for blocks and distributions are created, we know the number of options for every block $\mathcal{B}_{src}$, if command c is taken. Similar to the construction for distributions, we iteratively combine all source block MTBDDs with their available distributions, where every distribution is identified with an option. Instead of adding the command's label in every iteration, we add it once in the very end.

Algorithm 2 illustrates how command abstraction fits into the big picture of menu-based abstraction of P. The Menu-game is essentially given by the sum of the $\mathfrak{D}_c$ for all $c \in Cmd$. However, at that point, $\mathfrak{D}_{sys}$ may still represent unreachable states, does not represent transitions to the trap block, and may still have blocks which subsume states not satisfying any command's guard. All these issues can be taken care of in a few post-processing operations, though.

The invocation of REACH returns a BDD over $\underline{src}$ representing the set of blocks reachable from the initial block. Using this result, we can restrict $\mathfrak{D}_{sys}$ to the reachable blocks, and thereby reduce the number of blocks to consider in future analyses. Since REACH is in fact optional, we first of all illustrate ADDTRAP and FIXDEADLOCKS, before coming back to it in Section 3.3.5.

Algorithm 2 Program abstraction

```
1: procedure AbstractProgram($P$)                      ▷ Creates the MTBDD $\mathfrak{D}_{sys}$
2:     $\mathfrak{D}_{sys} = 0_{\mathfrak{D}}$
3:     for all $c \in Cmd$ do                          ▷ Combine abstract transitions
4:         $\mathfrak{D}_{sys} = \mathfrak{D}_{sys} + \textsc{AbstractCommand}(c)$
5:     end for
6:     $\mathfrak{D}_{sys} \leftarrow \mathfrak{D}_{sys} \cdot \textsc{Reach}(\mathfrak{D}_{sys}, \mathfrak{D}_{init})$    ▷ Restrict to reachable blocks
7:     AddTrap($\mathfrak{D}_{sys}, P$)                 ▷ Add transitions to trap-loop where due
8:     FixDeadlocks($\mathfrak{D}_{sys}$)               ▷ Add self-loops to deadlock-subsuming blocks
9:     return $\mathfrak{D}_{sys}$
10: end procedure
```

Adding Trap Block

So far, our Menu-game MTBDD $\mathfrak{D}_{sys}$ does not represent transitions to the trap vertex. Determining the player 2 vertices which should lead to the trap vertex is not a problem, though, and can be performed for every command c independently.

To this end, we determine the set of blocks $V_{1,\neg g_c} = \alpha\left(\llbracket \neg g_c \rrbracket\right)$ which do not satisfy the guard g_c of a command c. If such a block has a player 2 successor vertex reachable with action a_c, then this vertex must have the option to reach $v_p^{\perp}$. Let $\mathfrak{D}_{V_{1,\neg g_c}}$ be the BDD over $\underline{src}$ representing the set $V_{1,\neg g_c}$. We get the MTBDD representing the player 2 vertices occurring in $\mathfrak{D}_{sys}$ and reachable from $V_{1,\neg g_c}$ with a_c by

$$\mathfrak{D}_{V_{2,\neg g_c}} = \mathfrak{D}_{V_{1,\neg g_c}} \cdot \textsc{Set}_{\underline{act}}(enc_{Act}(a_c)) \cdot \mathfrak{D}_{V_2^{01}},$$

where

$$\mathfrak{D}_{V_2^{01}} := \textsc{ExistsAbstract}\left(\left(\underline{opt}, \underline{upd}, \underline{dst}\right), \mathfrak{D}_{sys} \neq 0_{\mathfrak{D}}\right)$$

is the set of player 2 vertices existing in $\mathfrak{D}_{sys}$. All player 2 vertices (for all commands) which should lead to the trap behaviour, are obtained through the sum

$$\mathfrak{D}_{V_{2,\neg g_{Cmd}}} := \sum_{c \in Cmd} \mathfrak{D}_{V_{2,\neg g_c}}.$$

The actual hurdle at that point is, that both $\mathfrak{D}_{sys}$ and $\mathfrak{D}_{V_{2,\neg g_{Cmd}}}$ use the option variables $\underline{opt} = (o_1, \ldots, o_l)$, which just suffice to distinguish between ordinary player 2 choices. We have to add another option variable to distinguish between ordinary choices and the choices leading to the trap behaviour. The concept is visualised in Figure 3.3, where the right MTBDD, illustrated as a triangle, encodes the transitions to the bottom states. Combining both MTBDDs like that yields a $\mathfrak{D}_{sys}$ with transition to the trap vertex.

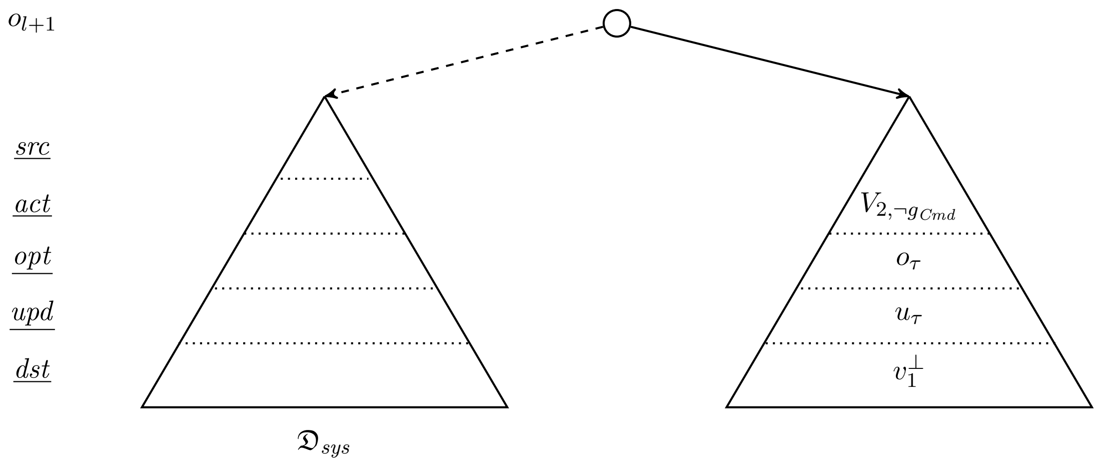


Figure 3.3.: Extending $\mathfrak{D}_{sys}$ with trap-successors

Of course we still have to represent the trap-cycle $v_1^\perp v_2^\perp v_p^\perp$. We don't go into detail about that though, as it amounts to simply adding a single transition from $v_1^\perp$ to itself, using the action $a_\mathcal{T}$ and option $o_\mathcal{T}$.

Fixing Deadlocks

It remains to sketch the last step of Algorithm 2, where we add the possibility of self-looping to those player 1 vertices which subsume deadlock states. According to Definition 3.8, the respective player 1 vertices are given by V_{dl}. Let $\mathfrak{D}_{V_{dl}}$ be the BDD over $\underline{src}$ representing this set and $\mathfrak{D}_{id}$ be the identity BDD over $(\underline{src}, \underline{dst})$, i.e. it represents the function

$$f_{\mathfrak{D}_{id}}[\underline{src} = s, \underline{dst} = d] = \begin{cases} 1 & \textit{if } s = d \\ 0 & \textit{otherwise} \end{cases}.$$

The fixed MTBDD is then obtained by

$$\mathfrak{D}_{sys} = \mathfrak{D}_{sys} + \mathfrak{D}_{V_{dl}} \cdot \mathfrak{D}_{id} \cdot \textsc{Set}_{\underline{act}}(enc_{Act}(a_\mathcal{T})) \cdot \textsc{Set}_{\underline{opt}}(enc_{Opt}(o_\mathcal{T})) \cdot \textsc{Set}_{\underline{upd}}(enc_{Upd}(u_\mathcal{T})),$$

where the latter summand encodes a self-loop with action $a_\mathcal{T}$, option $o_\mathcal{T}$ and update $u_\mathcal{T}$ for all vertices in V_{dl}.

3.3.5. Reachable State Space

It is not necessary but advisable to remove unreachable vertices from $\mathfrak{D}_{sys}$ since such states, by definition, do not contribute to probabilistic reachability. Accordingly, considering such vertices in the fixed point computation for probabilistic reachability only slows down.

Algorithm 3 illustrates the inner workings of the REACH-method, which takes a system's MTBDD $\mathfrak{D}_{sys}$ and a BDD $\mathfrak{D}_{init}$ representing a set of initial vertices – a single vertex in our case. The general idea is to keep track of a the set of vertices, known as *frontier*, found in the latest iteration, and only add new successors of the frontier to the set of reachable states.

Algorithm 3 Determining reachable state space

```
 1: procedure REACH($\mathfrak{D}_{sys}, \mathfrak{D}_{init}$)                ▷ Creates the MTBDD $\mathfrak{D}_{reach}$
 2:     $\mathfrak{D}_{SrcDst} = \textsc{ExistsAbstract}\left(\left(\underline{act}, \underline{opt}, \underline{upd}\right), \mathfrak{D}_{sys} \neq 0_{\mathfrak{D}}\right)$
 3:     $\mathfrak{D}_{reach} = \mathfrak{D}_{init}$
 4:     $\mathfrak{D}_{frontier} = \mathfrak{D}_{init}$
 5:     while $\mathfrak{D}_{frontier} \neq 0_{\mathfrak{D}}$ do                ▷ Until no new vertices exist
 6:         $\mathfrak{D}_{succ} = \textsc{ExistsAbstract}\left(\underline{src}, \mathfrak{D}_{frontier} \cdot \mathfrak{D}_{SrcDst}\right)$
 7:         $\mathfrak{D}_{frontier} = \textsc{ReplaceVar}\left(\underline{src}, \underline{dst}, \mathfrak{D}_{succ}\right) \cdot \neg\mathfrak{D}_{reach}$
 8:         $\mathfrak{D}_{reach} = \mathfrak{D}_{reach} + \mathfrak{D}_{frontier}$
 9:     end while
10:     return $\mathfrak{D}_{reach}$
11: end procedure
```

At the beginning, we construct an MTBDD $\mathfrak{D}_{SrcDst}$ which represents the player 1 vertex successor relation, i.e. $f_{\mathfrak{D}_{SrcDst}}\left[\underline{src} = enc_v(\mathcal{B}_{src}), \underline{dst} = enc_v(\mathcal{B}_{dst})\right] = 1$ if there is an action, option and update to reach $\mathcal{B}_{dst}$ from $\mathcal{B}_{src}$. We start with the knowledge of the initial vertices being reachable, i.e. $\mathfrak{D}_{reach} = \mathfrak{D}_{frontier} = \mathfrak{D}_{init}$.

In the loop, we determine the successors of vertices of the current frontier by multiplying the frontier with the successor relation and abstracting from the variables in $\underline{src}$. As a result the successors are represented by $\mathfrak{D}_{succ}$ over the variables $\underline{dst}$. Subsequently we set the frontier to the set of new successors, where, by multiplying with $\neg\mathfrak{D}_{reach}$, we ensure that the frontier contains only unexpanded vertices. Note that we replace the source variables by destination variables since the frontier is supposed to be defined over $\underline{src}$. Finally, we add the frontier to the set of reached vertices and continue looping until no vertices remain unexpanded.

3.4. Solving Menu-games

In the previous section we have seen how to obtain an MTBDD $\mathfrak{D}_{sys}$ representing a Menu-game. This section focuses on how we can perform the fixed point iterations on this symbolical representation.

3.4.1. Symbolical Value Iteration

In Section 3.1.2, we have seen that probabilistic reachability for Menu-games, amounts to computing the respective fixed points for stochastic games, see Section 2.1.8. Thus, to compute probabilistic reachability, we have to express the fixed point iterations in terms of operations on the system's MTBDD $\mathfrak{D}_{sys}$.

Algorithm 4 Value Iteration

```
1:  procedure $\textsc{ValIter}(\mathfrak{D}_{sys}, p_1max, p_2max, \mathfrak{D}_G)$
2:      $\mathfrak{D}^u_{sys} \leftarrow \textsc{Abstract}\left(+, \underline{upd}, \mathfrak{D}_{sys}\right)$                      ▷ Unlabel distributions
3:      $\mathfrak{D}^{01}_{Opt} \leftarrow \textsc{ExistsAbstract}\left(\underline{dst}, \mathfrak{D}^u_{sys} \neq 0_{\mathfrak{D}}\right)$          ▷ Valid options
4:      $\mathfrak{D}^{01}_{Act} \leftarrow \textsc{ExistsAbstract}\left(\underline{opt}, \mathfrak{D}^{01}_{Opt}\right)$                 ▷ Valid actions
5:      $\mathfrak{D}^{new}_{res} \leftarrow \mathfrak{D}_G$
6:      repeat
7:          $\mathfrak{D}^{old}_{res} \leftarrow \mathfrak{D}^{new}_{res}$
8:          $\mathfrak{D}^{new}_{res} \leftarrow \textsc{ValIterStep}\left(\mathfrak{D}^u_{sys}, p_1max, p_2max, \mathfrak{D}^{old}_{res}, \mathfrak{D}^{01}_{Act}, \mathfrak{D}^{01}_{Opt}\right)$
9:          $\mathfrak{D}^{new}_{res} \leftarrow \text{ITE}\left(\mathfrak{D}_G, 1_{\mathfrak{D}}, \mathfrak{D}^{new}_{res}\right)$         ▷ Goal always reaches itself
10:     until $f_{\mathfrak{D}^{old}_{res}} \approx_\delta f_{\mathfrak{D}^{new}_{res}}$                          ▷ Until fixed point reached
11:     return $\mathfrak{D}^{new}_{res}$
12: end procedure
```

Algorithm 4 illustrates such a symbolical procedure to compute the fixed point, where the Boolean parameters $p_1 max$ and $p_2 max$ indicate which players aim to maximise the reachability and the BDD $\mathfrak{D}_G$ represents the goal set over $\underline{dst}$.

Knowing that unlabelled distributions suffice for our analysis, we begin by abstracting from the distributions' updates to obtain a smaller MTBDD $\mathfrak{D}^u_{sys}$. This is semantically equivalent to summing up over the updates, as we have seen in Section 2.1.1. In addition, we construct the auxiliary BDDs $\mathfrak{D}^{01}_{Opt}$ and $\mathfrak{D}^{01}_{Act}$, which indicate the options and actions that actually exist in $\mathfrak{D}^u_{sys}$. This information is necessary for ValIterStep, to avoid considering invalid encodings when minimising or maximising over options or actions.

We begin the actual value iteration with the MTBDD $\mathfrak{D}^{new}_{res}$ over $\underline{dst}$, representing the valuation mapping all goal vertices to 1. Setting $\mathfrak{D}^{new}_{res} \leftarrow 0_{\mathfrak{D}}$ will work too, but be equal to $\mathfrak{D}_G$ after one iteration. Bear in mind that $v_1^\perp$ must be part of G when computing minimal probabilistic reachability. The invocation of ValIterStep corresponds to a single application of the respective transformer $pre^{\pm,\pm}_G$ on the current valuation, given by $\mathfrak{D}^{old}_{res}$. The resulting valuation $\mathfrak{D}^{new}_{res}$ is then adapted to ensure that goal vertices always have the reachability probability 1. Note that this modification is necessary to meet the definition, since otherwise, goal states without self-loops will have degraded reachability – adding self-loops is an alternative, too. The loop terminates once the reachability probabilities of the player 1 vertices do not change significantly. Due to the numerical nature of this approach, exact equality will not be reached in a finite number of iterations for many models. Therefore, in practice, some equality metric with respect to a difference δ must be employed.

3.4.2. Symbolical Valuation Transformer Application

The core part of value iteration is the application of a transformer $pre_G^{\pm,\pm}$ on the current valuation. Algorithm 5 describes a procedure which handles this. The innermost expression of a value transformer for stochastic games is always the computation of a weighted sum $\sum_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v')$ – a matrix-vector-multiplication if you will. The respective MTBDD $\mathfrak{D}_{MV}$, representing a mapping of options to probabilities, is obtained from summation over the destinations of $\mathfrak{D}^u_{sys} \cdot \mathfrak{D}^{old}_{res}$, which corresponds to the inner multiplication $\widehat{v_p}(v') \cdot w(v')$. Afterwards, conforming to the structure of $pre_G^{\pm,\pm}$, the two phases of non-determinism resolving follow.

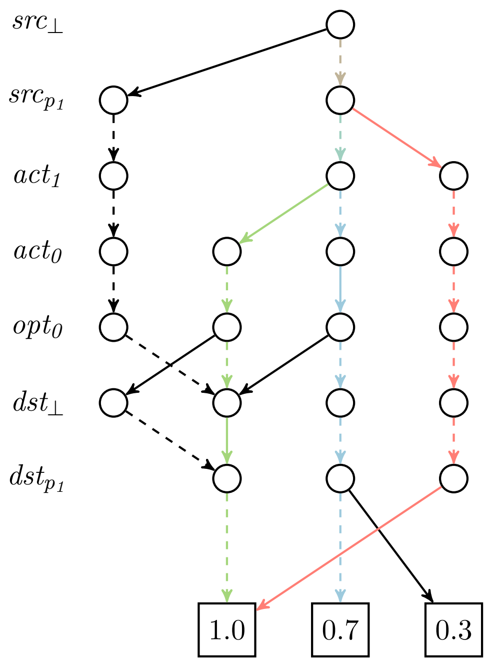
Algorithm 5 Value Iteration Step

```
 1: procedure ValIterStep($\mathfrak{D}^u_{sys}, p_1max, p_2max, \mathfrak{D}^{old}_{res}, \mathfrak{D}^{01}_{Act}, \mathfrak{D}^{01}_{Opt}$)
 2:     $\mathfrak{D}_{MV}$ = Abstract $\left(+, \underline{dst}, \mathfrak{D}^u_{sys} \cdot \mathfrak{D}^{old}_{res}\right)$        ▷ $\sum_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v')$
 3:
 4:     if $p_2max$ then                                                  ▷ Min-/Maximise over options
 5:         $\mathfrak{D}^{new}_{res} \leftarrow$ ITE $\left(\neg\mathfrak{D}^{01}_{Opt}, -1_{\mathfrak{D}}, \mathfrak{D}_{MV}\right)$
 6:         $\mathfrak{D}^{new}_{res} \leftarrow$ MaxAbstract $\left(\underline{opt}, \mathfrak{D}^{new}_{res}\right)$
 7:     else
 8:         $\mathfrak{D}^{new}_{res} \leftarrow$ ITE $\left(\neg\mathfrak{D}^{01}_{Opt}, 2_{\mathfrak{D}}, \mathfrak{D}_{MV}\right)$
 9:         $\mathfrak{D}^{new}_{res} \leftarrow$ MinAbstract $\left(\underline{opt}, \mathfrak{D}^{new}_{res}\right)$
10:     end if
11:
12:     if $p_1max$ then                                                  ▷ Min-/Maximise over actions
13:         $\mathfrak{D}^{new}_{res} \leftarrow$ ITE $\left(\neg\mathfrak{D}^{01}_{Act}, -1_{\mathfrak{D}}, \mathfrak{D}^{new}_{res}\right)$
14:         $\mathfrak{D}^{new}_{res} \leftarrow$ MaxAbstract $\left(\underline{act}, \mathfrak{D}^{new}_{res}\right)$
15:     else
16:         $\mathfrak{D}^{new}_{res} \leftarrow$ ITE $\left(\neg\mathfrak{D}^{01}_{Act}, 2_{\mathfrak{D}}, \mathfrak{D}^{new}_{res}\right)$
17:         $\mathfrak{D}^{new}_{res} \leftarrow$ MinAbstract $\left(\underline{act}, \mathfrak{D}^{new}_{res}\right)$
18:     end if
19:     return ReplaceVar $\left(\underline{src}, \underline{dst}, \mathfrak{D}^{new}_{res}\right)$           ▷ Returns $\mathfrak{D}^{new}_{res}$ over $\underline{dst}$
20: end procedure
```

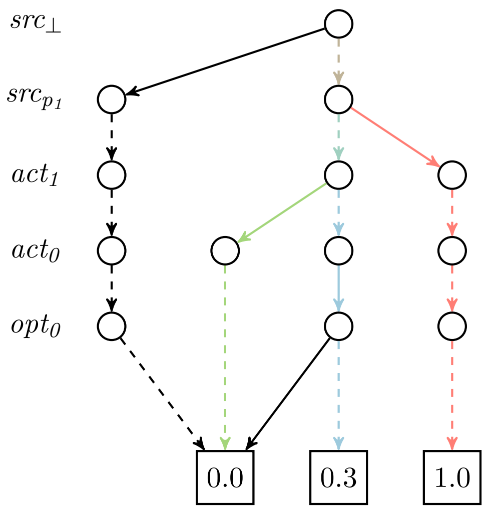
Player 2 chooses prior to player 1. Let us first consider the case that player 2 aims to maximise the reachability probability, i.e. we are computing the upper bound for minimal or maximal reachability. At first glance, it may seem to suffice to apply MaxAbstract over the options, to choose the maximising one. This, however, is not the case. In $\mathfrak{D}_{MV}$, both the invalid options and the valid options which result in the reachability probability 0, are mapped to 0. Thus prior to maximising, we have to ensure that no invalid option can be taken by mapping all invalid options to -1. Since every player 2 vertex has at least one option, it is guaranteed that -1 will not appear anymore after MaxAbstract. Dually, when minimising, the invalid options must be mapped to a value greater than 1, in our case 2, to ensure that invalid options cannot be chosen as long as valid alternatives exist. The subsequent MinAbstract ensures that the minimal valid option is chosen and no valuation yielding 2 occurs afterwards.

Analogously, the second half of the algorithm implements the choice of player 1, but operates on the result $\mathfrak{D}_{res}^{new}$ of the first choice. Lastly, the result's variables $\underline{src}$ are replaced by $\underline{dst}$, to be comparable to the result of the previous iteration, which is encoded over $\underline{dst}$, too.

Example 3.9. Let $\mathfrak{D}_{sys}$ be the Menu-game from Figure 3.2 and assume we are interested in finding the upper bound for maximal probabilistic reachability of B_1. In the following, we illustrate the first iteration step on the game's MTBDD.



(a) MTBDD $\mathfrak{D}_{sys}^{u}$



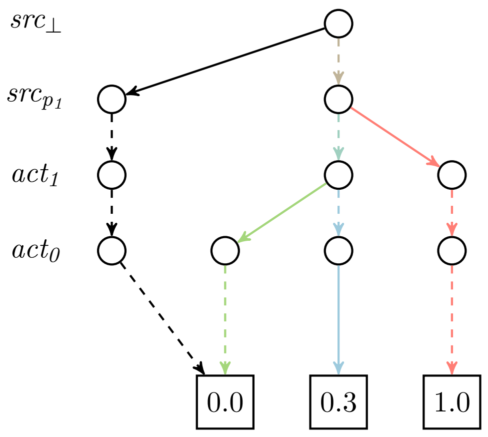
(b) MTBDD $\mathfrak{D}_{MV}$

Figure 3.4.: Unlabeling distributions and matrix-vector-multiplication

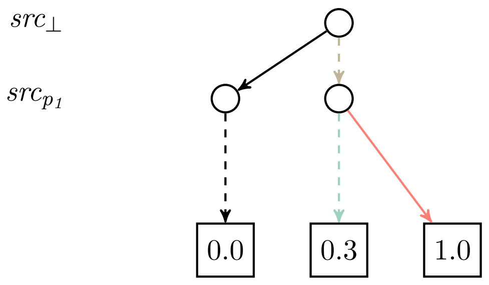
To begin with, we unlabel the system's distributions, see Figure 3.4a. Since no two updates of a player 2 vertex lead to the same destination, the result corresponds to ExistsAbstract $(\underline{upd}, \mathfrak{D}_{sys})$. Since we compute a bound for the maximal reachability we do not have to add $v_1^\perp$ to goal, such that $\mathfrak{D}_G = \text{Set}_{\underline{dst}}\,(enc_v(B_1))$.

Figure 3.4b illustrates the MTBDD $\mathfrak{D}_{MV}$, which represents the reachability probability achievable by the different options, based on the current valuation $\mathfrak{D}_{res}^{old} = \mathfrak{D}_G$. For clarity, we also illustrated the valid paths in the MTBDD leading to zero. For example, we see that going from B_0 to the trap state $v_1^\perp$ (green path) is a valid option which will result in the reachability probability 0. Furthermore, given the current valuation, the option on the blue path seems to merely yield the reachability 0.3. This is due to the fact that the current valuation maps B_0 to 0.

Now that all action-option combinations for every player 1 vertex are mapped to estimated reachability probabilities, we resolve the non-determinism corresponding to the value transformer for the upper bound of maximal probabilistic reachability, i.e. by maximising over both players' choices.



(a) $\mathfrak{D}_{res}^{new}$ after player 2 choice



(b) $\mathfrak{D}_{res}^{new}$ after player 1 choice

Figure 3.5.: Both players maximising over their choices

The result of selecting the maximising options is illustrated in Figure 3.5a. While there is actually not much of a choice, note that player 2 settles for option o_1, for the vertex (B_0, a), instead of o_2 since $0.3 > 0$ (blue path). Subsequently player 1 picks the maximising actions, which yields the valuation represented in Figure 3.5b. The first iteration resulted in assigning the maximal reachability probability 0.3 to block B_0. This, however, is not the fixed point and follow up iterations will show that the value in fact converges to 1. $\square$

3.5. Backward Refinement

In the previous sections, we have seen how a to construct a symbolic representation of Menu-games and perform value iteration on this structure. However, we have also seen that the bounds may turn out too coarse to be useful, e.g. $Pr^{max}(\Diamond G) \in [0, 1]$ does not give any new information. In Example 2.39 we already illustrated that this is due to our partition Q being too coarse. Since Q is induced from a set of predicates $\mathcal{P}$, this implies that the predicates do not suffice to distinguish between states which influence the reachability of the goal set. Consider our running example program P_{simple} from Listing 2.1. Using a predicate set which did not reference the predicate $run > 0$, states which have the actions b or c enabled could not be distinguished – see Example 3.2.

Therefore, this section introduces the *backward refinement* procedure, proposed by

[Wachter, 2011], which analyses a Menu-game, constructed with respect to a set of predicates $\mathcal{P}$, to derive additional predicates which will eventually improve the precision. In contrast to Wachter, we illustrate a fully symbolic approach to this procedure, avoiding any explicit representation.

3.5.1. Pivot Blocks

Let $\widehat{\mathcal{G}}$ be a Menu-game and $G^{\#}$ the set of goal vertices. Employing value iteration will leave us with the lower and upper bound valuations w^l and w^u for the probabilistic reachability of interest. Now, if $w^u(v_{init})$ and $w^l(v_{init})$ differ, there must be blocks in the system consisting of states which influence the reachability of $G^{\#}$ in two different ways, e.g. one subsumed state may never reach a goal state, while another may always do so. [Wachter, 2011] coined the term *pivot blocks* to denote such refinement candidates where precision is lost.

At first glance, it might seem a good idea to consider all blocks where the upper and lower bounds differ as pivot blocks. This is a bad idea though. For example, consider the Menu-game of our program P_{simple}, illustrated in Figure 3.1. We have seen that the bounds for maximal probabilistic reachability of B_2 are $[0, 1]$ for the initial state. However, refining block B_0 (if it weren't already a s single-state block) would not improve the bounds since the actual cause for the deviation lies in its successor block B_1, whose bounds propagate to B_0. If we were to split B_1 in such a way that states which enable different actions do not fall together anymore, there would be no reachable trap state and, as a result, the lower bound would improve.

In fact, a good indicator for a state not being a pivot state is that both the lower and upper bounds strategies for this block do not differ – this is true for B_0. If the way non-determinism is resolved influences the reachability, then the strategies for lower and upper bound must differ. Thus, a better criterion to identify pivot blocks is to consider states where the strategies for both lower and upper bound differ.

However, different strategies for a block do not imply different bounds, e.g. both strategies may lead to different goal blocks with probability 1. To avoid such spurious differences between strategies, we will have to ensure that strategies only differ, if the resulting bounds differ, too. Assuming we can avoid spurious differences, different strategies in a block will imply different bounds and qualify the blocks as pivot block.

Definition 3.10 (Pivot Block [Wachter, 2011]). Let (σ_1^l, σ_2^l) and (σ_1^u, σ_2^u) be the strategy pairs yielding the lower and upper bound valuations in a Menu-game. A reachable block B is a pivot block, if the (composed) strategies differ, i.e.

$$\sigma_2^l(\sigma_1^l(v)) \neq \sigma_2^u(\sigma_1^u(v)).$$

□

Note that the existence of a pivot block is guaranteed if the bounds for probabilistic reachability differ (in the initial block), because if there were no pivot block, i.e. the lower and upper strategies of block would be equal, the bounds would be equal, too.

3.5.2. Deriving Refinement Predicates

Let v_1 be a pivot block. The emerging question is which new predicates to introduce to avoid the current cause of losing precision, i.e. to split the block v_1 in such a way that the states causing the currently regarded uncertainty are separated.

In Menu-games, uncertainty is introduced by the player 2 non-determinism, which distinguishes behaviours of the states subsumed by a block. Thus in a pivot block v_1 for at least one of the player 2 vertices $v_2^l := \sigma_1^l(v_1)$ or $v_2^u := \sigma_1^u(v_1)$ different options must have been chosen.

If an action does not contribute to the uncertainty, i.e. $\sigma_2^l(v_2) = \sigma_2^u(v_2), v_2 \in \{v_2^l, v_2^u\}$, clearly no splitting is necessary. However, if the player 2 strategies differ, then the respectively reached distributions differ too. Consequently, a predicate must be introduced which allows to distinguish between both options, i.e. rebuilding the Menu-game with respect to the extended predicate set should group these behaviours in different blocks.

We distinguish between two kinds of reachable distributions – the trap-distribution leading to the trap block and the remaining ordinary distributions. In the previous section, we have seen that if a player 2 vertex has the option of reaching the trap behaviour, this is due to the predicate set not sufficing to distinguish between states which have different sets of actions enabled. Clearly, in such a case the guard of the respective action must be introduced as predicate.

In the other case, the upper and lower bound strategies in v_2^l or v_2^u lead to ordinary distributions over blocks only. Note that since since the distributions of such an player 2 vertex are induced by the same command (action), the weights of the distributions must be the same. As a result, they can only differ in their successors. Bear in mind that using a form of predicate abstraction, we distinguish blocks by the sets of predicates the satisfy. Considering this, we must introduce a predicate which splits the behaviours corresponding to these options into different blocks, i.e. a predicate which distinguishes the effects of the corresponding updates. In fact, in Definition 3.6 we have already used a construct guaranteeing that an expression e holds after an assignment E – the weakest precondition $WP_E(e) := e\,[Var/E(Var)]$.

Overall, we derive refinement predicates for a pivot block v_{pivot} as follows

$$\textsc{DerivePred}(v_{pivot}) := \textsc{DerivePred}(\sigma_1^l(v_{pivot})) \cup \textsc{DerivePred}(\sigma_1^u(v_{pivot})),$$

where for a player 2 vertex v_2

$$\textsc{DerivePred}(v_2) := \begin{cases} \emptyset & \text{if } \sigma_2^l(v_2) = \sigma_2^u(v_2) \\ \{g_c\} & \text{if } \sigma_2^l(v_2) = v_p^\perp \text{ or } \sigma_2^u(v_2) = v_p^\perp \\ \{WP_{E_i}(p)\} & \text{if } \sigma_2^l(v_2)(u_{a,i}, v_1') = \sigma_2^u(v_2)(u_{a,i}, v_1'') > 0 \text{ and } [\![p]\!]_{v_1'} \neq [\![p]\!]_{v_1''} \end{cases},$$

with c being the command corresponding to the action a_c chosen by the player 1 strategy, and v_1', v_1'' being the blocks distinguishable by a predicate p [Wachter, 2011].

3.5.3. Backward Refinement Procedure

Taken as a whole, the backward refinement procedure amounts to computing a Menu-game, obtaining valuations w^l and w^u for the lower and upper bound of the respective reachability property and repeating the process with respect to predicates derived from pivot blocks if the bounds turn out being too coarse. Figure 3.6 illustrates this scheme.

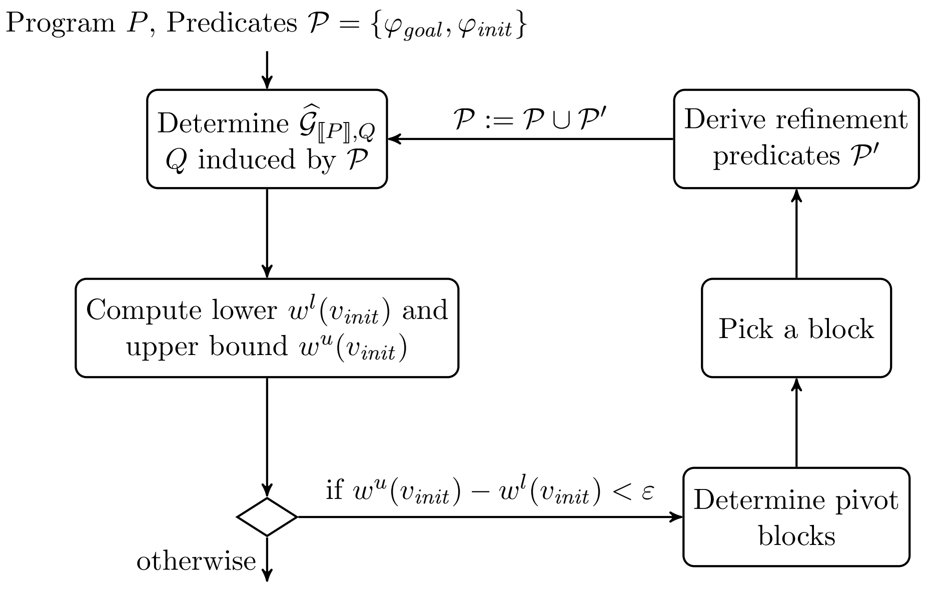

Program P, Predicates $\mathcal{P} = \{\varphi_{goal}, \varphi_{init}\}$
Determine $\widehat{\mathcal{G}}_{[\![P]\!],Q}$ Q induced by $\mathcal{P}$
$\mathcal{P} := \mathcal{P} \cup \mathcal{P}'$
Derive refinement predicates $\mathcal{P}'$
Compute lower $w^l(v_{init})$ and upper bound $w^u(v_{init})$
Pick a block
if $w^u(v_{init}) - w^l(v_{init}) < \varepsilon$
Determine pivot blocks
otherwise

Figure 3.6.: General structure of the backward refinement procedure

[Wachter, 2011] proves that the given predicate derivation function does indeed refine Q in such a way that the concrete states which caused the uncertainty for a pivot block, end up in different blocks afterwards, thus trivially guaranteeing the termination of the backward refinement procedure for finite-state programs.

Example 3.11. Let us walk through the process of computing the maximal probability of the system P_{simple} (from Listing 2.1) breaking, i.e. $Pr^{max}(\Diamond [\![phase = 3]\!])$. To this end we employ the backward refinement procedure using the predicates

$$\mathcal{P} = \{phase = 0, phase = 1, phase = 2, phase = 3, run \leq 0\}$$

for the initial Menu-game. Note that the single goal predicate $phase = 3$ would have sufficed too, but would have resulted in more refinement iterations. In contrast to Example 3.2, we additionally use the predicate $run \leq 0$ to avoid reaching a trap state. Additionally, the initial state expression must often be added to the predicates, to ensure that there exists only one initial block. However, for the given example this is not needed, as the initial state expression is already satisfied by only one block.

Figure 3.7 illustrates the refinement process. The initial Menu-game with respect to the given set of predicates is depicted in Figure 3.7a. For better readability, blocks are only labeled with the predicates they satisfy and the variables $phase$ and run are abbreviated by p and r. The maximal reachability probability bounds, with respect to the red tinted goal blocks, are annotated as intervals next to the blocks.

First Refinement Step

In contrast to Example 3.2, the lower bound for reaching a goal block is not zero anymore. Nevertheless, the interval $[0.0591, 1]$ is yet too imprecise to be meaningful. Let us analyse where precision is lost. In the initial Menu-game, there is only one block featuring player 2 non-determinism $B = \{p = 1, \neg (r \leq 0)\}$ – also indicated by thick borders. Clearly, this block is a pivot block since the strategies yielding the lower and upper bounds lead to different distributions, i.e. the left option is chosen for the lower and the right one for the upper bound.

The non-determinism comes from the states $s_1 = \{\mathit{phase} \mapsto 1, \mathit{run} \mapsto 2\}$ and $s_3 = \{\mathit{phase} \mapsto 1, \mathit{run} \mapsto 1\}$ being contained in the pivot block, and only s_3 having a b-successor to a state where $r \leq 0$ holds. To derive the refinement predicate splitting (at least) these states, we compute the weakest precondition of the distinguishing predicate $r \leq 0$ with respect to the update $u_{b,1}$, which leads to different blocks:

$$WP_{E_1}(r \leq 0) \equiv \mathit{run} - 1 \leq 0 \equiv \mathit{run} \leq 1,$$

where $E_1 = \{\mathit{run} \mapsto \mathit{run} - 1, \mathit{phase} \mapsto \mathit{phase}\}$ is the assignment corresponding to update $u_{b,1}$. We add this predicate to $\mathcal{P}$ and rebuild the Menu-game.

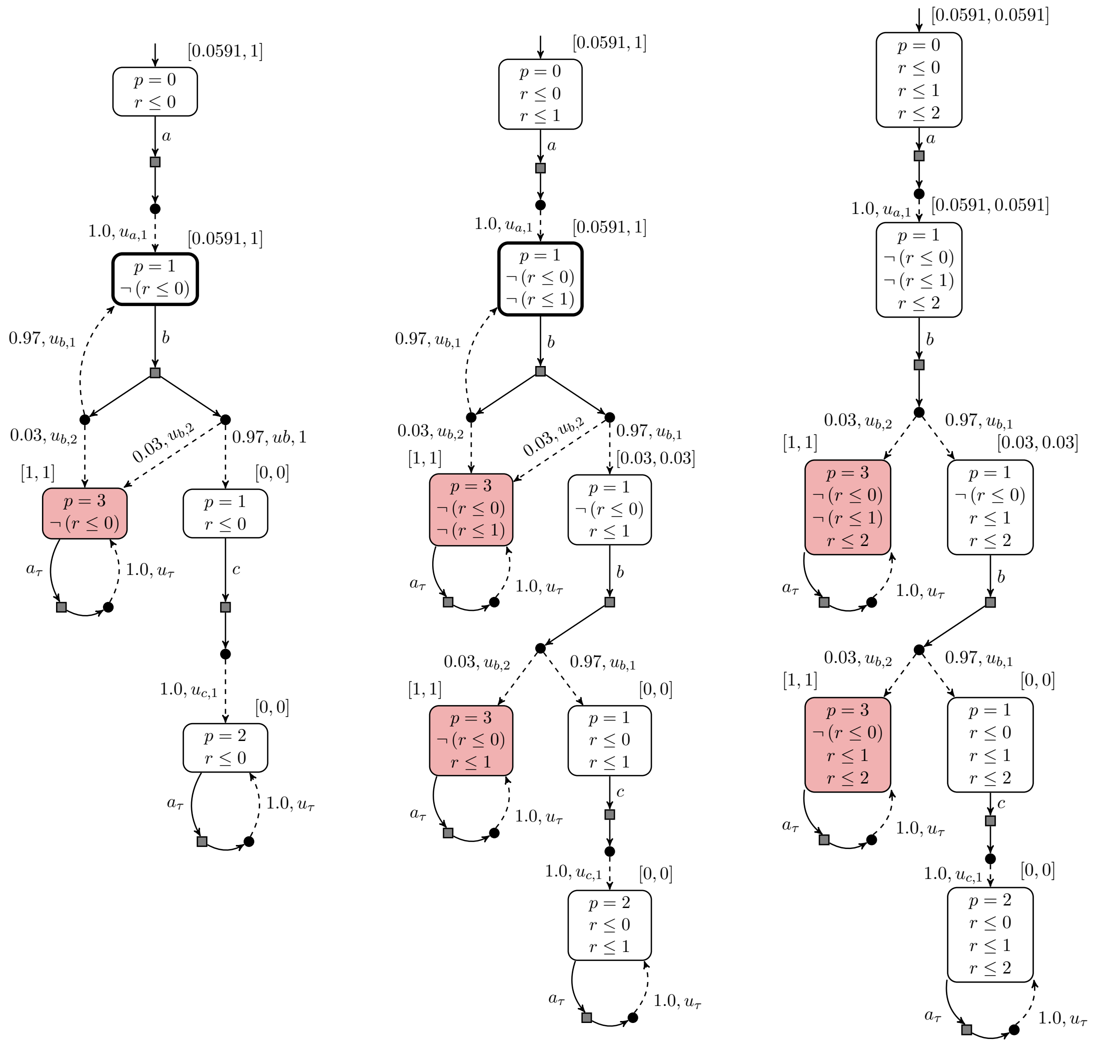
Second Refinement Step

Having split $B = \{p = 1, \neg (r \leq 0)\}$ into $B_1 = \{p = 1, \neg (r \leq 0), \neg (r \leq 1)\}$ and $B_2 = \{p = 1, \neg (r \leq 0), r \leq 1\}$, we still do not get better bounds than $[0.0591, 1]$. This time B_1 is the pivot state, where the update $u_{b,1}$ leads both the minimising and the maximising option to different blocks – distinguished by $r \leq 1$. However, in contrast to the initial Menu-game, the problem is that block B_1 contains states for which $\mathit{run} = 2$ holds and those where run is mapped to something greater. The respective predicate is derived as follows:

$$WP_{E_1}(r \leq 1) \equiv \mathit{run} - 1 \leq 1 \equiv \mathit{run} \leq 2.$$

Rebuilding the Menu-game another time, yields precise bounds $[0.0591, 0.0591]$ for the maximal probabilistic reachability of the system breaking.

The result may be somewhat disillusioning, considering that the number of blocks turned out to be equal to that of the concrete state space we have seen in Example 3.2. This level of refinement is usually not needed though. For example, we did in fact not even use the initial state expression as a predicate, thereby building a Menu-game which proves that the maximal probability of the system breaking is 0.0591 no matter what the initial value of run is, i.e. we would have obtained the very same small Menu-game if the initial state expression were just $\mathit{phase} = 0$, defining a system with infinitely many initial states.



(a) Initial Menu-game (b) Refined Menu-game (c) Final Menu-game

Figure 3.7.: Backward refinement of the Menu-game of P_{simple}

□

3.5.4. Deriving Strategies Symbolically

In the previous section we have seen that to derive predicates we need the lower and upper bound strategies for both players. In this section, we elaborate on how the symbolical value iteration can be extended to also yield strategies.

Minimal and Maximal Representatives

In Section 3.4.2, we used the MaxAbstract (or MinAbstract) operation to maximise (or minimise) over actions and options. This time, we are not only interested in the maximal and minimal values for a player's vertex but also the choice itself, as this corresponds to the notion of strategy. To this end, we introduce novel operations MaxAbstractRepresentative and its minimising counterpart, which do not abstract from variables but pick a unique representative which realises the maximal (or minimal) value. Formally, for an MTBDD $\mathfrak{D}$ over the variables $(\underline{x}, \underline{z})$, the operations are defined as follows:

- MaxAbstractRepresentative$(\underline{z}, \mathfrak{D})$ yields an BDD over $(\underline{x}, \underline{z})$ representing the function
$$f(\underline{x}, \underline{z}) := \begin{cases} 1 & \text{if } \underline{z} = z_{max}(\underline{x}) \\ 0 & \text{otherwise} \end{cases},$$
where z_{max} is the function choosing a maximal representative, i.e. $f_{\mathfrak{D}}(\underline{x}, z_{max}(\underline{x})) := \max_z f_{\mathfrak{D}}(\underline{x}, z)$.

- Dually, MinAbstractRepresentative$(\underline{z}, \mathfrak{D})$ yields an BDD over $(\underline{x}, \underline{z})$ representing the function
$$f(\underline{x}, \underline{z}) := \begin{cases} 1 & \text{if } \underline{z} = z_{min}(\underline{x}) \\ 0 & \text{otherwise} \end{cases},$$
where z_{min} is the function choosing a minimal representative, i.e. $f_{\mathfrak{D}}(\underline{x}, z_{min}(\underline{x})) := \min_z f_{\mathfrak{D}}(\underline{x}, z)$.

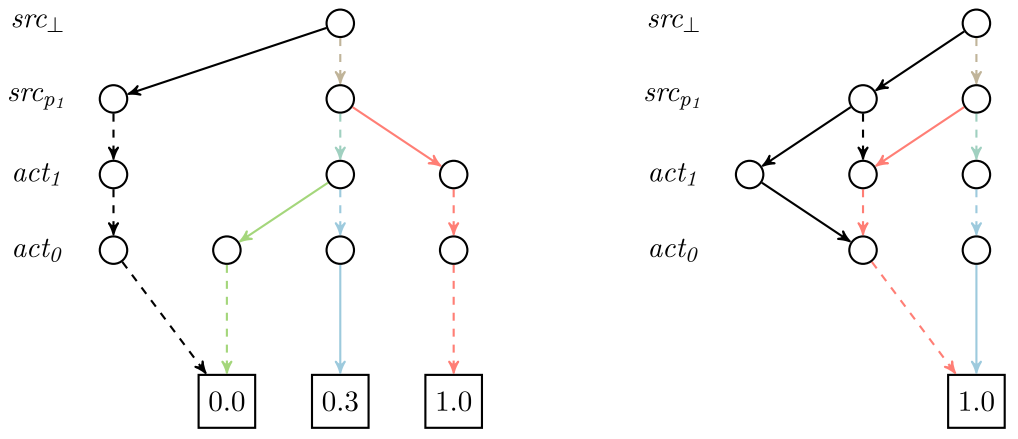
Example 3.12. Figure 3.8 illustrates the usage of MaxAbstractRepresentative on the MTBDD $\mathfrak{D}$, which we already used in Example 3.9. Again, we explicitly depict paths to the 0 terminal node, which correspond to valid paths of the respective model.

To begin with, let us focus on choosing the maximal representative for the block encoded by $(src\bot, src_{p_1}) = (0, 0)$. From Figure 3.8a it is easy to see that only two actions (green and blue path) come into question being the representative. Since the blue action realises the greater value we drop the green one, i.e. for this block only the blue action is mapped to 1 in the resulting MTBDD.

Furthermore, it is easy to see that the states encoded by $(src\bot, src_{p_1}) = (1, 0)$ and $(src\bot, src_{p_1}) = (0, 1)$ have only one valid action to choose from (black and red path). Accordingly these paths are kept unchanged by the representative picking.

It might be surprising though, that the resulting MTBDD has four paths to the 1-terminal node, i.e. the valuation $(1, 1, 1, 0)$ does not correspond to any valid path in $\mathfrak{D}$. This is due to the fact, that representatives are chosen for all paths – even those that are invalid with respect to the model. In fact, the MTBDD $\mathfrak{D}$ maps all actions of the block encoded by $(src\bot, src_{p_1}) = (1, 1)$ to zero, such that any action is a valid representative and any extension of the valuation $(1, 1)$ may occur in the result.

Thus, when picking representatives of actions or options, we have to multiply the result with $\mathfrak{D}^{01}_{Act}$, respectively $\mathfrak{D}^{01}_{Opt}$, to avoid considering invalid actions and options in our computation.



(a) Action choices $\mathfrak{D}$ (b) Maximising player 1 strategy

Figure 3.8.: Illustration of MaxAbstractRepresentative

□

Updating Strategies

With the notion of minimal and maximal representatives, we can derive strategies in every value iteration step. It remains to discuss what to do with the strategy of a previous iteration, i.e. whether to completely replace the old strategy by the new one or update separate parts of it.

Figure 3.9 illustrates why we cannot simply replace the old strategy when maximising. Let a correspond to the player 1 strategy of the current value iteration step for block B, yielding the upper bound probability 0.75. Then, in the next iteration, it seems like both action a and b are valid maximising representatives, since choosing b always leads to a state with the upper bound 0.75, too. This, however, is wrong since a strategy choosing b would actually realise the reachability value 0, never leaving B.

As a result, if the new maximising strategy for a block or player 2 vertex does not strictly improve the reachability probability, it should be discarded as it may build upon values propagated by other actions, like a in our case.

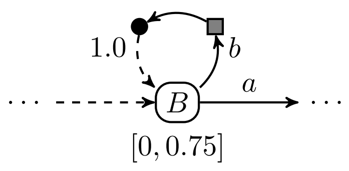


Figure 3.9.: Case to consider when updating strategies

Adapted Value Iteration Procedure

Using these concepts, we can now extend the value iteration step, from Algorithm 5, with the minimal and maximal representatives computation, to derive respective strategies. Additionally, we have to make sure to not simply replace the previous strategies, but update them correctly when maximising.

Algorithm 6 describes the extended procedure, which now consists of four parts and takes the strategies of the previous step as additional parameters:

Computing player 2 strategies In the first place, similar to the original algorithm, we compute the MTBDD $\mathfrak{D}_{MV}$, representing the values the different options yield. Subsequently, in lines 4 to 10, we determine the maximising (or minimising) options. In contrast to the original algorithm, we do not simply pick the optimal values but the representative options. Note that, as illustrated in Example 3.12, we multiply with the valid options $\mathfrak{D}^{01}_{Opt}$ to avoid storing strategies for invalid blocks.

Updating player 2 strategies We know that when maximising, we may only update the strategies of those player 2 vertices where the new strategy, given by the representative options, realises strictly greater reachability values. To this end, in lines 12 to 16, we compute the values the old and the new strategy realise for a player 2 vertex. The combined strategy is then given by the new strategy for the player 2 vertices which realise strictly greater values using the new strategy, and the old strategy for the remaining player 2 vertices.

Computing player 1 strategies Analogous to the computation of the player 2 strategy, the new player 1 strategy is given by the corresponding valid representatives.

Updating player 1 strategies When maximising, the combination of the old and new player 1 strategy is obtained similar to its player 2 counterpart. We compute for which blocks the new strategies yield improved results and replace the old strategy decisions for them by the new ones.

Finally, not only the reachability probability values are returned but also the updated strategies for both players.

Algorithm 6 Value Iteration Step Considering Strategies

```
1: procedure ValIterStep($\mathfrak{D}^u_{sys}, p_1max, p_2max, \mathfrak{D}^{old}_{res}, \mathfrak{D}^{01}_{Act}, \mathfrak{D}^{old}_{\sigma_1}, \mathfrak{D}^{old}_{\sigma_2}, \mathfrak{D}^{01}_{Opt}$)
2:     $\mathfrak{D}_{MV} = \textsc{Abstract}\left(+, \underline{dst}, \mathfrak{D}^u_{sys} \cdot \mathfrak{D}^{old}_{res}\right)$    ▷ $\sum_{v' \in E(v_p)} \widehat{v_p}(v') \cdot w(v')$
3:
4:     if $p_2max$ then    ▷ Min-/Maximise over options
5:         $\mathfrak{D}^{new}_{res} \leftarrow \text{ITE}\left(\neg\mathfrak{D}^{01}_{Opt}, -1_{\mathfrak{D}}, \mathfrak{D}_{MV}\right)$
6:         $\mathfrak{D}^{new}_{\sigma_2} \leftarrow \textsc{MaxAbstractRepresentative}\left(\underline{opt}, \mathfrak{D}^{new}_{res}\right) \cdot \mathfrak{D}^{01}_{Opt}$
7:     else
8:         $\mathfrak{D}^{new}_{res} \leftarrow \text{ITE}\left(\neg\mathfrak{D}^{01}_{Opt}, 2_{\mathfrak{D}}, \mathfrak{D}_{MV}\right)$
9:         $\mathfrak{D}^{new}_{\sigma_2} \leftarrow \textsc{MinAbstractRepresentative}\left(\underline{opt}, \mathfrak{D}^{new}_{res}\right) \cdot \mathfrak{D}^{01}_{Opt}$
10:    end if
11:
12:    if $p_2max \wedge f_{\mathfrak{D}^{old}_{\sigma_2}} \neq 0$ then    ▷ Update improvements when maximising
13:        $\mathfrak{D}^{old}_{p_2res} = \textsc{MaxAbstract}\left(\underline{opt}, \mathfrak{D}^{old}_{\sigma_2} \cdot \mathfrak{D}_{MV}\right)$
14:        $\mathfrak{D}^{new}_{p_2res} = \textsc{MaxAbstract}\left(\underline{opt}, \mathfrak{D}^{new}_{\sigma_2} \cdot \mathfrak{D}_{MV}\right)$
15:        $\mathfrak{D}^{new}_{\sigma_2} = \text{ITE}\left(\mathfrak{D}^{new}_{p_2res} > \mathfrak{D}^{old}_{p_2res}, \mathfrak{D}^{new}_{p_2res}, \mathfrak{D}^{old}_{p_2res}\right)$
16:    end if
17:    $\mathfrak{D}^{new}_{res} = \textsc{MaxAbstract}\left(\underline{opt}, \mathfrak{D}^{new}_{\sigma_2} \cdot \mathfrak{D}^{new}_{res}\right)$
18:
19:    if $p_1max$ then    ▷ Min-/Maximise over actions
20:        $\mathfrak{D}^{new}_{res} \leftarrow \text{ITE}\left(\neg\mathfrak{D}^{01}_{Act}, -1_{\mathfrak{D}}, \mathfrak{D}^{new}_{res}\right)$
21:        $\mathfrak{D}^{new}_{\sigma_1} \leftarrow \textsc{MaxAbstractRepresentative}\left(\underline{act}, \mathfrak{D}^{new}_{res}\right) \cdot \mathfrak{D}^{01}_{Act}$
22:    else
23:        $\mathfrak{D}^{new}_{res} \leftarrow \text{ITE}\left(\neg\mathfrak{D}^{01}_{Act}, 2_{\mathfrak{D}}, \mathfrak{D}^{new}_{res}\right)$
24:        $\mathfrak{D}^{new}_{\sigma_1} \leftarrow \textsc{MinAbstractRepresentative}\left(\underline{act}, \mathfrak{D}^{new}_{res}\right) \cdot \mathfrak{D}^{01}_{Act}$
25:    end if
26:
27:    if $p_1max \wedge f_{\mathfrak{D}^{old}_{\sigma_1}} \neq 0$ then    ▷ Update improvements when maximising
28:        $\mathfrak{D}^{old}_{p_1res} = \textsc{MaxAbstract}\left(\underline{act}, \mathfrak{D}^{old}_{\sigma_1} \cdot \mathfrak{D}^{new}_{res}\right)$
29:        $\mathfrak{D}^{new}_{p_1res} = \textsc{MaxAbstract}\left(\underline{act}, \mathfrak{D}^{new}_{\sigma_1} \cdot \mathfrak{D}^{new}_{res}\right)$
30:        $\mathfrak{D}^{new}_{\sigma_1} = \text{ITE}\left(\mathfrak{D}^{new}_{p_1res} > \mathfrak{D}^{old}_{p_1res}, \mathfrak{D}^{new}_{p_1res}, \mathfrak{D}^{old}_{p_1res}\right)$
31:    end if
32:    $\mathfrak{D}^{new}_{res} = \textsc{MaxAbstract}\left(\underline{act}, \mathfrak{D}^{new}_{\sigma_1} \cdot \mathfrak{D}^{new}_{res}\right)$
33:
34:    return $\left(\textsc{ReplaceVar}\left(\underline{src}, \underline{dst}, \mathfrak{D}^{new}_{res}\right), \mathfrak{D}^{new}_{\sigma_1}, \mathfrak{D}^{new}_{\sigma_2}\right)$    ▷ Returns $\mathfrak{D}^{new}_{res}$ over $\underline{dst}$
35: end procedure
```

To ensure that the lower and upper bound strategies do not differ, as assumed in Section 3.5.1, we always have to compute the lower bound first and then start the upper bound computation with the results of the lower bound.

3.5.5. Computing Pivot Blocks Symbolically

In Section 3.5.2, we have seen that besides the reachability probabilities we also need the respective strategies for both players. Using the procedure from the previous section we do now have all the necessary parts to compute the set of pivot blocks.

We know that a player 1 vertex v is a pivot block if the (composed) lower and upper bound strategies differ, i.e. $\sigma_2^l(\sigma_1^l(v)) \neq \sigma_2^u(\sigma_1^u(v))$ (see Definition 3.10). Using the strategy MTBDDs $\mathfrak{D}_{\sigma_1^l}, \mathfrak{D}_{\sigma_2^l}, \mathfrak{D}_{\sigma_1^u}$ and $\mathfrak{D}_{\sigma_2^u}$, obtained through value iteration for both the lower and upper bound, we can compute the MTBDDs representing the distributions reached in both cases for all blocks. The lower distributions for all blocks are obtained by combining both players' lower strategies and multiplying them with the system's MTBDD $\mathfrak{D}_{sys}^u$:

$$\mathfrak{D}_{v_p^l} = \textsc{MaxAbstract}\left(\left(\underline{act}, \underline{opt}\right), \mathfrak{D}_{sys}^u \cdot \mathfrak{D}_{\sigma_1^l} \cdot \mathfrak{D}_{\sigma_2^l}\right),$$

where the abstraction is only employed to get rid of the action and option variables, since for each block there exactly only one non-zero action-option pair anyways. Analogously, the upper bound distributions the different blocks reach are given by:

$$\mathfrak{D}_{v_p^u} = \textsc{MaxAbstract}\left(\left(\underline{act}, \underline{opt}\right), \mathfrak{D}_{sys}^u \cdot \mathfrak{D}_{\sigma_1^u} \cdot \mathfrak{D}_{\sigma_2^u}\right).$$

Now, finding the blocks where the lower and upper bound distributions differ amounts to computing $\mathfrak{D}_{v_p^l} \neq \mathfrak{D}_{v_p^u}$ and abstracting from the destinations afterwards:

$$\mathfrak{D}_{pivot} = \textsc{ExistsAbstract}\left(\underline{dst}, \mathfrak{D}_{v_p^l} \neq \mathfrak{D}_{v_p^u}\right).$$

4. Optimisation Opportunities

In the previous chapter we have seen procedures which make symbolical backward refinement amenable. Although functional, in a naive implementation, both the construction of a Menu-game and the symbolical value iteration scale rather bad, such that analyses take minutes, even for small numbers of predicates (< 20). This chapter elaborates on optimisation opportunities in both the Menu-game construction and value iteration, which overall significantly improve the feasibility of the approach. In the following, we often refer to the source-variables instance Var_0 as Var when the context is clear.

4.1. Optimising Abstraction

This section focuses on optimisation of the Menu-game construction. We point out things to consider when using a SMT-solver to enumerate transition constraint solutions, elaborate on how previous computations can be reused and state where unnecessary computation can be avoided.

4.1.1. Asserting Variable Ranges

We introduced the abstract transition constraint, see Definition 3.6, as the logical characterisation of a command, such that its solutions correspond to source and destination blocks of the abstract model. However, in the current state, the resulting over-approximation is coarser than necessary. For example consider the command c

$$[a]\ x \neq 1 \rightarrow 1.0 : (x' = x + 1),$$

where $x \in \{0, 1, 2\}$ is a bounded integer and the set of predicates is given by $\mathcal{P} = \{x \text{ is odd}\}$. The respective abstract transition constraint

$$\mathcal{R}_c^{\#} = \exists_{x_0 \in \mathbb{N}} (x_0 \neq 1) \wedge \left(b_1^0 \Leftrightarrow x_0 \text{ is odd}\right) \wedge \left(b_1^1 \Leftrightarrow x_0 + 1 \text{ is odd}\right),$$

has two solutions $\left(b_1^0 = 0, b_1^1 = 1\right)$ and $\left(b_1^0 = 1, b_1^1 = 0\right)$. However, knowing that the domain of x is $\{0, 1, 2\}$, the solution $\left(b_1^0 = 1, b_1^1 = 0\right)$ is clearly an over-approximating one, since the guard is only satisfied by the even values 0 and 2 for x_0. Adding the constraints $0 \leq x$ and $x \leq 2$ corresponding to the variable's domain would have sufficed to eradicate that solution.

It is generally a good idea to add constraints describing a variable's domain to reduce the number of solutions and accordingly the number of MTBDDs to create. Using the notation from Section 2.1.7, the additional constraints are given by

$$\bigwedge_{v \in Var} (\min(dom(v)) \leq v \wedge v \leq \max(dom(v)))\,[Var/Var_0]\,.$$

4.1.2. Exploitiing Incrementality

Modern SMT-solvers support *incremental solving*. This means that the solver maintains a stack of *clauses* instead of a flat representation and stores auxiliary clauses, which it derives during solving, in relation to the clauses they are derived from. The advantage of this becomes clear, when several similar formulas must be solved.

Assume we have a Boolean LIA formula φ_1, which we *push* on the solver's stack and let it enumerate all solutions. If, at some later point, we want to check a formula $\varphi_1 \wedge \varphi_2$ we only have to push φ_2 on the stack. Since the solver most likely derived auxiliary clauses when checking φ_1, the solving of $\varphi_1 \wedge \varphi_2$ will be significantly faster than it would have been if the stack was empty and the checking would have started from scratch. Furthermore, the incrementality allows *popping* of the clauses from the stack, such that only the auxiliary clauses related to the popped clauses are deleted, e.g. checking $\varphi_1 \wedge \varphi_3$ won't have to start from scratch if we simply pop φ_2 and push φ_3.

The emerging question is how to exploit the incrementality of SMT-solving in our use case. Using a single SMT-instance (a single stack) for all abstract transition constraints is clearly not desirable as no auxiliary clauses would be retained, due to the pushing and popping of whole transition constraints. For the same reason, creating a new instance, every time we want to check a formula, is a bad idea.

We settled for creating one SMT-instance for each command or abstract transition constraint, respectively, as this allows the solver to derive command-specific constraints. Most importantly, the stack increases monotonically with every refinement iteration such that previously derived auxiliary clauses can often be reused. For example, when a refinement step derives a new predicate p_{n+1}, we only have to extend the abstract transition constraint for a command c in the following way:

$$\begin{array}{cccccccccc}
 & g_c & & & & & & & & \\
\wedge & \big(b_1^0 \Leftrightarrow p_1\big) & \wedge & \dots & \wedge & \big(b_n^0 \Leftrightarrow p_n\big) & \wedge & \big(b_{n+1}^0 \Leftrightarrow p_{n+1}\big) & \\
\wedge & \big(b_1^1 \Leftrightarrow WP_{E_1}(p_1)\big) & \wedge & \dots & \wedge & \big(b_n^1 \Leftrightarrow WP_{E_1}(p_n)\big) & \wedge & \big(b_{n+1}^1 \Leftrightarrow WP_{E_1}(p_{n+1})\big) & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & , \\
\wedge & \big(b_1^k \Leftrightarrow WP_{E_k}(p_1)\big) & \wedge & \dots & \wedge & \big(b_n^k \Leftrightarrow WP_{E_k}(p_n)\big) & \wedge & \underbrace{\Big(b_{n+1}^k \Leftrightarrow WP_{E_k}(p_{n+1})\Big)}_{\text{new clauses}} &
\end{array}$$

where $deg(c) = k$ and E_i corresponds to the i-th assignment. It is easy to see that this amounts to merely pushing the new clauses

$$\big\{b_{n+1}^0 \Leftrightarrow p_{n+1}\big\} \cup \Big\{b_{n+1}^j \Leftrightarrow WP_{E_j}(p_{n+1}) \;\Big|\; j \in \{1, \dots, k\}\Big\}$$

on the solver's stack.

4.1.3. Relevant Predicates Optimisation

So far, when solving the abstract transition constraint, we have considered that the predicates holding in a block are unrelated to those holding its successor blocks, i.e.

in the abstract transition constraint we have always added all clauses $b_i^0 \Leftrightarrow p_i$ and $b_i^j \Leftrightarrow WP_{E_j}(p_i)$. This, however, is unnecessary since commands are often only related to a subset of all predicates. As a result, not all clauses of the abstract transition constraint have to be considered, since those corresponding to irrelevant predicates will always stay unchanged and respective b_i^j are therefore known beforehand. This allows us to compute several "similar" transitions from a single solution of the adapted transition constraint.

Concept

To get a better idea of which predicates have to be considered in the abstract transition constraint, we consider several base cases of commands.

Case 1 Let us first consider the command *true* $\to 1.0 : (x' = 1)$ with respect to the predicates

$$\mathcal{P} = \{\underbrace{x \geq 0}_{p_1}, \underbrace{x = y}_{p_2}, \underbrace{y \geq 0}_{p_3}\}.$$

Since the assignment only modifies the value of x, it does clearly not affect the validity of the predicate $y \geq 0$, as this one does not refer to x. Accordingly, $y \geq 0$ will hold in a successor if and only if it did so in the source block. Respectively, we refer to the set $\{x \geq 0, x = y\}$ as the *relevant destination predicates.* Dually, the set of *relevant source predicates* is empty, since the value assigned to x does not depend on the predicates that are true in the source block.

Bear in mind that generally the relevant source predicates must also contain all the predicates (indirectly) related to the assignment variable. This is rather difficult to see and thus deferred to the last case.

Figure 4.1 illustrates a solution of the abstract transition constraint, which discards the source block clauses $b_i^0 \Leftrightarrow p_i$ and only considers the destination block clauses $b_1^1 \Leftrightarrow 3 \geq 0$ and $b_2^1 \Leftrightarrow 3 = y$. The irrelevant destination predicates stay unchanged and can safely be multiplied with the solutions of such a reduced transition constraint.

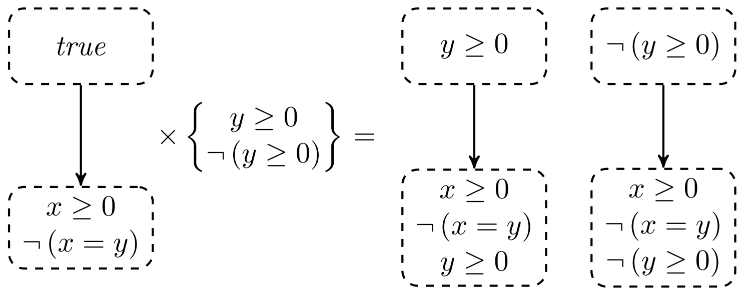


Figure 4.1.: Extending a solution with irrelevant destination predicates

To distinguish from actual blocks, we depicted the solutions with dashed lines. The difference to conventional blocks is that the dashed ones correspond to sets of blocks, where the validity of the "missing" predicates may take any value. For example, the rightmost solution represents, among others, a possible transition from the source block $\{\neg(x \geq 0), x = y, \neg(y \geq 0)\}$ to the destination block $\{x \geq 0, \neg(x = y), \neg(y \geq 0)\}$. Technically, the dashed blocks correspond to what is encoded in the MTBDD representing the solution.

Case 2 Let us now focus on the slightly modified command $\mathit{true} \rightarrow 1.0 : (x' = y - 1)$ but keep the predicates. Since the assignment still only affects the variable x the set of relevant destination predicates is the same as before: $\{x \geq 0, x = y\}$. However, this time the value assigned to x depends on y. As a result, the validity of source block predicates containing y influences the (in)validity of the relevant destination predicates and thus must be considered. Accordingly, the relevant source predicates are given by $\{x = y, y \geq 0\}$.

Figure 4.2 illustrates a solution of the abstract transition constraint, which considers the source block clauses $\{b_2^0 \Leftrightarrow x = y, b_3^0 \Leftrightarrow y \geq 0\}$ and the destination block clauses $\{b_1^1 \Leftrightarrow y - 1 \geq 0, b_2^1 \Leftrightarrow y - 1 = y\}$. As before, the irrelevant destination predicates stay unchanged and are multiplied with the solutions of the reduced transition constraint.

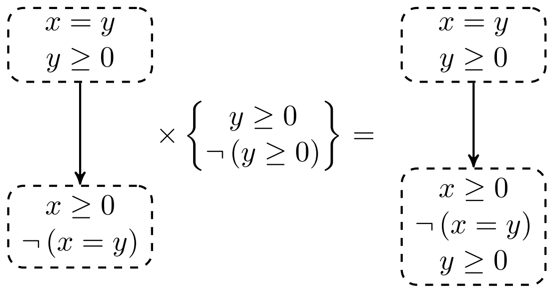


Figure 4.2.: Illustration of a more complex solution

Note that this may introduce invalid source blocks like $B = \{\neg(x \geq 0), x = y, y \geq 0\}$, corresponding to unsatisfiable combinations of predicates. However, as long as there is no valid block leading to an invalid one, they are not reachable and thus discarded eventually.

Case 3 We maintain the set of predicates, but in contrast to the previous cases we focus on the guard. To this end, we extend the command from the first case: $y \geq 0 \rightarrow 1.0 : (x' = 1)$. Assume we would not care for the guard and determine the relevant predicates as in the previous cases, i.e. consider the destination block clauses $\{b_1^1 \Leftrightarrow y - 1 \geq 0, b_2^1 \Leftrightarrow y - 1 = y\}$ but no source block clauses $b_i^0 \Leftrightarrow p_i$ since the value assigned to x is independent of other variables.

Figure 4.3 illustrates why this may introduce invalid successors for valid blocks. Consider the source-destination pairs given by the rightmost solution. Clearly, one solution corresponds to the valid source block $\{x \geq 0, \neg(x = y), \neg(y \geq 0)\}$ having the invalid successor $\{x \geq 0, x = y, \neg(y \geq 0)\}$. To avoid such results, predicates sharing a variable with the guard must be considered as relevant source predicates. In our case, this amounts to considering the predicates $y \geq 0$ and $x = y$ for source blocks, which removes the rightmost (problematic) solution.

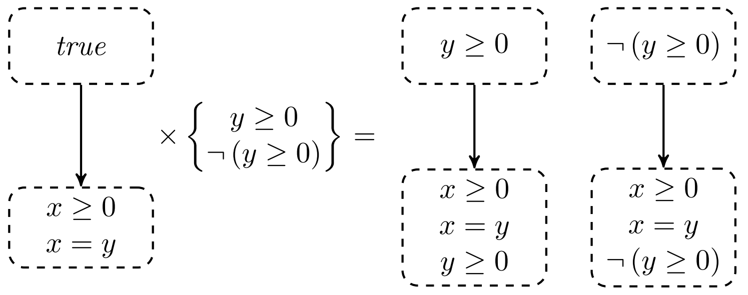


Figure 4.3.: A problematic transition constraint solution

Case 4 – Revising Case 1 There is a special case we have not considered yet, which technically renders our previous examples not quite correct. In the following we illustrate why all predicates – even indirectly – related to the assignment variable must be considered as relevant source predicates.

We come back to the command *true* $\rightarrow 1.0 : (x' = 1)$ from the first case, but this time we look at the predicate set $\mathcal{P} = \{y + 1 = 2, y + x = 2, x = 1\}$. According to the previous cases we, naively, let the set of relevant distribution predicates be $\{y + x = 2, x = 1\}$ and leave the set of relevant source predicates empty.

Figure 4.4 illustrates how failing to consider predicates related to the assignment variable may result in invalid successors.

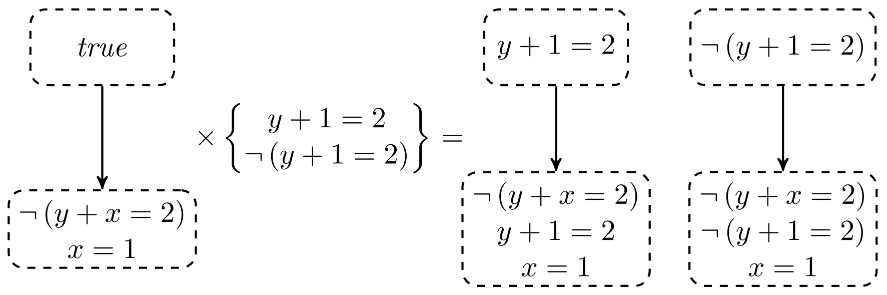


Figure 4.4.: Why predicates related to the assignment variable must be considered

According to the left solution, the valid source block $\{y + 1 = 2, \neg(y + x = 2), \neg(x = 1)\}$ must have a successor $\{y + 1 = 2, \neg(y + x = 2), x = 1\}$, which is clearly invalid as the combination of its predicates is unsatisfiable. This result is attributed to the fact that x is indirectly related to y and thus combinations of predicates containing x and y may constrain the values assigned to x. As a result, we must consider all predicates (even indirectly) related to the assignment variable as relevant source predicates.

Formalism

Our relevant predicates optimisation is inspired by the approach vaguely sketched in [Wachter et al., 2007], which seems to use a single (coarser) set of relevant successor predicates and neglects to consider predicates related to the left-hand side of an assignment as relevant source predicates.

Definition 4.1 (Relevant Predicates)**.** Let c be a command

$$[a]\ g \rightarrow p_1 : \mathit{Var}' = E_1 + \cdots + p_k : \mathit{Var}' = E_k,$$

and $\mathcal{P} = \{p_1, \ldots, p_n\}$ a set of predicates. For an assignment E_j, we define the sets of relevant source and destination predicates, respectively $\mathcal{P}^{src}_{E_j}$ and $\mathcal{P}^{dst}_{E_j}$, as follows:

$\mathcal{P}^{src}_{E_j}$**:** Predicates which indicate the (in)validity of predicates in successor blocks, i.e. predicates which share a variable with the right-hand side of E_j or the guard or are (indirectly) related to an assignment variable.

$\mathcal{P}^{dst}_{E_j}$**:** Predicates whose validity in successor blocks may be affected by E_j, i.e. predicates containing an assignment variable.

Instead of solving the original abstract transition constraint $\mathcal{R}^{\#}_c$, it suffices to solve

$$g \wedge \bigwedge_{p_i \in \mathcal{P}^{src}_c} (b^0_i \Leftrightarrow p_i) \wedge \bigwedge_{j=1}^{k} \bigwedge_{p_i \in \mathcal{P}^{dst}_{E_j}} \Big(b^j_i \Leftrightarrow \mathit{WP}_{E_j}(p_i)\Big),$$

where $\mathcal{P}^{src}_c := \bigcup_{j=1}^{k} \mathcal{P}^{src}_{E_j}$, without losing precision, given that solutions are manually adapted, such that predicates not in a $\mathcal{P}^{dst}_{E_j}$ retain their value. $\square$

4.1.4. Expression Decomposition

As illustrated in the previous section, rarely all predicates must be considered when solving the abstract transition constraint. However, predicates which couple several variables often unnecessarily impair the effectivity.

For example, consider the command $x > 0 \rightarrow 1.0 : (x' = x - 1)$ and the predicates

$$\mathcal{P} = \{x > 1, x = 1 \wedge y > 0, y = 0\}.$$

In this case, the relevant destination predicates correspond to $\{x > 1, x = 1 \wedge y > 0\}$ and, due to the coupling of x and y in $x = 1 \wedge y > 0$, all predicates are relevant source predicates. However, strictly sticking to the procedure for derivation of refinement predicates will often introduce such ones, e.g. a command's guard is often a conjunction of constraints over different variables.

To this end, we propose decomposition of predicates, which recursively splits a predicate at Boolean operators. For example, the predicate $x = 1 \wedge y > 0$ is decomposed into the predicates $x = 1$ and $y > 0$. As a result, instead of $\mathcal{P}$, we can equivalently use the set

$$\{x > 1, x = 1, y > 0, y = 0\},$$

without losing expressivity. In fact, we even get more expressivity, enabling the new set of predicates to distinguish between states where $\neg(x = 1) \wedge y > 0$ or $x = 1 \wedge \neg(y > 0)$ holds. Although the decomposition yields larger sets of predicates and thereby theoretically increases the complexity of the abstract transition constraint, we can decouple many variables and in practice get simpler transition constraints, using the relevant predicates optimisation.

Let us reconsider the command $x > 0 \to 1.0 : (x' = x - 1)$ with respect to the decomposed set of predicates. This time, both the set of relevant source and destination predicates are $\{x > 1, x = 1\}$, which clearly yields a simpler transition constraint.

Note that decomposition of an Boolean expression not necessarily decouples variables, e.g. the decomposition of an expression $x \geq y \vee x + 1 \geq y$ yields the predicates $\{x \geq y, x + 1 \geq y\}$ without decoupling x and y. Nevertheless, expression decomposition always increases the number of predicates to consider. Thus, an even better approach would be to only decompose expressions if the resulting predicates couple strictly less variables.

4.1.5. Unrelated Commands

Often, the introduction of a new refinement predicate only affects a few commands. For example, consider the command $x > 0 \to 1.0 : (x' = x - 1)$ from the previous section and the predicates $\{x > 1, x = 1, y > 0\}$. We know that employing the relevant predicates optimisation we can obtain the solutions by solving the abstract transition constraint with respect to the relevant source and destination predicates $\{x > 1, x = 1\}$. The irrelevant destination predicates, in our case $\{y > 0\}$, simply retain their value.

Now, if we were to add another refinement predicate $y = 0$, this would neither affect the relevant source predicates nor the relevant destination predicates, hence not affecting the solutions of the simplified abstract transition constraint, either. For such commands, *unrelated* to a refinement predicate, the new MTBDD representing the command is obtained from the old MTBDD, extended with the encoding of the new predicate retaining its (in)validity.

To exploit this circumstance, we separately cache the MTBDDs representing each command, instead of solely keeping a monolithic representation $\mathfrak{D}_{sys}$ of the system.

4.1.6. Reachable State Space as Constraint

We know that the addition of a refinement predicate can only split existing blocks of the current state space but never introduce “new” blocks. Accordingly, when solving the abstract transition constraint, we know that solutions of the transition constraint extended with respect to a new refinement predicate can only be extensions of solutions of the previous refinement iteration.

For example, let c be a command whose transition constraint solutions indicate that there are only three blocks $\left(b_1^0, b_2^0, b_3^0\right) \in \{(0,0,1), (0,1,1), (1,0,0)\}$ which satisfy its guard. As a result, on addition of a fourth predicate, we do not have to consider all 2^4

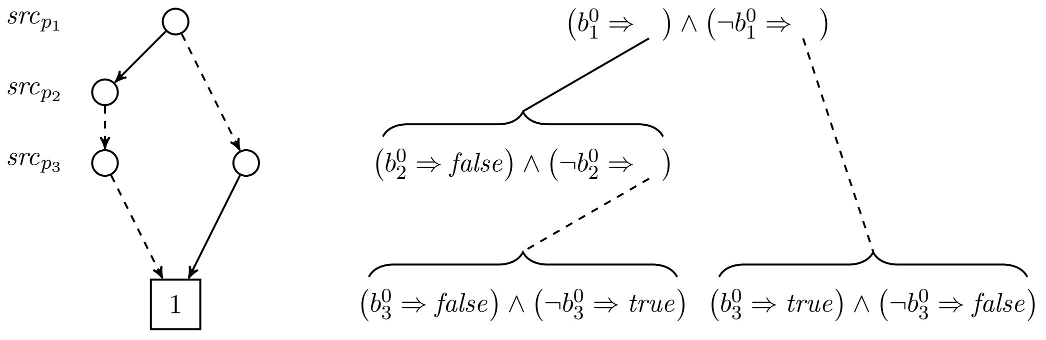
possible valuations for $\left(b_1^0, b_2^0, b_3^0, b_4^0\right)$, but only those extending previous solutions, i.e.

$$(0,0,1) \times \{0,1\}\,,\ (0,1,1) \times \{0,1\} \text{ and } (1,0,0) \times \{0,1\}\,.$$

To derive a constraint, which ensures that only extensions of the current solutions are taken into consideration, we can use the command's MTBDD $\mathfrak{D}_c$, which essentially corresponds to the solutions of $\mathcal{R}_c^{\#}$. The source blocks of a command can be obtained by existentially abstracting from all but the source predicate variables:

$$\mathfrak{D}_c^{src} = \textsc{ExistsAbstract}\left(\left(src_\perp, \underline{act}, \underline{opt}, \underline{upd}, \underline{dst}\right), \mathfrak{D}_c \neq 0_{\mathfrak{D}}\right).$$

The constraint can then be derived from interpreting the BDD $\mathfrak{D}_c^{src}$ as an expression, as illustrated in Figure 4.5.



src_{p_1}
src_{p_2}
src_{p_3}
1
$\left(b_1^0 \Rightarrow \quad\right) \wedge \left(\neg b_1^0 \Rightarrow \quad\right)$
$\left(b_2^0 \Rightarrow \mathit{false}\right) \wedge \left(\neg b_2^0 \Rightarrow \quad\right)$
$\left(b_3^0 \Rightarrow \mathit{false}\right) \wedge \left(\neg b_3^0 \Rightarrow \mathit{true}\right)$
$\left(b_3^0 \Rightarrow \mathit{true}\right) \wedge \left(\neg b_3^0 \Rightarrow \mathit{false}\right)$

(a) $\mathfrak{D}_c^{src}$ (b) Derived constraint

Figure 4.5.: Deriving the source predicate constraint from $\mathfrak{D}_c^{src}$

Note that successor blocks in the form of the destination bit-vectors $\left(b_i^j, \ldots, b_i^j\right), 1 \leq j \leq deg(c)$ can be constrained in a similar way.

4.2. Optimising Value Iteration

While the previous section illustrated optimisations of the abstraction and construction process, this section proposes techniques to speed up the actual analysis process.

4.2.1. Static Pre-computation of Reachability

We have already seen an example of value iteration needing infinitely many steps to reach a fixed point (Example 2.39) and accordingly introduced a respective termination criterion in our symbolical value iteration (Algorithm 4). However, using this criterion we will often get near-1 probability results when they should actually be exactly 1.

In this section, we propose symbolic procedures to compute the sets of blocks having exactly the reachability probabilities 0 and 1. More often than not this fixes probabilities for a significant part of the state space, thereby reducing the number of blocks to consider in the actual value iteration. Though our procedures analyse SGs, they are based on respective pre-processing algorithms for PA [Rutten et al., 2004].

Algorithm 7 Pre-computation Prob1

```
1: procedure Prob1($\mathfrak{D}_{sys}, p_1 max, p_2 max, \mathfrak{D}_G$)
2:     $\mathfrak{D}^{u,01}_{sys} \leftarrow$ Abstract$(+, \underline{upd}, \mathfrak{D}_{sys}) \neq 0_{\mathfrak{D}}$    ▷ Valid, unlabelled distributions
3:     $\mathfrak{D}^{01}_{Opt} \leftarrow$ ExistsAbstract$\left(\underline{dst}, \mathfrak{D}^{u,01}_{sys}\right)$    ▷ Valid options
4:     $\mathfrak{D}^{01}_{Act} \leftarrow$ ExistsAbstract$\left(\underline{opt}, \mathfrak{D}^{01}_{Opt}\right)$    ▷ Valid actions
5:     $\mathfrak{D}^{01}_{V_1} \leftarrow$ ExistsAbstract$\left(\underline{act}, \mathfrak{D}^{01}_{Act}\right)$    ▷ Valid player 1 vertices
6:     $\mathfrak{D}^{src}_{maybe} \leftarrow \mathfrak{D}^{01}_{V_1}$    ▷ Initially all blocks may be “yes”
7:     $maybeDone \leftarrow false$
8:     while $\neg maybeDone$ do    ▷ Outer fixed point
9:         $\mathfrak{D}^{dst}_{maybe} \leftarrow$ ReplaceVar$\left(\underline{src}, \underline{dst}, \mathfrak{D}^{src}_{maybe}\right)$
10:        $\mathfrak{D}^{src}_{yes} \leftarrow$ ReplaceVar$(\underline{src}, \underline{dst}, \mathfrak{D}_G)$    ▷ Initially, only goal blocks seem “yes”
11:        $yesDone \leftarrow false$
12:        while $\neg yesDone$ do    ▷ Inner fixed point
13:            $\mathfrak{D}^{dst}_{yes} \leftarrow$ ReplaceVar$\left(\underline{src}, \underline{dst}, \mathfrak{D}^{src}_{yes}\right)$
14:            $\mathfrak{D}_{Opt,\forall maybe} \leftarrow$ UniversalAbstract$\left(\underline{dst}, \mathfrak{D}^{u,01}_{sys} \Rightarrow \mathfrak{D}^{dst}_{maybe}\right)$
15:            $\mathfrak{D}_{Opt,\exists yes} \leftarrow$ ExistsAbstract$\left(\underline{dst}, \mathfrak{D}^{u,01}_{sys} \cdot \mathfrak{D}^{dst}_{yes}\right)$
16:            $\mathfrak{D}^{new}_{yes} \leftarrow \mathfrak{D}_{Opt,\forall maybe} \cdot \mathfrak{D}_{Opt,\exists yes}$    ▷ Options satisfying both conditions
17:
18:            if $p_2 max$ then    ▷ Determine actions realising 1-reachability
19:                $\mathfrak{D}^{new}_{yes} \leftarrow$ ExistsAbstract$\left(\underline{opt}, \mathfrak{D}^{new}_{yes}\right)$
20:            else
21:                $\mathfrak{D}^{new}_{yes} \leftarrow$ UniversalAbstract$\left(\underline{opt}, \mathfrak{D}^{new}_{yes} \vee \neg\mathfrak{D}^{01}_{Opt}\right) \cdot \mathfrak{D}^{01}_{Act}$
22:            end if
23:
24:            if $p_1 max$ then    ▷ Determine blocks realising 1-reachability
25:                $\mathfrak{D}^{new}_{yes} \leftarrow$ ExistsAbstract$\left(\underline{act}, \mathfrak{D}^{new}_{yes}\right)$
26:            else
27:                $\mathfrak{D}^{new}_{yes} \leftarrow$ UniversalAbstract$\left(\underline{act}, \mathfrak{D}^{new}_{yes} \vee \neg\mathfrak{D}^{01}_{Act}\right) \cdot \mathfrak{D}^{01}_{V_1}$
28:            end if
29:
30:            $\mathfrak{D}^{new}_{yes} \leftarrow \mathfrak{D}^{new}_{yes} \vee$ ReplaceVar$(\underline{src}, \underline{dst}, \mathfrak{D}_G)$    ▷ Goal is always “yes”
31:            if $f_{\mathfrak{D}^{new}_{yes}} = f_{\mathfrak{D}^{src}_{yes}}$ then    ▷ Inner fixed point reached?
32:                $yesDone \leftarrow true$
33:            end if
34:            $\mathfrak{D}^{src}_{yes} \leftarrow \mathfrak{D}^{new}_{yes}$
35:        end while
36:        if $f_{\mathfrak{D}^{src}_{maybe}} = f_{\mathfrak{D}^{src}_{yes}}$ then    ▷ Outer fixed point reached?
37:            $maybeDone \leftarrow true$
38:        end if
39:        $\mathfrak{D}^{src}_{maybe} \leftarrow \mathfrak{D}^{src}_{yes}$
40:    end while
41:    return $\mathfrak{D}^{src}_{maybe}$
42: end procedure
```

Algorithm 7 describes a procedure for computing the blocks with reachability 1, given a set of goal blocks and the players' objectives. In the following, we firstly explain its workings and then conclude with an example.

The general idea is to consider two sets of blocks: the ones of which we know that they realise the reachability probability 1 and the ones for which we don't know yet whether they do – respectively represented by $\mathfrak{D}^{src}_{yes}$ and $\mathfrak{D}^{src}_{maybe}$. Accordingly, we only consider those options leading to distributions which both stay in the "maybe"-blocks and at least one destination is in the "yes" set. To this end, we employ a nested fixed point iteration, where the inner iteration adds blocks which reach an "yes" block from $\mathfrak{D}^{src}_{yes}$ with probability 1, while the outer one removes blocks from $\mathfrak{D}^{src}_{maybe}$ which turn out to realise a reachability probability smaller than 1.

To begin with, we compute BDDs representing valid parts of the Menu-game represented by $\mathfrak{D}_{sys}$ and initialise the "maybe"-set to all blocks of the system (lines 2-7). Since, by definition, the reachability of a goal block is 1, we start the inner iteration with the "yes" set of goal blocks.

As mentioned above, the inner iteration only considers options leading to distributions, which do not leave the "maybe"-blocks and at least one destination is "yes". Note that, for the first condition, we cannot simply universally abstract $\mathfrak{D}^{u,01}_{sys}$ from the destination variables since this would also take the invalid transitions (mapped to 0) into consideration. Thus, in line 14, we use the implication to ensure that only valid destinations, which can actually be reached with a probability greater than 0, must be in the "maybe"-set. Subsequently, line 15 determines the options satisfying the second condition. The multiplication of both MTBDDs, corresponding to the intersection of the sets of options, gives us the options which may realise a 1-reachability.

Lines 18 to 28 select options and actions corresponding to the players' objectives. Note that when maximising, it suffices for a single choice realising the value 1 to exist, while when minimising, a choice realising reachability 1 will only be chosen if all choices realise the value 1. However, when universally abstracting, we again have to make sure that the invalid choices do not affect the outcome. Also, bear in mind that goal states may not reach other goal states with probability 1, thus not being contained in the new set of "yes"-blocks. As a result, we ensure that the goal blocks stay in "yes" in line 21.

Having determined new seemingly "yes"-blocks $\mathfrak{D}^{src}_{yes}$ we update the old "yes"-set and finish the inner iteration once the set does not change anymore. The blocks found to be in "yes" in the inner iteration are then used to restrict the set of "maybe"-blocks, essentially removing all blocks which do not seem to be "yes". Similar as for the inner iteration, the outer iteration updates the shrinking set of "maybe"-blocks until it stabilises.

Example 4.2. We illustrate the procedure on the Menu-game illustrated in Figure 4.6, letting B_5 be the only goal block. Note that we did not annotate the transition probabilities as their exact values are not relevant for the algorithm. Let us compute the blocks realising the lower bound 1 for maximal probabilistic reachability, i.e. let player 1 maximise and player 2 minimise.

Initially, the "maybe"-set $\mathcal{B}_{maybe} = \{B_0, B_1, B_2, B_3, B_4, B_5\}$ corresponds to all player

1 vertices and the "yes"-set $\mathcal{B}_{yes} = \{B_5\}$ is the goal set. Clearly, the only distributions leading to blocks from $\mathcal{B}_{yes}$ (and staying in $\mathcal{B}_{maybe}$) are $\{\mu_7, \mu_8, \mu_9\}$. However, while all options of the player 2 vertices v_6 and v_7 reach one of these distributions, v_5 does not. The only blocks leading to v_6 and v_7 are B_4 and B_5. Correspondingly, we extend $\mathcal{B}_{yes}$ with both, obtaining $\mathcal{B}_{yes} = \{B_4, B_5\}$. Subsequent steps proceed alike and are summarised in Table 4.1.

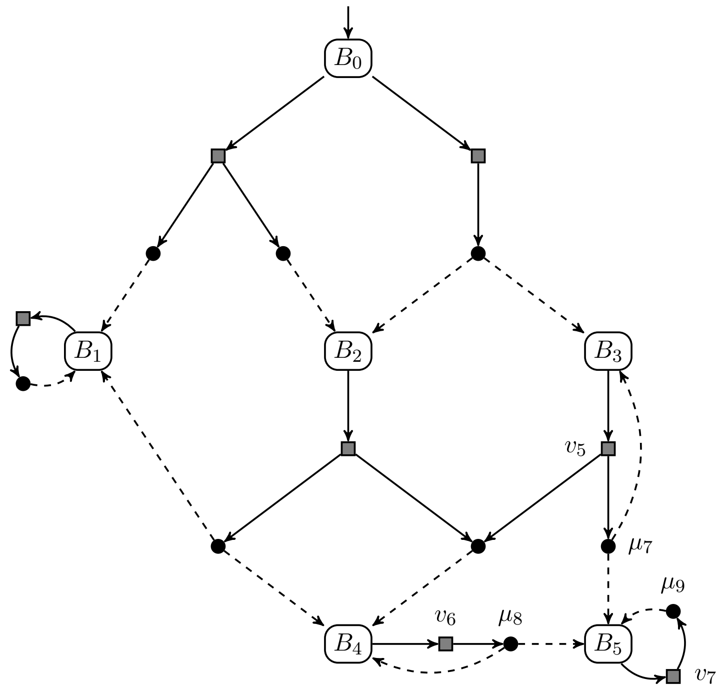


Figure 4.6.: A Menu-game with the goal set $\{B_5\}$

Outer Iteration	$\mathcal{B}_{maybe}$	Inner Iteration	$\mathcal{B}_{yes}$
#0	$B_0, B_1, B_2, B_3, B_4, B_5$	#0	B_5
		#2	B_4, B_5
		#3	B_2, B_3, B_4, B_5
		#4	B_0, B_2, B_3, B_4, B_5
		#5	B_0, B_2, B_3, B_4, B_5
#1	B_0, B_2, B_3, B_4, B_5	#0	B_5
		#1	B_4, B_5
		#2	B_3, B_4, B_5
		#3	B_3, B_4, B_5
#2	B_3, B_4, B_5	#0	B_5
		#1	B_4, B_5
		#2	B_3, B_4, B_5
		#3	B_3, B_4, B_5

Table 4.1.: PROB1 for lower bound of maximal probabilistic reachability

□

Knowing which blocks reach the goal set with probability 1, we incorporate them into the value iteration procedure by treating them like goal blocks, i.e. fix their reachability probability at 1 to avoid waiting for their value to converge. This effectively reduces the number of states to consider and the number of iterations needed.

Dually to PROB1, we propose a symbolic variant of PASS' explicit procedure PROB0, which determines the blocks where the reachability probability is 0. Here, the general idea is to determine "yes" blocks which have a reachability probability greater than 0 and return the complementary set. We don't go into detail though, as it is very similar to PROB1, the main difference being that a single fixed point iteration suffices.

Algorithm 8 Pre-computation PROB0

```
1: procedure PROB0($\mathfrak{D}_{sys}, p_1max, p_2max, \mathfrak{D}_G$)
2:     $\mathfrak{D}^{u,01}_{sys} \leftarrow$ ABSTRACT $(+, \underline{upd}, \mathfrak{D}_{sys}) \neq 0_{\mathfrak{D}}$    ▷ Valid, unlabelled distributions
3:     $\mathfrak{D}^{01}_{Opt} \leftarrow$ EXISTSABSTRACT $(\underline{dst}, \mathfrak{D}^{u,01}_{sys})$    ▷ Valid options
4:     $\mathfrak{D}^{01}_{Act} \leftarrow$ EXISTSABSTRACT $(\underline{opt}, \mathfrak{D}^{01}_{Opt})$    ▷ Valid actions
5:     $\mathfrak{D}^{01}_{V_1} \leftarrow$ EXISTSABSTRACT $(\underline{act}, \mathfrak{D}^{01}_{Act})$    ▷ Valid player 1 vertices
6:     $\mathfrak{D}^{src}_{yes} \leftarrow$ REPLACEVAR $(\underline{src}, \underline{dst}, \mathfrak{D}_G)$    ▷ Initially, only goal blocks seem "yes"
7:     $done \leftarrow false$
8:     while $\neg done$ do    ▷ Fixed point iteration
9:         $\mathfrak{D}^{dst}_{yes} \leftarrow$ REPLACEVAR $(\underline{src}, \underline{dst}, \mathfrak{D}^{src}_{yes})$
10:        $\mathfrak{D}^{new}_{yes} \leftarrow$ EXISTSABSTRACT $(\underline{dst}, \mathfrak{D}^{u,01}_{sys} \cdot \mathfrak{D}^{dst}_{yes})$    ▷ $\mathfrak{D}_{Opt,\exists yes}$ from PROB1
11:        if $p_2max$ then    ▷ Determine actions realising 1-reachability
12:            $\mathfrak{D}^{new}_{yes} \leftarrow$ EXISTSABSTRACT $(\underline{opt}, \mathfrak{D}^{new}_{yes})$
13:        else
14:            $\mathfrak{D}^{new}_{yes} \leftarrow$ UNIVERSALABSTRACT $(\underline{opt}, \mathfrak{D}^{new}_{yes} \vee \neg\mathfrak{D}^{01}_{Opt}) \cdot \mathfrak{D}^{01}_{Act}$
15:        end if
16:        if $p_1max$ then    ▷ Determine blocks realising 1-reachability
17:            $\mathfrak{D}^{new}_{yes} \leftarrow$ EXISTSABSTRACT $(\underline{act}, \mathfrak{D}^{new}_{yes})$
18:        else
19:            $\mathfrak{D}^{new}_{yes} \leftarrow$ UNIVERSALABSTRACT $(\underline{act}, \mathfrak{D}^{new}_{yes} \vee \neg\mathfrak{D}^{01}_{Act}) \cdot \mathfrak{D}^{01}_{V_1}$
20:        end if
21:        $\mathfrak{D}^{new}_{yes} \leftarrow \mathfrak{D}^{new}_{yes} \vee$ REPLACEVAR $(\underline{src}, \underline{dst}, \mathfrak{D}_G)$    ▷ Goal is always "yes"
22:        if $f_{\mathfrak{D}^{new}_{yes}} = f_{\mathfrak{D}^{src}_{yes}}$ then    ▷ Fixed point reached?
23:            $done \leftarrow true$
24:        end if
25:        $\mathfrak{D}^{src}_{yes} \leftarrow \mathfrak{D}^{new}_{yes}$
26:    end while
27:    return $\mathfrak{D}^{01}_{V_1} \cdot \neg\mathfrak{D}^{src}_{yes}$    ▷ Valid blocks not realising a reachability probability > 0
28: end procedure
```

Note that for the states found by either PROB1 or PROB0, we can also derive respective strategies by (similarly to Algorithm 6) storing the respective minimal and maximal representative choices instead of just abstracting from them. To this end, we make another (inner) iteration of the respective procedure after the fixed point is reached, where we store the representatives.

4.2.2. Reusing Previous Reachability Values

In Section 3.5.3, we illustrated the backward refinement process, which, in every refinement iteration, constructs a new Menu-game and starts the respective value iterations for lower and upper bound from scratch. However, using the pre-computation procedure Prob0, we can reuse the previous reachability results partially and thereby reduce the number of iterations needed for convergence.

Let us begin with Figure 4.7, illustrating why we can't just keep the previous reachability values without further modification. The depicted graph is a fragment of a greater Menu-game. The relevant information here is that, employing value iteration for maximal probabilistic reachability, we determined the bounds $[0, 0.9]$ for B_0. The lower bound is obviously attributed to the self-loop-option while the upper bound is propagated from the lower right successor.

Now, assume that a refinement step does not split B_0 but tightens the bounds of the lower right successor to $[0.2, 0.8]$. If we were to keep the reachability results from the previous refinement step and initialise the value iteration with the old values, the maximising option in block B_0 would be to take the self-loop, since at some point the 0.8 will propagate to the lower right successor and the self-loop will seem to yield the (better) reachability probability 0.9.

The real cause for this behaviour is not exactly the self-loop though, but the existence of a 0-reachability-strategy for the block B_0 – the same problem can occur with greater loops. As a result, when initialising the value iteration with previous reachability results, we may only keep the values for such blocks, which do not have a 0-reachability-strategy, i.e. the blocks not covered by Prob0 with both players minimising.

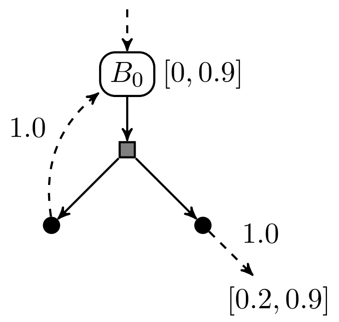


Figure 4.7.: A fragment of a Menu-game prior to refinement

4.2.3. Pivot-picking Policies

A crucial step of the backward refinement procedure is the choice of a pivot block, since picking the "right" blocks results in fewer refinement iterations. Although Section 3.5.5 illustrated how to determine the set of pivot blocks, it is not clear yet which one to actually use to derive refinement predicates. To this end, we propose three policies for picking pivot blocks.

Random Picking the "best" block is hard, as the choice heavily depends on the semantics of the (never built) concrete state space and the goal predicates. Thus picking a pivot block randomly doesn't seem so unreasonable.

Maximal deviation By definition, pivot blocks introduce imprecision. Thus, aiming to reduce imprecision, the maximal deviation policy picks a pivot state where the reachability bounds differ the most.

Nearest maximal deviation While the maximal deviation policy may pick a block far away from the initial block, usually unlikely to be visited, this policy performs a breadth-first search starting in the initial block and picks the nearest pivot block it encounters. If several blocks are found at the same depth, the maximal deviation policy is employed to resolve the choice. Here, the main idea is that nearer blocks are more likely to be visited and thus will more likely improve the precision.

4.2.4. Strategy-reachable Pivot Blocks

Not all pivot blocks attribute to the imprecision at the initial block. Tracing the lower and upper bound strategies, we can see where choices deviate and which ones attribute to the bounds at the initial block.

Thus, it is reasonable to only consider pivot states, which can actually be reached with the lower and upper bound strategies, as only these may contribute to the imprecision at the initial block, thereby avoiding refinement of not expedient pivot blocks.

4.2.5. Removing Goal Successors

Focusing on probabilistic reachability, it is irrelevant what happens once a goal block is reached. As a result, we can safely remove all outgoing transitions of goal blocks, potentially cutting off an irrelevant part of the system or at least reducing the system's MTBDD.

5. Symbolical Backward Refinement in Practice

In the previous chapters, we elaborated on a framework for symbolic analysis of probabilistic reachability. In this chapter, we evaluate our prototypical implementation of said approach. We begin with some details about our implementation, which may be helpful for reproducibility of our results. Afterwards, we illustrate the case studies, which will subsequently be used for evaluation. Besides evaluating the abstraction and value iteration, we also compare our implementation to the reference implementation PASS.

5.1. Implementation Details

This section gives insight on some implementation-specific decisions we made. Bear in mind, that these details are relevant to the evaluation results and must be considered for reproducibility.

Variable Ordering In Section 2.3, we have illustrated that the variable ordering has a significant impact on a MTBDD's size. To this end, in contrast to our illustrations, we initially order them as follows:

$$\underline{act} \prec \underline{upd} \prec src_{\bot} \prec dst_{\bot} \prec src_{p_1} \prec dst_{p_1} \prec \cdots \prec src_{p_n} \prec dst_{p_n} \prec \underline{opt}.$$

Note the interleaved ordering of source and destination variables. This heuristic was successfully applied in [Parker, 2002] to represent PA as MTBDDs and thus adapted by us for the Menu-game representation. Using the CUDD[1] library for our MTBDDs we also make use of its dynamic variable reordering when solving the games with our purely symbolic engine – the reordering strategy being *group sift* [Minato, 1993].

Avoiding Trap-successors and Concrete Deadlocks According to the Definition 3.8, a player 2 vertex must have a trap-vertex successor if the action taken to get to this vertex is not enabled for all states subsumed by the respective block. Additionally, self-loops must be introduced for blocks which subsume deadlock states.

Practice has shown, that most of the time the first refinement iterations will yield the commands' guards as refinement predicates, since blocks are usually so coarse that states enabling different sets of actions end up in the same block. Thus, to avoid wasting

[1]`http://vlsi.colorado.edu/~fabio/CUDD/`

resources for such unnecessary computations, we directly add the guards of a program to the initial set of predicates. As a result, blocks contain only states enabling the same set of actions, consequently making trap vertices unreachable. In addition, due to this grouping, all concrete deadlock states end up in the same block. Considering these circumstances, the AddTrap-procedure from Algorithm 2 may be skipped and the FixDeadlocks-procedure simplified to only fix (abstract) deadlock blocks.

5.2. Overview of Case Studies

To study the feasibility of our symbolical approach to backward refinement and the impact of the proposed optimisations, we applied it to several case studies, from the Prism website[2], which can also be found in the appendix.

Crowds Protocol The Crowds Protocol [Reiter and Rubin, 1998] is an anonymity protocol, modelling the routing of a message through a system of N members – the crowd. Instead of simply sending the message directly to the destination, the sender randomly selects a member from the crowd and forwards it to him with probability $1 - p_f$ or delivers the message directly to the destination with probability p_f. The probability for a recipient to be a bad crowd member is p_{bad}.

In contrast to a good member, which, like the sender, either forwards the message to another member or straight to the destination, a bad one does not adhere to the rules, sending the message directly to the designated destination, assuming that the sender of the message was the original sender.

Here, for $N = 5$, our property of interest is the minimal probability for a bad member to observe a message from the first crowd member. It is known that for the given parameter the resulting probability is 0.33. Note that the respective model is technically a DTMC, since no non-determinism is involved, but can be interpreted as a PA. Accordingly, the minimal and maximal probabilistic reachability coincide. For reference, the concrete model has 8607 states and checking it with Prism takes 830ms.

Randomised Consensus Shared Coin Protocol This case study models the shared coin protocol [Aspnes and Herlihy, 1990], where N processes return a preference for either 1 or 2. The consensus is realised by a shared counter, initially is set to 0, which is incremented or decremented by one, depending on the result of a process' coin toss. You can think of it as a collective random walk parametrised by the number of processes. The coin tossing continues until the shared counter's value leaves the interval $[-N \cdot K, N \cdot K]$, where $K > 1$ is another parameter of the model, and respectively determines the overall preference.

Here, for $N = 2, K = 2$, our property of interest is the minimal probability for the protocol to terminate with all processes giving the preference 1. It is known that this probability is equal to 0.38. Note that this model corresponds to a PA since it introduces

[2]`http://www.prismmodelchecker.org/casestudies/`

non-determinism by allowing several processes to toss their coins simultaneously. It has 272 states and checking it with PRISM takes 810ms.

Asynchronous Leader Election Protocol This case study models the leader election protocol of [Itai and Rodeh, 1990], where N processors on an asynchronous ring topology elect one amongst them as leader.

Initially all processors are active and perform the following sequence of instructions until they become inactive and only relay messages:

1. Toss a coin, and depending on the result, send either 0 or 1 to next processor.
2. Become inactive if 0 was sent but 1 is received from the preceding processor.
3. Send a counter around the ring to check whether other processors are still active. Start all over, if it is the case, otherwise become leader.

Here, for $N = 4$, we are interested in verifying that the minimal probability for the protocol to actually elect a processor is 1 – we know it is. Note that the non-determinism of this model is attributed to the asynchonicity of the communication channel, i.e. no guarantee on the proper ordering of the received messages is given. The concrete semantics has 3172 states and checking it with PRISM takes 1010ms.

Wireless LAN The last case study [Kwiatkowska et al., 2002] models the IEEE 802.11 Wireless LAN Collision Avoidance mechanism for two wireless stations. When two stations transmit a message in the same time frame, i.e. their messages collide on the transmission medium, both have to retransmit their message at some later point in time. To reduce the likelihood of another collision, both stations independently determine a waiting time, based on a randomised exponential backoff rule. The model is parametrised by a maximum number of collisions COL to support and the maximal time a transmission may take $TIME\_TRANS\_MAX$.

For this model, we are interested in the maximal probability for two collisions occur, if $COL = 3$ and $TIME\_TRANS\_MAX = 10$. The corresponding PA has 9003 states and checking it with PRISM takes 940ms.

5.3. Evaluation

We integrated our prototypical implementation of the symbolic backward refinement procedure in the Stochastic Reward Model Checker (STORM), developed by the MOVES-group[3] at RWTH University. This saved us writing a PRISM-language parser, and also allowed us to transform the symbolical representation of a Menu-game into an explicit one.

Based on the introduced case studies, we now evaluate our procedure and the impact of the proposed optimisations on the run time. Since the abstraction and probabilistic

[3]`http://moves.rwth-aachen.de/`

reachability analysis parts are independent we evaluate them separately. However, to justify the symbolic approach, we also compare the memory consumption of both the symbolic and the explicit analysis, since the main reason for using MTBDDs is not the speed of MTBDD-operations but their space-efficient storage scheme. Subsequently, we also evaluate the effect of our refinement policies on the number of refinement iterations and compare the performance of our approach to that of PASS.

The experiments were carried out on an Intel Core i7-4770 processor (3,4 GHz) with 8 GB RAM, however, single-threaded and allowed to use at most 2 GB memory. The solving of abstract transition constraints was accomplished by employing the Z3 SMT-solver [de Moura and Bjørner, 2008] in version 4.3.2. Bear in mind that all of our time measurements correspond to a wall clock and not to the pure CPU time.

5.3.1. Abstraction and Construction

The relevant information for evaluation of the abstraction and construction part is the time spent to solve all abstract transition constraints and construct the reachable part of the corresponding Menu-game. To this end, we compare how the abstraction optimisations from Section 4.1 impact the performance. We denote the following (incremental) setups in our diagrams:

Normal This corresponds to incrementally solving the original abstract transition constraint as illustrated in Section 4.1.2.

Range This setup extends the transition constraints of "Normal" with assertions of the variables' ranges (see Section 4.1.1).

Relevant In addition to the above optimisations, we now only consider relevant predicates in the transition constraint (see Section 4.1.3).

Decomp This is the same as "Relevant", however, with decomposition of predicates (see Seection 4.1.4). Note that the decomposition increases the number of predicates (and expressiveness), such that the number of refinement iterations is generally lower than in the upper cases.

Unrelated This setup also avoids solving transition constraints for unrelated commands (see Section 4.1.5).

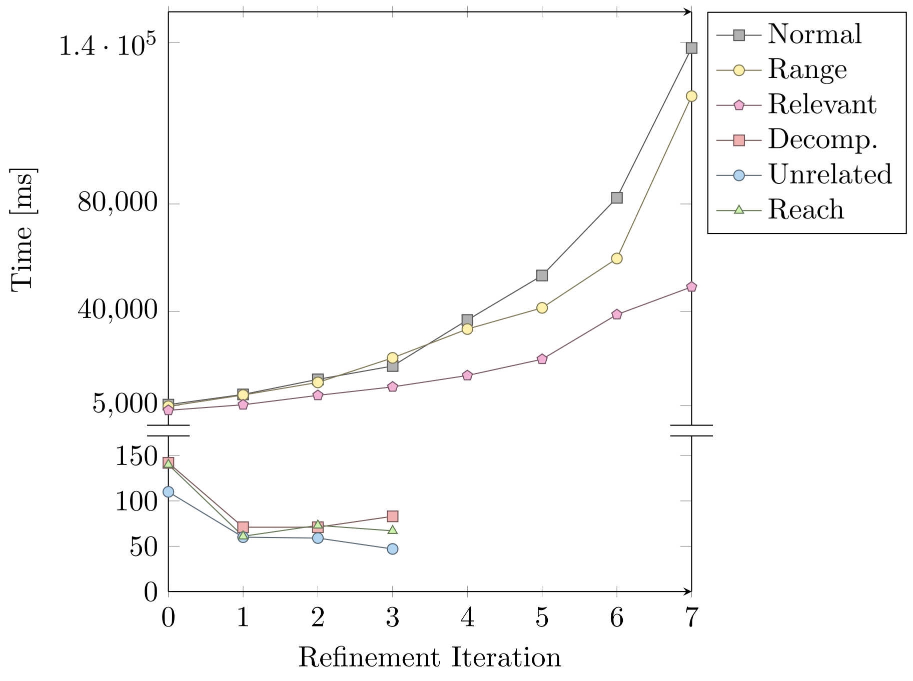
Reach This one has all the above optimisations but also adds the reachable state space constraint as seen in Section 4.1.6.

Besides the comparison of run times for the different setups, we also highlight the ratio of time spent on SMT-solving, MTBDD operations and computation of the reachable state space for the fastest setup, which will turn out to always be "Unrelated".

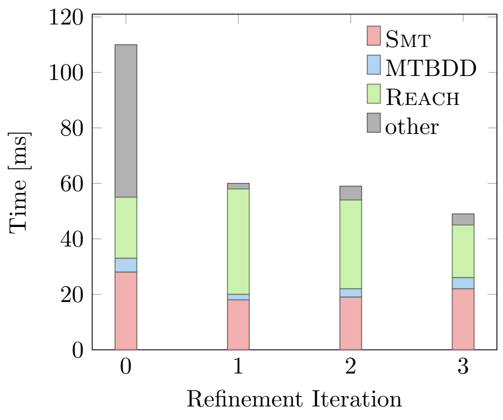
Crowds Protocol

Figure 5.1 depicts our measurements for the Crowds Protocol case study. It is easy to see that the decomposition of predicates is necessary to meet the increasing complexity of the transition constraints, since otherwise the SMT-solving time grows exponentially with the size of the predicate set.

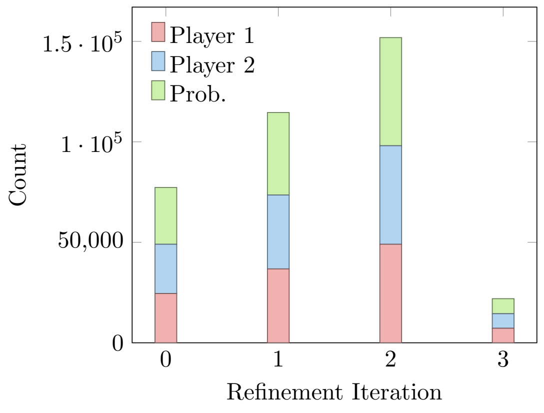
As expected, avoiding solving transition constraints of unrelated commands slightly increases the performance. However, at first glance unexpectedly, "Reach" performs worse. This is attributed to the effort needed to translate a BDD to a constraint.



(a) Menu-game construction time



(b) Detailed abstraction time



(c) State space evolution

Figure 5.1.: Crowds Protocol – abstraction measurements

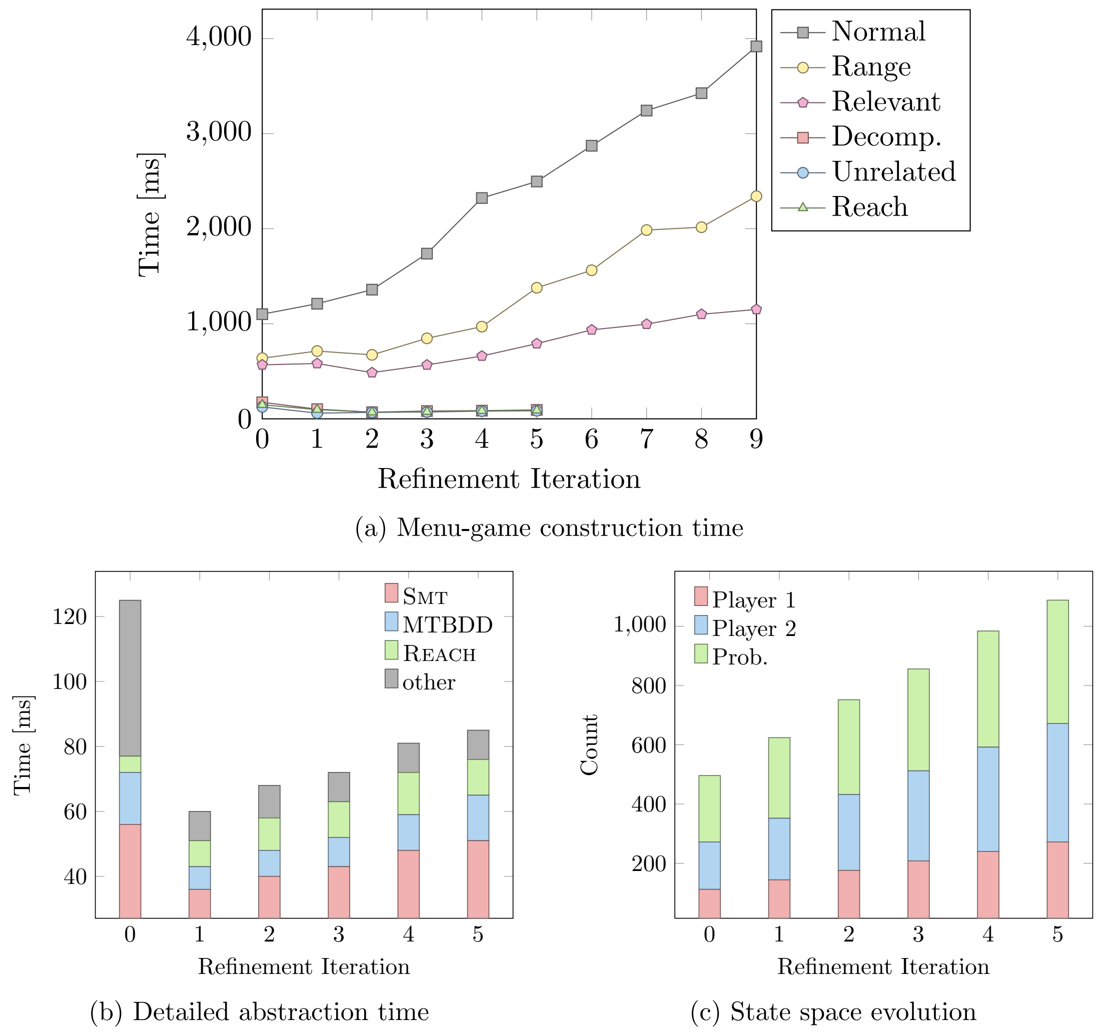
Overall, “Unrelated” performs best. Figure 5.1b indicates that for a model of this size, the analysis of the reachable state space takes even longer than solving the transition constraints and the time spent on constructing the game’s MTBDD is practically negligible. Despite the growing number of predicates the abstraction times hardly differ.

Note that the initial construction (refinement iteration zero) also comprises the predicate decomposition, partitioning of related variables, SMT-solver instantiation and initial creation of all necessary objects, which explains the big ratio spent on “other”.

The state space grows linearly until, in step 2, a refinement predicate is found, constraining the reachable space significantly. Keep in mind that the number of player 1 and player 2 vertices is equal, since the model is actually a DTMC.

Consensus Protocol

Figure 5.1 illustrates our measurements for the Consensus Protocol case study. Similar to “Crowds”, the first three setups scale badly, while the others perform equally good.



(a) Menu-game construction time

(b) Detailed abstraction time

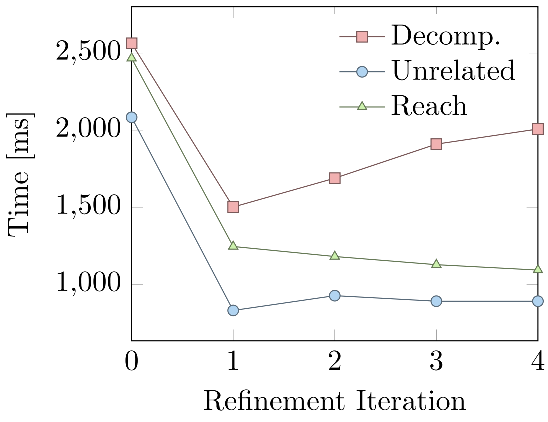
(c) State space evolution

Figure 5.2.: Consensus Protocol – abstraction measurements

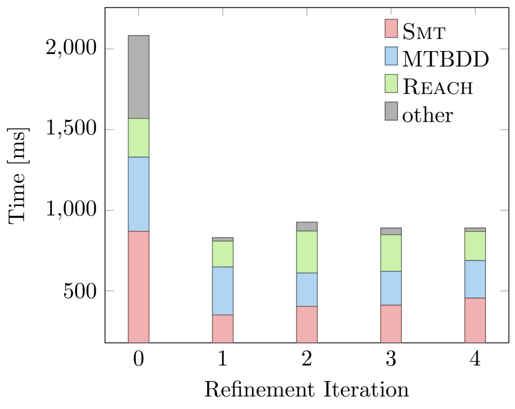
In contrast to “Crowds”, most of the time is spent on Smt-solving and MTBDD construction. We observe that both the abstraction time and state space grow linearly as no restricting predicate is found.

Leader Election Protocol

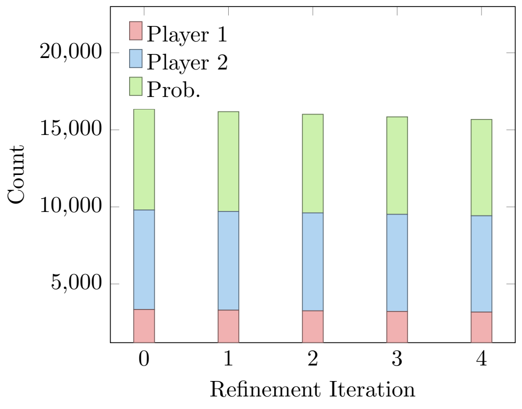
In contrast to the previous systems, the “Leader Election” is so complex that the first three setups do not even manage to solve the transition constraints in a reasonable time frame (30min) and are thus omitted in the following.



(a) Menu-game construction time



(b) Detailed abstraction time



(c) State space evolution

Figure 5.3.: Leader Election Protocol – abstraction measurements

Figure 5.3 illustrates how avoiding solving of transition constraints for unrelated commands outperforms the simple predicate decomposition. While “Decomp” grows linearly, “Unrelated” takes about constant time after the initial construction.

A similar effect can be observed for the ratio of sub procedures. After the initial construction, the run time distribution hardly changes – the Smt ratio increases slightly.

The refinement predicates seem to have not much of an effect on the state space which shrinks only slowly.

Wireless LAN Protocol

Similar to “Leader Election”, this is a rather complex model, where the first three setups turn out to be infeasible, see Figure 5.4. The results are quite similar, too, in the sense that “Unrelated” outperforms the others.

Nevertheless, many refinement iterations are needed to determine a “fine” enough set of predicates. During this time, besides the initial step, most of the time is spent on construction of the Menu-game’s MTBDD and computation of the reachable state space on this very representation.

The big number of refinement steps is actually attributed to the property we want to check. Since we are interested in the maximal probability for two collisions to occur, all possible backoff timings of the system must be considered. Accordingly, each refinement iteration only adds another predicate to distinguish backoff timings of both stations s, i.e. $\mathit{backoff}_s = i$ with $s \in 1, 2, i \in \{0, \ldots, 15\}$. Without the derivation of some restricting predicates, discarding spuriously reachable parts of the state space, such growth of the state space is unavoidable.

In contrast to the sightly smaller “Leader Election”, we observe that the growth of the state space is actually exponential. This is not surprising, considering that the added predicates enable all combinations of backoff timings.

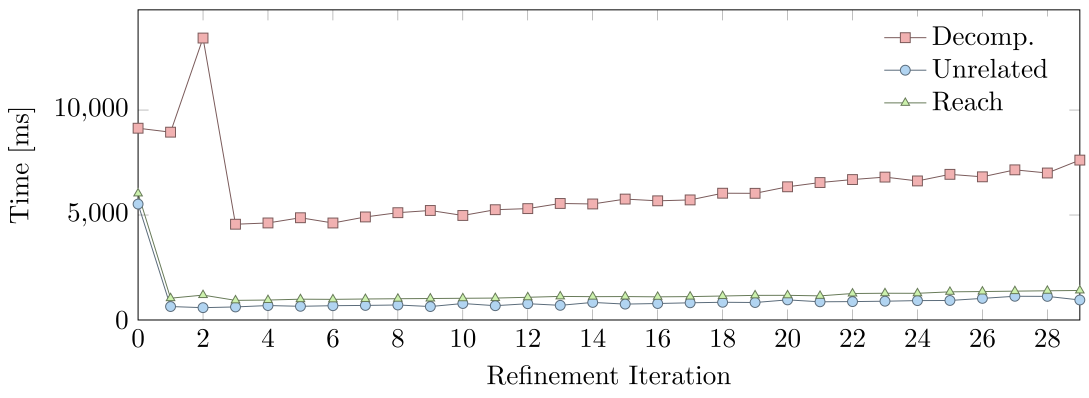
Conclusion

The most outstanding observation is the huge performance gap between “Relevant” and “Decomp”. Up to that point (“Relevant”) the abstraction time grows exponentially with the refinement count, but our experiments indicate it turning linear (almost constant) with “Decomp”. This points out that such an SMT-based abstraction scales badly if many variables are tightly coupled, i.e. if guards or assignments contain arithmetic expressions over many different variables. Luckily, in practice this is rarely the case.

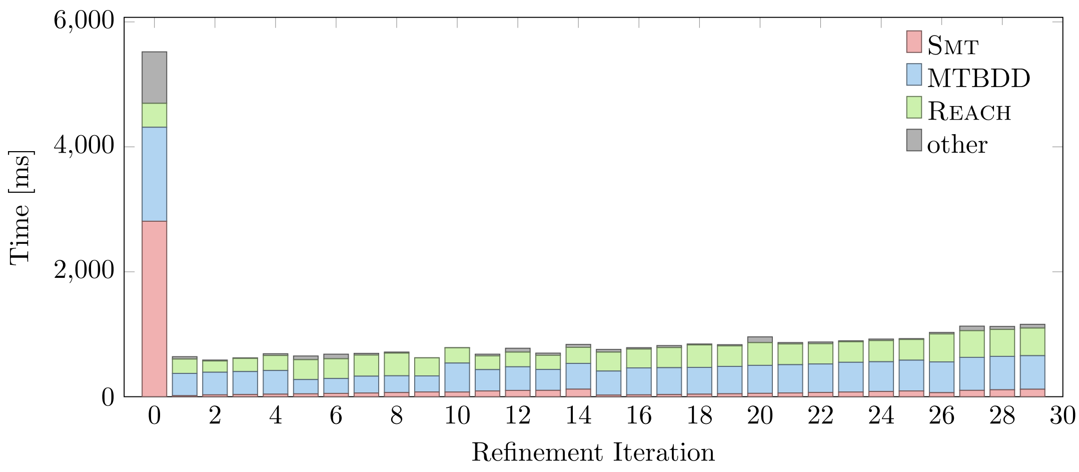
Furthermore, “Reach” turns out to not be an optimisation after all, which is essentially attributed to the effort of translating the BDD to a constraint but also due to our incremental use of the SMT-solver instances, which store auxiliary information and obviate big parts of the BDD.

Considering that the time spent on SMT-solving drops significantly after the initial construction and stays low, it seems that the incremental usage of SMT-solvers in combination with the “unrelated commands” optimisation are (besides predicate decomposition) key to a scalable abstraction. For example, in Figure 5.4, the SMT-usage hardly changes or affects the overall run time, despite the state space growing exponentially.

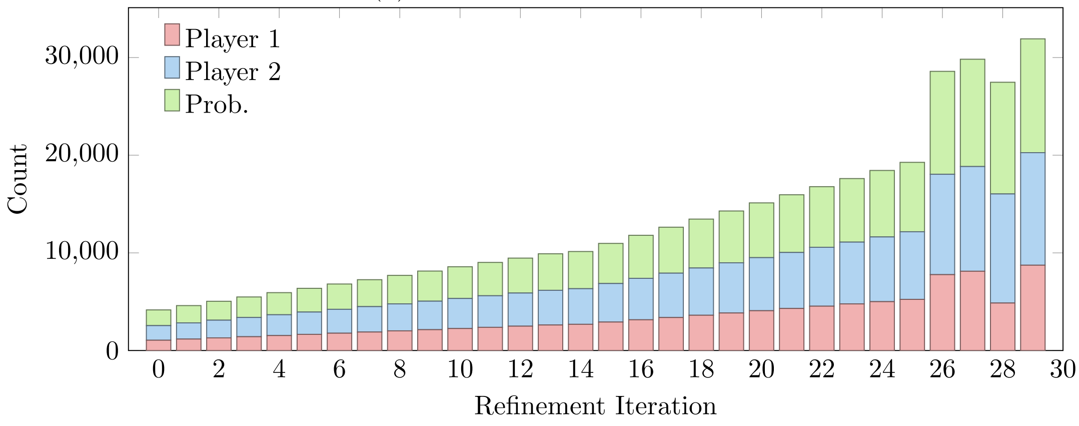
Unfortunately, our results also indicate that for most of our case studies (especially the more complex ones) the operations on MTBDDs become the bottleneck. This can be best seen in Figure 5.4b where about 80%-90% of the run time is spent on constructing the system’s MTBDD or computing the reachable state space. However, since the absolute time spent on these operations is rather small, it seems to be worthwhile to invest more time in finding ways to reduce the number of refinement iterations.



(a) Menu-game construction time



(b) Detailed abstraction time



(c) State space evolution

Figure 5.4.: Wireless LAN Protocol – abstraction measurements

When it comes to comparing the state spaces sizes of the Menu-games and the PA they over-approximate, it turns out that the number of player 1 vertices is indeed never greater than the number of concrete states in the respective PA. However, to be fair, the number of all players' vertices is clearly higher for the chosen case studies.

While this seems discouraging at first, we stress that the backward refinement procedure is clearly tailored towards huge or infinite systems. We, however, used case studies featuring (relatively) small state spaces, to allow for a reasonable visualisation of the measurements. In fact, as expected, Menu-games significantly outperform the analysis of concrete semantics (PA) for huge models. For example, the WLAN case study, for $COL = 20$ and $TIME\_TRANS\_MAX = 10000$, corresponds to a PA with 8690343 states, while its Menu-game only has 31914 vertices overall. It takes PRISM 11:38 minutes to construct the respective PA, while it takes STORM only 2:14 minutes for the complete symbolic backward refinement procedure.

5.3.2. Probabilistic Reachability Analysis

The most relevant information for the actual analysis is the time it takes. Analogous to the evaluation of the abstraction process, we evaluate the performance of the following setups:

Normal This corresponds to first pre-computing blocks with reachability zero or one, then incorporating these results in the value iteration. We consider this "Normal" since the pre-computations are necessary if we want to verify that a probability is exactly 1.

Reuse Res. In addition to the "Normal" case, we also reuse the reachability values of the previous refinement iteration (see Section 4.2.2).

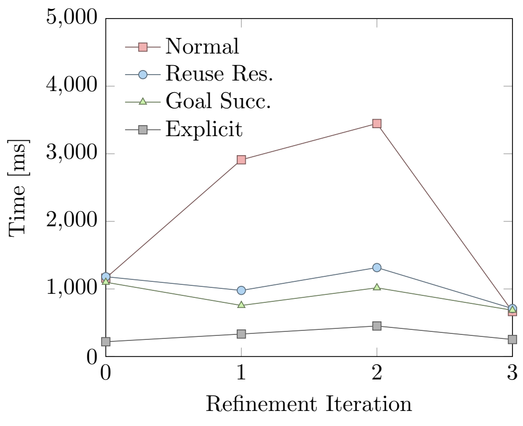
Goal Succ. Building on top of the above optimisations, this one also removes outgoing transitions of goal blocks (see Section 4.2.5).

Explicit This is a special case, where we still pre-compute the blocks with reachability zero or one symbolically, but then build an explicit representation of the system. The value iteration is then carried out explicitly for those states not already covered by the pre-computation results.

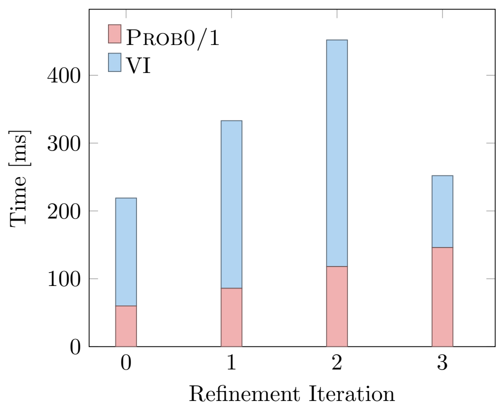
Similar as before, we also highlight the ratio of time spent for the pre-computations and the actual value iteration for the fastest setup – as we will see it will always be the explicit one.

Crowds Protocol

Figure 5.5 depicts our measurements for the Crowds Protocol case study. As to be expected, the initial analysis takes the same effort for all symbolical approaches, while the explicit approach outperforms all. Furthermore, the reuse of previous reachability values significantly improves the “Normal” analysis time. While the middle-setups stay approximately constant over the different refinement iterations, both “Normal” and “Explicit” grow. The small number of refinement iterations does not suffice to properly estimate the growth rate though – it is at least linear.



(a) Menu-game analysis time



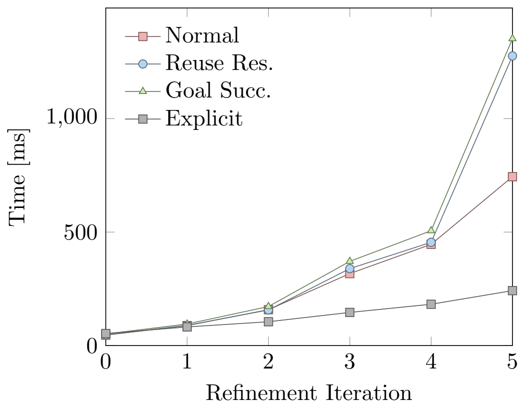
(b) Detailed analysis time

Figure 5.5.: Crowds Protocol – analysis measurements

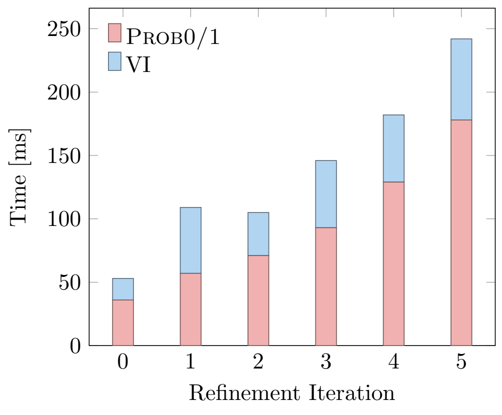
The most striking observation is the similarity of the evolution of the analysis time and the number of the game’s vertices, see Figure 5.1c. Actually, this is reasonable, since the bigger a system is the longer the propagation of analysis values may take.

Consensus Protocol

The measurements for the Consensus Protocol (see Figure 5.6) clearly show exponential growth for the symbolic value iterations. In contrast to “Crowds”, the “Normal” variant is the best symbolic one, but, as before, “Explicit” outperforms the rest and grows linearly with the number of refinement iterations.



(a) Menu-game analysis time



(b) Detailed analysis time

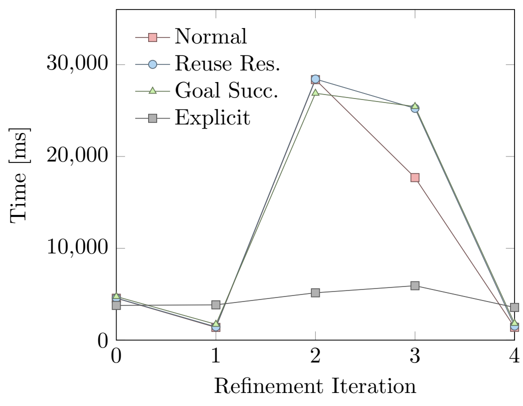
Figure 5.6.: Consensus Protocol – analysis measurements

This time, the ratio of pre-computation and actual value iteration is the other way around – the pre-computations take about 75% of each analysis. However, this appearance is deceiving. Since the respective Menu-game is a lot smaller than the one for "Crowds" the explicit value iterations will obviously take significantly less time. As a result, we can only say that evolution of the time spent on pre-computations corresponds to the growth of the state space, as seen in Figure 5.2c.

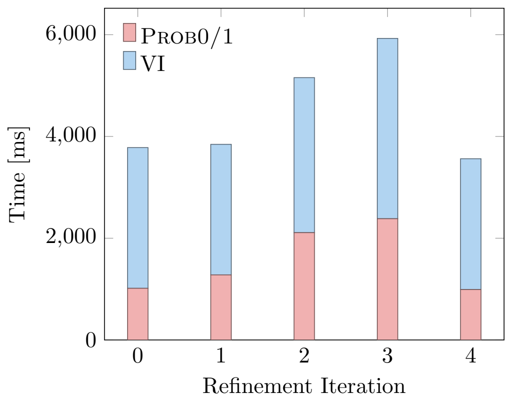
Yet, since the pre-computation is the same for both symbolic and explicit variants, it is easy to see that for a symbolic analysis the pre-computation percentage is a lot smaller, as the value iteration run time grows exponentially.

Leader Election Protocol

The measurements for this case study seem quite similar to those of "Crowds", however, this time all symbolic variants perform approximately equally bad, with "Normal" being the better one among them. As usual, "Explicit" outperforms the rest and grows linearly with the number of refinements.



(a) Menu-game analysis time



(b) Detailed analysis time

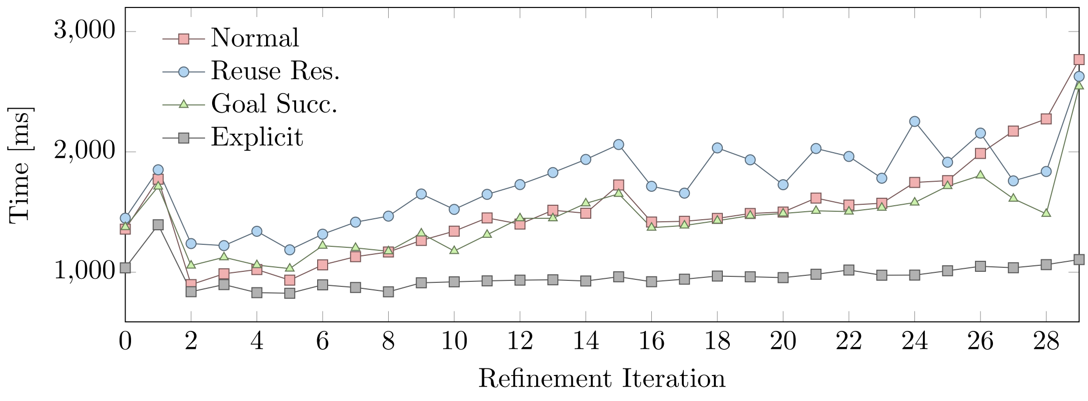
Figure 5.7.: Leader Election Protocol – analysis measurements

For this model, the pre-computation run time grows faster than expected, considering that the size of the state space stays approximately constant. However, knowing that for the property of interest the probability at the initial state must be 1, the reachability for many other states should be 1, too. Hence, there is more information to propagate than usually, which explains the slight growth. Overall, the run time does still seem to be proportional to the evolution of the size of the state space.

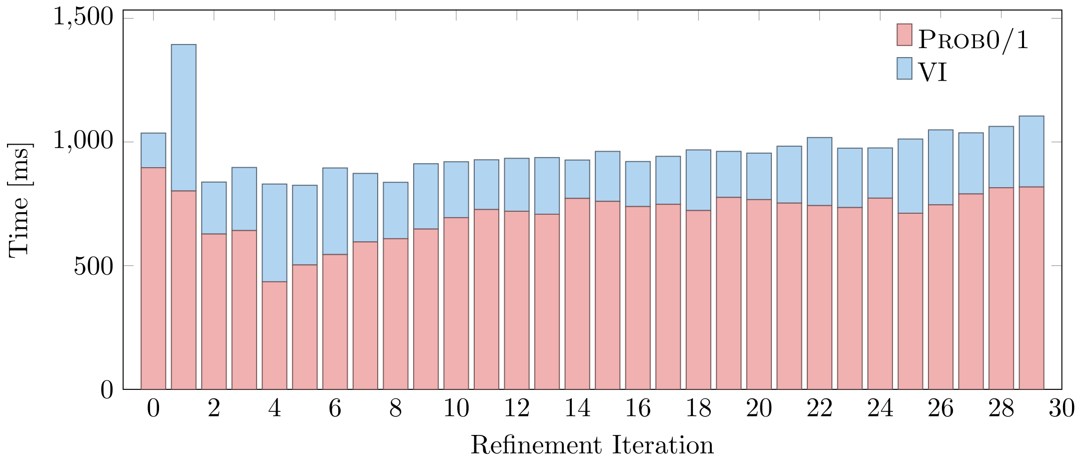
Wireless LAN Protocol

The results of this case study are visualised in Figure 5.8. The analysis times indicate an exponential growth of the run time for the symbolic approaches and linear growth for the outperforming "Explicit" variant.

Surprisingly, the pre-computation run time stays rather expensive at around 700ms over all refinement iterations – no clear trend is observable. It seems like the blocks with reachability 0 or 1 are hardly affected by the refinement predicates, which explains the lack of clear tendency.



(a) Menu-game analysis time



(b) Detailed analysis time

Figure 5.8.: Wireless LAN Protocol – analysis measurements

Conclusion

Overall, the value iteration time of the symbolical analyses seems to grow exponentially while the explicit value iteration and pre-computations scale linearly. To shed some light on this discrepancy, let us reconsider our measurements with respect to the MTBDD storage scheme and some additional information on the probability bounds.

We know that the cost of operations on an MTBDD is closely tied to the symmetries of the function it represents. However, during the value iteration, the game's vertices are assigned many different probabilities, which results in a significantly more irregular MTBDD than the one representing the actual system. Operations and reordering on this MTBDD are costly. This is backed by our findings that the most expensive instruction of ValIterStep (Algorithm 5) is the computation of $\mathfrak{D}_{MV}$ – the matrix-vector multiplication of the system's representation with the current valuation.

Furthermore, this assumption is in accord with our observation that the symbolical pre-computation does not suffer from such an exponential growth, but rather stays linear. This is reasonable since the pre-computations operate on BDDs that abstract from the actual probabilities and therefore only possess two terminal nodes, leveraging the symmetry-exploiting storing scheme.

Another indication arguing that the many different terminal nodes correlate with the bad scaling is that the first significant growth for the run times of symbolic variants in both "Crowds" and "Leader Election" corresponds to a change of the bounds from either 0 or 1 to some non-extremal value, e.g. from 0 to 0.24. Up to that point the computation of $\mathfrak{D}_{MV}$ is rather cheap as multiplication with many extremal entries does hardly affect the MTBDD's symmetry.

However, the different terminal nodes go hand in hand with an increased number of value iteration steps needed for convergence. In fact, it is the combination of many costly operations on an rather irregular MTBDD which cause the exponential growth for the symbolical approach. Since we can hardly avoid the high number of different terminal nodes, it seems worthwhile to research techniques which decrease the number of value iteration steps.

Since explicit representations are not affected by the actual valuations and the pre-computations seem to grow linearly, it makes sense that "Explicit" grows linearly, too.

5.3.3. Symbolic vs. Explicit Memory Usage

When it comes to pure speed, transforming the state space into an explicit representation seems to pay off. However, the actual argument for using symbolical representations is their space-efficient storage scheme. Figure 5.9 visualises the peak memory consumption for both the "Normal" symbolic approach and the one we denoted by "Explicit".

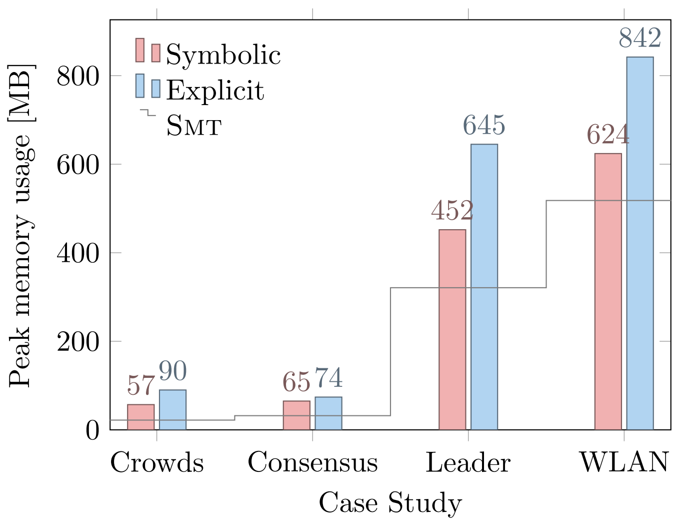


Figure 5.9.: Memory consumption of symbolical and explicit approaches

Note that both approaches use the same Smt-based construction procedure. Thus, to get a better idea of the memory usage, Figure 5.9 also indicates the amount of memory used by Smt-solver instances. At a first glance, the ratio of memory used for solver instances may seem surprisingly high. Remember, though, that our approach always flattens modular programs, which results in high numbers of commands and solver instances, respectively (> 100 for "Leader" and "WLAN").

It is easy to see that the symbolic approach uses significantly less memory than the explicit one. Furthermore, this is backed by the fact that "Explicit" is not a purely explicit approach but profits from the space-efficient symbolical construction of the state space. As a result, the difference to a purely explicit approach should be even bigger.

Hence, we conclude that for bigger models, memory constraints may render an explicit backward refinement impossible, while a symbolical approach may still be successful.

5.3.4. Pivot-picking Policies

In the following, we compare the impact of pivot picking policies on the number of refinement iterations. To this end, we distinguish between the policies RND (Random), MD (Maximal Deviation) and NMD (Nearest Maximal Deviation) proposed in Section 5.3.4. Note that we only consider strategy-reachable pivot states (see Section 4.2.4). Figure 5.10 visualises our measurements, where we additionally plot the refinement iterations needed by the reference tool Pass.

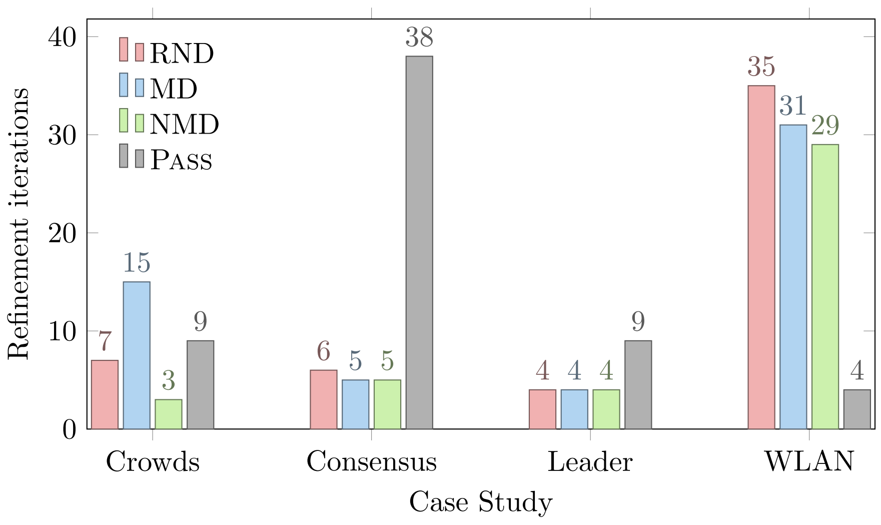


Figure 5.10.: Pivot policies' impact on number of refinements

As expected, picking the nearest pivot block tends to yield the most goal-oriented refinement predicates, resulting in small number of refinement iterations, while Pass tends to make more refinement iterations.

A closer look at Pass unveiled its use of an extension of our maximal deviation policy. Instead of simply picking the pivot state with maximal deviation, Pass also checks whether this pivot state is spurious or actually reachable in the concrete state space. To this end, it encodes the semantics of the most probable path in terms of an LIA formula and, in case of spuriousness, employs *interpolants* [McMillan, 2005] to derive refinement predicates, which make this and similar blocks unreachable in follow up refinement iterations.

Although, this may often increase the number of refinement iterations it may also derive several predicates at once, which otherwise would have taken several refinement iterations, using the "conventional" derivation procedure. The effect of this can best be seen in the WLAN case study.

5.3.5. Storm vs. Pass

A direct comparison of our prototypical implementation to Pass is rather difficult, considering that both use different representations and optimisations. We can compare the total time both implementations need to compute the probabilities for our case studies, though. To this end, we construct our model symbolically, but then distinguish between our best symbolic and explicit analysis variants and Pass.

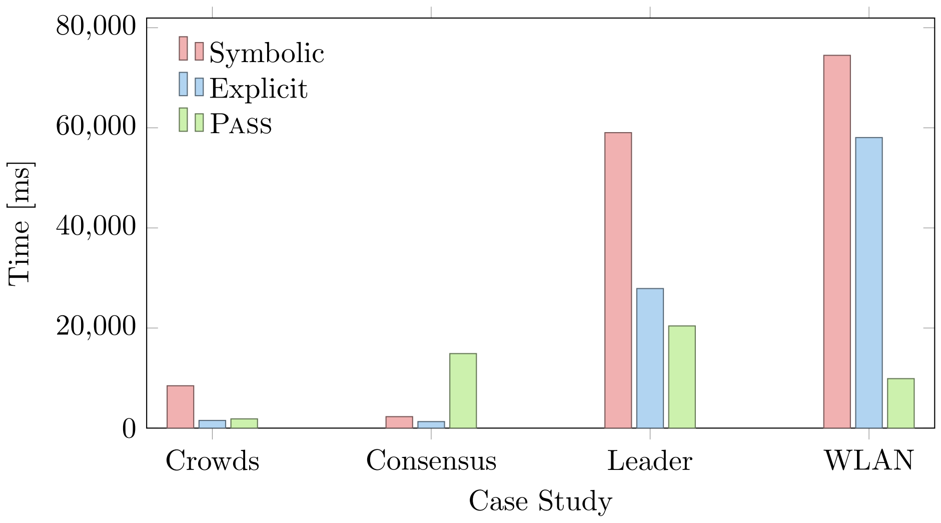


Figure 5.11.: Full reachability analysis run times of Storm and Pass

According to Figure 5.11, the results are mixed. While "Explicit" most of the time performs similar to "Pass" the "Symbolical" performs significantly worse. This is not surprising, considering the above evaluation of analysis times.

However, a clear trend can be observed. The more refinement iterations are involved, the more likely Pass will outperform our implementation. The WLAN case study is an example of such a case. It seems desirable to adapt Pass' predicate derivation procedure to both gain predicates which discard spurious blocks and potentially determine several predicates in a single refinement step.

6. Assume-guarantee Style Extension for Menu-games

Often, probabilistic systems are modelled in terms of several smaller but interacting components. While the state spaces of the single components are rather small, a composed system grows exponentially in the number of its components. To this end, *assume-guarantee* style compositional techniques [Pnueli, 1985] aim to *guarantee* the validity of properties of the composed systems from *assumptions* holding for its components.

For a system of two components, this concept can be captured by the following rule

$$\frac{L_1 \parallel A \models P \qquad L_2 \subseteq A}{L_1 \parallel L_2 \models P}(\text{A}\textsc{Sym}),$$

where L_1 and L_2 the systems components, A is an over-approximation of L_2 and P is some property to be satisfied [Komuravelli et al., 2012]. It suggests, that instead of verifying the property P for the composition $L_1 \parallel L_2$, if suffices to verify it for the composition $L_1 \parallel A$, given that A is an over-approximation of L_2. Note that the semantics of composition, satisfaction and over-approximation depend on the actual types of objects used to model components and properties.

In this chapter, we adapt this notion to Menu-games, such that in future work compositional probabilistic programs may be handled more adequately than just being flattened.

6.1. Composition of Probabilistic Automata

So far we only considered flattened PA, which consist of a single *module*. To allow for concurrent behaviour, we introduce the synchronisation of PA, each modelling a module, via shared action labels. We also introduce the cross product for labelled distributions, which yields a more accessible definition.

Definition 6.1 (Cross Product for Labelled Distributions)**.** Let $\mu_1 : U_1 \times S \to [0,1]$ and $\mu_2 : U_2 \times S \to [0,1]$ be labelled distributions. Their cross product $\mu_1 \times \mu_2 : U_1 \times U_2 \times S \to [0,1]$ is defined as

$$(\mu_1 \times \mu_2)\left((u_1, u_2), s\right) = \mu_1(u_1, s) \cdot \mu_2(u_2, s).$$

□

Definition 6.2 (Parallel Composition of PA [Kwiatkowska et al., 2010])**.** Let $\mathcal{A}_1 = (S_1, Act_1, U_1, \mathbf{P}_1, s_{init,1})$ and $\mathcal{A}_2 = (S_2, Act_2, U_2, \mathbf{P}_2, s_{init,2})$. Their parallel composition is again a PA

$$\mathcal{A}_1 \parallel \mathcal{A}_2 = \left(S_1 \times S_2, Act_1 \cup Act_2, U_1 \times U_2, \mathbf{P}, \left(s_{init,1}, s_{init,2}\right)\right),$$

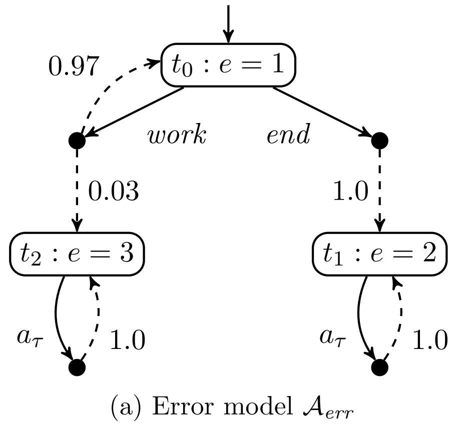
with $(\alpha, \mu_1 \times \mu_2) \in \mathbf{P}\left((s_1, s_2)\right)$ if and only if one of the following holds:

- $(\alpha, \mu_1) \in \mathbf{P}_1(s_1), (\alpha, \mu_2) \in \mathbf{P}_2(s_2)$ where $\alpha \in (Act_1 \cap Act_2) \setminus \{a_\tau\}$
- $(\alpha, \mu_1) \in \mathbf{P}_1(s_1), \mu_2 = 1.0 : (u_\tau, s_2)$ where $\alpha \in (Act_1 \setminus Act_2) \cup \{a_\tau\}$
- $\mu_1 = 1.0 : (u_\tau, s_1), (\alpha, \mu_2) \in \mathbf{P}_2(s_2)$ where $\alpha \in (Act_2 \setminus Act_1) \cup \{a_\tau\}$,

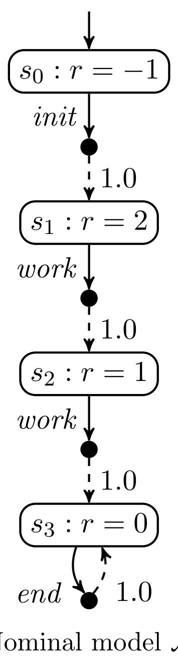
where a_τ acts as *internal*, non-synchronising action. □

Example 6.3. Reconsider our running example from Listing 2.1. It models a system which is initialised, does some work until *run* reaches zero and terminates. However, while working it may break with a certain probability and end up in an error state.

This kind of behaviour could also have been modelled in a compositional way. For example, we could have modelled the error behaviour as one PA and the nominal behaviour as another. Figure 6.1 depicts such a modelling, where we use r instead of *run* and use a variable e to distinguish between error states.



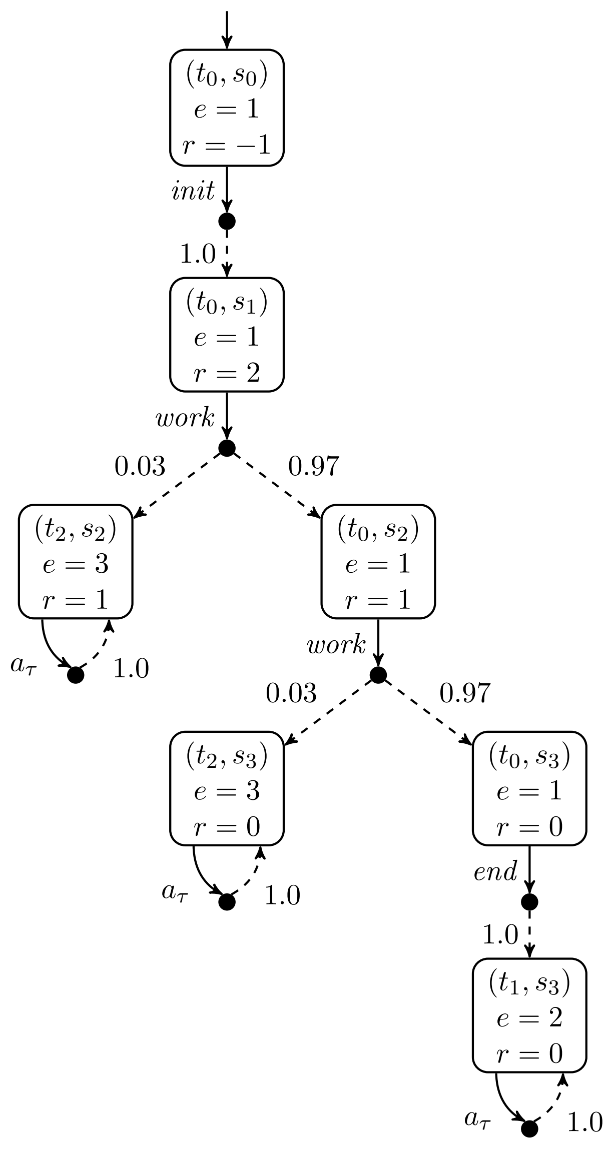
(a) Error model $\mathcal{A}_{err}$



(b) Nominal model $\mathcal{A}_{nom}$

Figure 6.1.: Compositional modelling

Since both the error model and nominal model share the actions *work* and *end*, these can only be taken simultaneously. Accordingly, the error model cannot take the *end* action unless the nominal model is in state s_3. Also, as the composed system is supposed to only break while working, the error state t_2 can only be reached by synchronising over the *work* action. The composed behaviour of these modules is visualised in Figure 6.2 and indeed corresponds to the PA $[\![P_{simple}]\!]$ from Figure 2.6.



(t_0, s_0) $e = 1$ $r = -1$
init
1.0
(t_0, s_1) $e = 1$ $r = 2$
work
0.03
0.97
(t_2, s_2) $e = 3$ $r = 1$
(t_0, s_2) $e = 1$ $r = 1$
a_τ
1.0
work
0.03
0.97
(t_2, s_3) $e = 3$ $r = 0$
(t_0, s_3) $e = 1$ $r = 0$
a_τ
1.0
end
1.0
(t_1, s_3) $e = 2$ $r = 0$
a_τ
1.0

Figure 6.2.: Composition $\mathcal{A}_{err} \parallel \mathcal{A}_{nom}$

□

6.2. Composition of Menu-games

We have seen that the general idea of assume-guarantee style verification is based on reasoning about a composed system $L_1 \parallel L_2$ from a simpler composition $L_1 \parallel A$, where A is an over-approximation of L_2 with respect to the property of interest. In the context of Menu-games a composed system $\mathcal{A}_1 \parallel \mathcal{A}_2$ is given by the parallel composition of PA. We aim to define a parallel composition $\mathcal{A}_1 \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$, analogous to $L_1 \parallel A$, where Q_2 is a partition of the state space of $\mathcal{A}_2$.

However, to keep the parallel composition symmetrical, we define a parallel composition for two Menu-games instead of a PA and a game. For that purpose, we lift the PA $\mathcal{A}_1$ to a game $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1}$. Note that technically S_1 is not a partition but we conveniently use it here to denote the partition $\{\{s\} \mid s \in S_1\}$.

To get an intuitively accessible definition, we aim to make is structurally similar to the composition of PA. Therefore, we first of all provide an alternative definition of Menu-games, which interprets the game's edges as a transition relation δ, similar to the approach in Section 3.2.

Definition 6.4 (Menu-game (Functional)). Let P be a probabilistic program, $[\![P]\!] = (S, Act, U, \mathbf{P}, s_{init})$ its semantics and

$$\widehat{\mathcal{G}}_{[\![P]\!],Q} = ((V, E), (V_1, V_2, V_p), U, v_{init})$$

its Menu-game with respect to a partition Q. An equivalent representation of the Menu-game is given by the tuple

$$(V_1, Act, Opt, U, \delta, v_{init}),$$

with

- the player 2 options are uniquely identified by $Opt := (V_2 \times V_p) \cap E$
- the transition relation $\delta : V_1 \times Act \times Opt \times Dist_U(V_1)$, where

$$(v, \alpha, \beta, \mu) \in \delta \iff (v, (v, \alpha)) \in E \text{ and } \beta = ((v, \alpha), \mu) \in E$$

□

Definition 6.5 (Parallel Composition of Menu-games). Let P_1 and P_2 be probabilistic program modules and

$$\widehat{\mathcal{G}}_{[\![P_1]\!],Q_1} = \big(V_1, Act_1, Opt_1, U_1, \delta_1, v_{init,1}\big) \tag{6.1}$$
$$\widehat{\mathcal{G}}_{[\![P_2]\!],Q_2} = \big(V_2, Act_2, Opt_2, U_2, \delta_2, v_{init,2}\big) \tag{6.2}$$

their Menu-games. The parallel composition is again a Menu-game

$$\widehat{\mathcal{G}}_{[\![P_1]\!],Q_1} \parallel \widehat{\mathcal{G}}_{[\![P_2]\!],Q_2} = \big(V_1 \times V_2, Act_1 \cup Act_2, Opt_1 \times Opt_2, U_1 \times U_2, \delta, \big(v_{init,1}, v_{init,2}\big)\big),$$

where $((v_1, v_2), \alpha, (\beta_1, \beta_2), \mu_1 \times \mu_2) \in \delta$ if and only if one of the following holds:

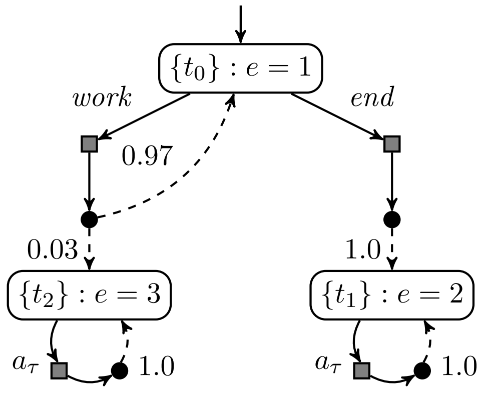
- $(v_1, \alpha, \beta_1, \mu_1) \in \delta_1, (v_2, \alpha, \beta_2, \mu_2) \in \delta_2$ where $\alpha \in (Act_1 \cap Act_2) \setminus \{a_\tau\}$

- $(v_1, \alpha, \beta_1, \mu_1) \in \delta_1, \mu_2 = 1.0 : (u_\tau, v_2), \beta_2 = \beta_1$ where $\alpha \in (Act_1 \setminus Act_2) \cup \{a_\tau\}$

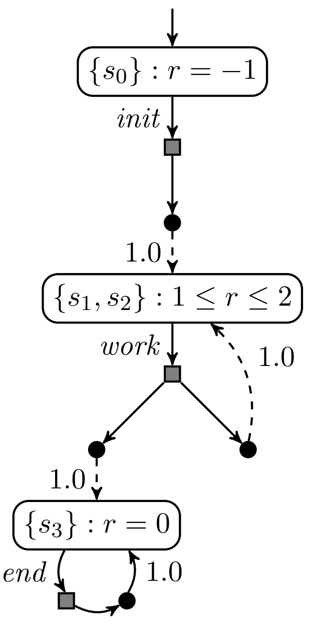
- $\mu_1 = 1.0 : (u_\tau, v_1), \beta_1 = \beta_2, (v_2, \alpha, \beta_2, \mu_2) \in \delta_2$ where $\alpha \in (Act_2 \setminus Act_1) \cup \{a_\tau\}$

and a_τ acts as *internal*, non-synchronising action. □

Example 6.6. Let us assume that the composition $\mathcal{A}_{err} \parallel \mathcal{A}_{nom}$ is too costly to compute and instead try to over-approximate it with Menu-games. Figure 6.3 illustrates both the lifted error model $\widehat{\mathcal{G}}_{\mathcal{A}_{err},S_{err}}$ and the Menu-game $\widehat{\mathcal{G}}_{\mathcal{A}_{nom},Q_{nom}}$ of the nominal model with respect to a partition $Q_{nom} = \{\{s_0\}, \{s_1, s_2\}, \{s_3\}\}$.



(a) Lifted error model



(b) Menu-game of nominal model

Figure 6.3.: Menu-games of components

The parallel composition of Menu-games is conceptually similar to the composition of PA with the main difference, that for Menu-games, synchronising actions lead to a cross product of successor options instead of successor distributions. Figure 6.4 visualises the parallel composition of $\widehat{\mathcal{G}}_{\mathcal{A}_{err},S_{err}}$ and Menu-game $\widehat{\mathcal{G}}_{\mathcal{A}_{nom},Q_{nom}}$.

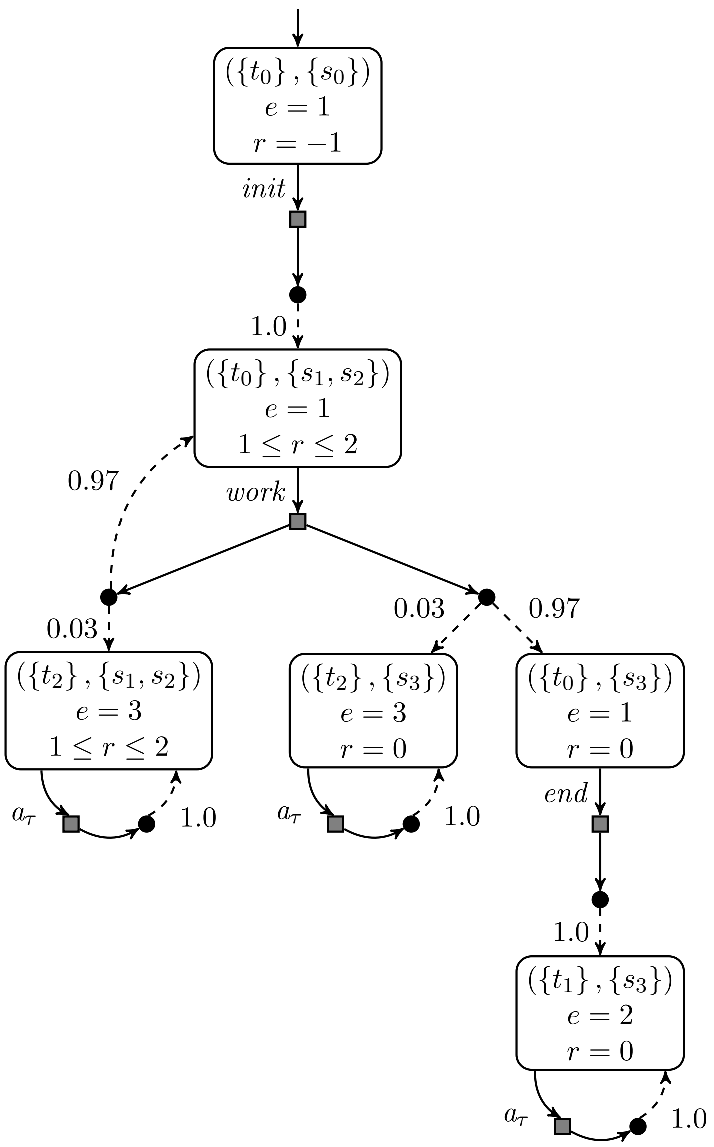


Figure 6.4.: Composition $\widehat{\mathcal{G}}_{\mathcal{A}_{err},S_{err}} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_{nom},Q_{nom}}$

□

6.3. Assume-guarantee Rule

As already mentioned above, we aim to over-approximate a composed system $\mathcal{A}_1 \parallel \mathcal{A}_2$ by a composition $\mathcal{A}_1 \parallel \widehat{\mathcal{G}}_{A_2,Q_2}$ – technically $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{A_2,Q_2}$. In this section, we propose an according assume-guarantee rule and reason about its correctness.

To begin with, consider two Menu-games G_A and G_B, where G_B can *simulate* every strategy pair (σ_1, σ_2) for G_A, i.e. (σ_1, σ_2) is applicable to G_B and induces the same DTMC (up to renaming). Then the reachability probability of a set of goal states in G_A is clearly over-approximated by G_B, since all (and possibly more) behaviour possible in G_A is covered. This is the basic idea behind our adaptation of the ASYM-rule for Menu-games.

Definition 6.7 (Assume-guarantee Rules for Menu-games)**.** Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be PA over the states S_1 and S_2. Then, for a partition Q_2 of S_2, the following rules can be applied:

$$\frac{\underset{\sigma_2}{\forall}\ \underset{\sigma_1}{\sup}\, Pr^{\sigma_1,\sigma_2}_{G_1 \| G_2}\left(\Diamond G^{\#}\right) \in [a,b] \qquad G_1 = \widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \qquad G_2 = \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}}{Pr^{max}_{\mathcal{A}_1 \| \mathcal{A}_2}\left(\Diamond G\right) \in [a,b]},$$

and

$$\frac{\underset{\sigma_2}{\forall}\ \underset{\sigma_1}{\inf}\, Pr^{\sigma_1,\sigma_2}_{G_1 \| G_2}\left(\Diamond G^{\#} \cup S_1 \times \left\{v_1^{\perp}\right\}\right) \in [a,b] \qquad G_1 = \widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \qquad G_2 = \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}}{Pr^{min}_{\mathcal{A}_1 \| \mathcal{A}_2}\left(\Diamond G\right) \in [a,b]},$$

where $G \subseteq S_1 \times S_2$. Note that, as before, $G^{\#}$ must be an exact representation of G in $G_1 \parallel G_2$. □

These rules say that for a PA $\mathcal{A}_1 \parallel \mathcal{A}_2$, composed of the components $\mathcal{A}_1$ and $\mathcal{A}_2$, the probabilistic reachability with respect to a goal set G is over-approximated by the composition of the component $\mathcal{A}_1$ (technically it is lifted to a game) and the Menu-game of component $\mathcal{A}_2$ with respect to a partition Q_2.

Let us look at why these rules are correct. In the previous chapters we have seen that probabilistic reachability for an $\mathcal{A}$ can be over-approximated by its Menu-game $\widehat{\mathcal{G}}_{\mathcal{A},Q}$ with respect to some partition Q, i.e. the following rules are known to be correct

$$\frac{\underset{\sigma_2}{\forall}\ \underset{\sigma_1}{\sup}\, Pr^{\sigma_1,\sigma_2}_{\widehat{\mathcal{G}}_{\mathcal{A},Q}}\left(\Diamond G^{\#}\right) \in [a,b]}{Pr^{max}_{\mathcal{A}}\left(\Diamond G\right) \in [a,b]}, \qquad \frac{\underset{\sigma_2}{\forall}\ \underset{\sigma_1}{\inf}\, Pr^{\sigma_1,\sigma_2}_{\widehat{\mathcal{G}}_{\mathcal{A},Q}}\left(\Diamond G^{\#} \cup \left\{v_1^{\perp}\right\}\right) \in [a,b]}{Pr^{min}_{\mathcal{A}}\left(\Diamond G\right) \in [a,b]}.$$

Analogously, a composed PA $\mathcal{A}_1 \parallel \mathcal{A}_2$ can be over-approximated by its Menu-game $\widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,Q}$ with respect to some partition Q of $S_1 \times S_2$, i.e. these rules are correct too:

$$\frac{\underset{\sigma_2}{\forall}\ \underset{\sigma_1}{\sup}\, Pr^{\sigma_1,\sigma_2}_{\widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,Q}}\left(\Diamond G^{\#}\right) \in [a,b]}{Pr^{max}_{\mathcal{A}_1\|\mathcal{A}_2}\left(\Diamond G\right) \in [a,b]}, \qquad \frac{\underset{\sigma_2}{\forall}\ \underset{\sigma_1}{\inf}\, Pr^{\sigma_1,\sigma_2}_{\widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,Q}}\left(\Diamond G^{\#} \cup \left\{v_1^{\perp}\right\}\right) \in [a,b]}{Pr^{min}_{\mathcal{A}_1\|\mathcal{A}_2}\left(\Diamond G\right) \in [a,b]}.$$

However, the Menu-game $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$ in our proposed assume-guarantee rules is rather different from that. The crucial question is why $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$ over-approximates $\mathcal{A}_1 \parallel \mathcal{A}_2$ with respect to probabilistic reachability.

The answer is, as initially indicated, that $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$ can simulate the Menu-game $\widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,S_1\times Q_2}$, of which we already know that it over-approximates $\mathcal{A}_1 \parallel \mathcal{A}_2$. Thus, the crucial part is proving that

$$\widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,S_1\times Q_2} \text{ is simulated by } \widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}.$$

The proof is rather tedious and thus not presented here but put in Section A of the appendix.

Example 6.8. Consider the modular system from Example 6.3. Using the assume-guarantee formalism, we can over-approximate the maximal probability of reaching an error state in $\mathcal{A}_{err} \parallel \mathcal{A}_{nom}$ by analysing the Menu-game $\widehat{\mathcal{G}}_{\mathcal{A}_{err},S_{err}} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_{nom},Q_{nom}}$ from Example 6.6.

To this end, we let $G = \{t_2\} \times S_{nom}$, since t_2 is the error state of the error model. Correspondingly, the abstract goal set is given by $G^{\#} = \{t_2\} \times Q_2$.

Figure 6.4 makes it easy to see that the bounds for maximal probabilistic reachability of $G^{\#}$ correspond to $[0.03, 1.0]$. Employing the respective assume-guarantee rule

$$\frac{\underset{\sigma_2}{\forall} \; \sup_{\sigma_1} Pr^{\sigma_1,\sigma_2}_{G_1 \parallel G_2}\left(\Diamond G^{\#}\right) \in [0.03, 1.0] \qquad G_1 = \widehat{\mathcal{G}}_{\mathcal{A}_{err},S_{err}} \qquad G_2 = \widehat{\mathcal{G}}_{\mathcal{A}_{nom},Q_{nom}}}{Pr^{max}_{\mathcal{A}_1 \parallel \mathcal{A}_2}\left(\Diamond G\right) \in [0.03, 1.0]}$$

yields that the maximal probabilistic reachability of G in $\mathcal{A}_{err} \parallel \mathcal{A}_{nom}$ is bounded by $[0.03, 1.0]$, too. This makes sense since the actual probability is 0.0591. □

7. Conclusion

7.1. Summary & Evaluation

In this thesis, we presented a fully symbolical variant of the (partially explicit) backward refinement procedure proposed in [Wachter, 2011], which is a form of predicate abstraction for probabilistic programs featuring non-determinism. In particular, we proposed optimisation techniques for both the abstraction and probabilistic reachability analysis. Based on several case studies, we evaluated the impact of these optimisations on the run time of our prototypical implementation and analysed the ratio of sub-procedures.

Our experiments indicate that the presented optimisations are crucial to make the symbolical approach feasible. Especially the combination of *relevant predicates* and *expression decomposition* turns out to be necessary for a fast and scalable abstraction of more complex models.

Furthermore, we showed that an explicit value iteration will usually outperform a symbolic approach if the explicit representation fits into memory. This is due to the fact that the symbolical representation, unlike the explicit one, heavily depends on the current valuation and is subject to change, but at the same time must be kept reduced and ordered. However, experiments have shown that the symbolical approach uses significantly less memory and may thus be able to verify models which are not amenable to the explicit approach.

We found that for both the model construction and especially the value iteration, MTBDD operations form the bottleneck of our approach. This was to be expected though, as the cost of MTBDD operations grows with increasing irregularity of the represented function but the MTBDDs during value iteration often become irregular and many of these costly operations must be performed.

Since neither Menu-games nor the backward refinement procedure exploit the composite nature of many models we also proposed an assume-guarantee style extension to alleviate this shortcoming. Its proper study and integration into an automatic refinement procedure is future work though.

Overall, we have found that (symbolical) backward refinement and the use of Menu-games has a right to exist, as it is computationally simpler than game-based abstraction but in contrast to other predicate abstraction techniques yields both lower and upper bounds for probabilistic reachability properties. Also, especially for large models, the state space of Menu-games turns out to be significantly smaller than its concrete counterpart.

Lastly, bear in mind that although backward refinement can solve probabilistic reachability for many infinite sized probabilistic models, it will not terminate for all instances. For example, the classical *Collatz-Conjecture* has been proven to be algorithmically undecidable [Conway, 1972]. Correspondingly, the backward refinement for $Pr^{min}(\Diamond [\![n = 1]\!])$ will never terminate for the following probabilistic program:

```
mdp

module collatz_on_int
    n    : int;
    [a] n>0 & n mod 2 = 0    -> 1.0 :(n'=n/2);
    [b] n>0 & n mod 2 = 1    -> 1.0 :(n'=3*n+1);
    [inv] phase=1 & run<=0   -> 1.0 :(n'=-n+1);
endmodule
```

Listing 7.1: Collatz conjecture extended to $\mathbb{Z}$

7.2. Future Work

In this section, we illustrate possible future work on extending our approach. First of all, there are several rather obvious ways to improve the overall performance of the prototype:

Topological Value Iteration The worst performing part of our symbolical approach is clearly the value iteration. The bad scaling is largely attributed to our primitive value iteration scheme which always updates the values of all vertices. Thus, a more sophisticated approach like [Dai et al., 2011], exploiting the topology of the game, may significantly improve its performance.

Parallelisation The computation of menu-based abstraction amounts to solving an abstract transition constraint $\mathcal{R}_c^\#$ for every command c. However, these constraints are independent, making their solution ideally suited for parallelisation. Moreover, one may consider using an MTBDD library like JINC [Ossowski, 2010], which supports multi-threading, to handle the complexity of MTBDD operations during value iteration.

Restrictive refinement predicates As mentioned in Section 5.3.4, the reference tool PASS checks whether a chosen pivot block is actually reachable in the concrete model. If such a block is spurious, PASS computes interpolants from a respective path formula, which make this and similar blocks unreachable in follow up refinement iterations. Such restrictive predicates may significantly affect the size of the abstract state space and as a result improve both the value iteration performance and the precision of obtained Menu-games.

Furthermore, aiming to avoid spurious blocks, it might be feasible to derive invariants from statical analysis of a probabilistic program and use these to restrict the abstract state space. The difficulty here is to make this work properly in combination with the relevant predicates optimisation, where irrelevant predicates are not considered by the SMT-solver but always retain their values. Due to this, pushing the invariants on the solver's stack would often have no effect. However, it is not clear how to apply an invariant to the constructed state space either.

Last but not least, is also remains to integrate our assume-guarantee extension into an automatic refinement procedure and evaluate its feasibility.

Appendix

A. Assume-guarantee Proof

Our Assume-guarantee rule is based on the assumption that $\widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,S_1\times Q_2}$ can be *simulated* by $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$, i.e. there exists a relation $\mathcal{R}$ between the games' vertices, such that the initial vertices are related and for a pair $(u, v) \in \mathcal{R}$, if there exists a α-successor u' of u, reachable with probability p, then there exists a α-successor v' of v, reachable with the same probability and $(u', v') \in \mathcal{R}$. In the following we conveniently write $\mathcal{R}(v, u)$ to denote $(v, u) \in \mathcal{R}$.

Definition 7.1 (Simulation)**.** Let

$$\mathcal{G}_1 = \left(V_1, Act_1, Opt_1, U_1, \delta_1, v_{init,1}\right) \tag{7.1}$$
$$\mathcal{G}_2 = \left(V_2, Act_2, Opt_2, U_2, \delta_2, v_{init,2}\right) \tag{7.2}$$

be Menu-games. We say that $\mathcal{G}_2$ simulates $\mathcal{G}_1$, denoted by $\mathcal{G}_1 \preceq \mathcal{G}_2$, if there exists a relation $\mathcal{R}$ such that $\left(v_{init,1}, v_{init,2}\right) \in \mathcal{R}$, and for two related vertices $(v_1, v_2) \in \mathcal{R}$, with $v_1 \in V_1$ and $v_2 \in V_2$, the following holds:

$$\forall_{(v_1,\alpha,\beta_1,\mu_1)\in\delta_1} \exists_{(v_2,\alpha,\beta_2,\mu_2)\in\delta_2} \forall_{v_1'\in V_1} \widehat{\mu_1}(v_1') > 0 \Rightarrow \exists_{v_2'\in V_2} \widehat{\mu_2}(v_2') = \widehat{\mu_1}(v_1') \wedge \mathcal{R}\left(v_1', v_2'\right).$$

□

Let $\mathcal{A}_1 = (S_1, Act_1, U_1, \mathbf{P}_1, s_{init,1})$ and $\mathcal{A}_2 = (S_2, Act_2, U_2, \mathbf{P}_2, s_{init,2})$ be PA and Q_2 a partition of S_2. Furthermore, let $G_1 := \widehat{\mathcal{G}}_{\mathcal{A}_1\|\mathcal{A}_2,S_1\times Q_2}$ and $G_2 := \widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$. We refer to the vertices of G_i as V_i and their transition relations as δ_i.

We claim that

$$\mathcal{R} := \left\{ (v_1, v_2) \in V_1 \times V_2 \;\middle|\; \begin{array}{c} v_1 = v_1^\perp \Rightarrow v_2 \in S_1 \times \left\{v_1^\perp\right\} \\ v_1 = (s_1, B_2) \Rightarrow v_2 = (s_1, B_2) \end{array} \right\}$$

is a simulation relation, such that $G_1 \preceq G_2$. The crucial difference between G_1 and G_2 is that G_1 has a single bottom state, while G_2 has several composed bottom states, which are related in $\mathcal{R}$. The remaining entries of $\mathcal{R}$ relate identically identified vertices of both games. Note that the initial vertex of both G_1 and G_2 is identified via $\left(v_{init,1}, \overline{v_{init,2}}\right)$ such that $(v_{init1}, v_{init2}) \in \mathcal{R}$.

Proof. Let $(v_1, \alpha, \beta_1, \mu_1) \in \delta_1$ and $\mathcal{R}(v_1, v_2)$. We show that there exists a corresponding tuple $(v_2, \alpha, \beta_2, \mu_2) \in \delta_2$ such that equally weighted vertices in the support of μ_1 and μ_2 are related. To this end we distinguish whether v_1 is a trap vertex or not.

Case $v_1 = v_1^\perp$**:** By Definition 3.1, the distributions must be $\mu_1 = v_1^p = 1.0 : (u_\tau, v_1')$ with $v_1' = v_1^\perp$. Since $\mathcal{R}(v_1, v_2)$, we know that $v_2 = (s_1, v_1^\perp) \in S_1 \times \left\{v_1^\perp\right\}$. Since trap states do not enable conventional actions but use a_τ they are not synchronised during $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1} \parallel \widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$ and, as a result, there exists a tuple $(v_2, a_\tau, \beta_2, 1.0 : (u_\tau, v_2')) \in \delta_2$ with $\left(s_1, v_1^\perp\right)$, such that $(v_1', v_2') \in \mathcal{R}$.

Case $v_1 = (s_1, B_2)$**:** There are three ways for an action α to be enabled in v_1. Either it is the result of synchronisation over a *shared* action, i.e. $\alpha \in (Act_1 \cup Act_2) \setminus \{a_\tau\}$ or a transition *private* to a subcomponent, i.e. $\alpha \in (Act_1 \setminus Act_2) \cup \{a_\tau\}$ or $\alpha \in (Act_2 \setminus Act_1) \cup \{a_\tau\}$.

Case α **shared:** α must be enabled in both s_1 and some state $s_2 \in B_2$.

Since $\mathcal{R}(v_1, v_2)$ we know that $v_2 = (s_1, B_2)$. Accordingly, there must be a tuple $(s_1, \alpha, \beta_{2,1}, \mu_{1,1})$ in the transition relation of $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1}$ and a tuple $(B_2, \alpha, \beta_{2,2}, \mu_{1,2})$ in the transition relation of $\widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$. Their parallel composition results in $(v_2, \alpha, (\beta_1, \beta_2), \mu_2)$ with $\mu_1 = \mu_2 = \mu_{1,1} \times \mu_{1,2}$. Being the same distributions they trivially lead to related successors (in $\mathcal{R}$).

Case α **private to** $\mathcal{A}_1$**:** In this case, according to Definition 6.5, $\mu_1 = \mu_{1,1} \times 1.0 : (u_\tau, B_2)$, because there is no $s_2 \in B_2$ such that α is enabled (not considering a_τ).

Since $\mathcal{R}(v_1, v_2)$ we know that $v_2 = (s_1, B_2)$. Accordingly, there must be a tuple $(s_1, \alpha, \beta_{2,1}, \mu_{1,1})$ in the transition relation of $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1}$ but block B_2 in $\widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$ does not enable α. According to the definition of parallel composition, this results in $(v_2, \alpha, (\beta_{2,1}, \beta_{2,1}), \mu_2) \in \delta_2$ with $\mu_2 = \mu_1$. Being the same distributions, their successors are in $\mathcal{R}$.

Case α **private to** $\mathcal{A}_2$**:** In this case, according to Definition 6.5, we have $\mu_1 = 1.0 : (u_\tau, \overline{s_1}) \times \mu_{1,2}$, because s_1 does not enable α (not considering a_τ).

Since $\mathcal{R}(v_1, v_2)$ we know that $v_2 = (s_1, B_2)$. Accordingly, there must be a tuple $(B_2, \alpha, \beta_{2,2}, \mu_{1,2})$ in the transition relation of $\widehat{\mathcal{G}}_{\mathcal{A}_2,Q_2}$ but block $\overline{s_1}$ in $\widehat{\mathcal{G}}_{\mathcal{A}_1,S_1}$ does not enable α. According to the definition of parallel composition, this results in $(v_2, \alpha, (\beta_{2,2}, \beta_{2,2}), \mu_2) \in \delta_2$ with $\mu_2 = \mu_1$. Being the same distributions, their successors are in $\mathcal{R}$.

$\square$

B. Raw Evaluation Data

B.1. Crowds Protocol

Ref. Iteration	Normal	Range	Relevant	Decomp.	Unrelated	Reach
#0	5336	4717	3237	142	110	140
#1	9145	8925	5289	71	60	61
#2	14748	13605	8789	71	59	73
#3	19705	22739	11944	84	47	67
#4	36826	33465	16176			
#5	53389	41349	22231			
#6	822291	59784	38854			
#7	138019	120123	49154			

Table 7.1.: Time (in ms) spent on abstraction and construction

	Time [ms]				Vertex count		
Ref. Iteration	SMT	BDD	REACH	other	Player 1	Player 2	Prob.
#0	28	5	22	55	24516	24516	28269
#1	18	2	38	2	36773	36773	41039
#2	19	3	32	5	49030	49030	53809
#3	20	4	19	4	7238	7238	7471

Table 7.2.: Detailed breakdown of “Unrelated”

Ref. Iteration	Normal	Reuse Res.	Goal Succ.	Explicit
#0	1161	1181	1099	219
#1	2911	978	756	333
#2	3447	1316	1017	452
#3	666	710	685	225

Table 7.3.: Time (in ms) spent on analysis

	Time [ms]	
Ref. Iteration	PROB0/1	VI
#0	60	159
#1	86	247
#2	118	334
#3	146	106

Table 7.4.: Detailed breakdown of “Explicit”

B.2. Randomised Consensus Shared Coin Protocol

Ref. Iteration	Normal	Range	Relevant	Decomp.	Unrelated	Reach
#0	1101	648	568	174	125	148
#1	1211	713	582	102	60	97
#2	1359	673	486	70	68	70
#3	1738	847	567	82	72	75
#4	2323	970	661	86	81	85
#5	2497	1379	791	94	85	90
#6	2873	1563	936			
#7	3243	1986	996			
#8	3426	2016	1100			
#9	3918	2342	1151			

Table 7.5.: Time (in ms) spent on abstraction and construction

	Time [ms]				Vertex count		
Ref. Iteration	Smt	BDD	Reach	other	Player 1	Player 2	Prob.
#0	56	16	5	48	112	160	224
#1	36	7	8	9	144	208	272
#2	40	8	10	10	176	256	320
#3	43	9	11	9	208	304	344
#4	48	11	13	9	240	352	392
#5	51	14	11	9	272	400	416

Table 7.6.: Detailed breakdown of "Unrelated"

Ref. Iteration	Normal	Reuse Res.	Goal Succ.	Explicit
#0	46	47	51	53
#1	88	89	95	82
#2	158	157	172	105
#3	317	339	371	146
#4	446	455	506	182
#5	744	1276	1352	242

Table 7.7.: Time (in ms) spent on analysis

	Time [ms]	
Ref. Iteration	Prob0/1	VI
#0	36	17
#1	57	25
#2	71	34
#3	93	53
#4	129	53
#5	178	64

Table 7.8.: Detailed breakdown of “Explicit”

B.3. Asynchronous Leader Election Protocol

Ref. Iteration	Decomp.	Unrelated	Reach
#0	2563	2083	2466
#1	1501	830	1245
#2	1688	926	1180
#3	1909	890	1127
#4	2007	890	1092

Table 7.9.: Time (in ms) spent on abstraction and construction

	Time [ms]				Vertex count		
Ref. Iteration	Smt	BDD	Reach	other	Player 1	Player 2	Prob.
#0	869	460	240	514	3340	6460	6548
#1	351	297	160	22	3298	6408	6474
#2	404	207	260	55	3256	6356	6400
#3	411	210	227	42	3214	6304	6326
#4	455	233	180	22	3172	6252	6252

Table 7.10.: Detailed breakdown of “Unrelated”

Ref. Iteration	Normal	Reuse Res.	Goal Succ.	Explicit
#0	4565	4582	4752	3781
#1	1390	1449	1724	3845
#2	28369	28433	26884	5156
#3	17700	25241	25437	5927
#4	1398	1536	1826	3562

Table 7.11.: Time (in ms) spent on analysis

	Time [ms]	
Ref. Iteration	Prob0/1	VI
#0	1015	2766
#1	1279	2566
#2	2109	3047
#3	2383	3544
#4	990	2572

Table 7.12.: Detailed breakdown of “Explicit”

B.4. Wireless LAN Protocol

Ref. Iteration	Decomp.	Unrelated	Reach
#0	9134	5521	6027
#1	8946	641	1038
#2	13429	587	1186
#3	4561	622	935
#4	4622	688	950
#5	4870	652	991
#6	4620	681	983
#7	4908	694	1000
#8	5111	715	1013
#9	5216	640	1023
#10	4978	785	1034
#11	5253	681	1044
#12	5305	776	1082
#13	5550	698	1123
#14	5526	837	1111
#15	5759	757	1116
#16	5676	785	1103
#17	5719	819	1114
#18	6038	844	1142
#19	6033	833	1173
#20	6344	958	1175
#21	6547	867	1149
#22	6689	876	1261
#23	6806	895	1274
#24	6623	923	1271
#25	6937	930	1341
#26	6819	1029	1355
#27	7151	1130	1374
#28	7001	1125	1391
#29	7616	1159	1406

Table 7.13.: Time (in ms) spent on abstraction and construction

	Time [ms]				Vertex count		
Ref. Iteration	Smt	BDD	Reach	other	Player 1	Player 2	Prob.
#0	2809	1505	383	824	1055	1497	1607
#1	16	357	230	38	1175	1655	1771
#2	30	362	180	15	1295	1813	1935
#3	35	368	210	9	1415	1971	2099
#4	41	379	240	28	1535	2129	2263
#5	46	229	318	59	1655	2287	2427
#6	53	239	316	73	1775	2445	2591
#7	59	270	338	27	1895	2603	2755
#8	67	268	363	17	2014	2761	2919
#9	76	257	289	18	2135	2919	3083
#10	75	464	224	22	2255	3077	3247
#11	92	324	220	45	2375	3235	3411
#12	101	378	235	62	2495	3393	3575
#13	102	334	227	35	2615	3551	3739
#14	121	412	257	47	2685	3647	3806
#15	26	385	303	43	2917	3945	4106
#16	30	431	302	22	3149	4243	4406
#17	35	430	322	32	3381	4541	4706
#18	41	428	360	15	3613	4839	5006
#19	47	438	329	19	3845	5137	5306
#20	54	448	364	92	4077	5435	5606
#21	60	453	332	22	4309	5733	5906
#22	68	456	325	27	4541	6031	6206
#23	75	476	329	15	4773	6329	6506
#24	84	476	339	24	5005	6627	6806
#25	92	493	331	14	5237	6925	7106
#26	65	492	448	24	7771	10277	10537
#27	102	528	427	73	8118	10723	10986
#28	111	535	431	48	8465	11169	11435
#29	120	537	442	60	8737	11522	11655

Table 7.14.: Detailed breakdown of “Unrelated”

Ref. Iteration	Normal	Reuse Res.	Goal Succ.	Explicit
#0	1358	1449	1375	1036
#1	1773	1851	1712	1394
#2	897	1238	1055	838
#3	987	1221	1124	897
#4	1022	1340	1059	830
#5	934	1186	1030	825
#6	1061	1316	1220	895
#7	1129	1416	1202	873
#8	1168	1466	1174	837
#9	1264	1650	1322	912
#10	1341	1522	1176	920
#11	1451	1647	1311	928
#12	1400	1728	1448	934
#13	1515	1828	1488	937
#14	1590	1937	1572	927
#15	1725	2061	1651	962
#16	1417	1714	1370	921
#17	1423	1657	1388	942
#18	1447	2033	1428	968
#19	1488	1934	1470	962
#20	1500	1727	1488	955
#21	1616	2028	1510	983
#22	1558	1962	1504	1018
#23	1573	1781	1536	975
#24	1746	2253	1579	976
#25	1761	1914	1716	1012
#26	1987	2156	1804	1049
#27	2173	1759	1611	1037
#28	2274	1837	1485	1063
#29	2767	2627	2544	1105

Table 7.15.: Time (in ms) spent on analysis

	Time [ms]	
Ref. Iteration	Prob0/1	VI
#0	896	140
#1	802	592
#2	628	210
#3	642	255
#4	435	395
#5	503	322
#6	545	350
#7	596	277
#8	609	228
#9	648	264
#10	694	226
#11	727	201
#12	720	214
#13	708	229
#14	772	155
#15	760	202
#16	739	182
#17	748	194
#18	723	245
#19	776	186
#20	767	188
#21	753	230
#22	743	275
#23	735	240
#24	773	203
#25	712	300
#26	746	303
#27	790	247
#28	815	248
#29	818	287

Table 7.16.: Detailed breakdown of “Explicit”

B.5. Memory Usage

	Procedure		
Case Study	Symbolic	Explicit	Smt-part
Crowds	57	90	22
Consensus	65	74	32
Leader	452	645	321
WLAN	624	842	518

Table 7.17.: Peak memory usage (in MB) of complete procedure

B.6. Storm vs. Pass

Case Study	Symbolic	Explicit	PASS
Crowds	8461	1532	1842
Consensus	2290	1301	14891
Leader	59041	27890	20416
WLAN	74493	58059	9885

Table 7.18.: Time (in ms) spent on complete procedure

C. Case Studies and Properties

C.1. Crowds Protocol

Property

$$Pr^{min}\left(\lozenge \llbracket observe0 > 1\rrbracket\right).$$

Prism model

```
mdp

// probability of forwarding
const double    PF = 0.8;
const double notPF = .2;  // must be 1-PF
// probability that a crowd member is bad
const double  badC = .167;
 // probability that a crowd member is good
const double goodC = 0.833;
// Total number of protocol runs to analyze
const int TotalRuns = 5;
// size of the crowd
const int CrowdSize = 5;

module crowds
    // protocol phase
    phase: [0..4] init 0;

    // crowd member good (or bad)
    good: bool init false;

    // number of protocol runs
    runCount: [0..TotalRuns] init 0;

    // observe_i is the number of times
    // the attacker observed crowd member i
    observe0: [0..TotalRuns] init 0;
    observe1: [0..TotalRuns] init 0;
    observe2: [0..TotalRuns] init 0;
    observe3: [0..TotalRuns] init 0;
    observe4: [0..TotalRuns] init 0;

    // the last seen crowd member
    lastSeen: [0..CrowdSize - 1] init 0;

    // get the protocol started
    [] phase=0 & runCount<TotalRuns -> 1: (phase'=1)
                                        & (runCount'=runCount+1)
                                        & (lastSeen'=0);

    // decide whether crowd member is good or bad
    [] phase=1 -> goodC : (phase'=2) & (good'=true)
                + badC : (phase'=2) & (good'=false);

    // if the current member is a good member,
    // update the last seen index (chosen uniformly)
    [] phase=2 & good -> 1/5 : (lastSeen'=0) & (phase'=3)
                       + 1/5 : (lastSeen'=1) & (phase'=3)
                       + 1/5 : (lastSeen'=2) & (phase'=3)
                       + 1/5 : (lastSeen'=3) & (phase'=3)
                       + 1/5 : (lastSeen'=4) & (phase'=3);

    // if the current member is a bad member,
    // record the most recently seen index
    [] phase=2 & !good & lastSeen=0 & observe0 < TotalRuns
      -> 1: (observe0'=observe0+1) & (phase'=4);
    [] phase=2 & !good & lastSeen=1 & observe1 < TotalRuns
      -> 1: (observe1'=observe1+1) & (phase'=4);
    [] phase=2 & !good & lastSeen=2 & observe2 < TotalRuns
      -> 1: (observe2'=observe2+1) & (phase'=4);
    [] phase=2 & !good & lastSeen=3 & observe3 < TotalRuns
      -> 1: (observe3'=observe3+1) & (phase'=4);
    [] phase=2 & !good & lastSeen=4 & observe4 < TotalRuns
      -> 1: (observe4'=observe4+1) & (phase'=4);

    // good crowd members forward with probability PF; deliver otherwise
    [] phase=3 -> PF : (phase'=1) + notPF : (phase'=4);

    // deliver the message and start over
    [] phase=4 -> 1: (phase'=0);
endmodule
```

Listing 7.2: Crowds Protocol

C.2. Randomised Consensus Shared Coin Protocol

Property

$$Pr^{min}\left(\Diamond \left[\!\left[\mathit{finished} \wedge \mathit{allCoinsEqual1} \right]\!\right]\right),$$

where

$$\begin{aligned} \mathit{finished} &:= \mathit{pc1} = 3 \wedge \mathit{pc2} = 3 \\ \mathit{allCoinsEqual1} &:= \mathit{coin1} = 1 \wedge \mathit{coin2} = 1. \end{aligned}$$

Prism model

```
mdp

// constants
const int N=2;
const int K;
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 coin
global counter : [0..range] init counter_init;

module process1
    // program counter
    // 0 = flip, 1 = write, 2 = check, 3 = finished
    pc1 : [0..3];

    // local coin
    coin1 : [0..1];

    // flip coin
    [] (pc1=0)  -> 0.5 : (coin1'=0) & (pc1'=1)
                 + 0.5 : (coin1'=1) & (pc1'=1);
    // write tails -1  (reset coin to add regularity)
    [] (pc1=1) & (coin1=0) & (counter>0)
       -> 1 : (counter'=counter-1) & (pc1'=2) & (coin1'=0);
    // write heads +1 (reset coin to add regularity)
    [] (pc1=1) & (coin1=1) & (counter<range)
    -> 1 : (counter'=counter+1) & (pc1'=2) & (coin1'=0);

    // decide tails
    [] (pc1=2) & (counter<=left) -> 1 : (pc1'=3) & (coin1'=0);
    // decide heads
    [] (pc1=2) & (counter>=right) -> 1 : (pc1'=3) & (coin1'=1);
    // flip again
    [] (pc1=2) & (counter>left) & (counter<right) -> 1 : (pc1'=0);
    // loop (all loop together when done)
    [done] (pc1=3) -> 1 : (pc1'=3);
endmodule

// construct remaining processes through renaming
module process2 = process1[pc1=pc2,coin1=coin2] endmodule
```

Listing 7.3: Consensus Protocol

C.3. Asynchronous Leader Election Protocol

Property

$$Pr^{min}\left(\Diamond\,[\![elected]\!]\right),$$

where

$$s1 = 4 \vee s2 = 4 \vee s3 = 4 \vee s4 = 4.$$

Prism model

```
mdp

const int N = 4; // number of processes
module process1
    // COUNTER
    c1 : [0..4-1];

    // STATES
    // 0 = make choice, 1 = have not received neighbours choice,
    // 2 = active, 3 = inactive, 4 = leader
    s1 : [0..4];

    // PREFERENCE
    p1 : [0..1];

    // VARIABLES FOR SENDING AND RECEIVING
    receive1 : [0..2];
    sent1 : [0..2];

    // pick value
    [] (s1=0) -> 0.5 : (s1'=1) & (p1'=0) + 0.5 : (s1'=1) & (p1'=1);

    // send preference
    [p12] (s1=1) & (sent1=0) -> (sent1'=1);
    // receive preference
    // stay active
    [p41] (s1=1) & (receive1=0) & !((p1=0) & (p4=1))
    -> (s1'=2) & (receive1'=1);
    // become inactive
    [p41] (s1=1) & (receive1=0) & (p1=0) & (p4=1)
    -> (s1'=3) & (receive1'=1);

    // send preference (can now reset preference)
    [p12] (s1=2) & (sent1=0) -> (sent1'=1) & (p1'=0);
    // send counter (already sent preference)
    // not received counter yet
    [c12] (s1=2) & (sent1=1) & (receive1=1) -> (sent1'=2);
    // received counter (pick again)
    [c12] (s1=2) & (sent1=1) & (receive1=2)
    -> (s1'=0) & (p1'=0) & (c1'=0) & (sent1'=0) & (receive1'=0);

    // receive counter and not sent yet
    [c41] (s1=2) & (receive1=1) & (sent1<2) -> (receive1'=2);
    // receive counter and sent counter
    // only active process (decide)
    [c41] (s1=2) & (receive1=1) & (sent1=2) & (c4=N-1)
    -> (s1'=4) & (p1'=0) & (c1'=0) & (sent1'=0) & (receive1'=0);
    // other active process (pick again)
    [c41] (s1=2) & (receive1=1) & (sent1=2) & (c4<N-1)
    -> (s1'=0) & (p1'=0) & (c1'=0) & (sent1'=0) & (receive1'=0);

    // send preference (must have received preference) and can now reset
    [p12] (s1=3) & (receive1>0) & (sent1=0) -> (sent1'=1) & (p1'=0);
    // send counter (must have received counter first) and can now reset
    [c12] (s1=3) & (receive1=2) & (sent1=1)
    -> (s1'=3) & (p1'=0) & (c1'=0) & (sent1'=0) & (receive1'=0);

    // receive preference
    [p41] (s1=3) & (receive1=0) -> (p1'=p4) & (receive1'=1);
    // receive counter
    [c41] (s1=3) & (receive1=1) & (c4<N-1) -> (c1'=c4+1) & (receive1'=2);

    // done
    [done] (s1=4) -> (s1'=s1);
    // add loop for processes who are inactive
    [done] (s1=3) -> (s1'=s1);
endmodule

// construct further stations through renaming
module process2=process1[s1=s2,p1=p2,c1=c2,sent1=sent2,receive1=receive2,
                         p12=p23,p41=p12,c12=c23,c41=c12,p4=p1,c4=c1]
endmodule
module process3=process1[s1=s3,p1=p3,c1=c3,sent1=sent3,receive1=receive3,
                         p12=p34,p41=p23,c12=c34,c41=c23,p4=p2,c4=c2]
endmodule
module process4=process1[s1=s4,p1=p4,c1=c4,sent1=sent4,receive1=receive4,
                         p12=p41,p41=p34,c12=c41,c41=c34,p4=p3,c4=c3]
endmodule
```

Listing 7.4: Leader Election Protocol

C.4. Wireless LAN Protocol

Property

$$Pr^{max}\left(\Diamond\, [\![col = 2]\!]\right).$$

Prism model

```
mdp

// COLLISIONS
const int COL = 2; // maximum number of collisions
```

```

// TIMING CONSTRAINTS
const int ASLOTTIME = 1;
const int DIFS = 3;
const int VULN = 1;
const int TRANS_TIME_MAX; // scaling up
const int TRANS_TIME_MIN = 4; // scaling down
const int ACK_TO = 6;
const int ACK = 4;
const int SIFS = 1;
// maximum constant used in timing constraints + 1
const int TIME_MAX = max(ACK_TO,TRANS_TIME_MAX)+1;

// CONTENTION WINDOW
// CWMIN =15 & CWMAX =16
// this means that MAX_BACKOFF IS 2
const int MAX_BACKOFF = 0;

// THE MEDIUM/CHANNEL
// FORMULAE FOR THE CHANNEL
// channel is busy
formula busy = c1>0 | c2>0;
// channel is free
formula free = c1=0 & c2=0;

module medium
    // number of collisions
    col : [0..COL];

    // medium status
    c1 : [0..2];
    c2 : [0..2];
    // ci = message associated with station i
    // 0 = nothing being sent, 1 = being sent correctly,
    // 2 = being sent garbled

    // begin sending message and nothing else currently being sent
    [send1] c1=0 & c2=0 -> (c1'=1);
    [send2] c2=0 & c1=0 -> (c2'=1);

    // begin sending message and  something is already being sent
    // in this case both messages become garbled
    [send1] c1=0 & c2>0 -> (c1'=2) & (c2'=2) & (col'=min(col+1,COL));
    [send2] c2=0 & c1>0 -> (c1'=2) & (c2'=2) & (col'=min(col+1,COL));

    // finish sending message
    [finish1] c1>0 -> (c1'=0);
    [finish2] c2>0 -> (c2'=0);
endmodule

module station1
    // clock for station 1
    x1 : [0..TIME_MAX];

    // local state
    s1 : [1..12];
    // 1 = sense, 2 = wait until free before setting backoff,
    // 3 = wait for DIFS then set slot, 4 = set backoff, 5 = backoff,
    // 6 = wait until free in backoff,
    // 7 = wait for DIFS then resume backoff, 8 = vulnerable,
    // 9 = transmit, 10 = wait for ACK_TO,
    // 11 = wait for SIFS and then ACK, 12 = done

    // BACKOFF (separated into slots)
    slot1 : [0..1];
    backoff1 : [0..15];

    // BACKOFF COUNTER
    bc1 : [0..1];

    // SENSE
    // let time pass
    [time] s1=1 & x1<DIFS & free -> (x1'=min(x1+1,TIME_MAX));
    // ready to transmit
    [] s1=1 & (x1=DIFS | x1=DIFS-1) -> (s1'=8) & (x1'=0);
    // found channel busy so wait until free
    [] s1=1 & busy -> (s1'=2) & (x1'=0);

    // WAIT UNTIL FREE BEFORE SETTING BACKOFF
    // let time pass (no need for the clock x1 to change)
    [time] s1=2 & busy -> (s1'=2);
    // find that channel is free
    // so check its free for DIFS before setting backoff
    [] s1=2 & free -> (s1'=3);

    // WAIT FOR DIFS THEN SET BACKOFF
    // let time pass
    [time] s1=3 & x1<DIFS & free -> (x1'=min(x1+1,TIME_MAX));
    // found channel busy so wait until free
    [] s1=3 & busy -> (s1'=2) & (x1'=0);
    // start backoff  first uniformly choose slot
    // backoff counter 0
    [] s1=3 & (x1=DIFS | x1=DIFS-1) & bc1=0
    -> (s1'=4) & (x1'=0) & (slot1'=0) & (bc1'=min(bc1+1,MAX_BACKOFF));

    // SET BACKOFF (no time can pass)
    // chosen slot now set backoff
    [] s1=4 -> 1/16 : (s1'=5) & (backoff1'=0 )
            + 1/16 : (s1'=5) & (backoff1'=1 )
            + 1/16 : (s1'=5) & (backoff1'=2 )
            + 1/16 : (s1'=5) & (backoff1'=3 )
            + 1/16 : (s1'=5) & (backoff1'=4 )
            + 1/16 : (s1'=5) & (backoff1'=5 )
            + 1/16 : (s1'=5) & (backoff1'=6 )
            + 1/16 : (s1'=5) & (backoff1'=7 )
            + 1/16 : (s1'=5) & (backoff1'=8 )
```

```
              + 1/16 : (s1'=5) & (backoff1'=9 )
              + 1/16 : (s1'=5) & (backoff1'=10)
              + 1/16 : (s1'=5) & (backoff1'=11)
              + 1/16 : (s1'=5) & (backoff1'=12)
              + 1/16 : (s1'=5) & (backoff1'=13)
              + 1/16 : (s1'=5) & (backoff1'=14)
              + 1/16 : (s1'=5) & (backoff1'=15);
    // BACKOFF
    // let time pass
    [time] s1=5 & x1<ASLOTTIME & free -> (x1'=min(x1+1,TIME_MAX));
    // decrement backoff
    [] s1=5 & x1=ASLOTTIME & backoff1>0
    -> (s1'=5) & (x1'=0) & (backoff1'=backoff1-1);
    [] s1=5 & x1=ASLOTTIME & backoff1=0 & slot1>0
    -> (s1'=5) & (x1'=0) & (backoff1'=15) & (slot1'=slot1-1);
    // finish backoff
    [] s1=5 & x1=ASLOTTIME & backoff1=0 & slot1=0
    -> (s1'=8) & (x1'=0);
    // found channel busy
    [] s1=5 & busy -> (s1'=6) & (x1'=0);

    // WAIT UNTIL FREE IN BACKOFF
    // let time pass (no need for the clock x1 to change)
    [time] s1=6 & busy -> (s1'=6);
    // find that channel is free
    [] s1=6 & free -> (s1'=7);

    // WAIT FOR DIFS THEN RESUME BACKOFF
    // let time pass
    [time] s1=7 & x1<DIFS & free -> (x1'=min(x1+1,TIME_MAX));
    // resume backoff (start again from previous backoff)
    [] s1=7 & (x1=DIFS | x1=DIFS-1) -> (s1'=5) & (x1'=0);
    // found channel busy
    [] s1=7 & busy -> (s1'=6) & (x1'=0);

    // VULNERABLE
    // let time pass
    [time] s1=8 & x1<VULN -> (x1'=min(x1+1,TIME_MAX));
    // move to transmit
    [send1] s1=8 & (x1=VULN | x1=VULN-1) -> (s1'=9) & (x1'=0);

    // TRANSMIT
    // let time pass
    [time] s1=9 & x1<TRANS_TIME_MAX -> (x1'=min(x1+1,TIME_MAX));
    // finish transmission successful
    [finish1] s1=9 & x1>=TRANS_TIME_MIN & c1=1 -> (s1'=10) & (x1'=0);
    // finish transmission garbled
    [finish1] s1=9 & x1>=TRANS_TIME_MIN & c1=2 -> (s1'=11) & (x1'=0);
    // WAIT FOR SIFS THEN WAIT FOR ACK

    // WAIT FOR SIFS i.e. c1=0
    // check channel and busy: go into backoff
    [] s1=10 & c1=0 & x1=0 & busy -> (s1'=2);
    // check channel and free: let time pass
    [time] s1=10 & c1=0 & x1=0 & free -> (x1'=min(x1+1,TIME_MAX));
    // let time pass
    // following guard is always false as SIFS=1
    // [time] s1=10 & c1=0 & x1>0 & x1<SIFS -> (x1'=min(x1+1,TIME_MAX));
    // ack is sent after SIFS
    [send1] s1=10 & c1=0 & (x1=SIFS | (x1=SIFS-1 & free))
    -> (s1'=10) & (x1'=0);

    // WAIT FOR ACK i.e. c1=1
    // let time pass
    [time] s1=10 & c1=1 & x1<ACK -> (x1'=min(x1+1,TIME_MAX));
    // get acknowledgement so packet sent correctly and move to done
    [finish1] s1=10 & c1=1 & (x1=ACK | x1=ACK-1)
    -> (s1'=12) & (x1'=0) & (bc1'=0);

    // WAIT FOR ACK_TO
    // check channel and busy: go into backoff
    [] s1=11 & x1=0 & busy -> (s1'=2);
    // check channel and free: let time pass
    [time] s1=11 & x1=0 & free -> (x1'=min(x1+1,TIME_MAX));
    // let time pass
    [time] s1=11 & x1>0 & x1<ACK_TO -> (x1'=min(x1+1,TIME_MAX));
    // no acknowledgement (go to backoff waiting DIFS first)
    [] s1=11 & x1=ACK_TO -> (s1'=3) & (x1'=0);

    // DONE
    [time] s1=12 -> (s1'=12);
endmodule

// STATION 2 (rename STATION 1)
module station2=station1[x1=x2, s1=s2, s2=s1, c1=c2, c2=c1,
                         slot1=slot2, backoff1=backoff2,
                         bc1=bc2, send1=send2, finish1=finish2]
endmodule
```

Listing 7.5: WLAN Protocol

Bibliography

[Alur and Henzinger, 1999] Alur, R. and Henzinger, T. (1999). Reactive Modules. *Formal Methods in System Design*, 15(1):7–48.

[Aspnes and Herlihy, 1990] Aspnes, J. and Herlihy, M. (1990). Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms*, 15(1):441–460.

[Bahar et al., 1997] Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1997). Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206.

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.

[Ball and Rajamani, 2002] Ball, T. and Rajamani, S. K. (2002). The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA. ACM.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999). Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK. Springer-Verlag.

[Bryant, 1986] Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691.

[Clarke et al., 2001] Clarke, E., Biere, A., Raimi, R., and Zhu, Y. (2001). Bounded Model Checking Using Satisfiability Solving. *Form. Methods Syst. Des.*, 19(1):7–34.

[Clarke and Emerson, 1982] Clarke, E. M. and Emerson, E. A. (1982). Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK. Springer-Verlag.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263.

[Condon, 1992] Condon, A. (1992). The Complexity of Stochastic Games. *Information and Computation*, 96:203–224.

[Conway, 1972] Conway, J. H. (1972). Unpredictable Iterations. In *Proceedings of the 1972 Number Theory Conference*, pages 49–52. University of Colorado, Boulder.

[Cook, 1971] Cook, S. A. (1971). The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA. ACM.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA. ACM.

[Cousot and Cousot, 1979] Cousot, P. and Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA. ACM.

[Cousot and Cousot, 1992] Cousot, P. and Cousot, R. (1992). Abstract Interpretation and Application to Logic Programs. *J. Log. Program.*, 13(2&3):103–179.

[Dai et al., 2011] Dai, P., , M., Weld, D. S., and Goldsmith, J. (2011). Topological Value Iteration Algorithms. *J. Artif. Int. Res.*, 42(1):181–209.

[de Moura and Bjørner, 2008] de Moura, L. M. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340.

[Dijkstra, 1976] Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall.

[Fujita et al., 1997] Fujita, M., McGeer, P. C., and Yang, J. C.-Y. (1997). Multi-Terminal Binary Decision Diagrams: An Efficient DataStructure for Matrix Representation. *Form. Methods Syst. Des.*, 10(2-3):149–169.

[Ganzinger et al., 2004] Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., and Tinelli, C. (2004). DPLL(T): Fast Decision Procedures. In *CAV*, pages 175–188.

[Graf and Saïdi, 1997] Graf, S. and Saïdi, H. (1997). Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK. Springer-Verlag.

[Hansson and Jonsson, 1994] Hansson, H. and Jonsson, B. (1994). A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6(5):512–535.

[Hermanns et al., 2008] Hermanns, H., Wachter, B., and Zhang, L. (2008). Probabilistic CEGAR. In *CAV*, pages 162–175.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Itai and Rodeh, 1990] Itai, A. and Rodeh, M. (1990). Symmetry Breaking in Distributed Networks. *Information and Computation*, 88(1).

[Katoen et al., 2010] Katoen, J.-P., van de Pol, J., Stoelinga, M., and Timmer, M. (2010). A Linear Process-Algebraic Format for Probabilistic Systems with Data. In Gomes, L., Khomenko, V., and Fernandes, J. M., editors, *ACSD*, pages 213–222. IEEE Computer Society.

[Katoen et al., 2009] Katoen, J.-P., Zapreev, I. S., Hahn, E. M., Hermanns, H., and Jansen, D. N. (2009). The Ins and Outs of the Probabilistic Model Checker MRMC. In *Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, QEST '09, pages 167–176, Washington, DC, USA. IEEE Computer Society.

[Kattenbelt et al., 2010] Kattenbelt, M., Kwiatkowska, M., Norman, G., and Parker, D. (2010). A Game-based Abstraction-Refinement Framework for Markov Decision Processes. *Formal Methods in System Design*, 36(3):246–280.

[Kattenbelt et al., 2008] Kattenbelt, M., Kwiatkowska, M. Z., Norman, G., and Parker, D. (2008). Game-Based Probabilistic Predicate Abstraction in PRISM. *Electr. Notes Theor. Comput. Sci.*, 220(3):5–21.

[Knuth and Yao, 1976] Knuth, D. and Yao, A. (1976). *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press.

[Komuravelli et al., 2012] Komuravelli, A., Păsăreanu, C. S., and Clarke, E. M. (2012). Assume-guarantee Abstraction Refinement for Probabilistic Systems. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 310–326, Berlin, Heidelberg. Springer-Verlag.

[Kwiatkowska et al., 2011] Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of Probabilistic Real-time Systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer.

[Kwiatkowska et al., 2010] Kwiatkowska, M., Norman, G., Parker, D., and Qu, H. (2010). Assume-Guarantee Verification for Probabilistic Systems. In Esparza, J. and Majumdar, R., 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.

[Kwiatkowska et al., 2002] Kwiatkowska, M., Norman, G., and Sproston, J. (2002). Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. In Hermanns, H. and Segala, R., editors, *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'02)*, volume 2399 of *LNCS*, pages 169–187. Springer.

[Kwiatkowska et al., 2006] Kwiatkowska, M. Z., Norman, G., and Parker, D. (2006). Game-based Abstraction for Markov Decision Processes. In *QEST*, pages 157–166.

[Lacan et al., 1998] Lacan, P., Monfort, J. N., Ribal, L. V. Q., Deutsch, A., and Gonthier, G. (1998). ARIANE 5 - The Software Reliability Verification Process. In Kaldeich-Schürmann, B., editor, *DASIA 98 - Data Systems in Aerospace*, volume 422 of *ESA Special Publication*.

[Lahiri et al., 2007] Lahiri, S. K., Ball, T., and Cook, B. (2007). Predicate Abstraction via Symbolic Decision Procedures. *Logical Methods in Computer Science*, 3(2).

[McMillan, 2005] McMillan, K. L. (2005). Applications of Craig Interpolation to Model Checking. In *ICATPN*, pages 15–16.

[Minato, 1993] Minato, S.-i. (1993). Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 272–277, New York, NY, USA. ACM.

[Ossowski, 2010] Ossowski, J. (2010). *JINC: a multi-threaded library for higher-order weighted decision diagram manipulation*. PhD thesis, University of Bonn.

[Parker, 2002] Parker, D. (2002). *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham.

[Pnueli, 1985] Pnueli, A. (1985). In Transition from Global to Modular Temporal Reasoning About Programs. In Apt, K. R., editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA.

[Queille and Sifakis, 1982] Queille, J. and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. In Dezani-Ciancaglini, M. and Montanari, U., editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg.

[Rabin, 1963] Rabin, M. O. (1963). Probabilistic Automata. *Information and Control*, 6(3):230–245.

[Reiter and Rubin, 1998] Reiter, M. K. and Rubin, A. D. (1998). Crowds: Anonymity for Web Transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1):66–92.

[Roscoe et al., 1997] Roscoe, A. W., Hoare, C. A. R., and Bird, R. (1997). *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Rutten et al., 2004] Rutten, J., Kwiatkowska, M., Norman, G., and Parker, D. (2004). *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.), volume 23 of *CRM Monograph Series*. American Mathematical Society.

[Segala, 1995] Segala, R. (1995). *Modeling and Verification of Randomized Distributed Real-time Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA. Not available from Univ. Microfilms Int.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309.

[Wachter, 2011] Wachter, B. (2011). *Refined Probabilistic Abstraction*. PhD thesis, Universität des Saarlandes.

[Wachter et al., 2007] Wachter, B., Zhang, L., and Hermanns, H. (2007). Probabilistic Model Checking Modulo Theories. In *QEST*, pages 129–140.