INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Model Checking Probabilistic Systems

David João Barros Henriques

Dissertação para obtenção do Grau de Mestre em

Matemática e Aplicações

Júri

Presidente:	Prof.ª Dra. Cristina Sernadas
Orientador:	Prof. Dr. Paulo Mateus
Vogal:	Prof. Dr. Carlos Caleiro
Vogal:	Prof. Dr. Jaime Ramos

Março de 2009

Resumo

Neste trabalho, apresentamos a implementação de uma ferramenta de verificação de modelos eficiente para fórmulas de uma lógica probabilística formal (EPPL) sobre circuitos digitais não fiáveis. Para aumentar a eficiência, capitalizamos em várias propriedades específicas destas estruturas; ainda assim, o programa mantém-se muito flexível, permitindo fácil adaptação a outros modelos mais complexos.

Também é introduzido um método para minimizar problemas de espaço em verificadores de modelos sobre um subconjunto de sistemas probabilísticos representáveis por redes Bayesianas. Para tal, consideramos factorizações dos processos estocásticos associados aos espaços de probabilidades gerados pelos sistemas.

São discutidas implicações de considerar uma extensão temporal sobre a lógica; é proposto um algoritmo de verificação para o caso temporal e são apresentadas opções de implementação.

Palavras-chave: Sistemas probabilísticos, verificação de modelos, circuitos digitais, lógica temporal, lógica probabilística.

Abstract

In this work, we present the implementation of an efficient model checking tool for formulas of a formal probabilistic logic (EPPL) over non-reliable digital circuits. In order to increase efficiency, we capitalize on several specific properties of these structures; however, the tool remains very open ended, allowing for adaptation to other, more complex, models.

A method to minimize space problems on model checkers over a subset of probabilistic systems representable by Bayesian networks, is also introduced. For this, we consider factorizations of stochastic processes associated with the probability spaces generated by the systems.

Implications of considering a temporal extension to the logic are discussed, a model checking procedure is proposed for the temporal case and implementation options are presented.

Keywords: Probabilistic systems, model checking, digital circuits, temporal logic, probabilistic logic.

Acknowledgments

I would like to express my sincerest thanks to all people that, through their support, made this thesis possible.

Above all else, I must thank my family for supporting me in all senses of the word, for shaping me as I am today and for possessing an apparently unending pool of affection and patience towards me.

I would also like to thank my advisor, Professor Paulo Mateus, for his extraordinary and contagious enthusiasm and, of course, for all his technical advise and time. A word of thanks should also go to Pedro Baltazar, for informally co-advising me whenever the need arose.

My thanks extend to my closer friends, both new (Bruno Montalto, Iris Ferreira, Luís Pereira, Manuel Martins, Rui Palma) and old-time (Ana Neves, Bruno Martins, Catarina Ferreira, Diana Sequeira, João Lopes, Tomaz Saraiva), for making these last few years so enjoyable and, ultimately, teaching me far more than any course could aspire to do.

Thank you all.

Ao meu irmão

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are numerous situations where reasoning about probabilistic systems is necessary, in fields as diverse as randomized algorithms, security, distributed systems, reliability or quantum computation.

When working with these systems, considering formal logics that capture their probabilistic behavior is a good way to formulate properties they may or may not satisfy. Furthermore, the development of model checking tools for formulas in such languages over structures generated by the systems is clearly beneficial when studying the satisfaction of said properties. In [11], [14] and [15], one such logic (PCTL) and the respective model checking tool (PRISM) are proposed. These are now widely accepted and used in several applications. PCTL, however, only allows for probabilistic reasoning over transitions and, in many cases, the probabilistic reasoning must be considered over states. EPPL [17] and it's temporal extension EpCTL [20] on the other hand, are logics that allow for quantitative probabilistic reasoning about states and we will consider them in this thesis.

The main focus of this work is to detail the development and implementation of a Model Checker for EPPL over digital circuits. Traditional digital circuits model checking tools assume that each logical gate in a circuit is completely reliable, deterministically providing an output for a given input. This is an increasingly unrealistic assumption; as hardware circuits become more and more miniaturized they also become more sensitive to noise from outside sources, either macrophysical or, in more recent years, microphysical. Due to it's stochastic nature, EPPL proves ideal to model these circuits.

Besides the model checking algorithm itself, we present several relevant results concerning both the state logic and it's temporal extension. This work is very well covered in [19] and, therefore, we will provide only the general guidelines. We derive a small model theorem for EPPL which allows us to propose a PSPACE SAT algorithm for the state logic. This SAT algorithm is then used to obtain a weakly

complete Hilbert calculus for EPPL. One such calculus is also obtained for the temporal extension EpCTL and a general model checker for this logic is proposed.

The presentation is structured as follows. In Chapter 2, we present the state logic, EPPL. Syntax and semantics are introduced. Afterwards, the small model theorem is derived and the SAT algorithm described. We finish the chapter presenting the complete Hilbert calculus for EPPL.

In Chapter 3 we explain the implementation of the model checking algorithm for EPPL for digital circuits. We start by setting notation and implementation constraints. We then describe a structure that accurately models faulty digital circuits and how it relates to EPPL. A presentation of the algorithm follows, along with a proof of its soundness and completeness. An informal complexity analysis is also provided. Since efficiency concerns are taken into account, several optimization options are then discussed and their soundness is explained.

In Chapter 4, we introduce the temporal extension EpCTL much in the same way we did with EPPL in Chapter 2: We start by defining the syntax and semantics of the logic, then proceed to present a complete Hilbert calculus for it. Finally, we present the general model checking algorithm for EpCTL and discuss integration of the MC for EPPL from the previous chapter in this extended algorithm.

Finally, in the Conclusion we assess our work and what remains to be done, as well as presenting some questions for further research.

Main contributions

The contributions of the thesis are twofold:

- Model checking noisy digital circuits: we improved the EPPL model-checking algorithm from EXPSPACE in the number of propositional symbols to PSPACE by taking advantage of realistic independence assumptions used when building the circuits.
- A tool that implements the previous algorithm, that is, a model checker that receives as input an EPPL formula and an unreliable digital circuit and outputs 1 if the formula is satisfied by the circuit and 0 otherwise.

Chapter 2

State Logic - EPPL

A probabilistic logic is necessary in order to reason about probabilistic states. We will consider for this purpose the Exogenous Probabilistic Propositional Logic (EPPL) [17]. In this chapter, a succinct description and a complete Hilbert calculus for EPPL is presented, as basic knowledge of the logic is required for the following chapters. For a more complete reference on the logic, please refer to [16],[17], [19], [18].

2.1 Syntax

The syntax of EPPL follows the exogenous approach described in [18]: We start with a base language (propositional in this case) and define a language at an higher level, taking base formulas as terms.

The formulas of the base language, *basic formulae*, are simply classical propositional formulas over a finite set of propositional symbols Λ that is our abstraction of the program variables and states. We introduce a set of *probabilistic terms* that represent real numbers to allow for quantitative reasoning.

The formulas of the second level, *global formulae*, allow us to perform probabilistic reasoning over basic formulae and probabilistic terms.

The syntax of the language is given by mutual recursion as presented in Table 2.1.

$\beta := p \,[\!]\, (\neg\beta) \,[\!]\, (\beta \Rightarrow \beta)$	basic formulae
$t := z \,[\!]\, 0 \,[\!]\, 1 \,[\!]\, (\int \beta) \,[\!]\, (t + t) \,[\!]\, (t.t)$	probabilistic terms
$\delta := (\Box\beta) \,[\!]\, (t \leq t) \,[\!]\, (\sim\delta) \,[\!]\, (\delta \supset \delta)$	global formulae

where $p \in \Lambda$, $z \in Z$.

Table 2.1: Syntax of EPPL

Although we use the syntax in Table 2.1 for all theoretical purposes in this work, the syntax of formulas in the tool itself differs very slightly in order to simplify the

actual implementation. The deviations are dully explained in the Annexes.

Classical abbreviations for propositional connectives like disjunction ($\beta_1 \vee \beta_2$), conjunction ($\beta_1 \wedge \beta_2$) and equivalence ($\beta_1 \Leftrightarrow \beta_2$) are used freely throughout this work for basic formulae.

Probability terms denote elements of the real numbers. We assume a finite set of real variables, Z, ranging over elements of algebraic real numbers. Together with the constants 0 and 1, addition and multiplication, we are able to express all algebraic real numbers. Finally *measure terms*, terms of the form $(\int \beta)$, intuitively represent the measure of the set of valuations that satisfy β.

Global formulae are either of the form $(\Box\beta)$, comparison formulas $(t_1 \leq t_2)$ or built upon these by the connectives $\sim, \supset$. Formulas like $(\Box\beta)$ allow us to check if all valuations induced by the sample space satisfy β. We will use the intuitive abbreviation $(\Diamond\beta)$ for $(\sim(\Box(\neg\beta)))$. $(\Diamond\beta)$ is satisfied if there is at least one valuation induced by the probability space that satisfies β but, despite the notation, it is not a modality, since we do not have formulas such as $\Box(\Box\beta)$.

Like we did for basic connectives, we will assume $\{\cup, \cap, \equiv\}$ and comparison operators $\{=, \geq, <, >\}$ introduced as abbreviations in the classical way.

Notions of occurrence and substitution of terms t and global subformulas δ_1 in the global formula δ are defined as usual. For the sake of clarity, we shall often drop parentheses in formulas and terms if it does not lead to ambiguity.

We also introduce the following sublanguage of probabilistic state formulas which do not contain any occurrence of measure terms:

$$\begin{array}{lcl} \kappa & := & (a \leq a) \,[\!]\, (\sim\kappa) \,[\!]\, (\kappa \supset \kappa) \\ \alpha & := & z \,[\!]\, 0 \,[\!]\, 1 \,[\!]\, (\alpha + \alpha) \,[\!]\, (\alpha.\alpha) \end{array}$$

Terms of this sublanguage will be called *analytical terms* and formulas will be called *analytical formulas*. This language is relevant because it is possible to apply the SAT algorithm for the existential theory of the real numbers to any analytical formula.

2.2 Semantics

The models of EPPL are tuples $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ where $(\Omega, \mathcal{F}, \mu)$ is a probability space and $\mathbf{X} = (X_p)_{p\in\Lambda}$ is a stochastic process over $(\Omega, \mathcal{F}, \mu)$ where each X_p is a Bernoulli random variable, that is, X_p ranges over $\mathbf{2} = \{0, 1\}$. Therefore, each $\omega \in \Omega$ induces a valuation v_ω over Λ such that $v_\omega(p) = X_p(\omega)$, for all $p \in \Lambda$. In addition, each basic EPPL formula β represents the measurable subset $\{\omega \in \Omega : \beta(v_\omega) = 1\}$, where $\beta(v_\omega)$ is the denotation of β by v_ω.

Moreover, each basic EPPL formula β also induces a Bernoulli random variable $X_\beta : \Omega \to \mathbf{2}$:

- $X_{(\neg\beta)}(\omega) = 1 - X_\beta(\omega)$;
- $X_{(\beta_1 \Rightarrow \beta_2)}(\omega) = \max((1 - X_{\beta_1}(\omega)), X_{\beta_2}(\omega))$.

A simple argument of structural induction shows that $\{\omega \in \Omega : \beta(v_\omega) = 1\} = \{\omega \in \Omega : X_\beta(\omega) = 1\}$.

Given an EPPL model $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ and assignement γ for real variables, the semantics of global formulas is defined in the following way:

- Denotation of probabilistic terms:
  - $[\![z]\!]_{m,\gamma} = \gamma(z)$; $[\![0]\!]_{m,\gamma} = 0$; $[\![1]\!]_{m,\gamma} = 1$;
  - $[\![(t_1 + t_2)]\!]_{m,\gamma} = [\![t_1]\!]_{m,\gamma} + [\![t_2]\!]_{m,\gamma}$; $[\![(t_1.t_2)]\!]_{m,\gamma} = [\![t_1]\!]_{m,\gamma}.[\![t_2]\!]_{m,\gamma}$;
  - $[\![(\int \beta)]\!]_{m,\gamma} = \mu(X_\beta^{-1}(1))$ is the probability of observing an outcome ω such that v_ω satisfies β.
- Satisfaction of global formulas:
  - $m, \gamma \Vdash (\Box\beta)$ iff $\Omega = X_\beta^{-1}(1)$;
  - $m, \gamma \Vdash (t_1 \leq t_2)$ iff $[\![t_1]\!]_{m,\gamma} \leq [\![t_2]\!]_{m,\gamma}$;
  - $m, \gamma \Vdash (\sim\delta)$ iff $m, \gamma \nVdash \delta$;
  - $m, \gamma \Vdash (\delta_1 \supset \delta_2)$ iff $m, \gamma \Vdash \delta_2$ or $m, \gamma \nVdash \delta_1$.

Closed terms are defined as terms where no real variables appear. A global formula involving only closed terms is called a *closed global formula*. As the denotation of closed terms is independent of the assignement, we will drop the assignement from the notation in such cases.

Remark 2.2.1. Let $V_m = \{v_\omega : \omega \in \Omega\}$ be the set of all valuations over Λ induced by m. Consider, for each $i \in \{0,1\}^{|\Lambda|}$, the set $B_i = \{v \in V_m : v(p_1) = i_1, \ldots, v(p_{|\Lambda|}) = i_{|\Lambda|}\}$ for $p_1, \ldots, p_{|\Lambda|} \in \Lambda$ and i_n the $n-th$ bit of i. Let $\mathcal{B}_m$ be the set of all such B. Observe that an EPPL model $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ induces a probability space $P_m = (V_m, \mathcal{F}_m, \mu_m)$ over valuations, where $\mathcal{F}_m \subseteq \mathbf{2}^{V_m}$ is the σ-algebra generated by $\mathcal{B}_m$ and μ_m is defined over $\mathcal{B}_m$ by $\mu_m(B) = \mu(\{\omega \in \Omega : v_\omega \in B\})$ for all $B \in \mathcal{B}_m$. Moreover, given a probability space over valuations, $P = (V, \mathcal{F}, \mu)$, we can construct an EPPL model $m_P = (V, \mathcal{F}, \mu, \mathbf{X})$ where $X_p(v) = v(p)$. Since $X_\beta^{-1}(1) = \{\omega \in \Omega : X_\beta(\omega) = 1\} = \{\omega \in \Omega : \beta(v_\omega) = 1\}$, it is easy to see that m and m_{P_m} satisfy precisely the same formulas.

Example 2.2.2. Consider an EPPL model describing the toss of two fair coins and checking if any of them comes up heads (=1). Each coin represents a probabilistic bit, as does the checking action, that is, the set of propositional symbols is $\Lambda = \{p_1, p_2, p_3\}$, where p_1 models one coin, p_2 models the other coin and p_3 models the checking action. The outcome of tossing the two coins and checking is described by $m = (\{000, 011, 101, 111\}, 2^{\{001,011,101,111\}}, \mu, \mathbf{X})$ where $\mu(xyz) = \frac{1}{4}$, $X_{p_1}(xyz) = x$ and $X_{p_2}(xyz) = y$ and $X_{p_3}(xyz) = z$ for all $x, y, z \in \{0,1\}$ s.t. $z = max(x, y)$. It is easy to see that $m \Vdash (1 + 1 + 1 + 1)(\int p_3) = (1 + 1 + 1)$ (which we abbreviate to $m \Vdash (\int p_3) = \frac{3}{4}$).

Example 2.2.3. Consider the more sophisticated experiment of tossing a fair coin until the outcome is heads(=1's). This process can be modeled by the structure $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ over a countable infinite set of propositional symbols $\Lambda = \{p_1, \ldots, p_n, \ldots\}$, where

$$\Omega = \{\underbrace{00\ldots0}_{k}111\ldots : k \geq 0\}, \mathcal{F} = 2^{\Omega},$$

and $X_i : \Omega \to \mathbf{2}$ is the state of the coin at time $i \in \mathbb{N}$, for all $p_i \in \Lambda$, with

$$\mu(\underbrace{00\ldots0}_{k}111\ldots) = \frac{1}{2^{k+1}} \quad \text{for } k \geq 0,$$

and zero otherwise.

m is *not* an EPPL model, as Λ is infinite. If we were to push the definition and consider the same semantics, allowing infinite sets of propositional symbols, we would have, for example, $m \Vdash (\Box(p_i \Rightarrow p_{i+1}))$, for all $i \in \mathbb{N}$.

Given the configuration $\underbrace{00\ldots0}_{k}111\ldots$ we could represent it by the basic formula $((\neg p_1) \wedge (\neg p_2) \wedge \ldots \wedge (\neg p_k) \wedge p_{k+1})$, but the configuration $0000\ldots$ could not be represented by any basic formula, because we don't allow infinitary conjunctions.

To avoid this limitation, we can group all variables of some index or higher in a single variable and consider instead the model $m' = (\Omega, \mathcal{F}, \mu, \mathbf{X}')$, over $\Lambda = \{p_1, \ldots, p_n\}$, such that $(\Omega, \mathcal{F}, \mu)$ as above and $\mathbf{X}' = (X'_i : \Omega \to \mathbf{2})_{1 \leq i \leq n}$ where X_i is the state of the coin at time $1 \leq i \leq n-1$ and X'_n is zero if the coin will never be in state "heads" from time n and one otherwise.

In this case, the basic formula $\beta_0 = ((\neg p_1) \wedge \ldots \wedge (\neg p_{n-1}) \wedge (\neg p_n))$ represents the configuration $0000\ldots$ and $m' \Vdash ((\int \beta_0) \leq 0)$, but $m' \nVdash (\Box(\neg\beta_0))$.

Given that we are working towards a complete Hilbert calculus for EPPL through a SAT algorithm, it is relevant to understand whether EPPL fulfills a small model theorem. If this is the case then an upper bound on the size of the satisfying models would imply the decidability of the logic (since it would be enough to search for models up to this bound).

2.3 Small model theorem

The technique used in [10] to obtain a small model theorem is adapted in [19] for EPPL.

Let δ be an EPPL formula. We denote the sets of inequalities, basic subformulas and propositional symbols occurring in δ by $iq(\delta)$, $bsf(\delta)$ and $prop(\delta)$, respectively. Given a formula δ and an EPPL model $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$, we define the following

relation on the sample space Ω:

$$\omega_1 \sim_\delta \omega_2 \text{ iff } X_p(\omega_1) = X_p(\omega_2) \text{ for all } p \in prop(\delta)$$

Lemma 2.3.1. The relation $\sim_\delta$ is a finite index equivalence relation on Ω and if $\omega_1 \sim_\delta \omega_2$ then $X_\beta(\omega_1) = X_\beta(\omega_2)$ for all $\beta \in bsf(\delta)$.

Proof. Clearly $\sim_\delta$ is an equivalence relation since equality is an equivalence relation as well. The set $prop(\delta)$ is finite, so, it allows only a finite number of different $\sim_\delta$ classes. Let $\omega_1, \omega_2 \in \Omega$ such that $\omega_1 \sim_\delta \omega_2$.

Base: true by definition.

Step:

- In the case $(\neg\beta)$ we have
  $X_{(\neg\beta)}(\omega_1) = 1 - X_\beta(\omega_1) = 1 - X_\beta(\omega_2) = X_{(\neg\beta)}(\omega_2)$;
- In the case $(\beta_1 \Rightarrow \beta_2)$ we have
  $X_{(\beta_1\Rightarrow\beta_2)}(\omega_1) = \max(1 - X_{\beta_1}(\omega_1), X_{\beta_2}(\omega_1)) = \max(1 - X_{\beta_1}(\omega_2), X_{\beta_2}(\omega_2)) = X_{(\beta_1\Rightarrow\beta_2)}(\omega_2)$.

□

Let $prop_\omega(\delta)$ be the subset of propositional symbols of δ such that $X_p(\omega) = 1$. We denote by $[\omega]_\delta$ the $\sim_\delta$ class of ω.

Lemma 2.3.2. Let $\omega \in \Omega$,

$$[\omega]_\delta = \left(\bigcap_{p \in prop_\omega(\delta)} \{\omega' : X_p(\omega') = 1\} \right) \bigcap \left(\bigcap_{p \in prop(\delta) \setminus prop_\omega(\delta)} \{w' : X_p(\omega') = 0\} \right).$$

Moreover, $[\omega]_\delta \in \mathcal{F}$.

Proof. For the first claim, observe that if $\omega_1 \sim_\delta \omega_2$ then $prop_{\omega_1}(\delta) = prop_{\omega_2}(\delta)$. Since $(X_p)_{p\in\Lambda}$ are random variables and $\mathcal{F}$ is closed to finite intersections we prove the last claim. □

Taking an EPPL model $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ and an EPPL formula δ, we define the *quotient model* $m/\sim_\delta = (\Omega', \mathcal{F}', \mu', \mathbf{X}')$ where:

- $\Omega' = \Omega/\sim_\delta$ is the finite set of $\sim_\delta$ classes;
- $\mathcal{F}' = 2^{\Omega'}$ is the power set σ-algebra;
- $\mu'(B) = \mu(\cup B)$ for all $B \in \mathcal{F}'$;
- $X'_p([\omega]_\delta) = X_p(\omega)$ for all $p \in \Lambda$.

The quotient model is well defined.

Proposition 2.3.3. Let $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ be an EPPL model and δ an EPPL formula, then $m/\sim_\delta = (\Omega', \mathcal{F}', \mu', \mathbf{X}')$ is a finite EPPL model .

Proof. By Lemma 2.3.1, Ω' is a finite set. If $B \in \mathcal{F}'$, then $\cup B = \cup\{[s]_\delta : [s]_\delta \in B\}$ is in $\mathcal{F}$ by Lemma 2.3.2. By definition we get that $\mu'([s]_\delta) = \mu([s]_\delta)$ and $\mu'(\Omega') = \mu(\cup\Omega') = \mu(\Omega) = 1$. So, μ' is a finite probabilistic measure over Ω'. □

If we prove that satisfaction is preserved by the quotient construction, then any satisfiable formula ξ has a finite discrete EPPL model of size bounded by the formula length $|\xi|$.

Theorem 2.3.4 (Small Model Theorem)**.** If δ is a satisfiable EPPL formula then it has a finite model using at most $2|\delta| + 1$ algebraic real numbers.

Proof. Let $m = (\Omega, \mathcal{F}, \mu, \mathbf{X})$ be an EPPL model of δ and $m' = m/\sim_\delta$ its quotient model, that is, a finite discrete EPPL model of size $2^{|prop(\delta)|} \in O(2^{|\delta|})$.

In the quotient model m', we need a real number for each valuation over $prop(\delta)$, therefore we need at most $2^{|\delta|}$ real numbers. We start by proving that m' satisfies δ. Note that m and m' agree in the denotation of probabilistic terms. The only non-trivial case are the terms $(\int \beta)$. For $\beta \in bsf(\delta)$ and using Lemma 2.3.2 we get

$$[\![\int \beta]\!]_{m',\gamma} = \mu'(X'_\beta = 1) = \mu(\cup\{X'_\beta = 1\}) = \mu(X_\beta = 1) = [\![\int \beta]\!]_{m,\gamma},$$

for any assignement γ of the real variables. By structural induction on terms of δ we get that m and m' agree on inequations.

For any subformula $\Box\beta$ of δ we have that $m, \gamma \Vdash \Box\beta$ iff $\Omega = X_\beta^{-1}(1)$ iff $\Omega' = (X'_\beta)^{-1}(1)$ iff $m', \gamma \Vdash \Box\beta$. Now, since m and m' agree on inequations $(t_1 \leq t_2)$ and modal formulas $\Box\beta$, we get by structural induction that m' is a model for δ.

Finally, we will simplify m' to obtain a model $m'' = (\Omega'', 2^{\Omega''}, \mu'', \mathbf{X}'')$ of δ such that $|\Omega''| \leq 2|\delta| + 1$. Let $bsf(\delta) = \{\beta_1, \ldots, \beta_k\}$ and $\Omega'_{\beta_i} = X'^{-1}_{\beta_i}(1) \subseteq \Omega'$. Observe that $k \leq |\delta|$. Then, we can build a system of $k+1$ equations

$$\begin{cases} \sum_{\omega \in \Omega'_{\beta_1}} x_\omega = \mu'(X_{\beta_1} = 1) \\ \ldots \\ \sum_{\omega \in \Omega'_{\beta_k}} x_\omega = \mu'(X_{\beta_k} = 1) \\ \sum_{\omega \in \Omega'} x_\omega = 1 \end{cases}$$

for which we know that there is a non-negative solution $x_\omega = \mu'(\{\omega\})$ for all $\omega \in \Omega'$. From linear programming it is well known that if a system of $k+1$ linear equations has a non-negative solution, then there is a solution ρ for the system with at most $k+1$ variables taking *positive values* [4]. Then, we can construct a model m''' such that $\Omega''' = \{\omega \in \Omega' : \rho(x_\omega) > 0\}$ and $\mu'''(\{\omega\}) = \rho(x_\omega)$. Observe that

$m''' \Vdash (t_1 \leq t_2)$ iff $m' \Vdash (t_1 \leq t_2)$ for each inequation $(t_1 \leq t_2)$ occurring in δ. However, it might be the case that $m''' \Vdash \Box\beta$ and $m' \nVdash \Box\beta$ for some subformula $\Box\beta$ of δ, since $\Omega''' \subseteq \Omega'$. Then, for each subformula $\Box\beta$ of δ such that $m' \nVdash \Box\beta$ and $m''' \Vdash \Box\beta$ there exists $\omega_\beta \in \Omega' \setminus \Omega'''$ such that $v_{\omega_\beta} \nVdash \beta$. We can now construct the model m'' where

$$\Omega'' = \Omega''' \cup \{\omega_\beta \in \Omega' \setminus \Omega''' : m' \nVdash \Box\beta \text{ and } m''' \Vdash \Box\beta\},$$

$$\mu''(\omega) = \begin{cases} \mu'''(\omega) & \text{if } \omega \in \Omega''' \\ 0 & \text{otherwise} \end{cases}$$

and $X''_p(\omega) = X'_p(\omega)$ for all $\omega \in \Omega''$. Clearly, $|\Omega''| \leq 2|\delta| + 1$ and $m'' \Vdash \delta$. Finally, from the first order theory of real ordered fields, if there is a model using real numbers for a real closed formula, then there is a model using only algebraic real numbers [1] (since the logic cannot specify transcendental real numbers with a single formula – we need an infinite number of formulas to be able to specify a transcendental real number), so the solution of the system can be made just with algebraic real numbers. □

The size of the representation of the algebraic real numbers can increase without any bound. Fortunately, thanks to the fact that the existential theory of the real numbers can be decided in PSPACE, we find a bound on the size of the real representations in function of the size of the formula, which will lead to a SAT algorithm for EPPL.

2.4 Decision algorithm for EPPL satisfaction

Given an EPPL formula δ we will denote by $iq(\delta)$ the set of all subformulas of δ of the form $(t_1 \leq t_2)$, by $bf_\Box(\delta)$ the set of all subformulas of δ of the form $\Box\beta$ and $at(\delta) = bf_\Box(\delta) \cup iq(\delta)$. By an *exhaustive conjunction ε of literals of $at(\delta)$* we mean that ε is of the form $\alpha_1 \cap \cdots \cap \alpha_k$ where each α_i is either a global atom or a negation of a global atom. Moreover, all global atoms or their negation occur in ε, so, $k = |at(\delta)|$. Given a global formula δ, we denote by $\delta^\alpha_{p_\alpha}$ the propositional formula obtained by replacing in δ each global atom α with a fresh propositional symbol p_α, and replacing the global connectives $\sim$ and $\supset$ by the propositional connectives $\neg$ and $\Rightarrow$, respectively. We denote by v_ε the valuation over propositional symbols p_α such that $v_\epsilon(p_\alpha) = 1$ iff α occurs not negated in ε.

Given an exhaustive conjunction ε of literals of $at(\delta)$, we denote by $lbf_\Box(\varepsilon)$ the set of basic formulas such that $\beta \in lbf_\Box(\varepsilon)$ if $\Box\beta$ occurs positively in ε (that is, not negated). Similarly, the set of basic formulas that occur nested by a $\sim\Box$ in ε is denoted by $lbf_{\Diamond\neg}(\varepsilon)$. Finally, we denote all the inequality literals occurring in ε by $liq(\varepsilon)$. Given a global formula α in $liq(\varepsilon)$ we denote by $\alpha^{(\int\beta)}_{\sum\{x_v : v \in V, v \Vdash \beta\}}$ the analytical formula where all terms of the form $(\int\beta)$ are replaced in α by $\sum_{v\in V, v\Vdash\beta} x_v$ where

each x_v is a fresh variable. We need a PSPACE SAT algorithm of the existential theory of the reals numbers, that we denote by $SatReal$. We assume that this algorithm either returns *no model*, if there is no solution for the input system of inequations, or a *solution array* ρ, where $\rho(x)$ is the solution for variable x. We denote by $var(\delta)$ the set of real logical variables that occur in δ. Given a solution ρ for a system with X variables and a subset $Y \subseteq X$, we denote by $\rho_{|Y}$ the function that maps each element y of Y to $\rho(y)$.

Algorithm 1: SatEPPL(δ)

Input: EPPL formula δ

Output: (V, μ) (denoting the EPPL model $m = (V, 2^V, \mu, \mathbf{X})$) and assignement γ or *no model*

compute $bf_\Box(\delta), iq(\delta)$ *and* $at(\delta)$*;*
foreach exhaustive conjunction ε of literals of $at(\delta)$ such that $v_\varepsilon \Vdash \delta^\alpha_{p_\alpha}$ **do**
  compute $lbf_\Box(\varepsilon)$, $lbf_{\Diamond\neg}(\varepsilon)$ and $liq(\varepsilon)$*;*
  foreach $V \subseteq 2^{prop(\delta)}$ such that $0 < |V| \leq 2|\delta| + 1$, $V \Vdash \wedge lbf_\Box(\varepsilon)$ and $V \not\Vdash \beta$ for all $\beta \in lbf_{\Diamond\neg}(\varepsilon)$ **do**
    $\kappa \longleftarrow \left(\sum_{v\in V} x_v = 1\right) \cap \left(\bigcap_{v\in V} 0 \leq x_v\right)$;
    foreach $\alpha \in liq(\varepsilon)$ **do**
      $\kappa \longleftarrow \kappa \cap \alpha^{(\int \beta)}_{\sum\{x_v : v\in V, v\Vdash\beta\}}$;
    end
    $\rho \longleftarrow SatReal(\kappa)$;
    if $\rho \neq$ *no model* **then**
      $\mu_\rho \longleftarrow \rho_{|\{x_v : v\in V\}}$;
      $\gamma_\rho \longleftarrow \rho_{|var(\delta)}$;
      return (V, μ_ρ) and assignement γ_ρ;
    end
  end
end
return (*no model*);

We claim that Algorithm 1 decides the satisfiability of an EPPL formula in PSPACE and to support this claim, we explain the algorithm and its soundness:

Given an EPPL formula δ, we start by computing (line 1) its global atoms $bf_\Box(\delta), iq(\delta)$ and $at(\delta)$. This sets take $O(|\delta|)$ space to store.

Now, we cycle over all exhaustive conjunctions ε such that $v_\varepsilon \Vdash \delta^\alpha_{p_\alpha}$. Observe that if δ has a model m, γ then, for all $\delta' \in at(\delta)$ this model is such that either $m, \gamma \Vdash \delta'$ or $m, \gamma \not\Vdash \delta'$. Therefore, there is an exhaustive conjunction ε such that m, γ is a model of ε and, in this case, $v_\varepsilon \Vdash \delta^\alpha_{p_\alpha}$. On the other hand, if δ has no model, then all ε such $v_\varepsilon \Vdash \delta^\alpha_{p_\alpha}$ have no model. Hence, to find a model for δ it is enough to find a model for an ε such that $v_\varepsilon \Vdash \delta^\alpha_{p_\alpha}$. At each step of the cycle of the second line of the algorithm, we only need to store one such ε, which requires only

polynomial space.

To check if there is an EPPL model for ε we start by computing $lbf_{\Box}(\varepsilon), lbf_{\Diamond\neg}(\varepsilon)$ and $liq(\varepsilon)$, which can be stored in $O(|\varepsilon|)$. Given Remark 2.2.1, it is enough to check for models where the state space Ω is given as a set of valuations of the basic propositional symbols. Moreover, thanks to the small model theorem (Theorem 2.3.4), it is enough to search for sets of valuations V such that $|V| \leq 2|\delta| + 1$. V has to satisfy the modal literals $\Box\beta$ and $\sim\Box\beta$ occurring in ε, that is: for all $\beta \in lbf_{\Box}(\varepsilon)$ and $v \in V$ we have that $v \Vdash \beta$, which can be written as $V \Vdash \wedge lbf_{\Box}(\varepsilon)$ and for all $\beta \in lbf_{\Diamond\sim}(\varepsilon)$ there exists $v \in V$ such that $v \nVdash \beta$ wich can be written as $V \nVdash \beta$ for all $\beta \in lbf_{\Diamond\neg}(\varepsilon)$. These are exactly the conditions in the guard of the second cycle of the program. In the body of this cycle we will check if there is a model of ε taking such V as the set of states, that is, if there is a solution for the inequations in $liq(\epsilon)$. Since we only have to store a set of valuations V with $|V| \leq 2|\delta| + 1$ at each step of the cycle, once again we need only polynomial space.

Next, we search for a model of the inequations in $liq(\varepsilon)$ having a set of states V. To this end we consider a fresh real logical variable x_v for each $v \in V$ representing its probability. The idea behind this step is to build an analytical formula κ that specifies the two probability constrains expressed in the fifth line and the inequations in $liq(\varepsilon)$. Two lines further, the formula κ is finished by replacing the terms $(\int \beta)$ in $liq(\varepsilon)$ by $\sum_{v \in V : v \Vdash \beta} x_v$. In line 9 we call the *SatReal* algorithm for a solution (model) to κ. Since $|\kappa|$ is polynomial on $|\delta|$ and the set of variables in κ is polynomially bounded by $|\delta|$, the *SatReal* will compute the solution in PSPACE. If such solution ρ exists, we have succeeded in finding a model for δ. Hence, we return (V, μ_ρ) and γ_ρ, where $\mu_\rho(v)$ is $\rho(x_v)$ and γ_ρ is the restriction of ρ to $var(\delta)$. As stated in Remark 2.2.1, this is enough to construct an EPPL model. If there is no solution ρ then we cannot find a solution for the set V of valuations, and have to try with another V. Finally, if for all ε and V we are not able to find a solution, then there is no model for δ.

2.5 Completeness of EPPL

It is shown in [18] that EPPL is not compact, therefore, it is impossible to obtain a strongly complete axiomatization for EPPL . Nevertheless, weak completeness is enough for verification purposes, since a program specification generates a finite number of hypothesis. The EPPL SAT algorithm allows us to show that the calculus presented in Table 2.2 is weakly complete.

The soundness of the calculus of Table 2.2 is straightforward, and so, we focus on the completeness result.

Theorem 2.5.1. The set of rules and axioms of Table 2.2 is a weakly complete axiomatization of EPPL.

Table 2.2: Complete Hilbert calculus for EPPL

Axioms:

- [**CTaut**] $\vdash_{\mathsf{EPPL}} (\Box\beta)$ for each valid propositional formula β;
- [**GTaut**] $\vdash_{\mathsf{EPPL}} \delta$ for each instantiation of a propositional tautology δ;
- [**Lift** $\Rightarrow$] $\vdash_{\mathsf{EPPL}} (\Box(\beta_1 \Rightarrow \beta_2) \supset (\Box\beta_1 \supset \Box\beta_2))$;
- [**Eqv** $\bot$] $\vdash_{\mathsf{EPPL}} (\Box\bot \Leftrightarrow \perp\!\!\!\perp)$;
- [**RCF**] $\vdash_{\mathsf{EPPL}} (t_1 \leq t_2)$ for each instantiation of a valid analytical inequality;
- [**Prob**] $\vdash_{\mathsf{EPPL}} ((\int \top) = 1)$;
- [**FAdd**] $\vdash_{\mathsf{EPPL}} (((\int(\beta_1 \wedge \beta_2)) = 0) \supset ((\int(\beta_1 \vee \beta_2)) = (\int \beta_1) + (\int \beta_2)))$;
- [**Mon**] $\vdash_{\mathsf{EPPL}} (\Box(\beta_1 \Rightarrow \beta_2) \supset ((\int \beta_1) \leq (\int \beta_2)))$;

Inference Rules:

- [**Mod**] δ_1, $(\delta_1 \supset \delta_2) \vdash_{\mathsf{EPPL}} \delta_2$.

Proof. As is common in completeness results, we use a contrapositive argument: if $\nvdash \delta$ then $\nVdash \delta$. A formula δ is said to be *consistent* if $\nvdash \sim\delta$. So, if we prove that every consistent formula δ has a model we get the completeness result. Observe that if $\nvdash \delta$ then $\nvdash \sim\sim\delta$, that is, $\sim\delta$ is consistent. Therefore, if we can prove it has a model, $\nVdash \delta$.

So, we will prove that every consistent formula δ has a model. Assume by contradiction that δ is consistent and the SAT algorithm returns *no model*. Let $A = \{\varepsilon \text{ exhaustive conjunction of literals } : v_\varepsilon \Vdash \delta^\alpha_{p_\alpha}\}$. By the completeness of propositional logic it follows that $\vdash (\vee_{\varepsilon\in A}\varepsilon^\alpha_{p_\alpha}) \Leftrightarrow \delta^\alpha_{p_\alpha}$, and by **GTaut** we have $\vdash \cup A \equiv \delta$. If δ is consistent then there is ε consistent, and if δ has no model, then the consistent ε has no model as well. If the SAT algorithm returns *no model* for ε it must be because of one of the following two causes: (i) it cannot find a V at the second **foreach**; (ii) for all viable V the *SatReal* algorithm returns *no model*. We will now show that for both cases we can contradict the consistency of ε.

In case (i) – no V can be found at the second **foreach** – it cannot be because $|V| > 2|\delta|+1$, thanks to the small model theorem. This means that if we remove the bound $0 < |V| \leq 2|\delta|+1$, and consider all possible sets of valuations the algorithm would also fail. In particular, take $V = 2^{prop(\delta)}$, it must happen (a) $V \nVdash \wedge lbf_\Box(\varepsilon)$ *or* (b) $V \Vdash \beta$ for some $\beta \in lbf_{\Diamond\neg}(\varepsilon)$. For case (a) we have that $\nVdash \wedge lbf_\Box(\varepsilon)$ and so, $\nVdash \beta$ for some $\beta \in lbf_\Box(\varepsilon)$, or equivalently $\Vdash \beta \Rightarrow \bot$. But by completeness of the propositional calculus we have that $\vdash \beta \Rightarrow \bot$, by **CTaut** we have that $\vdash \Box(\beta \Rightarrow \bot)$ and by **Lift**$\Rightarrow$ and **Eqv**$\bot$ we have that $\vdash \sim(\Box\beta)$ from which follows $\vdash \sim\varepsilon$ which contradicts the consistency of ε. In case (b) there is $\beta \in lbf_{\Diamond\neg}(\varepsilon)$ such that β is a tautology. Then, by **CTaut**, $\vdash \Box\beta$. From the last derivation we get $\vdash \sim\varepsilon$, which contradicts the consistency of ε.

In case (ii), the algorithm returns no model for all viable V computed in the

second **foreach**. Thanks to the small model theorem it means that the algorithm would also return no model for all V such that

$$V \Vdash \wedge lbf_{\Box}(\varepsilon) \text{ and } V \nVdash \beta \text{ for all } \beta \in lbf_{\Diamond\neg}(\varepsilon), \tag{2.1}$$

independently of the bound on the size of V. The sets of valuations satisfying (2.1) are closed for unions, and therefore there is the largest V fulfilling (2.1), say $V_{\max}$, and for this set the algorithm would return no model. Let $V^c = 2^{prop(\delta)} \setminus V_{\max}$, since ε is consistent it is easy to see that $\varepsilon' = \varepsilon \cap ((\cap_{v \in V^c} \Box\neg\beta_v))$ is consistent, where β_v is a propositional formula that is satisfied only by valuation v. Therefore, $\vdash \varepsilon' \supset \varepsilon$. Moreover, $\vdash \wedge lbf_{\Box}(\varepsilon) \Rightarrow \neg\beta_v$ for all $v \in V^c$, from which we derive that

$$\vdash (\cap_{\beta \in lbf_{\Box}(\varepsilon)} \Box\beta) \supset (\cap_{v \in V^c} \Box\neg\beta_v)$$

and so $\vdash \varepsilon \supset \varepsilon'$ and therefore $\vdash \varepsilon' \equiv \varepsilon$. Thus, if ε is consistent then ε' is also consistent, and if there is no model for ε then there is no model for ε' as well, and the algorithm will fail precisely in line where it returns a model.

By **RCF** we have $\vdash \sim\kappa^{x_v}_{(\int \beta_v)}$, where $\kappa^{x_v}_{(\int \beta_v)}$ is the formula κ where we replace each variable x_v by the term $\int \beta_v$ with β_v a propositional formula that is satisfied only by v. By **Prob**, **FAdd** and **Mon** we have $\vdash (\Box\neg\beta_v) \supset ((\int \beta_v) = 0)$, thus we can derive

$$\vdash \varepsilon' \supset (\cap_{v \in V^c}((\int \beta_v) = 0)). \tag{2.2}$$

From $\vdash \sim\kappa^{x_v}_{(\int \beta_v)}$ and **FAdd** and **RCF** we obtain that

$$\vdash \cap_{v \in V^c}((\int \beta_v) = 0) \supset \sim \cap_{\alpha \in liq(\delta)} \alpha. \tag{2.3}$$

Finally, by **CTaut** we have

$$\vdash \sim \cap_{\alpha \in liq(\delta)} \alpha \supset \sim\varepsilon'. \tag{2.4}$$

So, from (2.2), (2.3) and (2.4) we obtain with tautological reasoning $\vdash \varepsilon' \supset \sim\varepsilon'$ from which we conclude $\vdash \sim\varepsilon'$. This contradicts the consistency of ε' and thus, the consistency of δ. For this reason there must be a model for ε' and consequently, a model for δ. □

Chapter 3

Verifying Digital Circuits with EPPL

In this section, we discuss a model checker for EPPL with applications to simple unreliable digital circuits. Its actual implementation in the C programming language is presented in the appendix. To model unreliable digital circuits we devise and propose the concept of *probabilistic Boolean circuits* (PBC). A PBC is able to model any digital circuit with faulty logical gates, that is gates that are prone to an error with a certain known probability, which is reasonable, since actual hardware components explicitly state their reliability. We will show how EPPL can be used to achieve this goal.

3.1 Notation and conventions

At this point, it is convenient to make a disambiguation: for the rest of this chapter, the word "Model" (capitalized) will be used to denote EPPL models, whereas the word "model" (not capitalized) will be used without this connotation.

It has already been seen that each $\omega \in \Omega$ induces a valuation v_ω over Λ. In this chapter, this association will be injective, which amounts to saying that the sample space Ω is finite and for each valuation $v \in 2^\Lambda$, there is at most one $\omega_v \in \Omega$. This is a reasonable assumption because one only works with variables of the stochastic process $\mathbf{X}$ or continuous functions of these variables and is, therefore, unable to distinguish between different elements of Ω that induce the same valuation. Furthermore, as already observed in Remark 2.2.1, given any Model, there is always a Model over valuations that satisfies the same formulas. Due to this identification, the elements of the sample space will be abusively referred to as *valuations*. We will also assume that there are no valuations in Ω with zero measure, so it is possible that some valuations are not represented in Ω.

Under these assumptions, knowledge of the joint distribution of the random variables X_{p_i} is enough to describe an EPPL Model. Given $P_{(X_{p_1},\ldots,X_{p_n})}(X_{p_1},\ldots X_{p_n})$,

- $\Omega = \{x \in 2^{\Lambda} : P(X_{p_1} = x_1, ..., X_{p_n} = x_n) > 0\}$,
- $\mathcal{F} = 2^{\Omega}$,
- $\mu(\{\omega\}) = P(X_{p_1} = \omega_1, ..., X_{p_n} = \omega_n)$, Ω being finite, it is enough to define μ for each singleton set $\{\omega\} \subset \Omega$.

Since we have to deal with computer representation, probabilities will be represented by floating points and not symbolically by algebraic real numbers. This is not a major theoretical problem, as floating point numbers represent rational numbers, which are algebraic numbers; however, issues of lack of precision for using floating point representation cannot be easily avoided. In this work we do not preform an error analysis due to floating points (which is also the case for other model-checkers, like PRISM). However, we do try to minimize the number of operations that manipulate probabilities in order to slow the propagation of these errors.

3.2 Probabilistic Boolean circuits

Boolean circuits are the standard choice to model digital circuits. However, they lack structure to account for error in logical gates, which happens in practice. The following definition builds upon the concept of Boolean circuit, allowing a probabilistic error:

Definition 3.2.1. A *probabilistic Boolean circuit* (*PBC*) is a directed acyclic graph where each vertex i is labeled with a Bernoulli random variable R_i with expected value r_i, a fresh Boolean variable x_i and a Boolean function $g_i : \{0,1\}^{k+1} \to \{0,1\}$ with domain in the variables labeling vertices pointing to i.

A vertex whose indegree is zero is called an input vertex, and its labeling variable is called an input variable. A vertex whose outdegree is zero is called an output vertex, and its labeling variable is called an output variable. A vertex that is neither an input vertex nor an output vertex is called an internal vertex. If there is an arrow pointing from vertex i to vertex j, j is said to be a *child* of i, and i is said to be a *parent* of j. The set of parent vertices of a vertex i is denoted by $par(i)$.

Each non-input vertex i represents a logical gate that computes the Boolean function expressed by the formula f_i. This computation returns the correct output with probability r_i. The variable x_i does the double duty of identifying the vertex and representing the outcome of this (probabilistic) computation.

For implementation purposes, we notice that in order to specify a vertex, we only need the associated expected value r_i (that we will represent with a floating point between 0 and 1), the variable x_i and the Boolean function f_i (that we will

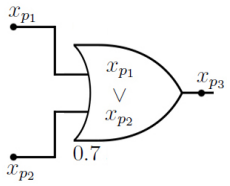

Figure 3.1: Sample gate

represent by a formula in the obvious way). We will henceforth adopt these syntactic representations of PBCs, which we shall call *Specifications of probabilistic Boolean circuit* (SPBC).

An arrow of a PBC or SPBC represents the dependency relation between the head (where the arrow points to) and the tail (where the arrow comes from): one needs the values of the propositional symbols in order to compute the value of the labeling function.

We will make use of graphical representations of SPBCs borrowing from standard notation of Boolean circuits and extending upon it: each vertex will be represented by a box either labeled with the function f_i or with a characteristic shape for widely used connectives. The expected value of the Bernoulli random variable R_i will be written under the box and it's labeling variable, representing the output of the probabilistic computation will be, as usual, placed after the box. The lines entering from the left of the box represent the outputs of other gates that are to be used as inputs. Consider, as an example, Figure 3.1: It represents a gate that computes $f_3(x_1, x_2) = max(x_1, x_2)$ (where x_1 and x_2 are given as input, coming from similar gates and f_3 is represented by $x_1 \vee x_2$), x_3 represents the result of the probabilistic computation that returns the value of f_3 with probability 0.7 and $(1-f_3)$ otherwise.

SPBCs play a central role of this work because each SPBC generates an EPPL Model where the stochastic process $\mathbf{X}$ is composed of random variables induced by the vertices, their labeling functions and their labeling real numbers. Essentially, $P(X_{p_1} = x_1, .., X_{p_n} = x_n)$ represents the probability of vertices taking the configuration $\langle x_1, ..., x_n \rangle$. Furthermore, as will be shown, these Models can be efficiently stored and checked. We will frequently refer to vertex i of a PBC by it's induced random variable X_{p_i}.

Example 3.2.2. Recall the EPPL model introduced by Example 2.2.2. It is simple to see that the SPBC presented in Figure 3.2 induces this model. Observe that all the relevant information is much more efficiently encoded in the PBCS than it would be if we explicitly wrote out the probability function.

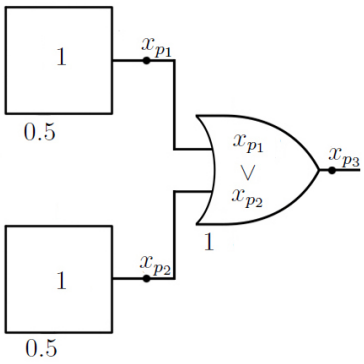

Figure 3.2: SPBC for double coin tossing example

3.3 Factorizations and Bayesian networks

In general, an EPPL Model representation requires an exponential space in $|\Lambda|$ (which we will denote throughout this section by n, unless stated otherwise). This is due to the existence of Models where each pair of propositional symbols are dependent; for such Models, even the most efficient representation would have a worst case scenario that would require writing out explicitly all of the values corresponding to the probability attributed to each valuation, occupying $O(2^n)$ space.

Fortunately, SPBCs are built in such a way that some degree of independence is maintained between the variables. We will explore this property in order to get an efficient representation of the EPPL model associated to each PBC. For this, we will need a very simple proposition.

Lemma 3.3.1. Let $X_1, ...X_n$ be discrete random variables. Then, for all $\langle x_1, ..., x_n\rangle \in \Omega$,

$$P(X_1 = x_1, ..., X_n = x_n) = \prod_{i=1}^{n} P^*(X_i = x_i | X_{i-1} = x_{i-1}, ..., X_1 = x_1),$$

where

$$P^*\left(X_i = x_i | X_{i-1} = x_{i-1}, ..., X_1 = x_1\right) = \\ = \begin{cases} P(X_i = x_i | X_{i-1} = x_{i-1}, ..., X_1 = x_1) & \text{if } P(X_{i-1} = x_{i-1}, ..., X_1 = x_1) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Proof. The proof is presented by induction in n.

Base: $P(X_1 = x_1) = P^*(X_1 = x_1)$

Step: If $P(X_1 = x_1, ..., X_{n-1} = x_{n-1}) = 0$, then $P(X_1 = x_1, ..., X_n = x_n) = 0$ because $\{\omega : X_1(\omega) = x_1, ...X_n(\omega) = x_n\} \subset \{\omega : X_1(\omega) = x_1, ..., X_{n-1}(\omega) = x_{n-1}\}$.

If $P(X_1 = x_1, ..., X_{n-1} = x_{n-1}) \neq 0$, then $P(X_1 = x_1, ..., X_n = x_n) = P^*(X_n = x_n | X_1 = x_1, ..., X_{n-1} = x_{n-1}) P(X_1 = x_1, ..., X_{n-1} = x_{n-1})$ by definition of con-

ditioned probability. By induction hypothesis, $P(X_1 = x_1, ..., X_{n-1} = x_{n-1}) = \prod_{i=1}^{n-1} P^*(X_i = x_i | X_{i-1} = x_{i-1}, ..., X_1 = x_1)$. Therefore, $P(X_1 = x_1, ..., X_n = x_n) = \prod_{i=1}^{n} P^*(X_i = x_i | X_{i-1} = x_{i-1}, ..., X_1 = x_1)$. □

We shall henceforth drop the superscript in P^* whenever no ambiguity arises. A rewriting of a joint distribution function in the above form is called a *factorization.*

Since representing the conditional distributions $P(X_{p_i} | X_{p_{i-1}}, ..., X_{p_1})$ in the worst case may take $O(2^i)$ space, using factorizations is not, in general, an efficient way of representing EPPL models, as we could need $\sum_{i=1}^{n} O(2^i) = O(2^{n+1})$ space. However, in a SPBC, each variable induced by a vertex depends exclusively on the variables of vertices that point to it, and so $P(X_{p_i} | X_{p_{i-1}}, ..., X_{p_1}) = P(X_{p_i} | par(X_{p_i}))$.

The structure we just described is known as a *Bayesian network*:

Definition 3.3.2. A *Bayesian network* relative to a set of random variables is a directed acyclic graph where each vertex is labeled with one of the variables and the joint distribution of those variables can be written as the product of the conditioned probability of each vertex and its parents:

$P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i | par(X_i))$.

Bayesian networks have the expressive power to determine all EPPL Models since one can always consider the Bayesian network where the parents of the vertex labeled with X_i are all vertexes with label $X_k, k < i$, effectively performing a factorization. However, we have seen that, in the worst case, there are no gains in space representation from using this approach relative to using the explicit description of the probability distribution. Nevertheless, Bayesian networks are a well studied object and perhaps some results from their general theory could yield interesting consequences when studying SPBCs.

Fortunately, SPBCs have another property that further reduces the complexity of the Model Checking algorithm: the probability of correctly computing the Boolean functions expressed by the labeling formulas, i.e. $P(X_{p_i} = \varphi_i(par(X_{p_i}))$, is known. With this information, we can prove the following result:

Theorem 3.3.3. Let $(X_{p_1}, ..., X_{p_n})$ be the variables induced by the vertices of a SPBC and $\langle x_1, ..., x_n \rangle$ a valuation, then:

$$P(X_{p_1} = x_1, ..., X_{p_n} = x_n) = \prod_{i=1}^{n} r_i \delta_{x_i,\varphi_i} + (1 - r_i)\delta_{1-x_i,\varphi_i}$$

where r_i is the real number labeling i and $\delta_{x_i,\varphi_i} = \begin{cases} 1 & \text{if } x_i = \varphi_i(x_1, ..., x_{i-1}) \\ 0 & \text{if } x_i \neq \varphi_i(x_1, ..., x_{i-1}) \end{cases}$

Proof. In this proof, we denote by $par(X_{p_i}) = par(x_i)$ the set $\{x \in \Omega : X_{p_i} = x_i, X_{p_i} \in par(X_{p_i})\}$

In view of Lemma 3.3.1, it suffices to prove that $P(X_{p_i} = x_i | X_{p_{i-1}} = x_{i-1}, ..., X_{p_1} = x_1) = r_i \delta_{x_i,\varphi_i} + (1 - r_i)\delta_{1-x_i,\varphi_i}$.

Using the Total Probabilities theorem with the partition $\Omega = \Omega_{x_i=\varphi_i} \cup \overline{\Omega_{x_i=\varphi_i}}$ where $\Omega_{x_i=\varphi_i} = \{x \in \Omega : x_i = \varphi_i(x_{i-1}, .., x_1)\}$, we have:

$$\begin{gathered} P\big[X_{p_i} = x_i | X_{p_{i-1}} = x_{i-1}, ..., X_{p_1} = x_1\big] = P\left[X_{p_i} = x_i | par(X_{p_i}) = par(x_i)\right] = \\ P\left[X_{p_i} = \varphi_i(par(X_{p_i}))\right] \times P\left[X_{p_i} = x_i | par(X_{p_i}) = par(x_i), X_{p_i} = \varphi_i(par(X_{p_i}))\right] + \\ P\left[X_{p_i} \neq \varphi_i(par(X_{p_i}))\right] \times P\left[X_{p_i} = x_i | par(X_{p_i}) = par(x_i), X_{p_i} \neq \varphi_i(par(X_{p_i}))\right] = \\ r_i \times P\left[x_i = \varphi_i(par(x_i))\right] + (1 - r_i) \times P\left[x_i \neq \varphi_i(par(x_i))\right] \end{gathered}$$

and

$$P\left[x_i = \varphi_i(par(x_i))\right] = \begin{cases} 1 & \text{if } x_i = \varphi_i(x_1, ..., x_{i-1}) \\ 0 & \text{if } x_i \neq \varphi_i(x_1, ..., x_{i-1}) \end{cases} = \delta_{x_i,\varphi_i},$$

$$P\left[x_i \neq \varphi_i(par(x_i))\right] = \begin{cases} 0 & \text{if } x_i = \varphi_i(x_1, ..., x_{i-1}) \quad \text{i.e. } (1 - x_i) \neq \varphi_i(x_1, ..., x_{i-1}) \\ 1 & \text{if } x_i \neq \varphi_i(x_1, ..., x_{i-1}) \quad \text{i.e. } (1 - x_i) = \varphi_i(x_1, ..., x_{i-1}) \end{cases}$$

$= \delta_{(1-x_i),\varphi_i}$,

□

This Theorem provides a very simple way of computing the probability of any given valuation, as we need only to check, for each vertex, if the denotation of the labeling formula by the valuation yields the same value as the denotation of the induced variable by the valuation, and return the real number labeling the vertex or its complement, according to the result. The product of all such values will be the desired probability.

3.4 The basic algorithm

We now describe the basic algorithm for model checking EPPL formulas in EPPL models induced by SPBCs.

Let δ be an EPPL global formula. In the following algorithm, the arrays $bf_{\Box}(\delta) = \{\lambda_1, ..., \lambda_k\}$,$sub_{\int}(\delta) = \{m_1, ..., m_s\}$, $pst(\delta) = \{t_1, ..., t_r\}$,$gsf(\delta) = \{\delta_1, ..., \delta_u, \delta\}$ denote, respectively, the list of subformulas of δ of the form $\Box\beta$, the list of measure subterms of δ (of the form $\int \beta$), the list of probabilistic subterms of δ that are not measure terms, the ordered tuple of basic subformulas of δ.

The symbol ω^i represents the $i-th$ valuation in 2^{Λ} for some enumeration of valuations over Λ.

$\beta(\omega^i)$ represents the algorithm that computes the denotation of β by ω^i, which is well known to take $O(|\beta_i|)$ time.

$fact(\omega^i)$ represents the algorithm that computes $P(X_{p_1} = \omega_1^i, ..., X_{p_n} = \omega_n^i)$. As we can see from Theorem 3.3.3, this algorithm takes $O(|fact|)$ time, where $|fact|$ is the size of the factorization induced by the PBC. Observe that $O(|fact|) = O(n \cdot |\beta|)$, with $|\beta|$ being the length of the largest labeling formula of the PBC and n the number of propositional connectives in Λ. One should keep in mind that whenever this algorithm returns 0, it should be interpreted as "$\omega^i \notin \Omega$", because we have no elements with measure zero in Ω.

The first part of the algorithm runs through all subformulas of the form $\Box\beta_i$ and through all valuations checking, for each valuation, both if it is in Ω and if it does *not* satisfy the basic formula β_i. If both answers are positive for any valuation, there is $\omega \in \Omega$ such that $\omega \notin X_{\beta}^{-1}(1)$; otherwise, $\Omega = X_{\beta}^{-1}(1)$. The values of $\Box\beta$ are stored in a Boolean $|bf_{\Box}(\delta)|$ array B.

Then, the algorithm runs through all subterms of the form $\int \beta_i$ and through all valuations, checking for each valuation if it is in Ω and if it satisfies the basic formula β_i. If both answers are positive, it then adds the measure of the valuation to $M(i)$, the $i-th$ entry of the real $|sub_{\int}(\delta)|$ array M, where the values of $\int \beta$ are stored.

The third part performs all other term evaluations, it stores them in a real $|pst(\delta)|$ array T.

Finally, the algorithm evaluates all global subformulas to a Boolean $|gsf(\delta)|$ array G, where $G(i) = 1$ iff $\chi_{fact}, \gamma \Vdash \delta_i$, for all $1 \leq i \leq |gsf(\delta)|$, where χ_{fact} is the model induced by $fact$ and return as output $G(|gsf|)$.

Algorithm 2: CheckEPPL($fact, \gamma, \delta$)

Input: Factorization $fact$, assignement γ and a formula δ

Output: Boolean value $G(|gsf(\delta)|)$

for $i = 1$ **to** $|bf_{\Box}(\delta)|$ **do**
  λ_i is $\Box\beta$;
  $B(i) = 1$;
  for $j = 1$ **to** 2^n **do**
    $If(\beta(\omega^j) == 0$ && $fact(\omega^j) > 0)$ $\{B(i) = 0; break;\}$;
  end
end
for $j = 1$ **to** $|sub_{\int}(\delta)|$ **do**
  m_i is $\int \beta$;
  $M(i) = 0$;
  for $j = 1$ **to** 2^n **do**
    $If(\beta(\omega^j) == 1)$ $\{M(i) = M(i) + fact(\omega^j);\}$;
  end
end
for $i = 1$ **to** $|pst(\delta)|$ **do**
  switch t_i **do**
    case $z:$ $T(i) = \gamma(z)$:;
    case 0 **or** $1:$ $T(i) = t_i$;
    case $(t_j + t_l):$ $T(i) = T(j) + T(l)$;
    case $(t_j.t_l):$ $T(i) = T(j).T(l)$;
    case $(m_j + t_l):$ $T(i) = M(j) + T(l)$;
    case $(m_j.t_l):$ $T(i) = M(j).T(l)$;
    case $(t_j + m_l):$ $T(i) = T(j) + M(l)$;
    case $(t_j.m_l):$ $T(i) = T(j).M(l)$;
  end
end
for $i = 1$ **to** $|gsf(\delta)|$ **do**
  switch δ_i **do**
    case $(\Box\beta_j):$ $G(i) = B(j)$;
    case $(t_j \leq t_l):$ $G(i) = (T(j) \leq T(l))$;
    case $(m_j \leq t_l):$ $G(i) = (M(j) \leq T(l))$;
    case $(t_j \leq m_l):$ $G(i) = (T(j) \leq M(l))$;
    case $(m_j \leq m_l):$ $G(i) = (M(j) \leq M(l))$;
    case $(\delta_j \supset \delta_l):$ $G(i) = \max(1 - G(j), G(l))$;
    case $(\sim\delta_j):$ $G(i) = 1 - G(j)$;
  end
end

Assuming that all basic arithmetical operations take $O(1)$ time, we see that the first part of the algorithm takes $\sum_{\Box\beta_i \in bf_{\Box}(\delta)} O(2^n)(O(|\beta_i|) + O(|fact|)) =$

$O(2^n)(\sum_{\Box\beta_i \in bf_\Box(\delta)} O(|\beta_i|) + \sum_{\Box\beta_i \in bf_\Box(\delta)} O(|fact|)) = O(2^n)O(|\delta|)O(|\delta| \cdot |fact|) = O(2^n \cdot |\delta|^2 \cdot |fact|)$.

The second part takes, likewise, $\sum_{\int \beta_i \in sub_\int(\delta)} O(2^n)(O(|\beta_i|)+O(|fact|)) = O(2^n \cdot |\delta|^2 \cdot |fact|)$.

The third cycle runs $O(|\delta|)$ times, each case being a basic operation or taking $O(|\delta|)$ time (since both $|sub_\int(\delta)| < |\delta|$ and $|pst(\delta)| < |\delta|$). The cycle runs in $O(|\delta|^2)$.

Finally, the last cycle runs $O(|\delta|)$ times, each case taking $O(|\delta|)$ time (since $|sub_\int(\delta)| < |\delta|$, $|pst(\delta)| < |\delta|$ and $|bf_\Box(\delta)| < |\delta|$). The cycle runs in $O(|\delta|^2)$.

All four rounds of the algorithm take $O(2^n \cdot |\delta|^2 \cdot |fact|)$ or less time. So the algorithm runs in $O(2^n \cdot |\delta|^2 \cdot |fact|)$ as well.

This is slightly less time efficient than the algorithm proposed in [19]. However, the present algorithm does not require writing out explicitly an EPPL Model, a structure that takes exponential space in $|\Lambda|$, nor it requires at any step the use of such large structures. In fact, none of the arrays B, M, T, G take more than $O(|\delta|)$ space and the input itself takes $O(|\gamma|) + O(|\delta|) + O(|fact|)$. So, the algorithm takes $O(|\gamma| + |\delta| + n \cdot |\beta|)$ space, whereas the one proposed in [19] needs $O(2^n \cdot |\delta| + \gamma)$. This is what we gain from using PBCs, at the cost of a little temporal efficiency.

Proposition 3.4.1. $\chi_{fact}, \gamma \Vdash \delta$ if and only if Algorithm 2 returns 1.

Proof. We will first prove that $B(i) = \begin{cases} 1 & \text{if } \chi_{fact}, \gamma \Vdash \lambda_i \\ 0 & \text{if } \chi_{fact}, \gamma \nVdash \lambda_i \end{cases}$:

If $B(i) = 1$, then, for all $\omega^j \in 2^\Lambda$, either $fact(\omega^j) = 0$ or $fact(\omega^j) \neq 0$ and $\lambda_i(\omega^j) = 1$. In the first case, $\omega^j \notin \Omega$, and we need not consider it. In the second case, $\omega^j \in \Omega$ and $\omega^j \in X_{\lambda_i}^{-1}(1)$. Since we run trough all $\omega^j \in 2^\Lambda$, we also exhaust all $\omega^j \in \Omega$, and each one verifies $\omega^j \in X_{\lambda_i}^{-1}(1)$, meaning that $\Omega = X_{\lambda_i}^{-1}(1)$, i.e., $\chi_{fact}, \gamma \Vdash \lambda_i$.

If $B(i) = 0$, then for at least one ω^j, $fact(\omega^j) > 0$ and $\lambda_i(\omega^j) = 0$. Since $fact(\omega^j) > 0$, $\omega^j \in \Omega$, but because $\lambda_i(\omega^j) = 0$, $\omega^j \notin X_{\lambda_i}^{-1}(1)$, which means there is $\omega^j \in \omega$ such that $\omega \notin X_{\lambda_i}^{-1}(1)$, i.e., $\chi_{fact}, \gamma \nVdash \lambda_i$.

We now prove that $M(i) = [\![m_i]\!]_{\chi_{fact},\gamma}$:

In the conditions of our sample space, $[\![m_i]\!]_{\chi_{fact},\gamma} = \mu(X_{\lambda_i}^{-1}(1)) = \mu(\{\omega \in \Omega : \beta(\omega) = 1\}) = \sum_{\omega \in \Omega : \beta(\omega)=1} \mu(\{\omega\})$. Since $fact(\omega) = \mu(\{\omega\})$ if $\omega \in \Omega$, 0 otherwise,

$$M(i) = \sum_{\substack{\omega^j \in 2^\Lambda : \\ \beta_i(\omega^j)=1}} fact(\omega^j) = \sum_{\substack{\omega^j \in \Omega : \\ \beta_i(\omega^j)=1}} fact(\omega^j) + \sum_{\substack{\omega^j \notin \Omega : \\ \beta_i(\omega^j)=1}} fact(\omega^j) = \sum_{\substack{\omega^j \in \Omega : \\ \beta_i(\omega^j)=1}} \mu(\{\omega^j\}) + \sum_{\substack{\omega^j \notin \Omega : \\ \beta_i(\omega^j)=1}} 0 =$$

$$\sum_{\substack{\omega^j \in \Omega : \\ \beta_i(\omega^j)=1}} \mu(\{\omega^j\}) = \mu(\bigcup_{\substack{\omega^j \in \Omega : \\ \beta_i(\omega^j)=1}} \{\omega^j\}) = \mu(\{\omega \in \Omega : \beta(\omega) = 1\} = [\![m_i]\!]_{\chi_{fact},\gamma}.$$

It is now straightforward to check that $T(i) = [\![t_i]\!]_{\chi_{fact},\gamma}$. We omit the rest of the proof, as it is a simple but lengthy exercise of induction over $|\delta|$.

□

3.5 Optimizations

The tool produced in the scope of this work is not a blind implementation of the algorithm of the previous section: several optimizations are used in order to reduce the time (in average) it takes to model check a formula. Unfortunately, none of these will actually reduce the worst case complexity in which the program runs. In this section, we will justify the major implementation deviations from the algorithm and prove their soundness.

3.5.1 Equivalent SPBCs

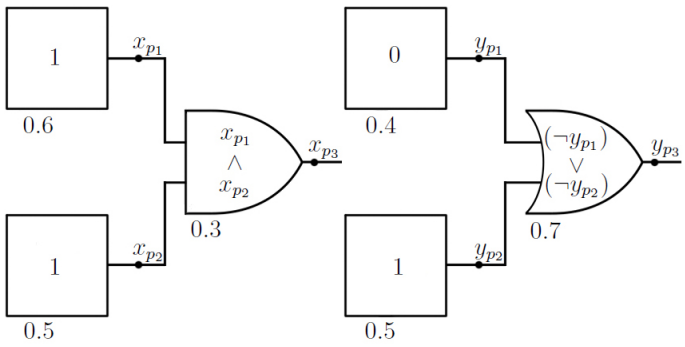

Figure 3.3: Equivalent SPBCs

Example 3.5.1. Consider the SPBCs represented in Figure 3.3 . It is easy to see that the joint distributions of $(X_{p_1}, X_{p_2}, X_{p_3})$ and $(Y_{p_1}, Y_{p_2}, Y_{p_3})$ are as presented in Table 3.1.

We have seen that, under our assumptions, knowledge of the joint distribution is enough to fully describe the EPPL model. Since both SPBCs from Figure 3.3 induce the same distribution function, we conclude that they generate the same EPPL model.

The previous example shows that different SPBCs can induce the same EPPL model. Notice that PBCs corresponding to these SPBCs are not equal, yet they

Table 3.1: Distributions of $\mathbf{X}$ and $\mathbf{Y}$

$\omega = \langle\omega_1\omega_2\omega_3\rangle$	$P(X_{p_1}=\omega_1, X_{p_2}=\omega_2, X_{p_3}=\omega_3)$	$P(Y_{p_1}=\omega_1, Y_{p_2}=\omega_2, Y_{p_3}=\omega_3)$
000	$(1-0.6)\times(1-0.5)\times 0.3 = 0.06$	$0.4\times(1-0.5)\times(1-0.7) = 0.06$
001	0.14	0.14
010	0.06	0.06
011	0.14	0.14
100	0.09	0.09
101	0.21	0.21
110	0.21	0.21
111	0.09	0.09

still induce the same EPPL model. We shall say that two SPBCs induce the same EPPL model are *equivalent*.

Equivalence between SPBCs is obviously an equivalence relation.

Lemma 3.5.2. Let $B_1=(\{X_{p_i}\}_{p_i\in\Lambda}, E, \{L_{p_i}\}_{p_i\in\Lambda})$,$B_2=(\{Y_{p_i}\}_{p_i\in\Lambda}, E', \{L'_{p_i}\}_{p_i\in\Lambda})$ be SPBCs that have equal underlying graphs and equal labels for real numbers and formulas in all vertices except one pair (X_{p_i}, Y_{p_i}), where X_i is labeled with the real number r_x and the formula φ_x and Y_i is labeled with r_y, φ_y.

If either $(\varphi_x \Leftrightarrow \varphi_y)$ (propositionally) and $r_x = r_y$ or $(\varphi_x \Leftrightarrow (\neg\varphi_y))$ and $r_x = 1 - r_y$, then B_1, B_2 are equivalent.

Proof. B_1 and B_2 are equivalent as long as the factorizations induced by them are equal. As all terms in the factorizations except the $i-th$ are equal, it suffices to prove that $P(X_{p_i}|X_{p_{i-1}}..., X_{p_1}) = P(Y_{p_i}|Y_{p_{i-1}}, ..., Y_{p_1})$ for all $\omega \in \Omega$.

Let $\omega = \langle\omega_1, ..., \omega_n\rangle \in \Omega$, then

$$P_{B_1}\left[X_{p_i} = \omega_i | X_{p_{i-1}} = \omega_{i-1}, ..., X_{p_1} = \omega_1\right] = r_i^{B_1}\delta_{\omega_i,\varphi_i^{B_1}} + (1 - r_i^{B_1})\delta_{1-\omega_i,\varphi_i^{B_1}}.$$

- Case $r_i^{B_1} = r_i^{B_2}$ and $\varphi_i^{B_1} \Leftrightarrow \varphi_i^{B_2}$:

  $\varphi_i^{B_1}(\omega_1, ..., \omega_{i-1}) = \varphi_i^{B_2}(\omega_1, ..., \omega_{i-1})$ because $(\varphi_i^{B_1} \Leftrightarrow \varphi_i^{B_2})$.

  $$\delta_{\omega_i,\varphi_i^{B_1}} = \begin{cases} 1 & \text{if } \omega_i = \varphi_i^{B_1}(\omega_1, ..., \omega_{i-1}) \\ 0 & \text{if } \omega_i \neq \varphi_i^{B_1}(\omega_1, ..., \omega_{i-1}) \end{cases} = \begin{cases} 1 & \text{if } \omega_i = \varphi_i^{B_2}(\omega_1, ..., \omega_{i-1}) \\ 0 & \text{if } \omega_i \neq \varphi_i^{B_2}(\omega_1, ..., \omega_{i-1}) \end{cases}$$
  $= \delta_{\omega_i,\varphi_i^{B_2}}$.

$$r_i^{B_1}\delta_{\omega_i,\varphi_i^{B_1}} + (1 - r_i^{B_1})\delta_{1-\omega_i,\varphi_i^{B_1}} = r_i^{B_1}\delta_{\omega_i,\varphi_i^{B_2}} + (1 - r_i^{B_1})\delta_{1-\omega_i,\varphi_i^{B_2}} =$$

$$r_i^{B_2}\delta_{\omega_i,\varphi_i^{B_2}} + (1 - r_i^{B_2})\delta_{1-\omega_i,\varphi_i^{B_2}} = P_{B_1}(Y_{p_i} = \omega_i | Y_{p_{i-1}} = \omega_{i-1}, ..., Y_{p_1} = \omega_1)$$

- Case $r_i^{B_1} = 1 - r_i^{B_2}$ and $\varphi_i^{B_1} \Leftrightarrow (\neg\varphi_i^{B_2})$:

$\varphi_i^{B_1}(\omega_1, ..., \omega_{i-1}) = 1 - \varphi_i^{B_2}(\omega_1, ..., \omega_{i-1})$ because $(\varphi_i^{B_1} \Leftrightarrow (\neg\varphi_i^{B_2}))$.

$$\delta_{\omega_i, \varphi_i^{B_1}} = \begin{cases} 1 & \text{if } \omega_i = \varphi_i^{B_1}(\omega_1, ..., \omega_{i-1}) \\ 0 & \text{if } \omega_i \neq \varphi_i^{B_1}(\omega_1, ..., \omega_{i-1}) \end{cases} = \begin{cases} 0 & \text{if } \omega_i = \varphi_i^{B_2}(\omega_1, ..., \omega_{i-1}) \\ 1 & \text{if } \omega_i \neq \varphi_i^{B_2}(\omega_1, ..., \omega_{i-1}) \end{cases}$$
$$= \delta_{1-\omega_i, \varphi_i^{B_2}}.$$

$$r_i^{B_1}\delta_{\omega_i, \varphi_i^{B_1}} + (1 - r_i^{B_1})\delta_{1-\omega_i, \varphi_i^{B_1}} = r_i^{B_1}\delta_{1-\omega_i, \varphi_i^{B_2}} + (1 - r_i^{B_1})\delta_{1-(1-\omega_i), \varphi_i^{B_2}} =$$

$$(1 - r_i^{B_2})\delta_{1-\omega_i, \varphi_i^{B_2}} + r_i^{B_2}\delta_{\omega_i, \varphi_i^{B_2}} = P_{B_1}(Y_{p_i} = \omega_i | Y_{p_{i-1}} = \omega_{i-1}, ..., Y_{p_1} = \omega_1)$$

So, for each $\langle \omega_1, ..., \omega_n \rangle \in \Omega$, the $i - th$ factor is also equal.

□

Proposition 3.5.3. Let $B_1 = (\{X_{p_i}\}_{p_i \in \Lambda},\ E,\ \{L_{p_i}\}_{p_i \in \Lambda})$, $B_2 = (\{Y_{p_i}\}_{p_i \in \Lambda},\ E',\ \{L'_{p_i}\}_{p_i \in \Lambda})$ be SPBCs that have equal underlying graphs and equal labels for real numbers and formulas in all vertices except they may differ in any number of pairs (X_{p_i}, Y_{p_i}), in the conditions of Lemma 3.5.2. Then B_1, B_2 are equivalent.

Proof. The result follows by transitivity of the equivalence relation and Lemma 3.5.2.

□

Henceforth, we will assume that all SPBCs' labeling real numbers are at least $\frac{1}{2}$. This can be done because we can always build a SPBC with this property that is equivalent to any SPBC given: In any vertex of the original SPBC with labeling number r less than $\frac{1}{2}$, we change the labeling formula φ to $(\neg\varphi)$ and labeling number $1 - r$.

3.5.2 Deterministic gates

Suppose a SPBC where one of the vertices is labeled with the real number 1. This amounts to saying that the connective of the digital circuit that is modeled by that vertex is completely reliable, deterministically providing an output for given sets of input (rigorously, providing the correct output for given sets of input with probability one, but since the sample space is finite and does not have elements with measure zero, we allow ourselves the language abuse).

We will call a vertex with this property a *deterministic gate*. Deterministic gates will allow some simplifications in our algorithm, essentially removing one symbol from Λ, for all implementation purposes. The idea is that if $x_i = \varphi_i(x_{i-1}, ..., x_1)$ with $r = 1$, then we can substitute all instances of X_{p_i} in all formulas that need to be verified by $\varphi_i(X_{p_{i-1}}, ..., X_{p_1})$.

Definition 3.5.4. Let $X_{p_n}, ..., X_{p_1}$ be discrete random variables. $X_{p_{i+1}}$ is said to be *completely dependent of* $X_{p_i}, ..., X_{p_1}$ *by means of* f if

$$\mu(\{\omega : X_{p_{i+1}}(\omega) = f(X_{p_i}(\omega), ..., X_{p_1}(\omega))\}) = 1$$

where μ denotes the measure over the probability space underlying in the random vector $X_{p_n}, ..., X_{p_1}$.

Proposition 3.5.5. Given SPBC B that has one deterministic gate, the random variable X_{p_i} induced by that vertex is completely dependent of $par(X_{p_i})$ by means of φ_i when seen as a Boolean formula.

Proof. For all $\omega = \langle \omega_1, ..., \omega_n \rangle \in \Omega$,

$$P(X_{p_i} = \omega_i | X_{p_{i-1}} = \omega_{i-1}, ..., X_{p_1} = \omega_1) = r_i \delta_{\omega_i, \varphi_i} + (1 - r_i)\delta_{1-\omega_i, \varphi_i} =$$

$$= \delta_{\omega_i, \varphi_i} = \begin{cases} 1 & \text{if } \omega_i = \varphi_i(\omega_1, ..., \omega_{i-1}) \\ 0 & \text{if } \omega_i \neq \varphi_i(\omega_1, ..., \omega_{i-1}) \end{cases}$$

Let us suppose, by contradiction, that $\mu(\{\omega : X_{p_i}(\omega) = \varphi_i(par(X_{p_i}))\}) \neq 1$.
Then, there is $W = \{\omega : X_{p_i} \neq \varphi_i(par(X_{p_i}))\}$ such that $\mu(W) > 0$.
As W is finite, there is $\omega^j = \left\langle \omega_1^j, ..., \omega_n^j \right\rangle \in W$ such that $\mu(\{\omega^j\}) > 0$.

$$P\left[X_{p_i} = \omega_i^1 | par(X_{p_i}) = par(\omega_i^1)\right] = \delta_{\omega_i^j, \varphi_i^j} = 0 \text{ (because } \omega^j,\ \omega_i^j \neq \varphi(par(\omega_i^j)))$$

On the other hand, as $P(par(X_{p_i}) = par(\omega_i^j)) > 0$ because $\omega^j \in \Omega$:

$$P\left[X_{p_i} = \omega_i^j | par(X_{p_i}) = par(\omega_i^j)\right] = \frac{P\left[X_{p_i} = \omega_i^j, par(X_{p_i}) = par(\omega_i^j)\right]}{P\left[par(X_{p_i}) = par(\omega_i^j)\right]} > 0$$

because $\omega^j \in \Omega$. This is a contradiction.

□

Remark 3.5.6. At this point, it is convenient to notice that, since our sample space is finite and has no elements with measure zero, verifying $m, \gamma \Vdash \Box\beta$ is the same as verifying if $[\![\int \beta]\!]_{m,\gamma} = 1$, indeed; if $m, \gamma \Vdash \Box\beta$, then $\Omega = X_\beta^{-1}(1)$, therefore

$1 = \mu(\Omega) = \mu(X_\beta^{-1}(1))$, i.e, $[\![\int \beta]\!]_{m,\gamma} = 1$. On the oher hand, if $m, \gamma \nVdash \Box\beta$, $\Omega \neq X_\beta^{-1}(1)$ and there is $\omega \in \Omega$ such that $\omega \in X_\beta^{-1}(0)$. Since $\mu(\omega) > 0$ and $X_\beta^{-1}(1) \cap X_\beta^{-1}(0) = \phi$, $[\![\int \beta]\!]_{m,\gamma} = X_\beta^{-1}(1) < 1$.

In [18], $\Box\beta$ is actually introduced as an abbreviation of $\int \beta = 1$, but slight differences in semantics do not allow us to take this shortcut in this work.

Finally, we can claim the following result:

Proposition 3.5.7. Let $X_{p_{i+1}}$ be completely dependent of $X_{p_i}, ..., X_{p_1}$ by means of f, m be an EPPL model induced by a SPBC, γ an assignement and δ a global formula. Then

$$m, \gamma \Vdash \delta \text{ iff } m, \gamma \Vdash \delta^{p_{i+1}}_{f(p_i,...,p_1)}$$

Proof. Notice that all nonbasic subformulas of δ remain unchanged by the substitution. Therefore, if we prove both that $[\![\int \beta]\!]_{m,\gamma} = [\![\int \beta^{p_{i+1}}_{f(p_i,...,p_1)}]\!]_{m,\gamma}$ and that $m, \gamma \Vdash \Box\beta$ iif $m, \gamma \Vdash \Box\beta^{p_{i+1}}_{f(p_i,...,p_1)}$, the result will follow.

In view of Remark 3.5.6, we need only prove that $[\![\int \beta]\!]_{m,\gamma} = [\![\int \beta^{p_{i+1}}_{f(p_i,...,p_1)}]\!]_{m,\gamma}$.
$[\![\int \beta]\!]_{m,\gamma} = \mu(X_\beta^{-1}(1)) = \mu(\{\omega \in \Omega : X_\beta(\omega) = 1\})$
$[\![\int \beta^{p_{i+1}}_{f(p_i,...,p_1)}]\!]_{m,\gamma} = \mu(X^{-1}_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}(1)) = \mu(\{\omega \in \Omega : X_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}(\omega) = 1\})$

We show that $\{\omega \in \Omega : X_\beta(\omega) = 1\} = \{\omega \in \Omega : X_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}(\omega) = 1\}$:

Since $\mu(\{\omega : X_{i+1}(\omega) = f(X_i(\omega), ..., X_1(\omega))\}) = 1 = \mu(\Omega)$ and Ω is finite and has no elements with measure zero, by elementary measure theory, $\{\omega : X_{i+1}(\omega) = f(X_i(\omega), ..., X_1(\omega))\} = \Omega$.

Let $\omega' \in \{\omega \in \Omega : X_\beta(\omega) = 1\}$. As $\omega' \in \Omega$, $X_{p_{i+1}}(\omega') = f(X_{p_i}(\omega'), ..., X_{p_1}(\omega'))$. Then, it must be the case that $X_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}(\omega') = 1$, as all functions in the recursive definition of $X_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}$ are evaluated in the same way as in X_β, and $f(X_{p_i}(\omega'), ..., X_{p_1}(\omega'))$ agrees with $X_{p_{i+1}}(\omega')$.

Therefore, $\omega' \in \{\omega \in \Omega : X_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}(\omega) = 1\}$.

On the other hand, let $\omega' \in \{\omega \in \Omega : X_{\beta^{p_{i+1}}_{f(p_i,...,p_1)}}(\omega) = 1\}$. By the same reasoning, *mutatis mutandis*, we conclude that $\omega' \in \{\omega \in \Omega : X_\beta(\omega) = 1\}$. □

This proposition allows the following simplification in the algorithm: by replacing all variables induced by deterministic gates with the function induced by the corresponding formula, the number of valuations that have to be considered halves for each deterministic gate, as one of each of the two valuations that only differ in the deterministic variable would have measure zero, and, therefore, doesn't need to be considered.

Remark 3.5.6 shows that, given the particular structure of our sample space, we may consider subformulas of the form $\Box\beta$ as $\int \beta = 1$. We refrain from doing

so in our implementation for efficiency reasons: As we can see from the basic algorithm, we always have to run exhaustively trough all 2^n valuations in 2^Λ in order to compute $[\![\int \beta]\!]_{\chi_{fact},\gamma}$, whereas to compute $\chi_{fact}, \gamma \Vdash \Box\beta$, the computation ends as soon as we find a suitable valuation. The process of computing $\chi_{fact}, \gamma \Vdash \Box\beta$ can be further refined using deterministic gates, and the program explores this property:

Theorem 3.5.8. Given a basic formula β and a SPBC induced EPPL Model χ_{fact}, then $\chi_{fact}, \gamma \nVdash \Box\beta$ iff $(\neg\beta^{p_i\in D}_{\varphi_i(p_{i-1},\ldots,p_1)})$ is satisfiable as a propositional formula, where D denotes the set of deterministic gates in the SPBC.

Proof. If $(\neg\beta^{p_i\in D}_{\varphi_i(p_{i-1},\ldots,p_1)})$ is satisfiable, there is a valuation $\omega' \in 2^{(\Lambda\setminus D)}$ s.t. $(\neg\beta^{p_i\in D}_{\varphi_i(p_{i-1},\ldots,p_1)})(\omega') = 1$. Let $\omega \in 2^\Lambda$ be the valuation that coincides with ω' in all $p_i \in \Lambda \setminus D$ and that has $\omega_i = \varphi_i(\omega_{i-1}, \ldots, \omega_1)$ for all $p_i \in D$ (this computation should be made by index order).

$fact(\omega) = \prod_n^{i=1} r_i\delta_{\omega_i,\varphi_i} + (1-r_i)\delta_{1-\omega_i,\varphi_i} = \prod_{p_i\in D} r_i\delta_{\omega_i,\varphi_i} + (1-r_i)\delta_{1-\omega_i,\varphi_i} * \prod_{p_i\in\Lambda\setminus D} r_i\delta_{\omega_i,\varphi_i} + (1-r_i)\delta_{1-\omega_i,\varphi_i} = \prod_{p_i\in D}(1+0) * \prod_{p_i\in\Lambda\setminus D} r_i\delta_{\omega_i,\varphi_i} + (1-r_i)\delta_{1-\omega_i,\varphi_i} = \prod_{p_i\in\Lambda\setminus D} r_i\delta_{\omega_i,\varphi_i} + (1-r_i)\delta_{1-\omega_i,\varphi_i}$. Each r_i and $1-r_i$ in the last product is greater than zero, and one of the δ in each term is 1. It follows that $fact(\omega) > 0$, that is, $\omega \in \Omega$.

$(\neg\beta)(\omega) = 1$, as we have seen in the proof of Proposition 3.5.7.

Since $\mu(\omega) > 0, \omega \in \Omega$, then $\chi_{fact}, \gamma \nVdash \Box\beta$ iff there is $\omega \in \Omega$ s.t. $\omega \in X_\beta^{-1}(0)$, i.e., there is $\omega \in \Omega$ s.t. $\omega \in X^{-1}_{(\neg\beta)}(1)$, i.e. there is $\omega \in \Omega$ s.t. $(\neg\beta)(\omega) = 1$, which we have just proved.

If $(\neg\beta^{p_i\in D}_{\varphi_i(p_{i-1},\ldots,p_1)})$ is not satisfiable, for all valuations $\omega' \in 2^{(\Lambda\setminus D)}$ we have that $\beta^{p_i\in D}_{\varphi_i(p_{i-1},\ldots,p_1)}(\omega') = 1$.

For each $\omega' \in 2^{(\Lambda\setminus D)}$ only the $\omega \in 2^\Lambda$ built in the same way as in the first part of the proof verifies $fact(\omega) > 0$ (i.e., $\omega \in \Omega$): if $\omega_i \neq \varphi_i(\omega_{i-1}, \ldots, \omega_1)$ for some $p_i \in \Lambda\setminus D$, the corresponding term in $fact(\omega)$, $r_i\delta_{\omega_i,\varphi_i} + (1-r_i)\delta_{1-\omega_i,\varphi_i} = 1*0+0*1 = 0$, and $fact(\omega) = 0$. Furthermore, there are no more valuations in Ω than those of this form or there would be $\omega' \in 2^{(\Lambda\setminus D)}$ s.t. $\beta^{p_i\in D}_{\varphi_i(p_{i-1},\ldots,p_1)}(\omega') = 0$.

Once again, by proposition 3.5.7, $\beta(\omega) = 1$. This is true for all $\omega \in \Omega$, and so $\chi_{fact}, \gamma \Vdash \Box\beta$

□

The previous proposition allows us to compute $\chi_{fact}, \gamma \Vdash \Box\beta$ with a single application of a SAT algorithm for propositional logic. Although still in the same complexity class of the direct computation of $\chi_{fact}, \gamma \Vdash \Box\beta$, the SAT problem is very well studied, and many clever algorithms have been proposed to solve it. Upon the use of one such algorithm, we improve our own, since our approach in the basic algorithm was to run through all of the 2^n valuations.

3.6 EPPL MC tool

3.6.1 Syntax differences

Although operational aspects of the actual implementation of the tool are left for the Appendix, there are differences between the syntax of EPPL and that of the actual tool that we will discuss at this point.

In practice, the tool parses strings inputed by the user and, since ASCII encoding does not allow the representation of all EPPL symbols, we need to find intuitive substitutes. There are also slight differences in parenthesis that ease the implementation of the parsing tool.

We start by defining the syntax for basic formulas:

$$\beta := \mathsf{var} \,[]\, (\tilde{}\,\beta) \,[]\, (\beta \&\& \ \beta) \,[]\, (\beta \ | \ \ \beta) \,[]\, (\beta \ => \ \ \beta) \,[]\, (\beta \ <=> \ \ \beta)$$

Since the parser is quite strict, when using the tool, we advise not to deviate from the above syntax. This includes droping parenthesis or assuming associativity. One should pay close attention to spacing: there is a whitespace after each connective.

The syntax should be clear: var stands for any variable name, ~ stands for $\neg$, && stands for $\wedge$, | stands for $\vee$, => stands for $\Rightarrow$ and <=> stands for $\Leftrightarrow$. Variable naming is duly covered in the Appendix.

For terms, the syntax is as follows:

$$t := \{\mathsf{term}\} \,[]\, \{0\} \,[]\, \{1\} \,[]\, \{\$\beta\} \,[]\, \{t+ \ t\} \,[]\, \{t. \ t\}$$

The syntax is straightforward; the only connective that needs explaining is \$, which stands for $\int$. Remember the variable naming conventions for term variables. The seemingly redundant {} around the variables and constants are necessary in order to contextualize the parser.

Finally, for global formulas, the syntax is:

$$\delta := [\#\beta] \,[]\, [t < \ \ t] \,[]\, [t > \ \ t] \,[]\, [t\ll t] \,[]\, [t\gg t] \,[]\, [t= t] \,[]$$

$$[!\delta] \,[]\, [\delta \&\& \ \delta] \,[]\, [\delta \ || \ \ \delta] \,[]\, [\delta \ ==> \ \ \delta] \,[]\, [\delta \ <==> \ \ \delta]$$

Once again, the peculiar parenthesis are necessary to contextualize the parser. As for connectives, <, > and = stand for themselves, # stands for $\square$, $\ll$,$\gg$ stand for $\leq$, $\geq$ and !, &&, ||, ==>, <==> stand for $\sim$, $\cap$, $\cup$, $\supset$, $\equiv$, respectively.

Take, for example, axiom **FAdd** from Table 2.2. For this tool's purposes, it should be written as:

[[{\$(β_1& β_2)} = {0}] $\supset$ [{\$($\beta_1 \vee \ \beta_2$)} = {{\$$\beta_1$} + {\$$\beta_2$}}]]

As for SPBCs, the file with the description should contain one line for each vertex. The line starts with the name of a fresh labeling variable (that will be

identified with the induced random variable), followed by an "=", a whitespace, then comes the labeling basic formula (with syntax as above), followed by another whitespace and, finally, the labeling float.

The SPBC file from Example 2.2.2, for instance, would be:

Xp1 = 1 0.5

Xp2 = 1 0.5

Xp3 = (Xp1 | Xp2) 1

3.6.2 A simple case study

We now introduce, as an example of application of the MC tool, a simple hypothetical case study on reliability and quality control.

Gates that compute usual Boolean functions (like conjunction or disjunction of multiple inputs) are massively produced due to their many applications. However, gates for more unusual functions are often required. In such cases, one of the approaches used is to build a circuit that computes the function and have the whole circuit acting as a single gate. This is usually done automatically, through the function's minterm form[1] (also known as sum-of-terms form).

To build a minterm representation, one takes all configurations of inputs that yield result 1 and, for each of them, considers the product of all variables that take value 1 in that configuration and 1 minus the variables that take value 0 in that configuration. The minterm representation is the sum of all such products.

Example 3.6.1. Consider the Boolean function represented in table 3.2. Since only the configurations $\langle 0,0,1\rangle, \langle 0,1,0\rangle$ and $\langle 1,0,0\rangle$ yield the result 1, its minterm form is

$$f(x,y,z) = (1-x)(1-y)z + (1-x)y(1-z) + x(1-y)(1-z)$$

.

So, Figure 3.4 represents a circuit that computes this function.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Table 3.2: Function f

[1] the minterm form for a Boolean function is an analog of the disjunctive normal form for a formula that represents it.

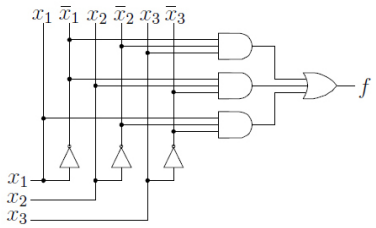

Figure 3.4: Circuit representing Boolean function f

Now, suppose a circuit manufacturing company (we will call it Hypothetical Co. for ease of reference) that produces the above circuit for two different purposes, say a medical device controller and a toy controller. An independent entity preforms quality control on the circuits produced. Actual quality control over a lot of circuits is preformed in a way very similar to the following:

- two produced circuits are randomly chosen.
- a (uniform) random input is chosen (from $\{0,1\}^k$, k the number of inputs of the circuit).
- the input is fed to both circuits and the output is read.
- if the outputs disagree, the circuits fail the test. Otherwise, they pass.

The lot is rejected if a significant (specified by the buyer) proportion of failures occur. These controls usually test for very high reliabilities so that the probability of a false positive ca be neglected.

The objective of Hypothetical Co. is to produce the cheapest possible circuits that still pass the quality control. Cheaper components are less reliable than expensive ones. We will show how to use our MC tool to ease the decision making procedure.

Suppose the toy circuit board has an accepted failure ratio of $\frac{1}{1000}$ and the medical device circuit board has an accepted failure ratio of $\frac{5}{100000}$. Suppose also that **Not** gates are easy to make and very reliable, so we will assume they always perform the computation correctly. **And** and **Or** gates, on the other hand, have two versions: a cheap one, with 0.9999 reliability and an expensive one, with 0.999999.

The cheapest circuit is described in our tool as:

X1 = 1 0.5
X2 = 1 0.5
X3 = 1 0.5
NX1 = (~X1) 1
NX2 = (~X2) 1
NX3 = (~X3) 1
A1 = ((NX1 & NX2)& X3) 0.9999
A2 = ((NX1 & X2)& X3) 0.9999

A3 = ((X1 & NX2)& NX3) 0.9999
O1 = ((A1 | A2) | A3) 0.9999

The first three lines represent the input (uniformly distributed in $\{0,1\}^3$), the three next ones represent the **Not** gates (that are completely reliable), then we have the three **And** gates and finally the **Or** output gate, all with the lowest (0.9999) reliability available. In order to simulate the quality control, we simply replicate the circuit, which is the same as running an equal one in paralel:

X1 = 1 0.5
X2 = 1 0.5
X3 = 1 0.5
NX1 = (~X1) 1
NX2 = (~X2) 1
NX3 = (~X3) 1
A1 = ((NX1 & NX2)& X3) 0.9999
A2 = ((NX1 & X2)& X3) 0.9999
A3 = ((X1 & NX2)& NX3) 0.9999
O1 = ((A1 | A2) | A3) 0.9999
Y1 = 1 0.5
Y2 = 1 0.5
Y3 = 1 0.5
NY1 = (~Y1) 1
NY2 = (~Y2) 1
NY3 = (~Y3) 1
A4 = ((NY1 & NY2)& Y3) 0.9999
A5 = ((NY1 & Y2)& Y3) 0.9999
A6 = ((Y1 & NY2)& NY3) 0.9999
O2 = ((A1 | A2) | A3) 0.9999

This is the SPBC that we introduce in our tool. We now want to check the property "If the inputs on both boards agree, then will the outputs also agree with probability greater than the security parameter p?", which is expressed by the formula:

$$\int (((X1 \Leftrightarrow Y1) \wedge (X2 \Leftrightarrow Y2) \wedge (X3 \Leftrightarrow Y3)) \Rightarrow (O1 \Leftrightarrow O2)) > p$$

In the tool's notation:

[{$((((X1 <=> Y1)& (X2 <=> Y2))& (X3 <=> Y3)) => (O1 <=> O2))} > {p }]

Now, if we check this formula using the security parameter for the toy $p = 0.999$, the MC returns that the property holds for the introduced circuit, meaning that Hypotetical Co. can use the cheaper version of the circuit for this purpose. If we run the MC with the stricter security parameter of 0.99995, the test fails, meaning that it is not wise to produce the cheap circuits for the medical devices.

This does not mean that the medical devce circuit has to be the most expensive possible. Playing a little with the MC we can show that the **Or** gate or one of the **And** gates (and no more) can be substituted by the cheaper version, without significant reduction of the circuit reliability, producing a more affordable circuit.

Chapter 4

Towards EpCTL

In this chapter we define the computation tree extension to EPPL, called exogenous probabilistic computation tree logic (EpCTL) and present a weakly complete Hilbert calculus and a model checking algorithm for it.

4.1 Syntax

The syntax of EpCTL can be easily obtained from the syntax of EPPL. The idea is that at the level of global formulas, we also introduce the usual CTLmodalities. In table 4.1 we present the syntax by mutual recursion and recall the definition of classical formulas and probability terms.

Table 4.1: Syntax of EpCTL

$\beta := p \,[\!]\, \neg\beta \,[\!]\, (\beta \Rightarrow \beta)$	basic formulae
$t := z \,[\!]\, 0 \,[\!]\, 1 \,[\!]\, (\int \beta) \,[\!]\, (t + t) \,[\!]\, (t.t)$	probabilistic terms
$\delta := (\Box\beta) \,[\!]\, (t \leq t) \,[\!]\, (\sim\delta) \,[\!]\, (\delta \supset \delta) \,[\!]\, (\mathsf{EX}\delta) \,[\!]\, (\mathsf{AF}\delta) \,[\!]\, (\mathsf{E}[\delta \mathsf{U} \delta])$	global formulae

where $p \in \Lambda$, $z \in Z$.

The basic formulas, probabilistic terms and global formulas without temporal modalities have the same intuitive meaning as in EPPL. Each modality is composed by two symbols: the first one is either E or A, and the second one either X, F or U. The first symbol quantifies over computation paths starting at some state s: one of the symbols is the existential quantification E (for "there **e**xists") and the other one, the universal quantification A (for "**a**ll"). The second symbol is used to describe temporal behavior: X stands for "ne**x**t", F for "sometime in the **f**uture" and U for "**u**ntil". So (EXδ) holds if there exists a path starting at s such that δ holds in the next state along that path; (AFδ) holds if for all paths starting at s there exists a state in those paths where δ holds; (E[δ_1Uδ_2]) if there exists a path leaving s such

that there is a state s' along that path where δ_2 holds and δ_1 holds in all states between s and s' (excluding s'). The other temporal connectives combinations and the temporal symbol G (for always in the future) are introduced as usual:

- $\mathsf{AX}\delta \equiv (\sim\mathsf{EX}(\sim\delta))$
- $\mathsf{EF}\delta \equiv \mathsf{E}[(\sim \perp\!\!\!\perp)\mathsf{U}\delta]$
- $\mathsf{AG}\delta \equiv (\sim\mathsf{EF}(\sim\delta))$
- $\mathsf{EG}\delta \equiv (\sim\mathsf{AF}(\sim\delta))$
- $\mathsf{A}[\delta_1\mathsf{U}\delta_2] \equiv (\sim\mathsf{E}[(\sim\delta_2)\mathsf{U}(\sim\delta_1 \cap \sim\delta_2)] \cap (\sim\mathsf{EG}(\sim\delta_2)))$

4.2 Semantics

A *probabilistic Kripke structure* is a tuple $M = (S, R, L)$ where S is a set of states, $R \subseteq S \times S$ is a total transition relation (called the *accessibility relation*) and L is a map from S to the set of EPPL models. A reader familiar with CTL should notice the similarities between the semantics of both logics.

The satisfaction relation is defined by structural induction:

- $M, s \Vdash \Box\beta$ iff $L(s) \Vdash \Box\beta$;
- $M, s \Vdash (t_1 \leq t_2)$ iff $L(s) \Vdash (t_1 \leq t_2)$;
- $M, s \Vdash (\sim\delta)$ iff $M, s \nVdash \delta$;
- $M, s \Vdash (\delta_1 \supset \delta_2)$ iff $M, s \nVdash \delta_1$ or $M, s \Vdash \delta_2$;
- $M, s \Vdash (\mathsf{EX}\delta)$ iff $M, s' \Vdash \delta$ with $(s, s') \in R$;
- $M, s \Vdash (\mathsf{AF}\delta)$ iff for all path π over R starting in s there exists $k \in \mathbb{N}$ such that $M, \pi_k, L \Vdash \delta$;
- $M, s \Vdash (\mathsf{E}[\delta_1\mathsf{U}\delta_2])$ iff there exists a path π over R starting in s and $k \in \mathbb{N}$ such that $M, \pi_k, \Vdash \delta_2$ and $M, \pi_i, \Vdash \delta_1$ for every $i < k$.

Example 4.2.1. Consider the coin tossing - heads checking Example 2.2.2 with only one coin now. We allow ourselves to non-deterministically toss the coin first or look at the face up first. We can model the outcome space with the set of propositional symbols $\Lambda = \{p_c, p_l, p_o\}$ that will induce Bernoulli random variables. $p_c = 1(= 0)$ stands for "coin shows heads (tails)", $p_l = 1(= 0)$ stands for "coin has (not) been looked at",$p_o = 1(= 0)$ stands for " if coin has been looked at, outcome was heads (tails)".

The experiment can be modeled by a probabilistic Kripke structure $M = (S, R, L)$ with $S = \{s_1, s_2, s_3, s_4, s_5\}$, $R = \{(s_1, s_2), (s_1, s_3), (s_2, s_4), (s_3, s_5)\}$ and $L(i) = (\Omega_i, 2^{\Omega_i}, \mu_i, \mathbf{X})$ for $i = 1, \ldots, 5$ such that

$\Omega_1 = \{(1,0,0)\}$ and $\mu_1(1,0,0) = 1$;

$\Omega_2 = \{(1,0,0),(0,0,0)\}$ and $\mu_2(1,0,0) = \mu_2(0,0,0) = 1/2$;

$\Omega_3 = \{(1,1,1)\}$ and $\mu_3(1,1,1) = 1$;

$\Omega_4 = \{(1,1,1),(0,1,0)\}$ and $\mu_4(1,1,1) = \mu_4(0,1,0) = 1/2$;

$\Omega_5 = \{(1,1,1),(0,1,1)\}$ and $\mu_5(1,1,1) = \mu_5(0,1,0) = 1/2$;

The formula $\mathsf{EF}((\int((\neg p_c) \wedge p_l \wedge p_o)) \geq 0)$ states that there is some course of actions where the coin has been checked as heads and yet the tails face is up. The formula $(((\int(\neg p_c)) \geq 0) \supset \mathsf{AG}(\Box(p_c \Leftrightarrow p_o)))$ states that if the tails face of the coin is up, then, for any course of actions, the checked value and the face of the the coin will agree.

Recall that a CTL model is a *Kripke Structure*, a tuple $M = (S, R, L)$ where S is the set of states, $R \subseteq S \times S$ is the accessibility relation and $L : S \to 2^{\Xi}$ maps each state to a valuation over Ξ.

4.3 Completeness of EpCTL

A complete axiomatization for EpCTL is obtained simply by joining a CTL calculus and the EPPL axioms:

Table 4.2: Complete Hilbert calculus HC_{EpCTL} for EpCTL

- Axioms
  - [**EPPL**] all valid EPPL formulas;
  - [**Taut**] all instantiations of propositional tautologies;
  - [**EX**] $\vdash \mathsf{EpCTL}\ \mathsf{EX}(\delta_1 \cup \delta_2) \Leftrightarrow \mathsf{EX}\delta_1 \cup \mathsf{EX}\delta_2$
  - [**X**] $\vdash_{\mathsf{EpCTL}} \mathsf{AX}(\top) \cap \mathsf{EX}(\top)$
  - [**EU**] $\vdash_{\mathsf{EpCTL}} \mathsf{E}[\delta_1 \mathsf{U} \delta_2] \equiv \delta_2 \cup (\delta_1 \cap \mathsf{EXE}[\delta_1 \mathsf{U} \delta_2])$
  - [**AU**] $\vdash_{\mathsf{EpCTL}} \mathsf{A}[\delta_1 \mathsf{U} \delta_2] \equiv \delta_2 \cup (\delta_1 \cap \mathsf{AXA}[\delta_1 \mathsf{U} \delta_2])$
  - [**AG1**] $\vdash_{\mathsf{EpCTL}} \mathsf{AG}(\delta_3 \supset ((\sim\delta_2) \cap \mathsf{EX}\delta_3)) \supset (\delta_3 \supset (\sim\mathsf{A}[\delta_1 \mathsf{U} \delta_2]))$
  - [**AG2**] $\vdash_{\mathsf{EpCTL}} \mathsf{AG}(\delta_3 \supset ((\sim\delta_2) \cap (\delta_1 \supset \mathsf{AX}\delta_3))) \supset (\delta_3 \supset (\sim\mathsf{E}[\delta_1 \mathsf{U} \delta_2]))$
  - [**AG3**] $\vdash_{\mathsf{EpCTL}} \mathsf{AG}(\delta_1 \supset \delta_2) \supset (\mathsf{EX}\delta_1 \supset \mathsf{EX}\delta_2)$
- Inference rules
  - [**MP**] $\delta_1, (\delta_1 \supset \delta_2) \vdash_{\mathsf{EpCTL}} \delta_2$
  - [**AGen**] $\delta_1 \vdash_{\mathsf{EpCTL}} \mathsf{AG}\delta_1$

As for EPPL, the soundness of these axioms is straightforward and we refrain

from presenting it. Te completeness is a bit harder and, once again, we follow the proof in [19].

First, we notice that EpCTL incorporates CTL

Lemma 4.3.1. $\vdash_{\mathsf{CTL}} \widetilde{\delta}$ *then* $\vdash_{\mathsf{EpCTL}} \delta$.

Proof. The result follows from $HC_{\mathsf{CTL}} \subseteq HC_{\mathsf{EpCTL}}$ □

Now we prove that reasoning in EpCTL over EPPL formulas coincides with EPPL reasoning.

Theorem 4.3.2. Let δ be an EPPL formula. Then

$$\vdash_{\mathsf{EpCTL}} \delta \qquad \text{iff} \qquad \vdash_{\mathsf{EPPL}} \delta.$$

Proof. If $\vdash_{\mathsf{EpCTL}} \delta$ then $\vDash_{\mathsf{EpCTL}} \delta$ by soundness of EpCTL. Let m be an EPPL model and $M = (\{1\}, R, L)$ a temporal model such that $L(1) = (m, \gamma)$ for some assignement γ. Hence, $M, 1 \Vdash \delta$ iff $m, \gamma \Vdash \delta$. So, $\vDash_{\mathsf{EPPL}} \delta$. By completeness of EPPL we get that $\vdash_{\mathsf{EPPL}} \delta$. The converse follows from $HC_{\mathsf{EPPL}} \subseteq HC_{\mathsf{EpCTL}}$. □

Consider the subset of EPPL formulas composed by modal formulas $\Box\beta$ and inequalities $(t_1 \leq t_2)$. We will call this set the set of *atomic formulas* of EPPL and denote it by $at(\mathsf{EPPL})$. Now take Ξ to be a countable set of propositional symbols used to write CTL formulas and consider a bijective map $\lambda : at(\mathsf{EPPL}) \to \Xi$. This bijection can be extended by structural induction to a map $\lambda^* : \mathsf{EpCTL} \to \mathsf{CTL}$ that translates EpCTL formulas to CTL formulas:

- $\lambda^*(\Box\beta) \equiv \lambda(\Box\beta)$
- $\lambda^*((t_1 \leq t_2)) \equiv \lambda((t_1 \leq t_2))$
- $\lambda^*(\delta_1 \supset \delta_2) \equiv (\lambda^*(\delta_1) \supset \lambda^*(\delta_2))$
- $\lambda^*(\sim\delta) \equiv (\sim\lambda^*(\delta))$
- $\lambda^*(\mathsf{EX}\delta) \equiv (\mathsf{EX}\lambda^*(\delta))$
- $\lambda^*(\mathsf{AF}\delta) \equiv (\mathsf{AF}\lambda^*(\delta))$
- $\lambda^*(\mathsf{E}[\delta_1 \mathsf{U} \delta_2]) \equiv (\mathsf{E}[\lambda^*(\delta_1)\mathsf{U}\lambda^*(\delta_2)])$

We denote by $\widetilde{\delta}$ the translation of the EpCTL formula δ by λ^*.

Satisfaction is defined as expected. The map λ^* can also be used to translate a probabilistic Kripke structure $M = (S, R, L)$ to the CTL model $\widetilde{M} = (S, R, L')$, where $\xi_i \in L'(s)$ iff $M, s \Vdash \lambda^{*^{-1}}(\xi_i)$ for all CTL propositional symbol $\xi_i \in \Xi$.

Lemma 4.3.3. Let $M = (S, R, L)$ be an EpCTL model. Then,

$$M, s \Vdash_{\mathsf{EpCTL}} \delta \qquad \text{iff} \qquad \widetilde{M}, s \Vdash_{CTL} \widetilde{\delta}.$$

Proof. The proof follows by induction on δ.

- Base: If δ is $(\Box\beta)$ or $(t_1 \leq t_2)$ then $M, s \Vdash \delta$ iff $\widetilde{M}, s \Vdash \widetilde{\delta}$ by definition.
- Step:
  - If δ is $(\sim\delta_1)$ then $M, s \Vdash (\sim\delta_1)$ iff $M, s \nVdash \delta_1$ iff $\widetilde{M}, s \nVdash \widetilde{\delta}_1$ iff $\widetilde{M}, s \Vdash (\sim\widetilde{\delta})$ and $\widetilde{\delta} = (\sim\widetilde{\delta}_1)$.
  - If δ is $(\delta_1 \supset \delta_2)$ then $M, s \Vdash \delta$ iff $M, s \nVdash \delta_1$ or $M, s \Vdash \delta_2$ iff $\widetilde{M}, s \nVdash \widetilde{\delta}_1$ or $\widetilde{M}, s \Vdash \widetilde{\delta}_2$ iff $\widetilde{M}, s \Vdash \widetilde{\delta}$ and $\widetilde{\delta} = (\widetilde{\delta}_1 \supset \widetilde{\delta}_2)$.
  - If δ is $\mathsf{EX}\delta_1$ then $M, s \Vdash \mathsf{EX}\delta_1$ iff there is $(s, s') \in R$ such that $M, s' \Vdash \delta_1$ iff there is $(s, s') \in R$ such that $\widetilde{M}, s' \Vdash \widetilde{\delta}_1$ iff $\widetilde{M}, s \Vdash (\mathsf{EX}\widetilde{\delta}_1)$ and $\widetilde{\delta} = (\mathsf{EX}\widetilde{\delta}_1)$.
  - If δ is $(\mathsf{AF}\delta_1)$ then $M, s \Vdash (\mathsf{AF}\delta_1)$ iff for all path π in R starting in s there is $i \geq 0$ such that $M, \pi_i \Vdash \delta_1$ iff for all path π in R starting in s there is $i \geq 0$ such that $\widetilde{M}, \pi_i \Vdash \widetilde{\delta}_1$ iff $\widetilde{M}, \pi_i \Vdash (\mathsf{AF}\widetilde{\delta}_1)$ and $\widetilde{\delta} = (\mathsf{AF}\widetilde{\delta}_1)$.
  - If δ is $\mathsf{E}[\delta_1 \mathsf{U} \delta_2]$ then $M, s \Vdash (\mathsf{E}[\delta_1 \mathsf{U} \delta_2])$ iff there is a path π in R starting in s and $i \geq 0$ such that $M, \pi_i \Vdash \delta_2$ and $M, \pi_j \Vdash \delta_1$ for all $0 \geq j < i$ iff there is a path π in R starting in s and $i \geq 0$ such that $\widetilde{M}, \pi_i \Vdash \widetilde{\delta}_2$ and $\widetilde{M}, \pi_j \Vdash \widetilde{\delta}_1$ for all $0 \geq j < i$ iff $\widetilde{M}, s \Vdash \mathsf{E}[\widetilde{\delta}_1 \mathsf{U} \widetilde{\delta}_2]$ and $\widetilde{\delta} = \mathsf{E}[\widetilde{\delta}_1 \mathsf{U} \widetilde{\delta}_2]$.

□

We need one final remark before we are able to prove the completeness of HC_{EpCTL}. Like before, we denote by $at(\delta) = \{\delta_1, ..., \delta_k\}$ the set of atomic EPPL formulas in δ. We define, for $i \in \{0, 1\}^k$, the formula

$$\phi_i = \bigcap_{j=1}^{k} \varphi_j \qquad \text{where} \qquad \varphi_j = \begin{cases} \delta_j & \text{if the } j\text{-th bit of } i \text{ is } 1 \\ (\sim\delta_j) & \text{otherwise} \end{cases}$$

Let $\Psi = \bigcup_{i \in A} \phi_i$, where A is the set of Boolean k-vectors such that ϕ_i is an EPPL consistent formula. Of course, $\vdash_{\mathsf{EPPL}} \Psi$ and so, by the previous theorem, $\vdash_{\mathsf{EpCTL}} \Psi$.

Now let δ be an EpCTL formula such that $\Vdash_{\mathsf{EpCTL}} \delta$ and suppose that $M = (S, R, L)$ is a CTL model such that $M, s \nVdash (\mathsf{AG}\widetilde{\Psi} \supset \widetilde{\delta})$. Then $M, s \Vdash \mathsf{AG}\widetilde{\Psi}$ and $M, s \nVdash \widetilde{\delta}$. Consider the EpCTL model $M' = (\{s\} \cup S', R', L')$ such that $S' = \{s \in S : (s, s') \in R\}$ is the subset of states reachable from the state s, R' is the restriction of R to $\{s\} \cup S'$, and $L'(s')$ is the EPPL model that satisfies one of the formulas ϕ_i with $i \in A$ and such that $M, s' \Vdash \widetilde{\phi}_i$. This model exists since $\widetilde{\Psi}$ is satisfied in all $s' \in S'$ and all ϕ_i in Ψ are consistent. Therefore, $M', s \nVdash (\mathsf{AG}\widetilde{\Psi} \supset \delta)$. But, $\vdash_{\mathsf{EpCTL}} \Psi$ and by soundness $\Vdash_{\mathsf{EpCTL}} \Psi$. So, $M', s \nVdash \delta$ which is a contradiction. We conclude that if $\Vdash_{\mathsf{EpCTL}} \delta$ then $\Vdash_{\mathsf{CTL}} (\mathsf{AG}\widetilde{\Psi} \supset \widetilde{\delta})$.

Theorem 4.3.4. The axiomatization HC_{EpCTL} is complete.

Proof. Let $\Vdash_{\mathsf{EpCTL}} \delta$. By the previous discussion, $\Vdash_{\mathsf{CTL}} (\mathsf{AG}\widetilde{\Psi} \supset \widetilde{\delta})$. Using the completeness of HC_{CTL} we have $\vdash_{\mathsf{CTL}} (\mathsf{AG}\widetilde{\Psi} \supset \widetilde{\delta})$. From Lemma 4.3.1 we get $\vdash_{\mathsf{EpCTL}} (\mathsf{AG}\Psi \supset \delta)$. Hence, we are able do the following derivation in EpCTL:

1)	$\vdash \Psi$	Theorem
2)	$\vdash (\mathsf{AG}\Psi)$	**AGen**
3)	$\vdash (\mathsf{AG}\Psi \supset \delta)$	Theorem
4)	$\vdash \delta$	Modus Ponens $2, 3$

Therefore, HC_{EpCTL} is complete. □

4.4 Model checking algorithm for EpCTL

The EpCTL MC algorithm can easily be obtained from the CTL MC algorithm, replacing the propositional symbol checking step with the EPPL model checker. Like in the CTL MC for Kripke structures, in order to check a formula δ, we label each state of the probabilistic Kripke structure with the set of subformulas of δ satisfied by the EPPL model labeling that state, starting with simpler subformulas. This labeling is straightforward for all non temporal connectives.

For formulas of the form $\mathsf{EX}\delta$, we label with $\mathsf{EX}\delta$ all states that have a successor labeled with δ. We denote this procedure by $LabelEX$.

For formulas of the form $\mathsf{E}[\delta_1 \mathsf{U} \delta_2]$, we reason backwards: We consider the converse relation R^{-1}, the set of states labeled by δ_2 (S') and proceed by labeling with $\mathsf{E}[\delta_1 \mathsf{U} \delta_2]$ all states reached by at least one path (considering R^{-1}) starting in one element of S' where every state is labeled with δ_1. We denote this procedure by $LabelEU$

Instead of considering AF, we explain the procedure for EG, since checking $\mathsf{AF}\delta$ is the same as checking $\sim\mathsf{EG}\sim\delta$. To check $\mathsf{EG}\delta$ over $M = (S, R, L)$, consider the substructure $M' = (S', R', L')$ obtained from the original probabilistic Kripke structure where we remove all states not labeled with δ and restrict R and L in the obvious way. Now, we take the decomposition of (S', R') into strongly connected components. It should be clear that, in any non-trivial of these components, $\mathsf{EG}\delta$ holds (since all states of the cycle are labeled with δ) and we label them accordingly. We can now work backwards using the converse relation R'^{-1} to likewise label all states that can be reached by a path in which each state is labeled with δ. Then, $\mathsf{EG}\delta$ will also hold in the respective states of the original structure M and only in those (because there are no more states labeled by δ). We denote this procedure by $LabelEG$.

In the model-checking algorithm for EpCTL described below, we denote by $sub_{\mathsf{CTL}}(\delta)$ the ordered list of global subformulas of δ.

Algorithm 3: alg:epctl

Input: temporal model $M = (S, R, L)$ and EpCTL formula δ
Output: M, with states where δ is satisfied labeled with δ

for $i = 1$ **to** $|sub_{CTL}(\delta)|$ **do**
  switch δ_i **do**
    case $(\Box\beta)$ **or** $(t_1 \leq t_2)$: **forall** $s \in S$ **do**
      if $CheckEPPL(L(s), \delta_i)$ **then**
        $Label(s) = Label(s) \cup \{\Box\beta\}$ **or** $Label(s) \cup \{t_1 \leq t_2\}$;
      end
    end
    case $(\sim\delta_j)$: **forall** $s \in S$ **do**
      if $\delta \notin s$ **then**
        $Label(s) = Label(s) \cup \{\sim\delta_j\}$;
      end
    end
    case $(\delta_j \supset \delta_l)$: **forall** $s \in S$ **do**
      if $\delta_j \notin s$ **or** $\delta_l \in s$ **then**
        $Label(s) = Label(s) \cup \{\delta_j \supset \delta_l\}$;
      end
    end
    case $(\mathsf{EX}\delta_j)$: $LabelEX(\delta_i)$;
    case $(\mathsf{E}[\delta_j \mathsf{U} \delta_l])$: $LabelEU(\delta_i)$;
    case $(\mathsf{AG}\delta_j)$: $LabelAG(\delta_i)$;
  end
end

The soundness of this algorithm can be checked following the proof in [6] with minor adjustments, namely in the first **case**. If we use the algorithm presented in [19] to preform the EPPL checking, the temporal model checking procedure will run in $O(|\delta| \cdot (|\delta| \cdot |\Omega| \cdot |S| + |R|))$ time: the outer **for** runs $O(\delta)$ times, the EPPL MC cases take $O(2^n \cdot |\delta| \cdot |S|)$ time and the other cases take $O(|S| + |R|)$ time.

It is clear that Algorithm 3 can be adapted to check EpCTL formulas over models $M = (S, R, L)$ using Algorithm 2 instead of the general EPPL MC algorithm if we restrict ourselves to structures where the image of the map L is in the set of EPPL models induced by SPBCs.

Unfortunately, programs written with the probabilistic programing language presented in [19] do not generate such models and it is unlikely that a really useful language that generates them exists. In fact, a simple reatribution of a variable has the potential to break the acyclicity of a SPBC. We could circumvent this problem by introducing a new variable for each such command, but this would be unfeasible by obvious efficiency reasons. Another (more likely) solution, to be exploited in future work, is to use MTBDDs: Since a MTBDD is a canonical way to represent a

function $f : \{0,1\}^n \to \mathbb{R}$, an EPPL model can be represented by a MTBDD using as little space as possible. This representation can be further refined in some cases by using factorizations to decompose the set of propositional symbols into independent subsets, and representing each one with a (smaller) MTBDD. However, although canonical, these representations do not avoid taking exponential space in $|\Lambda|$.

Chapter 5

Conclusion

In this work, we presented in detail the implementation of a PSPACE model checking tool for non-reliable digital circuits using a very expressive probabilistic logic. The model checker itself is a flexible tool, with the potential to be adapted for more complex structures, albeit at the cost of computational resources, mainly space. Since it has been designed with efficiency concerns, several optimizations were implemented and thoroughly justified.

We also introduced an alternative way of representing EPPL models generated by probability spaces over valuations through the factorization of the joint probability distribution of the variables of the associated stochastic process. Although no efficiency improvement is guaranteed, only a little independency between the variables is needed to make this procedure worthwhile.

It is possible to expand the model checker to arbitrary EPPL models, despite the fact that doing so breaks the PSPACE complexity. We have already proposed in Chapter 4 an approach to this problem using MTBDDs. This data structure can easily be integrated in the present tool and seems promising in terms of efficiency. Another useful research path would be the development of a programming language that generates probabilistic Kripke structures whose labeling functions returned only SPBC generated EPPL models. We leave these approaches to be explored.

Building upon the last suggestion, the implementation of the extension of the MC to check over EpCTL formulas could also prove valuable. In fact, as we have seen in Chapter 4, a simple adaptation of any standard CTL model checker is enough to successfully implement an EpCTL one.

Overall, our work provides a useful and effective tool with clear applications in circuit reliability and furthers some previous work. It also raises a few interesting questions and problems which we believe may constitute an interesting topic of

further research.

Bibliography

[1] S. Basu, R. Pollack, and M. Roy. *Algorithms in Real Algebraic Geometry*. Springer, 2003.

[2] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE ToC*, 35(8):677–691, 1986.

[3] R. Chadha, L. Cruz-Filipe, P. Mateus, and A. Sernadas. Reasoning about probabilistic sequential programs. *Theoretical Computer Science*, 379(1-2):142–165, 2007.

[4] V. Chvátal. *Linear programming*. Freeman, 1983.

[5] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi terminal binary decision diagrams: an efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[6] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. *Handbooks of automated reasoning*, 190(3), 2007.

[7] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[8] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *STOC*, pages 169–180, 1982.

[9] R. Fagin and J. Y. Halpern. Reasoning about knowledge and probability. *J. ACM*, 41(2):340–367, 1994.

[10] R. Fagin, J. Y. Halpern, and N. Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87(1/2):78–128, 1990.

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

[12] H. Hermanns, M. Kwiatkwoska, G. Norman, D. Parker, and M. Siegle. On the use of *MTBDD*s for performability analysis and verification of stochastic systems. *Journal of logic and algebric programming*, 56(1-2):23–67, 2003.

[13] A. F. Karr. *Probability*. Springer-Verlag, 1993.

[14] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic model checking in practice: case studies with *PRISM*. *SIGMETRICS Perform. Eval. Rev.*, 32(4):16–21, 2005.

[15] Marta Kwiatkowska, Gethin Norman, and David Parker. *PRISM*: Probabilistic symbolic model checker. In *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 200–204, London, UK, 2002. Springer-Verlag.

[16] P. Mateus and A. Sernadas. Exogenous quantum logic. In *Proceedings of CombLog'04, Workshop on Combination of Logics: Theory and Applications*, pages 141–149, 2004.

[17] P. Mateus and A. Sernadas. Weakly complete axiomatization of exogenous quantum propositional logic. *Information and Computation*, 204(5):771–794, 2006.

[18] P. Mateus, A. Sernadas, and C. Sernadas. Exogenous semantics approach to enriching logics. In G. Sica, editor, *Essays on the Foundations of Mathematics and Logic*, volume 1, pages 165–194. Polimetrica, 2005.

[19] P.Baltazar and P.Mateus. Verifying probabilistic systems with *EpCTL*. *Awaiting publication*, 2008.

[20] P.Baltazar, P.Mateus, R.Nagarajan, and N.Papanikolaou. Exogenous probabilistic computation tree logic. *Electronic Notes in Theoretical Computer Science*, pages 1635–1790, 2001.

[21] A. Sernadas and C. Sernadas. *Foundations of logic and theory of computation*. College Publications, 2008.

EPPL model checker - User Manual

Welcome to the EPPL model checking tool for probabilistic Boolean circuits for unix-based OS.

This tool was developed as part of a master's degree thesis and we advise reading the main reference before using the program. This is only an operational manual, but we shall assume the reader is familiar with the basics of formal logic and model checking when employing technical terms.

Operational guide

If you want to use the MC as a black box, you need no compiling options, just open the console, specify the path up to the MCEPPL folder and type

```
path\MCEPPL\make run
```

If you choose to edit any of the files used, type

```
path\MCEPPL\make clean
```

followed by

```
path\MCEPPL\make
```

to recompile.

If you wish to add any files, you should probably edit the makefile in the folder.

When running the program, if you see something similar to Figure 1, you should be fine.

```
./EPPL
Welcome to the EPPL model checking tool for PBCs.
What do you wish to do?

1. Load a new PBC specification.
2. Model check an EPPL formula over the loaded PBC.
3. Get the probability of a basic formula in the loaded PBC
4. Quit.
```

Figure 1: EPPL MC tool environment

The current version of the MC allows you do do three operations: to load a new SPBC, to model check a global formula over the EPPL model generated by the loaded SPBC or to check the probability of a basic (propositional) formula over the EPPL model generated by the loaded SPBC.

SPBCs and EPPL formulas are syntaticaly complex, and it is very easy to make a mistake when writing them. For this reason, the program reads it's input from previously created files. This way, if you make a mistake, you won't need to rewrite the whole thing again.

A few notes on variable naming. One important thing you have to remember when writing formulas and SPBCS is *all variables and term variables must end with a whitespace*. This is a convention very easily overlooked, so please pay close attention to it.
Variable names cannot include the symbols "=", whitespaces (other than the ending one) or start with a "0" or a "1". So "var8 " is an acceptable name, but neither "var 8 " nor "var8" are. Avoid assigning your variables exotic names and everything will run smoothly.

The syntax of formulas and SPBCs is thoroughly explained in the main reference.

Computational representation of data structures

Even well designed algorithms can be severely hampered by a bad choice of data structure representations. To smooth the comprehension of the program, we now describe and justify the ones we use in our implementation:

- **Variables**: Variables are internally represented by positive ints; each time the parser reads a variable name that is not yet initialized, it increments a counter and initializes a fresh variable labeled by that number.
- **Term Variables**: Term variables are represented by negative ints lower than -30; each time the parser reads a term variable name that is not yet initialized, it decrements a counter and initializes a fresh term variable labeled by that number.

- **assignements**: assignements are connected lists of a structure consisting of one int (the internal representation of the term variable), one pointer to the real name of the term variable (as specified by the user), one float (the value attributed to the variable) and one pointer to the next term variable (figure 2).

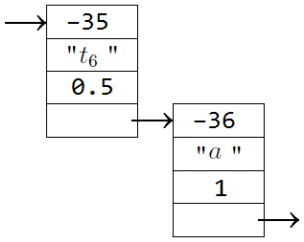

Figure 2: Part of an assignement

- **Connectives**: Connectives are represented by a structure composed of one of 23 reserved negative ints, standing for the connective itself and up to two pointers to the arguments of the connective;

- **Formulas**: Formulas (both global and basic) are represented by trees of connectives. Variables are abusively represented as 0-ary connectives, but easily distinguished because they are labeled with positive ints instead of negative ints. This representation allows for both an efficient interpretation of formulas (often, knowledge of the interpretation of one of the subtrees is enough to interpret the whole formula) and for easy inclusion of subformulas, an operation that will be commonly used in the program. It occupies linear space in the number of connectives and variables in the formula, having one node for each of these symbols (figure 3).

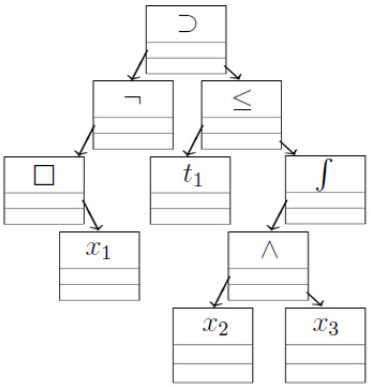

Figure 3: Representation of $\left[[\neg\Box(x_1)] \supset \left[t_1 \leq \{\int(x_2 \vee x_3)\}\right]\right]$

- **Valuations**: Valuations are represented by arrays of $n+1$ ints. The value of the variable labeling vertex i is represented in the $i-th$ element of the

array. A 2 represents a "don't-care" value, allowing for several valuations to be represented on the same array.

- **Factor**: A factor represents a mix between one vertex of a SPBC and the respective term in the product that describes the joint distribution of the random variables of the stochastic process. Is is a structure that consists of one int (the labeling variable), one float (the labeling real number), one pointer to the labeling formula, one pointer to the real name of the variable (as specified by the user) and one pointer to the next vertex, by numeric order of labeling variables.

- **Factorizations**: A factorization is the connected list of all factors. It is not a representation of the SPBC itself, because each "vertex" does not store the information of who its children are. This information is not relevant when computing the probabilities and is, therefore, discarded. In a factorization, there is a factor for each vertex of the SPBC, so this data structure occupies linear space in the number of vertices of the SPBC. Factorizations are actually the only structure needed to represent the EPPL model, so, as far as space is concerned, this is a very reasonable data structure. As we will see, time complexity will not be sacrificed by this decision (figure 4).

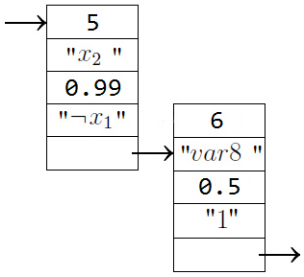

Figure 4: Part of a factorization

- **MTBDD**: MTBDDs are structures used to represent real valued Boolean functions in a canonic form. In this program, we use an external package with the implementation of these structures and of the functions that manipulate them. Despite having many proprieties that are not necessary in the present program, MTBDDs will play a major role in implementing a temporal extension of this model checker, and therefore, it was decided to use this package instead of others that are more limited in scope (figure 5).

In Table 1, at the end of the document, we present the correspondence between the symbols in the syntax described in the main reference, the actual symbols used in the implementation and their internal representation.

$x_1x_2x_3x_4$	$f(x_1, x_2, x_3, x_4)$
0100	3
0101	3
0111	2
1000	4
1010	2
1011	1
otherwise	0

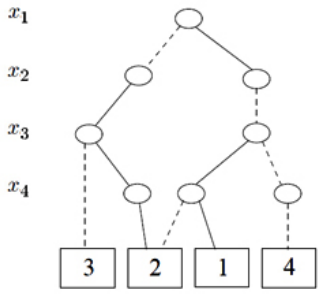

Figure 5: MTBDD for a Boolean function

Quick library of functions

We now provide a very brief description of the main functions in our program.

- factorization createfactorization(char* filename, int* n, deterministics dep)

  createfactorization takes a file where a SPBC is specified and recursively builds a factorization that will be stored in memory. Also taken as arguments are a pointer to an allocated int that keeps track of how many variables have been created so far, and a pointer to the list of deterministic gates and their respective formula (this list starts as empty).

- pformula createpformula(char* string, factorization fact, deterministics det)

  createpformula creates a basic (propositional) formula from a string (either the labeling formula in a SPBC description, a subformula of a global formula or an isolated basic formula in a file). As described in the previous section, formulas are trees of connectives, and the internal representation of variables are positive int. Therefore, one needs the factorization as a translator tool from the names of variables given as input to their internal representation. If any of the variables in the formula are induced by deterministic gates, they are substituted by the respective propositional formula in the process. This way, no variables induced by deterministic gates ever appear in internal representation of formulas.

- gformula createterm(char* string, list_terms list, factorization fact, deterministics det, int* t))

  createterm creates a term from a string . The second argument is used as a translator from input given term variable names to their internal representation. The factorization and list of deterministic gates are given as input

because it may be necessary to call createpformula in order to parse a basic subformula β in a term of the form $\int \beta$. The pointer to the allocated int keeps track of how many term variables have been created so far.

- gformula creategformula(char* string, factorization fact, list_terms terms, deterministics det, int* t)

  creategformula creates a global formula from a string (the sole line in a file). The other arguments are only needed because this function will call createterm and createpformula.

- float Measure(pformula formula, factorization fact, int n)

  Measure returns the probability of a given basic formula. It is essentially a cycle that, in each iteration, computes the next valuation that satisfies the formula (actually, it computes several valuations in each cycle, since we allow "don't care" values), uses a product-sum algorithm and the method proposed in Proposition 3.3.3 of the main reference to compute its (their) probability and adds it to the sum of the previously computed probabilities.

- float ProductSum(int* val, factorization fact, float p, int i, int n)

  ProductSum is a well known algorithm, used to compute a product over a set of terms where information someway propagates in a directed way from term to term (in our case, the information of the $i-th$ entry of a valuation may be needed only in factors with index equal or greater than i). Although this function is called an exponential number of times in $|\Lambda|$ (thus, not reducing the complexity class), this is still much more efficient than running through each valuation in $2^{|\Lambda}$ and checking its probability individually.

- int Box(pformula formula, int n)

  Box computes the satisfaction of $\Box\beta$, where β is represented by formula). Since createpformula substitutes all deterministic gate induced variables with their respective formula, by Theorem 3.5.8 of the main reference, all we need to do is run a conventional SAT algorithm over the negation of formula.

- float evaluateterm(gformula formula, list_terms terms, factorization fact, int n)

  evaluateterm recursively evaluates a propositional term, using the assignement given as input (which is stored in terms) for evaluating term variables and the

function Measure for measure terms.

- int evaluategformula(gformula formula, factorization fact, int n, list_terms terms)

  evaluategformula mimics the last part of Algorithm 2, evaluating global subformulas of the input formula using the above functions. However, unlike Algorithm 2, instead of starting by evaluating all subformulas, this routine tries to evaluate as little of them as possible, in order to reduce computing time.

EPPL *Symbol*	Computer Symbol	Internal Representation
$\cap$	&&	-1
$\cup$	||	-2
$\supset$	==>	-3
$\equiv$	<==>	-4
$\sim$	!	-5
$\Box$	#	-6
$\int$	$	-7
$<$	<	-8
$>$	>	-9
$\leq$	$\ll$	-10
$\geq$	$\gg$	-11
$=$	=	-12
$\wedge$	&	-13
$\vee$	|	-14
$\Rightarrow$	=>	-15
$\Leftrightarrow$	<=>	-16
$\neg$	~	-17
0 (propositional)	0	-18
1 (propositional)	1	-19
0 (term)	0	-20
1 (term)	1	-21
$+$	+	-22
.	*	-23

Table 1: Correspondence between syntaxes and internal representation