Reference: ……………

UNIVERSITY OF MOHAMED BOUDIAF – M'SILA

FACULTY OF MATHEMATICS AND INFORMATICS

Department of Computer Science

Thesis

Submitted in partial fulfilment of the requirements for the degree of

***DOCTORATE 3rd Cycle* in Computer Science**

Option: Advanced Information Systems

By:

Hichem DEBBI

Subject

Systems Analysis using Model Checking with Causality

Presented Publicly: 14/03/2015, **to the jury:**

B. Bouderah	Prof., University of M'sila	**President**
M. Bourahla	Dr., University of M'sila	**Reporter**
M. Ahmed-Nacer	Prof., Univeristy of USTHB	**Examiner**
A. Bilami	Prof., University of Batna	**Examiner**
D. Mihoubi	Prof., University of M'sila	**Examiner**

Academic Year: 2014/2015.

I would like to dedicate this thesis to the soul of my father, and my mother who really encouraged me to be here today; the person who always gave me hope.

Acknowledgements

First, my thanks go to my supervisor, Dr. Mustapha Bourahla for his ultimate support. He gave me the opportunity to learn so much about the domain of formal methods and formal verification. Thanks to him, I got much experience about a domain that I investigated and discovered for the first time. The thanks also go to the members of the jury, who give me the honor to evaluate my work. I would like first to thank Pr. Brahim Bouderah for being the president of this jury, and also the rest of the members of the jury, beginning by Pr. Mohamed Ahmed-Nacer, Pr. Azeddine Bilami and finally Pr. Douadi Mihoubi.

I would like also to thank my colleagues, especially Abdallah Arioua and Bilal Lounnas. Special thanks go to Abdallah Arioua for his valuable advises and concerns. My thanks also go to Pr. Stefan Leue from Konstanz University, for hosting me as an intern in his group, and I would like also to thank all the members of his group. I learned so much from them, and thanks to them I gained a good experience in my research field that helped me to complete my thesis. I would like to thank another person who inspired me as well; this person is Marly Roncken from Portland State University.

Finally, I'm very grateful for my family (mother, brothers and sisters) for their unlimited motivation and support. Without their help and support, I would never have this educational degree. Special thanks go to my brother Ali for his guidance and help with many issues.

ملخص

فحص النموذج هو واحد من الأساليب الأكثر شهرة التي تستخدم للتحقق من صحة الأنظمة . إعطاء نموذج النظام مجموعة من المواصفات التي هي مجموعة من الخصائص الرسمية للنظام، يتحقق البرنامج الخاص من صحة الخصائص , في حالة عدم توافق الخضائص، يتم إنشاء مثال مضاد يمثل مسار الخطأ. بفضل هذه الميزة، وهي القدرة على توليد مثال مضاد في حالة عدم توافق الخاصية، يستخدم التحقق من النماذج على نطاق واسع في فحص وتحليل وتصحيح النظم المعقدة.

ظهر فحص النموذج الاحتمالي كامتداد لنموذج التحقق الكلاسيكي للتحقق من الأنظمة التي تحمل سلوكا ستوكاستيكيا .فحص النموذج الاحتمالي يوظف العديد من الخوارزميات العددية لحساب ما ان كان الاحتمال يوافق الخاصية الاحتمالية، وبالتالي فإنه يمكن تحديد ما إذا كانت الخاصية الاحتمالية توافق أم لا العتبة المعطاة .في حالة انتهكت عتبة الاحتمال، يتم إنشاء مثال مضاد .

نظهر في هذه الأطروحة أنه مهمة توليد الأمثلة المضادة في مجال التحقق من النموذج الاحتمالي لديها جانب كمي .كما هو الحال في فحص النموذج التقليدي، في النموذج الاحتمالي التحقق من المثال المعاكس يجب أن يكون صغير والأكثر تأشيرا ليكون سهلا في التحليل . ومع ذلك، أمثلة معاكسة صغيرة وإرشادية فقط لا تكفي لفهم الخطأ، خصوصا أن المثال المعاكس الاحتمالي يتكون من مسارات متعددة وأنه احتمالي بطبيعته. ولذلك، فإن تحليل الأمثلة المعاكسة الاحتمالية أمر ضروري لفهم أفضل لسبب الخطأ.

تتناول هذه الأطروحة لأول مرة مهمة مكملة لتوليد الأمثلة المضادة الاحنمالية وهو تحليل الأمثلة المضادة الاحتمالية . نقترح العديد من الأساليب التشخيصية للأمثلة المعاكسة الاحتمالية باستخدام مفاهيم تتعلق بنظرية السببية . في وجود مثال مضاد لصيغة الاحتمالية أن لا يتوافق على نموذج احتمالي، ترشد هذه الأساليب المقترحة المستخدم إلى أكثر الأجزاء ذات الصلة من النموذج الذي أدت إلى الخطأ . نقيم طرقنا باستخدام العديد من دراسات الحالة.

في التحقق من النموذج الاحتمالي، تم تناول العديد من دراسات الحالة في العديد من المجالات. في السنوات الأخيرة كان هناك أيضا حضور كبير لاستخدام نموذج احتمالي لتقدير التدابير الكمية التي تساعدنا على فهم وتحليل ديناميكية وأداء النظم البيولوجية، والأنظمة المتطورة مثل أنترنت الأشياء. مع تزايد أهمية الفحص الاحتمالي للنماذج كإطار رسمي للتحقق التحليل الكمي للنظم الاحتمالية، نختم أطروحتنا باظهار كيف يمكن لهذه المجالات المختلفة أن تستفيد من أهمية الفحص الاحتمالي للنماذج من خلال دراسة إمكانية تطبيقه على اثنين من المجالات المختلفة، تحليل العلاج الطبي و تطبيقات معالجة الأحداث المركبة.

Abstract

Model checking is one of the most famous formal methods used for the verification of finite-state systems. Given a system model and such specification which is a set of formal proprieties, the model checker verifies whether or not the model meets the specification. In case the specification is not satisfied, a counterexample is generated as an error trace.

Probabilistic model checking has appeared as an extension of model checking for analysing systems that exhibit stochastic behaviour. Probabilistic model checking employs many numerical algorithms to compute the probability of the satisfaction of given temporal property, and thus it could determine whether a probabilistic property is satisfied or not given such threshold. In case the probability threshold is violated, a counterexample is generated.

In this thesis, we show that the task of counterexamples generation in probabilistic model checking has a quantitative aspect. As it is in conventional model checking, in probabilistic model checking the generated counterexample should be small and indicative to be easy for analysing. However, generating small and indicative counterexamples only is not enough for understanding the error, especially that probabilistic counterexample consists of multiple paths and it is probabilistic. Therefore, the analysis of probabilistic counterexamples is inevitable to better understand the error.

This thesis addresses for the first time the complementary task of counterexample generation in probabilistic model checking, which is the counterexample analysis. We propose many aided-diagnostic methods for probabilistic counterexamples based on notions related to causality theory. These methods guide the user to the most relevant parts of the model that led to the error. We evaluate our methods using many case studies.

In probabilistic model checking, several case studies in several domains have been addressed . In recent years there has been also a great attend to use probabilistic model checking to analyse the dynamic and the performance of biological systems. With the growing importance of probabilistic model checking as a formal framework for the verification and quantitative analysis of probabilistic systems, we investigate this importance by showing its applicability on two different domains, medical treatment analysis and probabilistic Complex Event Processing (CEP).

Résumé

Model checking est l'une des méthodes les plus utilisées pour la vérification des systèmes finis. Etant donné un modèle d'un système et une telle spécification, qui est un ensemble de propriétés formelles, le model checker (l'outil de vérification) vérifie si oui ou non le modèle satisfait la spécification. Dans le cas où la spécification n'est pas satisfaite, un contre-exemple est généré en tant que trace d'erreur.

Le probabilistic model checking est apparu comme une extension de model checking pour les systèmes qui présentent un comportement stochastique. Probabilistic model checking emploie de nombreux algorithmes numériques pour calculer la probabilité de la satisfaction de la propriété temporelle donnée, et donc pour déterminer si une propriété probabiliste est satisfait ou pas étant donné un tel seuil de probabilité. Dans le cas où le seuil de probabilité est violé, un contre-exemple est généré.

Dans cette thèse, nous montrons que la génération des contre-exemples dans le probabilistic model checking a un aspect quantitatif. Comme le model checking classique, dans le probabilistic model checking le contre-exemple généré devrait être significatif pour être facile à analyser. Cependant, la génération des contre-exemples indicatifs seulement n'est pas suffisante pour la compréhension de l'erreur, d'autant que les contre-exemples probabilistes se composent de chemins multiples et sont tout d'abord probabilistes. Par conséquent, l'analyse de contre-exemples probabilistes est nécessaire afin de mieux comprendre l'erreur.

Cette thèse aborde pour la première fois la tâche complémentaire de génération de contre-exemple, qui est l'analyse de contre-exemple dans le probabilistic model checking. Nous proposons de nombreuses méthodes de diagnostic pour les contre-exemples probabilistes basées sur des notions de la théorie de la causalité. Étant donné un contre-exemple pour la propriété probabiliste qui ne détient pas plus de modèle probabiliste, ces méthodes guident l'utilisateur pour les parties les plus pertinentes du modèle qui ont conduit à l'erreur. Nous évaluons nos méthodes en utilisant plusieurs études de cas.

Dans le probabilistic model checking, plusieurs études de cas dans plusieurs domaines ont étés réalisées. Au cours des dernières années, il y a eu une grande tendance à utiliser le probabilistic model checking pour analyser la dynamique et la performance des systèmes biologiques. En raison de l'importance notable de probabilistic model checking comme un outil formel pour la vérification et l'analyse quantitative des systèmes probabilistes, nous finissons en examinant cette importance et montrant différents domaines susceptibles de bénéficier de probabilistic model checking en étudiant son applicabilité sur deux domaines différents, le traitement médical et la technologie orientée évènement, Le Complex Event Processing (CEP) probabiliste.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

System Verification is an important task to do before building any information and communication system. System verification is the process of checking that such a system behaves correctly by making sure that the system meets design *specification* . To this end we employ such methods like formal methods to prove the correctness of a system to get more precise verification and gain more time. *Formal methods* refer to the branch of applied mathematics for modelling and analysing information and communication systems, and its efficiency is due mainly to employing mathematical rigour. *Model checking* is a formal method used for the verification of finite-state systems. Given a system model and such specification which is a set of formal proprieties, the model checker (model checking tool) verifies whether or not the model meets the specification. This set of properties is given usually in a logical formalism; the mostly used is the temporal logic such as *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)*. In case the specification is not satisfied, an error trace called *counterexample* is generated as an indication for the error.

Thanks to this feature, which is the ability to generate a counterexample when the property is not satisfied, model checking has been widely used for analysing and debugging complex systems. Roughly speaking, a counterexamples is an error trace, by analysing it the user can locate the source of the error. Counterexample generation has been investigated since the birth of model checking and it has its origins in graph theory trough the problem of fair cycle and *Strongly Connected Component (SCC)* detection. Due to its importance to the designer, many works have investigated the problem of generating small counterexamples, because small counterexamples are usually easy to be analysed, and thus are the most likely for debugging purpose. Regardless of the quality of the counterexample generated, analysing a counterexample is not a trivial task; therefore, *counterexample analysis* is inevitable for *debugging*, because it helps the designer to better understand the failure, and thus leads to redesign a robust system.

Probabilistic model checking has appeared as an extension of model checking for analysing systems that exhibit stochastic behaviour. Several case studies in several domains have been addressed from randomized distributed algorithms and network protocols to biological systems and cloud computing environments. These systems are described usually using *Discrete-Time Markov Chains (DTMC)*, *Continuous Time Markov Chains (CTMC)* or *Markov Decision Processes (MDP)*, and verified against properties specified in *Probabilistic Computation Tree Logic (PCTL)* or *Continuous Stochastic Logic (CSL)*. In recent years there have been also a great attend to use probabilistic model checking for the estimation of quantitative measures that help us to understand and analyse the dynamic and the performance of biological systems, and evolving systems such as quantum computing and internet of things.

For counterexample generation in *probabilistic model checking*, many algorithms and approaches have been proposed. But unlike the previous methods proposed for conventional model checking that generate the counterexample as a single path ending with bad state representing the failure, the task in probabilistic model checking is quite different. The counterexample in probabilistic model checking is a set of evidences or diagnostic paths that satisfy path formula and their probability mass violates the probability threshold. As the linear representation of counterexamples in CTL model checking has been debated, where tree representation has been proposed as a better representation, the path-based representation of probabilistic counterexamples has also been debated.

As it is in conventional model checking, in probabilistic model checking the generated counterexample should be small and most indicative to be easy for analysing. In probabilistic model checking, this task is more challenging since the counterexample consists of multiple paths. However, generating small and indicative counterexamples only is not enough for understanding the error. Therefore, as it was done in conventional model checking, counterexample analysis for probabilistic model checking is highly required to locate the causes of the error, especially that *probabilistic counterexample* consists of multiple paths instead of single path, and it is probabilistic. In probabilistic model checking the returned counterexample delivers a quantitative information, therefore counterexample analysis should deliver quantitative explanations as well.

This thesis explores mainly for the first time the debugging of probabilistic counterexamples as a complementary task for counterexample generation. In this thesis, we propose novel methods based the definition of *causality* by Halpern and Pearl, and its quantitative extensions *responsibility* and *blame* , as well as using *regression* to reason formally about the causes, and deliver quantitative diagnoses for the error generated while modelling and analysing stochastic systems. In addition we showed the importance of using probabilistic model checking for verifying and analysing probabilistic and non-deterministic systems by investigating two case studies in two different domains.

1.2 Contributions

The main contribution is that we address for the first time the analysis of probabilistic counterexamples through adopting theory of causality to generate the causes and estimate their effect on the error. Before going through, we survey some works on counterexamples in model checking from different aspects, generation, debugging and its usefulness for other purposes to give a better understanding and historical overview about the problem. Besides, we show how probabilistic model checking has become an important task to verify the correctness and analyse the complex stochastic systems of now-days. To do so, we address two applications domains in which probabilistic model checking plays an important role. Below we cite the contributions in detail:

This thesis emphasis the usefulness of counterexamples and its importance to engineers especially for debugging. Detailed history about counterexamples generation and analysis is presented in both conventional model checking and probabilistic model checking. We mainly focus on two aspects, counterexample generation and counterexample analysis. Nevertheless, other important uses of counterexamples are presented. We believe that the survey presented here on counterexamples covers most important and recent works on counterexamples. Most of the works presented in this survey, especially for counterexample debugging stand as a basic background to our work for debugging probabilistic counterexamples.

This thesis proposes novel methods for analysing counterexamples of probabilistic models. All the methods presented here are based on the theory of causality by Halpern and Pearl. This theory has been adopted in many domains including model checking, especially for explanation purpose. We also adopt two more key notions about quantitative measures of causality, which are responsibility and blame to give weights to the causes generated by our methods. Our methods can be applied to the majority of probabilistic models, Discrete-time Markov chains (DTMCs), Continuous time Markov chains (CTMCs) and Markov decision process (MDPs).

Our approach does not ignore the previous approaches of generating probabilistic counterexamples, but instead it is based on them. Our approach for error explanation is based directly on the most indicative counterexamples. Hence the input of our algorithms is got from counterexample generation tool, which is in our work *DiPro* tool. While this tool aims to deliver small and indicative counterexamples, our aim is to extract the most relevant causes from these counterexamples.

Our methods are applied on many case studies and report extensive experiments that showed their effectiveness. Even with respect to large models, our methods show promising results in term of diagnoses generated and especially time execution, which proves that it is possible to deliver a practical debugging tool based on the methods proposed for probabilistic models.

This thesis shows how many tools can be used in complementary way to analyse probabilistic models. Although probabilistic model checker like *PRISM* does not generate counterexamples, it is still used by other tools like DiPro for computing the probabilities, and thus is still important for generating counterexamples. Our methods perform directly on the counterexamples generated by DiPro. There is also AMOS tool for regression analysis that acts on the output of one of the methods proposed.

This thesis introduces for the first time a context or application domain in which all the theoretical notions causality, responsibility and blame are adopted together. Although causality and responsibility have been adopted before in other practical domains, for blame this is the first attempt. Since the difference between responsibility and blame has been debated in theoretical and philosophical literature, where they are sometimes even considered the same thing, we believe that this thesis enriches the discussion about this issue and could be very helpful for theoretical community through showing real examples about adopting responsibility and blame in complementary way.

In this thesis we show how probabilistic model checking can play a major role in verifying the correctness and analysing quantitative properties of stochastic and non-determinism systems. We show how we can use probabilistic model checking as a formal verification framework for probabilistic CEP applications. *Probabilistic CEP* refers to the set of applications of treating events under uncertainty. We also show the usefulness of probabilistic model checking for the first time in a non-trivial domain, which is the *medical treatment* . We show how probabilistic model checking serves as a formal and logical framework for modelling and analysing medical treatment problems described as MDPs. We will show the effectiveness of our approach for modelling and especially cost-effective analysis of MDPs through investigating a case study that concerns the optimal timing of living donor liver transplantation.

1.3 Outline

In chapter 2, we give an introduction to model checking, probabilistic model checking and causality. Hence, this chapter presents some important definitions that we find along the thesis and served as a background for our work.

Since counterexample is in the heart of our thesis, we dedicate chapter 3 for surveying major aspects and problems related to counterexample in model checking.

In chapter 4, we present our methods for analysing probabilistic counterexamples of probabilistic models. Two algorithms are proposed for each type of probabilistic models (deterministic and non-deterministic), Discrete Time Markov Chain and Continuous Time Markov Chain (CTMCs) from a side and Markov Decision Processes (MDPs) from another . Experimental results for both of the algorithms proposed are presented.

In chapter 5, we present an additional method for explaining probabilistic counterexamples using causality and regression in complementary way. Experimental results are also presented and compared to the previous method.

In chapter 6, we introduce the use of probabilistic model checking for modelling and analysing systems that exhibit stochastic and non-determinism behaviour. We investigate its applicability on two different domains, probabilistic Complex Event Processing (CEP) and medical treatment analysis.

Finally a conclusion and future works and directions are presented.

1.4 Own Publications

The first work was published in the Journal of Computing and Information Technology (CIT) [71]. It included the first approach proposed for debugging probabilistic counterexamples. The second work which is an extension of the first one by proposing the notion of criticality and responsibility that goes together with the notion of cause having the highest probability was published in the Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign (Memocode) [72]. The work that investigates the use of probabilistic model checking for medical treatment analysis was published in the International Journal of Biomedical Engineering and Technology (IJBET)[74]. The work in which we proposed an approach for verifying probabilistic CEP applications was published in the First International Symposium on Informatics and its Applications(ISIA) [73].

Chapter 2

Definitions and Theoretical Background

2.1 Model Checking

2.1.1 Introduction

System verification is an important task to do before building any information and communication system. System verification is the process of checking that such a system behaves as it is intended by making sure that the system meets design specification. The specification constitutes of set of elementary properties obtained from the supposed normal behave of the system, such as , a system should never crashes or it should always complete such task ..., and a defect arises once the system does not fulfil this specification.

One of the most known and most successful methods used in systems verification are formal methods. Formal methods are mathematical based methods for modelling and analysing information and communication systems, and its efficiency is due mainly to employing mathematical rigour. The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration) about the use of formal methods concludes that:

"Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers."

One of the formal methods that have known a great success over other formal methods is *model checking* . Model checking is an automatic technique for verifying the correctness of finite-state systems in exhaustive manner. Given a model of the system, the role of model checker is to check whether the model meets such specification and thus it always terminate with a Yes/no answer. In case the specification is not satisfied, a counterexample is generated as an error trace, where the error could emerge from incorrect modelling of the system or from the specification itself . To do so, model checking algorithms explored all the possible states the system could be in, and therefore with large systems this could be very consuming especially from memory perspective, this problem is called "state explosion problem". Actually, this is the main problem of model checking that has been dealt with since the birth of model checking. The model checking process consists of the main following tasks:

Modelling: This task aims to deliver a model of the system using some model description language that can be accepted by a model checker. Despite the language used, it generally enables the representation of the system as finite-state automata, where states comprise information about the current values of variables and transitions describe how the system evolves from one state into another. In model checking, we refer to the transition system describing the behaviour of the system *Kripke structure* .

Specification: Before performing verification, a set of properties that should be satisfied by the model must be delivered. This set of properties is given usually in a logical formalism, the mostly used is *temporal logic* since it is capable of representing how the system's behaviour evolves over time, where temporal logic formula is interpreted in the term of Kripke structure. Temporal logic is an extension of traditional propositional logic with modal operators. According to what we assume about time, temporal logics are either linear (Linear Temporal Logic (LTL)), or branching(tree) (Computation Tree Logic (CTL)). Using temporal logics we can express two main types of properties:

Safety properties. state that something bad never happens, a good example of that is mutual exclusion that states that having two processes in their critical section simultaneously should never happen.

Liveness properties. state that something good eventually happens, As an example: a message sent will be eventually received.

Verification: this task is the task in which model checking takes a place, the model checker employing such algorithms explores all the states of the model to check the correctness of the properties at hand. In case negative result is delivered, the phase of analysis comes to help to find the source of the error given the counterexample generated, which can lead to redesign the system and replicate the verification process. The algorithms used for the verification are ranged in two main categories, explicit state algorithms that run on the entire model, or symbolic algorithm that employee symbolic techniques such as binary

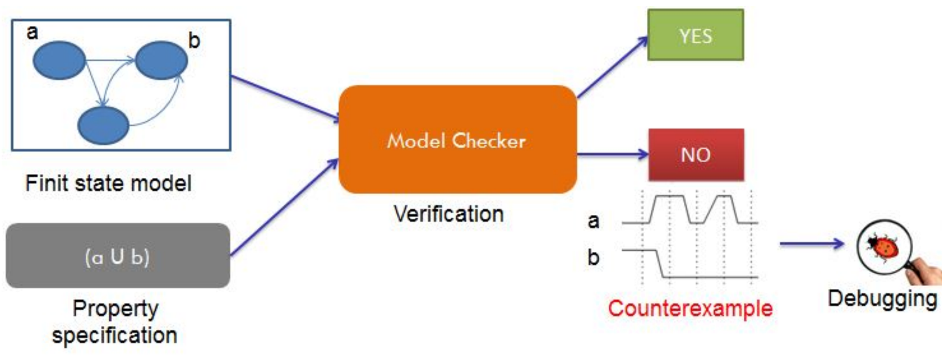


Fig. 2.1 Model Checking

decision diagrams for representing the state space.

2.1.2 Preliminaries and Definitions

Definition 2.1.1. (**Kripke Structure**) A Kripke structure is a tuple $M = (AP, S, s_0, R, L)$ that consists of a set AP of atomic propositions, a set S of states, $s_0 \in S$ an initial state, a total transition relation $R \subseteq S \times S$ and a labelling function $L : W \rightarrow 2^{AP}$ that labels each state with a set of atomic propositions.

Definition 2.1.2. (**Büchi Automaton**) A Büchi automaton is a tuple $B = (S, s_0, E, \sum, F)$ where S is a finite set of states, $s_0 \in S$ is the initial state, $E \subseteq S \times S$ is the transition relation,$\sum$ is a finite alphabet, and $F \subseteq S$ is the set of accepting or final states.

We use Büchi automaton to define a set of infinite words of an alphabet. A path is a sequence of states $(s_0 s_1 ..., s_k)$, $k \geq 1$ such that $(s_i, s_{i+1}) \in E$ for all $1 \leq i < k$. A path $(s_0 s_1 ..., s_k)$ is a cycle if $s_k = s_1$, the cycle is accepting if it contains a state in F. A path $(s_0 s_1 ..., s_k ....s_l)$ where $l > k$ is an accepting if $s_k ... s_l$ forms an accepting cycle. We call a path that starts at the initial state and reaches an accepting cycle an accepting path or counterexample (see Fig 2.1). A minimal counterexample is an accepting path with minimal number of transitions.

Linear Temporal Logic (LTL) and Computation Tree Logic (CTL)

The syntax of LTL state formula over the set AP are given as follows :

$$\varphi ::= true | a | \neg \phi | \phi_1 \wedge \phi_2 | \bigcirc \phi | \phi_1 U \phi_2$$

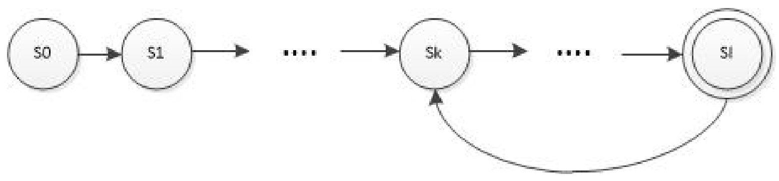


Fig. 2.2 Accepting path (Counterexample)

where $a \in AP$ is an atomic proposition. The Other Boolean connectives can be simply derived using the Boolean connectives $\neg$ and $\wedge$. The *eventual* operator F and the *always* operator and G can be easily derived using the temporal operator U.

Given a path $\pi = s_0 s_1 \ldots$ and an integer $j \geq 0$, where $\pi[j] = s_j$, T, such that $Words(\varphi) = \{\pi \in (2^{AP})^{w})\sigma \models \varphi\}$, the semantics of LTL formulas for infinite words over 2^{AP} are given as follows:

$\pi \models true \Leftrightarrow true$
$\pi \models a \Leftrightarrow a \in L(s_0)$
$\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi$
$\pi \models \varphi_1 \wedge \varphi_2 \Leftrightarrow s \models \varphi_1 \wedge s \models \varphi_2$
$\pi \models \bigcirc\phi \Leftrightarrow \pi[1] \models \phi$
$\pi \models \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq 0 . \pi[j] \models \phi_2 \wedge (\forall 0 \leq k < j . \pi[k] \models \phi_1)$

Verifying whether a finite state system described in Kripke structure A_M satisfies an LTL property φ reduces to the verification that $A = A_M \cap A_{\neg\varphi}$ has no accepting path, where $A_{\neg\varphi}$ refers to the Büchi automaton that violates φ, $L_\omega(A) = Words(\neg\varphi)$. We call this procedure a test of emptiness. So, In case $A_M \cap A_{\neg\varphi} \neq \emptyset$ a counterexample is generated.

We use the Computation Tree Logic (CTL) for specifying properties of systems described using Kripke Structures. The CTL formulas are evaluated over infinite computations produced by Kripke structure K. A computation of a Kripke structure is an infinite sequence of states $s_0 s_1, \ldots$ such that $s_i, s_{i+1} \in R$ for all $i \in N$. We denote by $Paths(s)$ the set of all paths starting at s. The syntax of CTL state formula over the set AP are given as follows :

$$\phi ::= true | a | \neg\phi | \phi_1 \wedge \phi_2 | \exists\varphi | \forall\varphi$$

where $a \in AP$ is an atomic proposition and φ is a path formula. The path formulas are formed according to the following grammar:

$$\varphi ::= \bigcirc\phi | \phi_1 U \phi_2$$

We denote by $K,s \models \phi$ the satisfaction of CTL formula at a state s of K. The semantics defined by the satisfaction relation for a state formula are given as follows

$K,s \models true \Leftrightarrow true$
$K,s \models a \Leftrightarrow a \in L(s)$
$K,s \models \neg\phi \Leftrightarrow s \not\models \phi$
$K,s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \wedge s \models \phi_2$
$K,s \models \exists\varphi \Leftrightarrow$ for some $\pi \in Paths(s)$, $\pi \models \varphi$
$K,s \models \forall\varphi \Leftrightarrow$ for all $\pi \in Paths(s)$, $\pi \models \varphi$

Given a path $\sigma = s_0 s_1 ...$ and an integer $j \geq 0$, where $\sigma[j] = s_j$, The semantics of path formulas are given as follows:

$K,\pi \models \bigcirc\phi \Leftrightarrow \pi[1] \models \phi$
$K,\pi \models \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq 0.\pi[j] \models \phi_2 \wedge (\forall 0 \leq k < j.\pi[k] \models \phi_1)$

In case the Kripke structure violates the specification $K \not\models \phi$ a counterexample will be generated. For CTL model checking , the counterexample is also expected to be a Kripke structure, just it should be suitable for explaining the violation [51].

Both LTL and CTL are considered as sub-logics or fragments of the logic CTL* introduced by Emerson and Halpern [57]. CTL is the subset of CTL* where each path operator $\bigcirc$ and U must immediately preceded by path quantifiers A or E, whereas LTL is the subset of CTL* that consists of formulas that have the form Af, where f is a path formula in which the only state formulas are just atomic propositions [60]. $ACTL$ is the analogue fragment of CTL and thus of CTL*, where the only quantifier allowed is A. Using CTL* we can express formulas of the form $A(FGp) \vee AG(EFp)$, which is a disjunction of LTL and CTL formula.

Generally two types of properties can be expressed using temporal logics: *Safety* and *Liveness*. Safety proprieties state that something bad never happens, a simple example of that is the LTL formula $G\neg error$ that means that error is never occurred. Liveness properties state that something good eventually happens, a simple example of that is the CTL formula $(AGreq \rightarrow AFgrant)$ that means that every request is eventually granted.

Strongest Connected Component

A graph is a pair $G = (V,E)$, where V is a et of states and $E \subseteq V \times V$ is the set of edges. A path is a sequence of states $(v_1,...,v_k)$, $k \geq 1$ such that $(v_i, v_{i+1}) \in E$ for all $1 < i \leq k$. Let π be a path, the length of π is defined by the number of transitions and is denoted by $[\pi]$. We say that we can reach a vertex u from a vertex v if there exists a path from v to u. We define a

Strongly Connected Component (SCC) as a maximal set of states $C \subseteq V$ such that for every pair of vertices $u, v \in C$, u and v are mutually reachable. A SCC C is trivial if $C = \{v\}$, or otherwise C is non-trivial if for every $u, v \in C$ there is a non-trivial path from u to v.

2.2 Probabilistic Model Checking

2.2.1 Introduction

In real life, systems are subject to phenomena of stochastic nature. For instance, it is usually impossible to totally guarantee the correctness of "it is impossible that the process fails" or "a message sent never lost". As a result, we should guarantee instead that " with 0.01 chance the process will fail " or "with chance 0.99 the message will not be lost" . Probabilistic model checking has appeared as an extension of model checking for analysing this kind of systems that exhibit stochastic behaviour. In probabilistic model checking, the model is constructed by assigning probabilities to the transitions between states of the system, and the specifications will be also subjected to deal with probabilities thresholds.

In all information and communication systems, Markov chains have been proved as the most probabilistic operational model, in probabilistic model checking are also the most used. Roughly speaking, Markov chains are transition systems with probability distributions over the states. In probabilistic model checking , the probabilistic systems are usually described using Discrete-Time Markov Chains (DTMC) or ContinuousTime Markov Chains (CTMC) , and Markov Decision Processes (MDP) for non-diterminismtic systems, and verified against properties specified in Probabilistic Computation Tree Logic (PCTL) [108] or Continuous Stochastic Logic (CSL) [31, 32]. While we use PCTL for specifying properties of DTMCS, we use CSL for specifying properties of CTMCs. probabilistic model checking process like conventional model checking consists of the same steps, modelling, specification and verification (see Fig 2.3).

While the algorithms used in conventional model checking are based on graph analysis, in probabilistic model checking the algorithms used reduce to solving a system of linear equations. Nevertheless, there are other model checkers [112, 183] that employ statistical execution sampling for the verification . We also find other approaches that tend to use automata-based techniques [174]and tableau-based techniques [64] to check linear temporal logic (LTL) formulae.

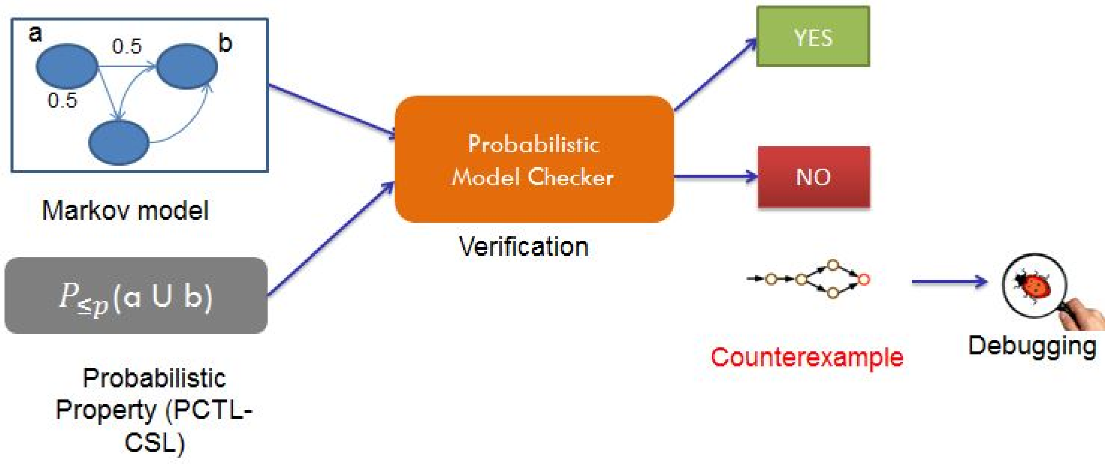


Fig. 2.3 Probabilistic Model Checking

2.2.2 Preliminaries and Definitions

Probabilistic Models

Definition 2.2.1. **(Discrete Time Markov Chain (DTMC))** A *Discrete-Time Markov Chain (DTMC)* is a tuple $D = (S, s_{init}, P, L)$, such that S is a finite set of states, $s_{init} \in S$ the initial state, $P : S \times S \to [0,1]$ represents the transition probability matrix, $L : S \to 2^{AP}$ is a labelling function that assigns to each state $s \in S$ the set $L(s)$ of atomic propositions.

An infinite path σ is a sequence of states $s_0 s_1 s_2 \ldots$, where $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. A finite path is finite prefix of an infinite path. We define a set of paths starting from a state s_0 by $Paths(s_0)$. The underlying σ-algebra is formed by the cylinder sets which are induced by finite paths in $Paths(s_0)$. The probability of this *cylinder set* is:

$$P(\sigma \in Paths(s_0) | s_0 s_1 \ldots s_n \text{is a prefix of } \sigma) = \prod_{i \leq 0 < n} P(s_i, s_{i+1}) \tag{2.1}$$

Definition 2.2.2. **(Continuous Time Markov Chain (CTMC))** A *Continuous Time Markov Chain (CTMC)* is a tuple $C = (S, s_{init}, \mathfrak{R}, L)$, such that S is a finite set of states, $s_{init} \in S$ the initial state, $\mathfrak{R} : S \times S \to \mathbb{R}_{\geqslant 0}$ represents the transition rate matrix, $L : S \to 2^{AP}$ is a labelling function that assigns to each state $s \in S$ the set $L(s)$ of atomic propositions.

Comparing to DTMC, the main difference is that with DTMC we have the transition probability matrix that corresponds to discrete-time steps, whereas with CTMC, the transition can occur in real-time, and thus are presented by the transition rate matrix, where every time rate of transition from s to s' is given by $\mathfrak{R}(s, s')$. This parameter represents a negative exponential distribution that contributes to computing the transition probability within t time units.

Definition 2.2.3. (Markov Decision Process (MDP)) A *Markov Decision Process (MDP)* is a tuple $M = (S, s_{init}, A, P, L)$, where S is a finite set of states, $s_{init} \in S$ is the initial state, A is a set of actions , $P : S \times A \times S \rightarrow [0,1]$ is a probability transition function such that for every state $s \in S$ and an action $\alpha \in A$: $\sum_{s' \in S} P(s, \alpha, s') \in \{0,1\}$, and $L : S \rightarrow 2^{AP}$ is a labelling function that assigns to each state $s \in S$ a set of atomic propositions.

At each state s, the probability of moving to a successor state s' by taking an action α is given by $P(s, \alpha, s')$. We say that an action α is enabled in state s if and only if $\sum_{s' \in S} P(s, \alpha, s') = 1$, otherwise the action α is disabled. For each state $s \in S$ there is at least one action enabled. We denote the set of actions enabled from a state s by $A(s)$.

An *infinite path* σ is an infinite sequence $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 ...$ with $\alpha_i \in A(s_i)$ such that $P(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \geqslant 0$. We define the set of infinite paths starting from a state s_0 by $Paths(s_0)$. A *finite path* is finite prefix of an infinite path. We denote by $FinitePaths(s_0)$ the finite paths starting from a state s_0. For Discrete-time Markov chains (DTMCs), the underlying σ-algebra is formed by the cylinder sets which are induced by $FinitePaths(s_0)$. For MDPs, computing the probabilities of paths must rely on the resolution of non-determinism, which is performed by a scheduler. A scheduler d resolves the non-determinism by taking in each state one of the enabled actions $\alpha \in A(s)$, thus resulting in DTMC for which the probability of paths is measurable. Then, the underlying σ-algebra is formed by the cylinder sets which are induced by finite paths under this scheduler denoted $FinitePaths_d(s_0)$. The probability of this cylinder set is computed by using the following formula :

$$P_d(\sigma \in FinitePaths_d(s_0) | \sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_{n-1}} s_n) = \prod_{i \leq 0 < n} P(s_i, \alpha_i, s_{i+1}) \tag{2.2}$$

Probabilistic Logics

The *Probabilistic Computation Tree Logic (PCTL)* [108] has appeared as an extension of CTL for the specification of systems that exhibit stochastic behaviour. We use the PCTL for defining quantitative properties of DTMCs. PCTL state formulas are formed according to the following grammar:

$$\phi ::= true | a | \neg \phi | \phi_1 \wedge \phi_2 | \mathbf{P}_{\sim p}(\varphi)$$

Where $a \in AP$ is an atomic proposition, φ is a path formula, **P** is a probability threshold operator, $\sim \in \{<, \leq, >, \geq\}$ is a comparison operator, and p is a probability threshold. The path formulas φ are formed according to the following grammar:

$$\varphi ::= \phi_1 \mathbf{U} \phi_2 | \phi_1 \mathbf{W} \phi_2 | \phi_1 \mathbf{U}^{\leq n} \phi_2 | \phi_1 \mathbf{W}^{\leq n} \phi_2$$

Where ϕ_1 and ϕ_2 are state formulas and $n \in N$. As in CTL, the temporal operators (**U** for strong until, **W** for weak (unless) until and their bounded variants) are required to be immediately preceded by the operator **P**. The PCTL formula is a state formula, where path formulas only occur inside the operator **P**. The operator **P** can be seen as a quantification operator for both the operators $\forall$ (universal quantification) and $\exists$ (existential quantification), since the properties are representing quantitative requirements.

The semantics of a PCTL formula over a state s (or a path σ) in a DTMC model $D = (S, s_{init}, P, L)$ can be defined by a satisfaction relation denoted by $\models$. The satisfaction of $\mathbf{P}_{\sim p}(\varphi)$ on DTMC depends on the probability mass of set of paths satisfying φ. This set is considered as a countable union of cylinder sets, so that, its measurability is ensured.

The semantics of PCTL state formulas for DTMC is defined as follows:

$s \models true \Leftrightarrow true$
$s \models a \Leftrightarrow a \in L(s)$
$s \models \neg\phi \Leftrightarrow s \not\models \phi$
$s \models \phi_1 \wedge \phi_2 \Leftrightarrow s \models \phi_1 \wedge s \models \phi_2$
$s \models \mathbf{P}_{\sim p}(\varphi) \Leftrightarrow P(s \models \varphi) \sim p$

Given a path $\sigma = s_0 s_1 ...$ in D and an integer $j \geq 0$, where $\sigma[j] = s_j$, The semantics of PCTL path formulas for DTMC is defined as for CTL as follows:

$\sigma \models \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq 0 . \sigma[j] \models \phi_2 \wedge (\forall 0 \leq k < j . \sigma[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{W} \phi_2 \Leftrightarrow \sigma \models \phi_1 \mathbf{U} \phi_2 \vee (\forall k \geq 0 . \sigma[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{U}^{\leq n} \phi_2 \Leftrightarrow \exists 0 \leq j \leq n . \sigma[j] \models \phi_2 \wedge (\forall 0 \leq k < j . \sigma[k] \models \phi_1)$
$\sigma \models \phi_1 \mathbf{W}^{\leq n} \phi_2 \Leftrightarrow \sigma \models \phi_1 \mathbf{U}^{\leq n} \phi_2 \vee (\forall 0 \leq k \leq n . \sigma[k] \models \phi_1)$

The satisfaction of $\mathbf{P}_{\sim p}(\varphi)$ on DTMC depends on the probability mass of set of paths satisfying φ. This set is considered as a countable union of cylinder sets, so that, its measurability is ensured. A formula $\mathbf{P}_{\sim p}(\varphi)$ is satisfied on an MDP M if only if for every $d \in D$: $\mathbf{P}_d(\varphi) \sim p$, where D represents the set of all schedulers and $\mathbf{P}_d(\varphi)$ represents the probability of the set of all finite paths satisfying φ under the scheduler d.

The semantics of PCTL state and of path formulas for MDPs are defined as the same as for DTMCs, except that for model checking of MDPs we have to consider either maximizing or minimizing schedulers. let $P_{max}(\varphi)$ be the maximal probability of φ where $P_{max}(\varphi) = max\{P_d(\varphi)|d \in D\}$, and dually the minimal probability $P_{min}(\varphi)$ be the minimal probability of φ where $P_{min}(\varphi) = min\{P_d(\varphi)|d \in D\}$. For instance for properties of upper threshold , it is evident that $(M \not\models P_{\leq p}(\varphi)) \Leftrightarrow P_{max(\varphi)>p}$.

For specifying properties of CTMC, we use The *Continuous Stochastic Logic (CSL)* . CSL has the same syntax and semantics as PCTL, except that in CSL, the time bound in bounded until formula can be presented of an interval of non-negative reals. Before verifying CSL properties over CTMC, the CTMC has to be transformed to its embedded DTMC. Therefore, further description of CTMC model checking is not addressed in this thesis. We refer to [31, 32] for further details.

2.3 Causality

2.3.1 Introduction

All the definitions of causality in literature are mainly based on *counterfactual* reasoning, event A is a cause of event B if, had A not happened then B would not have happened. Following the intuition about cause by [122] that states that a cause is something that makes a difference, or an event that leads to an effect, Lewis proposed his approach for counterfactual modelling using the possible worlds semantics. By assuming an actual world and other alternative worlds, Lewis's definition for a counterfactual statement between two events a and b denoted $a \rightarrow b$ is subjected to the relation of comparative similarity between the actual world and the alternative ones. The causal dependence between a and b is concluded, if only if among all the worlds, we should find that the most similar one in which $\neg a$ there should be $\neg b$, which means that in order to say that $a \rightarrow b$, a and b should appear together in similar worlds, otherwise there is no causal dependence.

Unfortunately, this definition does not capture all the cases we face in real world. Let us take the major known example of Suzy and Billy who both pick up rocks and throw them at a bottle. Suzy's rock gets there first, shattering the bottle. Since both throws are perfectly accurate, Billy's would have shattered the bottle had it not been pre-empted by Suzy's throw. Thus, according to the counterfactual condition, Suzy's throw is not a cause for shattering the bottle. Halpern and Pearl [104] have addressed this issue by taking A to be a cause of B if B counterfactually depends on A under some contingency. For example, Suzy's throw is a cause of the bottle shattering because the bottle shattering counterfactually depends on Suzy's throw, under the contingency that Billy does not throw. Both of the definitions have been widely used in many domains especially for explanation.

2.3.2 Preliminaries and Definitions

Halpern and Pearl have extended the Lewis counterfactual model [140] to what they refer to as structural equations for modelling the causal influence made by random variables. According to them, any phenomena we notice around us can be described by random variables,while some variables have direct influence on each other called *endogenous variables* , other can be seen as contributing factors and called *exogenous variables* . Let us take an example of forest fire caused by match lit by an arsonist. As endogenous variables we could easily see that we have endogenous variable F represents the effect (fire) and L lighting represents the cause, but we can not ignore the existence of other factors that represent the set of exogenous variables such as: the dry of wood or the enough oxygen in the air.

Halpern and Pearl introduced the *causality model* M as a tuple $M = (U,V,F)$, where random variables are partitioned into two sets, the exogenous variables U, whose values are determined by factors outside the model M, but they should be represented to encode the context, and the endogenous variables V whose values are determined by set of functions F, where each $f_{V_i} \in F$ is a mapping from $U \cup (V \setminus V_i)$ to V_i. Thus, each f_{V_i} tells us the value of V_i given the values of all other variables in $U \cup V$.

We call a Boolean combination of primitive events a basic causal formula. We call a Boolean combination of basic causal formulas a causal formula. A causal formula φ is true or false in causal model, given a context. We write $(M,\overrightarrow{u}) \models \varphi$ if φ is true in causality model M given a context $\overrightarrow{u}$, where $\overrightarrow{u}$ represents a setting for the variables in U. We write $(M,\overrightarrow{u}) \models [\overrightarrow{Y} \leftarrow \overrightarrow{y}](X = x)$, where $\overrightarrow{Y} \subset V$, if X has a value x in M given a context $\overrightarrow{u}$ and the assignment $\overrightarrow{y}$ to $\overrightarrow{Y}$. The formulas that are allowed to be causes for φ are ones of the form $X_1 = x_1 \wedge ... \wedge X_k = x_k$ which is abbreviated to the form $\overrightarrow{X} = \overrightarrow{x}$. With all these definitions in hand, we can now give the definition of an actual cause by Halpern and Pearl.

Definition 2.3.1. (**Actual Cause**) we say that $\overrightarrow{X} = \overrightarrow{x}$ is an *actual cause* of φ in $(M,\overrightarrow{u})$ if the following holds:

1. $(M,\overrightarrow{u}) \models (\overrightarrow{X} = \overrightarrow{x}) \wedge \varphi$
2. There exists a partition $(\overrightarrow{Z},\overrightarrow{W})$ of V with $\overrightarrow{X} \subseteq \overrightarrow{Z}$ and some setting $(\overrightarrow{x}',\overrightarrow{w}')$ of the variables in $(\overrightarrow{X}',\overrightarrow{W}')$ such that if $(M,\overrightarrow{u}) \models Z = z^*$ for $Z \in \overrightarrow{Z}$, then
   a-$(M,\overrightarrow{u}) \models \left[\overrightarrow{X} \leftarrow \overrightarrow{x}',\overrightarrow{W} \leftarrow \overrightarrow{w}'\right] \wedge \neg\varphi$
   b- $(M,\overrightarrow{u}) \models \left[\overrightarrow{X} \leftarrow \overrightarrow{x},\overrightarrow{W} \leftarrow \overrightarrow{w}',\overrightarrow{Z}' \leftarrow \overrightarrow{z^*}\right] \wedge \varphi$ for all subsets $\overrightarrow{Z}'$ of $\overrightarrow{Z}$.

3. The set of variables $\overrightarrow{X}$ is minimal (no subset of $\overrightarrow{X}$ satisfies the conditions 1 and 2).

The first condition states that both $\overrightarrow{X} = \overrightarrow{x}$ and φ are true in the current context, given the variables $\overrightarrow{X}$ and their values $\overrightarrow{x}$. The second condition states that any change on $(\overrightarrow{X}, \overrightarrow{W})$ will change φ from true to false, changing $\overrightarrow{W}$ will have no effect on φ as long as the values of $\overrightarrow{X}$ are kept at the current values, even if all subsets $\overrightarrow{Z}'$ of $\overrightarrow{Z}$ are set to their original values in the current context. Minimality condition ensures that only elements in the conjunction $\overrightarrow{X} = \overrightarrow{x}$ are essential for changing φ from true to false. As a consequence of this condition, we can refer to a cause as $X = x$, because the causes can always be taken to be single conjuncts [10].

Based on this definition, [20] introduced the definition of *responsibility* . Responsibility extends the concept of all-or-nothing of the actual cause $X = x$ for the truth value of φ in $(M, \overrightarrow{u})$.

Definition 2.3.2. (**Responsibility**) The degree of responsibility of $X = x$ for the value of φ in $(M, \overrightarrow{u})$, denoted $dr((M, \overrightarrow{u}), X = x, \varphi)$ is 0, if $X = x$ is not a cause for φ, and otherwise is $1/(|\overrightarrow{W}| + 1)$.

That is, the degree of responsibility measures the number of changes that have to be made in $\overrightarrow{u}$ in order to make φ counterfactually depend on X, where $\overrightarrow{W}$ refers to the set of variables that satisfies the condition 2) in Definition 2.3.1. It is easy to adapt this definition to the example of Suzy and Billy throwing a rock on a bottle. Suzy's throw has a responsibility 1/2 for bottle shattering, since there is a contingency has to be made (Billy does not throw) in order to make the bottle shattering counterfactually dependent on Suzy's throw. Roughly speaking, the degree of responsibility of a cause A for B where B counterfactually depends on A is taken to be $1/(N+1)$, where N represents the minimal number of changes that must take place to obtain a contingency. Let us consider a vote that takes place between two persons A and B, where the number of voters is 11. let us consider that all of them voted to A, this makes the result $11 - 0$. By considering the mechanism of winning is majority-mechanism, each voter has a degree of responsibility $(1/(5+1)) = 1/6$, where 5 represents the number of voters that have to change their minds by voting to B in order to make each vote is critical. The degree of responsibility has been proved that it could provide a good measure for the degree of fault-tolerance in a system as well as in coverage [21].

Causality and responsibility rely on the assumption that the context is well-defined, when it comes to uncertainty about the context in which variables should have such values, we face in addition a question of blame, which action should be blamed for such outcome. According to Chockler et al. [20], we model this uncertainty as a pair (K, Pr), where K is a set of situations $(M, \overrightarrow{u})$ and Pr is a probability distribution over K, then the degree of blame that a setting X to x has for φ is computed based on the responsibility of $X = x$ for φ over the situations $(M, \overrightarrow{u}) \in K$.

Definition 2.3.3. (Blame) The degree of blame of $X \leftarrow x$ for φ relative to (K, Pr), denoted $db(K, Pr, X \leftarrow x, \varphi)$ is $\sum_{(M,\overrightarrow{u}) \in K} dr((M_{X \leftarrow x}, \overrightarrow{u}), X = x, \varphi) Pr((M, \overrightarrow{u}))$

Because responsibility is directly based on the definition of actual cause, it also represents actual state, whereas blame is relative to an epistemic state, which is a set of situations before the action considering as a cause takes place. Let us consider an example of two hunters hunting a prey. If we consider that we gave them two rifles where one has live bullets, the other has blanks, and the situation is that neither of them know about it. By firing on the prey, only the one who has live bullets is considered as a cause for the prey's death, and thus has responsibility 1. However, both of them share the blame for prey's death $1/2$. This example pretty shows the intuitive difference between the two definitions.

Chapter 3

Counterexamples in Model Checking

3.1 Introduction

One of the major advantages of Model checking over other formal methods its ability to generate a counterexample when the model falsifies such specification. The counterexample is an error trace, by analysing it the user can locate the source of the error. The original algorithm for counterexample generation was proposed by [52] and was implemented in most symbolic model checkers. This algorithm of generating linear counterexamples for ACTL, which is a fragment of CTL was later extended to handle arbitrary ACTL properties using the notion of tree-like counterexamples [53]. Since then, many works have addressed this issue in conventional model checking.

Counterexample generation has its origins in graph theory trough the problem of fair cycle and Strongly Connected Component (SCC) detection, because model checking algorithms of temporal logics employ cycle detection , and technically a finite system model is determining a transition graph [60]. The original algorithm for fair cycle detection in LTL and CTL model was proposed by [84], since then, many variants of this algorithm and new alternatives were proposed for LTL and CTL model checking. In section 3.2 we will investigate briefly the problem of fair cycles and SCCs detection.

While the early works introduced by Clarke and Emerson [83] have investigated the problem of generating counterexample in general manner, which leads to practical implementation within well known model checkers , The open problem that emerged was the quality of the counterexample generated and how it really serves the purpose. As a result, in the last decade many papers have considered this issue, earlier in term of structure by Clarke [53] by proposing the notion of tree-like counterexamples to handle ACTL properties, and followed by the works investigating the quality of the counterexample mostly in term of length to be useful for debugging. Hence, in section 3.3, we will investigate the methods proposed for generating minimal, small and indicative counterexamples in conventional model checking. Model checking algorithms are classified in two main categories, explicit and symbolic. Where explicit algorithms are applied directly on the transition system, symbolic algorithms employ specific data structures. Generally, the explicit algorithms are adopted for LTL model checking, whereas symbolic algorithms are adopted for CTL model checking. In this section, The algorithms for generating small counterexamples are presented with respect to each type of algorithms.

However, generating small and indicative counterexamples only is not enough for understanding the error. Therefore, counterexamples analysis is inevitable for debugging. Many works in conventional model checking have addressed the analysis of counterexamples to better understand the error. Therefore, we will investigate the counterexample analysis in conventional model checking in section 3.4.

The most important thing about counterexample, is that it does not just serve as a debugging tool, but it is also used to refine the model checking process itself through the technique called Counterexample Guided Abstraction Refinement(CEGAR)[58]. CEGAR is an automatic method in the verification proposed to tackle the problem of state-explosion problem and it is based on the information obtained from the counterexamples generated. In section 3.5, we will show how counterexample contributes to this famous method of verification. In addition, we will show in section 3.6 how counterexample serves as a good tool for generating test cases.

Although counterexample generation is in th heart of model checking, not all the model checkers existed deliver counterexamples to the user. In section 3.7, we will review the famous tools that generate counterexamples. We conclude by citing brief open problems and future directions.

3.2 Cycle Detection Algorithms

Counterexample generation has its origins in graph theory trough the problem of existing cycle detection, since cycle detection is in the heart of model checking algorithms, either *explicit* or *symbolic* model checking . Varied algorithms were proposed for both LTL and CTL model checking in the literature. Explicit state model checking is based on Büchi automaton which is a type of ω-automata. The fairness condition relies on several sets of accepting states, where the acceptance condition is visiting the acceptance set infinitely often. So, a run is accepting if only if it contains a state in every accepting set infinitely often. As a result, the emptiness of the language is based on checking the non-existence of the faire cycle or equivalently the fair non-trivial *strongly connected component (SCC)* that intersects each accepting set. In case of non-emptiness, the accepting run is a sign of property failure, and as a result is returned as an error trace. We call this error trace a counterexample. So, the counterexample is typically presented by a finite stem followed by a finite cycle. Varied algorithms were proposed to find the counterexample in reasonable time, where finding the shortest counterexample has been proved that is a NP-Complete problem [52, 117].

For finding the fair SCC, depth first search (DFS) and Breath First Search (BFS) algorithms are used. The main algorithm employing DFS is the Tarjan's algrithm [170] that is based on manipulating the states of the graph explicitly. This algorithm was used to generate counterexamples by many other approaches [66, 173]. This algorithms runs in linear time, but as the number of states variables grows, it becomes simply infeasible. As a result, the symbolic-based algorithms are proposed as a solution.

In contrast to explicit algorithms, symbolic algorithms [41, 43] employ BFS and can describe large sets in compact manner through using characteristic functions. Several symbolic algorithms were proposed for computing the set of states that contains all the fair SCCs, without enumerating them [60, 116, 172]. We refer to these algorithms as SCC-hull algorithms. Currently, most of symbolic model checkers are employing Emerson's algorithm due to its high performance, and it was proven by [91] that both of the algorithms [83] and [52] can work in complementary way. Other works [118, 181] proposed algorithms based on enumerating the SCCs, we refer to these algorithms as symbolic SCC-enumeration algorithms.

Different approaches for generating counterexamples are proposed regarding the two types presented before. Clarke et al. [52] proposed a hull-based approach based on Emerson's algorithm by searching a cycle in a fair SCC close to the initial state. Since there is no guarantee to find terminal SCCs close to the initial state, finding *short counterexamples* was still a trade-off and an open problem, and thus it was investigated later by many researchers as we will show in the next section. The other approach by Hojati [116] was also employed by other works for generating counterexamples which employee isolations techniques of the SCCs [130]. Using Emerson's algorithm in combinatory way with SCC-Enumeration algorithm is possible, but is still not guaranteed to get a counterexample of short length. Ravi et al. [152] introduced a careful analysis of each type of these algorithms.

3.3 Finding Short Counterexamples

3.3.1 Explicit Algorithms

A counterexample in the *Büchi automaton* is a path $\sigma = \beta\gamma$ where β is a path without loop from the initial state to an accepting state, and γ is a loop around this accepting state. So that, a minimal counterexample is simply a counterexample with a minimal number of transitions. More formally, a counterexample $\sigma = \beta\gamma$ is minimal if $(|\beta| + |\gamma|) \leq (|\beta'| + |\gamma'|)$. With respect to this definition, a counterexample has at least one transition. Many algorithms consider the issue of generating counterexamples given Büchi automaton [119, 153, 175]. All these works employ Nested-Depth First Search, but they are not capable of finding a minimal counterexample. Although minimal counterexamples can be computed in polynomial time using minimal paths algorithms, the main drawback in fact is memory, where the resulting büchi automaton to be checked for emptiness is usually very huge, the thing that makes storing all the minimal paths to be compared so difficult.

Just recently new methods were proposed to compute *minimal counterexample* in Büchi automaton [96, 97, 107]. [97] proposed a DFS algorithm that runs in $\bigcirc(n\widehat{2})$ and they showed that $\bigcirc(nlogn)$ is sufficient, although DFS algorithms are memory consuming in general. This is due to the optimizations added using interleaving. Since the algorithms are based on exploring transitions backwards, adapting this method in practice is very difficult, especially by considering some restrictions. While this method requires more memory as SPIN does, [96, 107]proposed a method that does not use more memory than SPIN does. While the first one uses DFS and its time complexity is exponential [107], in [96] they proposed BFS algorithm with some optimisations able of computing the minimal counterexample in polynomial time. Hansen et al. [106] also proposed a method for computing minimal counterexamples based on Dijkstra algorithm for detecting strongly connected components. A novel approach was proposed by [127] for generating short counterexamples based on analysing the entire model and defining which events have more contribution to the error, these events are called crucial. In addition to generating short counterexamples, the technique helps with reducing the state space. The main drawback of this method is how to identify if such set of events are crucial and really lead to the error.

3.3.2 Symbolic Algorithms

The original algorithm for counterexample generation in symbolic model checking was proposed by [52]and was implemented in most symbolic model checkers. This algorithm of generating linear counterexamples for a fragment of ACTL was later extended to handle arbitrary ACTL properties using the notion of tree-like counterexamples [53]. Since then, many works have addressed the issue of computing short counterexamples in symbolic model checking [59, 146, 158].

[158] Proposed some criteria that should be met for the büchi automaton to accept shortest counterexamples. They proved that this criteria is satisfied in the approach proposed by [59] just for future time LTL specification, and thus they proposed an approach that met the criteria proposed for LTL specifications with past. The algorithm proposed employs breadth-first reachability check with BDD-based symbolic model checker.

The authors in [146] proposed a black-box based technique that masks some parts of the system in order to give understandable counterexample to the designer. So the work does not just tend to produce minimal counterexamples, but rather, it delivers small indicative counterexample of good quality to be analysed in order to get the source of the error. The most drawback of this method is that the generalization of counterexample generation from symbolic model checking to Black Box model checking could lead to non-uniform counterexamples that do not meet the behaviour of the system intended. While all of these works are applied to unbounded model checking [146, 158], the works [151, 166, 167] consider bounded model checking using SAT solvers, through lifting assignments produced by a SAT solver. This actually makes the quality of the counterexample generated is dependent on the SAT solver itself. Other works have investigated the use of heuristics algorithms for generating the counterexamples [80, 169]. Although heuristics were not widely used, they gave pretty good results, and were also used latter for generating probabilistic counterexamples, as we will see later.

3.4 Counterexamples for Debugging

One of the major advantages of model checking over other formal methods its ability to generate a counterexample when the model falsifies such specification. The counterexample represents an error trace; by analysing it the user can locate the source of the error, and as Clarke wrote : The counterexamples are invaluable in complex systems and some people use model checking just for this feature [56].

However, generating small and indicative counterexamples only, as has been presented in the previous section is not enough for understanding the error. Therefore, counterexamples explanation is inevitable. Error explanation is the task of discovering why the system exhibits this error trace. Many works have addressed the automatic analysis of counterexamples to better understand the failure. Error explanation ranges in two main categories . The first is based on the error trace itself, through considering the small number of changes that have to be made in order to ensure that the given counterexample is no longer exhibited, and thus of course these changes represent the sources of that error. The second is based on comparing successful executions with the erroneous one in order to find the differences, and thus those differences are considered as candidate causes for the error. Kumar et al. [133] have introduced a careful analysis of the complexity of each type. For the first type, they showed using three models(Mealy machines, extended finite state machines, and push-down automata) that this problem is NP-complete. For the second type, they provided a polynomial algorithm using Mealy machine and push-down automata, but solving the problem was difficult with extended finite state machines.

Error explanation methods are successfully integrated in model checkers such as SLAM [33] and Java Path finder [40]. SLAM takes time execution less than JP does and can achieve completeness in finding the causes, but according to Groce [102], this also could be harmful. The error explanation process has many drawbacks, the main one is that the counterexample consists usually of huge number of states and transitions and involves many variables. The second is that model checker usually floods the designer with multiple counterexamples, without any some sort of classification, which makes choosing the helpful counterexample difficult. Besides, a single counterexample it might not be enough to understand the behaviour of the system. Analysing set of counterexamples together is an option but the problem is that it requires much effort, and even though, the set of counterexamples to be analysed could contain the same diagnostic information, which makes analysing this set of counterexamples a waste of time. The last and the main important problem in error explanation is that not all the events that occur on the error trace are of importance for the designer, and so locating these events which are critical is the goal behind error explanation. In this section, we survey some works with respect to the two categories.

3.4.1 Computing the Minimal Number of Changes

Jin et al. [125] proposed an algorithm for analysing the counterexamples based on the local information, by segmenting the events of the counterexamples in two main segments, fated and free. The fated segments refer to the events that obviously have to occur in the executions, and the free segments refer to the events that may be avoiding them could not exhibit the error, and thus they are candidate to be causes. The approach is mainly based on computing the fated segments by considering the containing of the variables named controlling, which are considered to be critical and have more control on the environment. Then, the free segments will be simply those that are not fated.

Wang et al. [176] also proposed a method that works just on the fail run without considering successful runs. The idea is about looking at the predicates candidate for causing the failure in the error trace. To do so, they use weakest pre-condition computation, the technique that is widely used in predicate abstraction. This computation aims to find the minimal number of conditions that should be met in order to not let the program violate the assertion. This results in a set of predicates that contradict with each other, by comparing how these predicates contradict to each other, we can find the cause for the assertion failed, and map it back to the real code.

Using the notion of causality introduced by Halpern and Pearl, Beer et al. [36] introduced an approach for explaining LTL counterexamples and was implemented as a feature in the IBM formal verification platform RuleBase PE. Given the error trace, the causes for the violation are highlighted visually as red dots on the error trace itself. The Question asked was : what values of signals on the trace cause it to falsify the specification? Following the definition of Halpern and Pearl, they refer to such set of pairs of state-variable as bottom-valued pairs whose values should be switched to make such state-variable pair critical, the pair is critical if changing the value of the variable in this state no longer produces a counterexample. This pair represents the cause for the first failure of the LTL formula given the error trace, where they argue that the first failure is the most relevant to the user. Nevertheless, the algorithm proposed computes an over-approximation of set of causes not just the first cause that occurred. Many Other works also provided error explanation methods in the context of C programs [171, 185, 186].

3.4.2 Comparing Counterexamples with Successful Runs

This is the most adopted method for error explanation and was successfully featured in many model checkers such as SLAM and PathFinder. Groce et al. [101] have proposed an approach for counterexamples explanation based on computing a set of faulty runs called negatives, in which the counterexample is included, and comparing it to a set of correct runs called positives. Analysing the common features and differences could lead to get a useful diagnostic information. Their algorithms were implemented in the JAVA pathfinder model checker. Based on Lewis counterfactual theory of causality [140] and distance metrics, the authors in [100] have proposed a semi-automated approach for isolating errors in ANSI C programs, by considering the alternative worlds as programs executions and the events as propositions about those executions. The approach relies on finding causal dependencies between predicates of a program. A predicate *a* is causally dependent on *b* given the faulty execution iff executions in which the removal of a cause *a* also removes the effect *b* are more likely than execution where *a* and *b* do not appear together. For finding these traces, which are as close as possible to the faulty one, they employed distance metric. In [45] they extended the original approach for comparing a counterexample with closest successful run through combining distance metric with predicate abstraction in order to generate explanations for abstract counterexamples. They argue that even for abstract counterexample, abstract state-space makes the explanation more informative. Renieris and Reiss [154] also introduced a method based on distance metric to select the closest correct runs to the faulty one and they provided a quantitative method for evaluating their methods.

Ball et al. [34] proposed an effective approach that is currently featured in SLAM model checker. Their method is based on the same principle of finding successful runs to be compared with the counterexample. The interesting difference here is that it generates error trace per error cause, which makes the diagnostic more easier since there will not be causal dependencies in the traces generated. It is clear that this method will require the invocation of the model checker each time a cause for the error is found. Finally, the causes are reported as erroneous transitions that do not occur in any correct trace. Copty et al. [63] proposed a framework for debugging counterexamples as they refer to it as counterexample Wizard in the context of symbolic LTL. The technique employed three main capabilities: multi-value counterexample annotation, constraint-based debugging, and multiple counterexample generation. But in contrast to the work by Ball et all, the model checker is not invoked each time an error cause is found, but instead it gets all the data needed together to start the analysis.

While all of these works addressed safety properties, Kumazawa and Tamai [134] attended to explain errors for liveness properties that involve more computational complexity. For that reason, the counterexample is represented as an infinite trace and not a finite one, and the witnesses to be compared with this counterexample are infinite as well. The method also employee shortest paths algorithms. Many similar works for counterexamples analysis have been done [62, 70, 89, 103, 149, 163–165].

3.5 Counterexample Guided Abstraction Refinement (CEGAR)

The main challenge in model checking is the state explosion problem. Dealing with this issue is in the heart of model checking, and was addressed at the beginning of the model checking and still. Many methods were proposed to tackle this issue, the most famous are: symbolic algorithms, partial order reduction, Bounded Model Checking (BMC)and abstraction. Among these techniques, abstraction is considered as the most general and flexible for handling the state explosion problem [54]. Abstraction is about hiding or simplifying some details about the system to be verified, even removing some parts from it that are considered irrelevant for the property under consideration. The central idea is that verifying simplified or abstract model is more efficient than the entire model. Evidently, this abstraction has a price which is losing some information, and the best abstraction methods are the ones which control this loss of information. Over-approximation and under-approximation are two main key concepts for that problem. There were many abstraction methods in the literature [65, 99, 142] , where the last one is widely famous and was adopted in the symbolic model checker NuSMV.

Given the possible loss of information caused by the abstraction, inventing some refinements methods of the abstract model is necessary. The most known method for abstraction is a *Counterexample-Guided Abstraction Refinement (CEGAR)* proposed by Clarke et al. [54] as an extension of the original one [55]. In this approach, the counterexample plays the crucial role for finding the right abstract model. The process of CEGAR consists of three main steps: The first is to generate an abstract model using one of the abstractions techniques [46, 54, 61] given a formula φ. The second step is about checking the satisfaction of φ, if it is satisfied then the model checker stops and returns that the concrete or the original model satisfies the formula, if it is not satisfied, a counterexample will be generated. The counterexample generated is in the abstract model, so we have to check if it is also a counterexample in the concert model, and thus the concert model does not satisfy the property φ. Otherwise, the counterexample is called a spurious and the abstraction must be carried out based on this counterexample. The final step is to refine the model until no spurious counterexample is found. This is how the technique gets its name, refining the abstract model using the spurious counterexample.

In the literature we find many extensions for CEGAR depending on the type of predicates and application domains: Large program executions [132], Non-Disjunctive abstractions [143], Propositional Circumscription [123]. The CEGAR technique itself has been

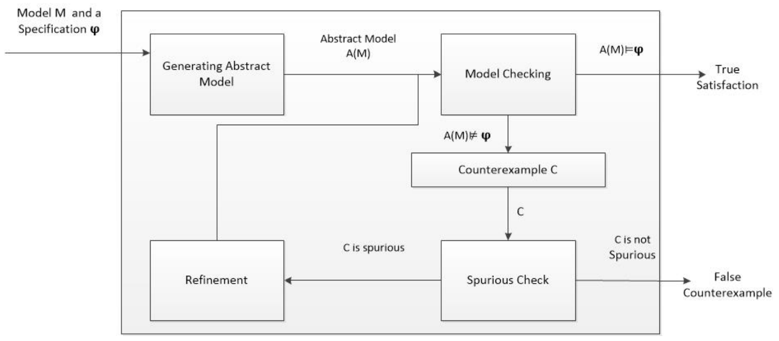


Fig. 3.1 Counterexample Guided Abstraction Refinement Process

used to find bugs in complex and large systems [38]. The idea is based on gathering and saving information during the abstract model checking process in order to generate short counterexamples in the case of the failure. This is could be helpful for large models that make generating counterexamples using standard BMS intractable. CEGAR currently is implemented in many tools such as NuSMV[15], SLAM and BLAST[2].

3.6 Counterexamples for Test Cases Generation

Counterexample generation gives the opportunity for model checking to be adopted and used in different domains, one of the most domains in which the model checking has been adapted is *test cases* generation . Roughly speaking, testing is an automating method used to verify the quality of software. When we use model checking to generate test cases, this is called model-based testing. The central idea about using model checking for testing [44, 86] is about interpreting counterexamples as test cases, and then test data and some expected results are extracted from these tests using such execution framework. Since the goal of counterexamples is to help the designer to find the source of the error given some specification, thus they are very useful as test cases [93].

A test describes the behaviour of the test case intended: the final state, the states that should be traversed to reach the final state. The test purpose is specified in temporal logic and then converted to what is called a never-claim by negation; to assert that the test purpose never becomes true. So, the counterexample generated after the verification process will describe how the never-claim is violated, which a description of how test purpose is fulfilled. Many approaches for creating never-claims based on coverage criteria (called *"trap properties"*)[94] are proposed. *Coverage* criteria aims to find how such a system is exercised given a specification in order to get the states that were not traversed during the test; in this

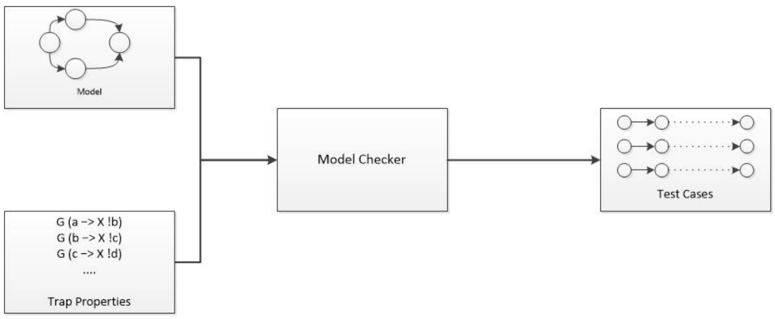


Fig. 3.2 Coverage based test case generation [93]

context we call this a specification a test suit. With regard to "trap properties", we find the work of Gargantini and Heitmeyer that addressed the coverage of SCR Specifications [94], the work of Heimdahl et al. that addressed the coverage of transition systems globally [111], the coverage of Control and Data Flow by Hong and Lee [120] and the Coverage of Abstract State Machines [95]. These approaches and other different than coverage-based approaches including requirements-based testing [85] and Mutation-based [21] are well studied in [42]. [92] proposed several effective techniques to improve the quality of the test cases generated in model checking-based testing, especially requirements based testing, and apply them on different types of properties in many industrial case studies.

3.7 Tools

Practically, all successful model checkers are able to output counterexamples in varying formats [51]. In this section we will try to survey the tools supporting counterexample generation and study their effectiveness.

Berkeley Lazy Abstraction Software Verification Tool (BLAST) [2] is a software model checking tool for C programs. BLAST has the ability to generate counterexamples and furthermore it employed the CEGAR for the verification. BLAST it is not just a CEGAR-based model checker but also can be used for generating test cases. BLAST shows promising results with safety properties of programs with medium size.

CBMC [3] is well known Bounded Model Checker for Ainci-C and C++ programs. CBMC performs symbolic simulation on the programs and employees a SAT solver in the verification procedure, when the specification is falsified, a counterexample in the form of states with variables valuation leading to these states is rendered to the user.

JavaPathfinder [11] is a famous software model checking tool for Java programs. JavaPathfinder is an effective virtual machin-based tool that verifies the program along all the possible executions. Due to its ability of dealing with most of JAVA language features because it runs on byte-code level, JavaPathfinder can generate a detailed report on the error in case of the property is violated. Besides, the tool gives the ability to generate test cases.

SPIN [18] is a model checker mostly known for the verification of systems that exhibit high interaction between processes. The systems are described using the description language, Process Meta Language (PROMELA), and verified against properties specified in LTL. By applying a depth-first search algorithm on the intersection product of the model and the büchi automaton representing the LTL formula, a counterexample is generated in case an accepted cycle is detected. SPIN offers an interactive simulator that helps to understand the cause of the failure by showing the processes and their interactions in order.

NuSMV [15] is a symbolic model checker that appeared as an extension of the Binary Decision Diagrams(BDD)-based model checker SMV. NuSMV includes both LTL and CTL for specification analysis, and combines SAT and BDD techniques for the verification. NuSMV can deliver a counterexample in well readable way using XML format by indicating the states of the trace and the variables with their new values that cause the transitions.

UPPAAL [19] is a verification framework for real-time systems. The systems can be modelled as networks of timed automata extended with data types and synchronisation channels and the properties are specified using a Timed CTL(TCTL). UPPAAL can find and generate counterexamples in highly visualisation graphic mode as message sequence charts that indicate the events with respect to their order.

PRISM [17] is a probabilistic model checker used for the analysis of systems that exhibit stochastic behaviour. The systems are described as Discrete-Time Markov Chains (DTMCs), Continuous-Time Markov Chains (CTMCs) or Markov Decision Processes(MDPs) using guarded command language, and verified against probabilistic properties expressed in Probabilistic Computation Tree Logic (PCTL) and Continuous Stochastic Logic (CSL) and can be extended with rewards. Another successful probabilistic model checker extended with rewards is the Markov Reward Model Checker (MRMC) [13]. Although, both model checkers have shown high effectiveness and have been applied for numerous case studies, they lake the generation of counterexamples. Nevertheless, they have been used by recent tools (DiPro [7] and COMICS [4]) for generating the counterexamples.

3.8 Conclusion

In this chapter, we surveyed the counterexamples in model checking from many aspects, generation and debugging, and we have seen the usefulness of counterexamples for other purposes like CEGAR and test cases generation. Although counterexamples have not treated as a particular subject in the beginning of model checking, but was treated as a related problem to fair cycle detection algorithms; in the recent years, counterexamples have been treated as a standalone and a fundamental problem. Although the form of tree has been proposed for the counterexamples of the fragment ACTL by Clarke, we see that this approach has not been adopted in model checkers during the last years, but instead model checkers are still based on generating linear counterexamples.

We can also see that the techniques based on counterexamples can directly benefit from any advancement and new proposition for generating small and indicative counterexamples in considerable time. It is not possible to cover all the issues related to counterexamples in model checking . However, we hope that we surveyed most important issues for counterexamples that could stand as a good starting point for new research works in this field.

Chapter 4

Causal Analysis of Probabilistic Counterexamples

4.1 Introduction and Related Works

Unlike the previous methods proposed for conventional model checking that generate the counterexample as a single path ending with a bad state representing the failure, the task in probabilistic model checking is quite different. The counterexample in probabilistic model checking is a set of evidences or diagnostic paths that satisfy path formula and their probability mass violates the probability threshold. As it is in conventional model checking, in probabilistic model checking the generated counterexample should be small and most indicative to be easy for analysing. In probabilistic model checking this task is more challenging, since the counterexample consists of multiple paths.

Various approaches for probabilistic counterexample generation have been proposed. The authors in [24, 27] introduced an approach for counterexample generation for DTMC and CTMC against timed reachability proprieties using heuristics guided and directed explicit state space search. In complementary work [25], with the intuition that single scheduler makes an MDP as DTMC, they proposed an approach for counterexample generation for MDPs based on existing methods for DTMC. They introduced more complete work in [26] for generating counterexamples for DTMC and CTMC as what they refer to as diagnostic sub-graphs. Based on all the previous works, they built a tool DiPro [28] for generating indicative counterexamples for DTMC, CTMC and MDPs. These heuristic-based algorithms showed a great efficiency in term of counterexample quality, nevertheless, the tool DiPro implementing these algorithms takes usually a long time for producing the counterexample. This tool can be jointly used with the model checkers PRISM [115] and MRMC

[128] , and can render the counterexamples in text formats as well as in graphical mode.

Similar to the previous works, [105] has proposed the notion of smallest most indicative counterexample that reduces to the problem of finding K shortest paths. In a weighted digraph transformed from the DTMC model, and given initial state and the target states, the strongest evidences that form the counterexample are selected using extensions of K-shortest paths algorithms for an arbitrary number k. Instead of generating *path-based counterexamples* , the authors in [179] have proposed a novel approach for DTMCs and MDPs based on *critical subsystems* using SMT solvers and mixed integer linear programming. Critical subsystem is simply a part of the model (states and transitions) that are considered relevant because of its contribution to exceeding the probability bound. The problem has been shown that is NP-Complete. Another work always based on the notion of critical subsystem is proposed to deliver abstract counterexamples with less number of states and transitions using hierarchical refinement method. Based on all of these works, the authors in [124] proposed the COMICS tool for generating the critical subsystems that induce the counterexamples.

There are also many other works that addressed special cases for generating counterexamples in probabilistic model checking. In [30], the authors proposed an approach for finding sets of evidences for bounded probabilistic LTL properties on Markov Decision Processes (MDP) that behave differently from each other giving significant diagnostic information. While their method is also based on K-shortest path, the main contribution is about selecting the evidences or the witnesses with respect to main five criteria cited in the paper in addition to the high probability. While all of the previous works for counterexample generation are explicit-based, the authors in [178] proposed a symbolic method using bounded model checking. In contrast to the previous methods, this method lakes the selection of the *strongest evidences* first, since the selection is performed in arbitrary order. Another approach for counterexample generation that uses *Bounded Model Checking (BMC)* has been proposed [39]. Unlike the previous work that uses conventional SAT solvers, The authors used a SMT-solving approach to put some constraints on the paths selected, in order to get more abstract counterexample that consists of strongest paths. Counterexample generation for probabilistic LTL model checking has been addressed in [157] and probabilistic CEGAR has been also addressed [113]. Comprehensive representation of the counterexamples using *regular expressions* has been addressed in [69]. Sine regular expressions deliver compact representations, they can help to deliver short counterexamples, besides, they are widely known and easy for understanding, sot that they will give more benefits as a tool for error explanation. for further details on counterexample generation in probabilistic model checking we refer to [20].

Although most of the works presented here aim to generate counterexamples in probabilistic model checking that help with debugging, we find the most interesting work and practically helpful for debugging that of [180]. Instead of relying on the state space search resulted from the parallel composition of the modules, this work suggests to rely directly on the guarded command language used by the model checker, which is more likely and helpful for debugging purpose. The authors always employee the critical subsystem technique but to identify the smallest set of guarded commands contributing to the error.

While the research on debugging of probabilistic counterexamples is in its first stage, in conventional model checking, many works have proposed techniques and algorithms for discovering error causes from counterexamples, hence presenting them to the user in a comprehensive way. Most of these works consider the existence of successful runs or executions to be compared with the error trace, with the assumption that the more successful execution closed to the error trace indicates the causes of the error [34, 100, 101, 186]. Based on Lewis counterfactual theory of causality [140] and distance metrics, the authors in [100] have proposed semi-automated approach for isolating errors in ANSI C programs by considering the alternative worlds as programs executions and the events as propositions about those executions. Unlike the previous works that require multiple executions, the work [176] introduced a technique performed on a single concrete execution path using a weakest precondition algorithm. While all of these works addressed safety properties, some of works attended to explain errors for liveness properties that involve more computational complexity [134].

There are several works have used the definition of causality in the context of model checking. We found the most closely related to our work that of [36, 48]. They used the definition of causality for explaining LTL counterexamples[36], where the method proposed was implemented as a feature in the IBM formal verification platform RuleBase PE . Unlike addressing the question in [36] "what causes a system to falsify a specification?" In the context of coverage [48], the question addressed was "what causes a system to satisfy specification?" In this aim, they adapted the definition of causality and its quantitative measure, responsibility.

The definition of causality has also been used by [135]. They adopted the definition of causality to event orders for generating fault trees from probabilistic counterexamples, where the selection of traces forming the *fault tree* are restricted to some *minimality condition* . To do so, they proposed the event order logic to reason about boolean conditions on the occurrence of events, where the cause of the hazard in their context is presented as a Event Order Logic (EOL) formula, which is a conjunction of events, and the event are simply actions leading from state to another. In [90], they extended their approach by integrating causality in explicit-state model checking algorithm to give a causal interpretation for sub- and super-sets of execution traces, the thing that could help the designer to get a picture about what is going on. They proved the applicability of their approach to many industrial size *PROMELA* models. In [139], they aimed to extended the causality checking approach to probabilistic counterexamples by computing the probabilities of events combination, but they still consider the use of causality checking of qualitative PROMELA models.

From all what precedes, we find the most closely related to our work from causality adoption perspective that of [36, 48]. They adopted the definition of causality in the same way for the same goal, which is discovering error causes from counterexamples already generated, in addition both of our works do not consider the existence of other counterexamples or successful runs to compare with. The main difference here is that the causality is adopted in conventional model checking for linear counterexamples where the causes generated are qualitative, whereas our work adopted the definition in probabilistic model checking where the error has a quantitative aspect. From probabilistic model checking perspective, we find the most closely related work that of [135]. While they adopted the same definition of causality for the same aim, which is debugging probabilistic models, the main difference is that they seek a counterexample of causal representation that could be given at the end in the form of fault tree, rather than searching for the causes in a counterexample already generated. While both of our works are classified in the category of path-based counterexamples according to [20], the work of [180] employees critical sub-systems which are fractions of probabilistic models, and rely directly on the *guarded command language* used by the model checkers like PRISM in order to deliver a well-understandable counterexample.

In this chapter we propose our approach for the analysis of probabilistic counterexamples. To this end, we adopt the definition of causality introduced by [104] as well as the definition of responsibility and blame [47]. We will focus on probabilistic safety properties with upper threshold of the form $P_{\leq p}(\phi_1 \mathbf{U} \phi_2)$. The properties with lower threshold can be easily transformed to properties with upper threshold [26, 105]. We should mention that explaining the violation of PCTL/CSL upper threshold properties reduces to the explanation of exceeding the probability threshold over the model. Our approach does not ignore the previous approaches of generating probabilistic counterexamples, but instead it is based on them. Our approach for error explanation is based directly on the most indicative counterexamples [26, 28, 105]. In this chapter, two algorithms for analysing probabilistic counterexamples are presented and applied on many case studies, the first is performed on counterexamples for DTCMs and CTMCs, the second is performed on counterexamples for MDPs.

4.2 Probabilistic Counterexamples

The probabilistic counterexample is generated when a PCTL/CSL property is not satisfied. The probabilistic property $\phi = \mathbf{P}_{\leq p}(\varphi)$ is refuted when the probability mass of the paths satisfying φ exceeds the bound p. Therefore, a probabilistic counterexample for the property ϕ is formed by a set of paths starting at state s and satisfying the path formula φ. We denote these paths by $Paths(s_0 \models \phi)$. The counterexample can be formed of set of finite paths where each path $\sigma = s_0 s_1 ... s_n$ is a prefix of an infinite path from $Paths(s_0 \models \phi)$ satisfying the formula φ. We denote these paths by $FinitePaths(s_0 \models \phi)$.

It is clear that we can get a set of probabilistic counterexamples, noted $PCX(s_0 \models \phi)$, which is a set of any combination from $FinitePaths(s_0 \models \phi)$ that their probability mass exceeds the bound p. Among all these probabilistic counterexamples, we are interested by the most indicative one. The most indicative counterexample is minimal counterexample (has the least number of paths from $FinitePaths(s_0 \models \phi)$) and its probability mass is the highest among all other minimal counterexamples. We denote the most indicative probabilistic counterexample by $MIPCX(s_0 \models \phi)$. We should note that the most indicative probabilistic counterexample may not be unique.

Lemma 4.2.1. Let $MIPCX(s_0 \models \phi)$ be a most indicative probabilistic counterexample. Every finite path $\sigma \in MIPCX(s_0 \models \phi)$ is critical. Which means $\forall \sigma : MIPCX(s_0 \models \phi) - \sigma$ (removing any path σ from $MIPCX(s_0 \models \phi)$ will render the result not a counterexample.

For the counterexample to have high probability, it should consist of paths that carry high probabilities from $FinitePaths(s_0 \models \phi)$. The path σ having the highest probability over all these paths is called *strongest path* and is defined as follows: for every path $\sigma' \in FinitePaths(s_0 \models \phi) : P(\sigma) \geq P(\sigma')$. The strongest path also may not be unique.

Lemma 4.2.2. A most indicative probabilistic counterexample $MIPCX(s_0 \models \phi)$ contains at least one strongest path $\sigma \in FinitePaths(s_0 \models \phi)$.

Lemma 4.2.3. For every path $\sigma \in MIPCX(s_0 \models \phi)$ on which the property $\phi_1 \mathbf{U} \phi_2 (\phi_1 \mathbf{U}^{\leq n} \phi_2)$ is satisfied, the right state sub-formula (ϕ_2) is satisfied in the last state of σ.

Example 1 Let us consider the example of DTMC shown in 4.1 and the property $P_{\leq 0.5}(\varphi)$, where $\varphi = (a \vee b) U (c \wedge d)$. The property above is violated in this model ($s_0 \not\models \mathbf{P}_{\leq 0.5}(\varphi)$), since there exists a set of paths satisfying φ whose probability mass is higher than the probability bound (0.5). Any combination from $FinitePaths(s_0 \models \phi)$ having probability mass higher than 0.5, is a valid counterexample including the whole set. For instance, we can find three counterexamples:

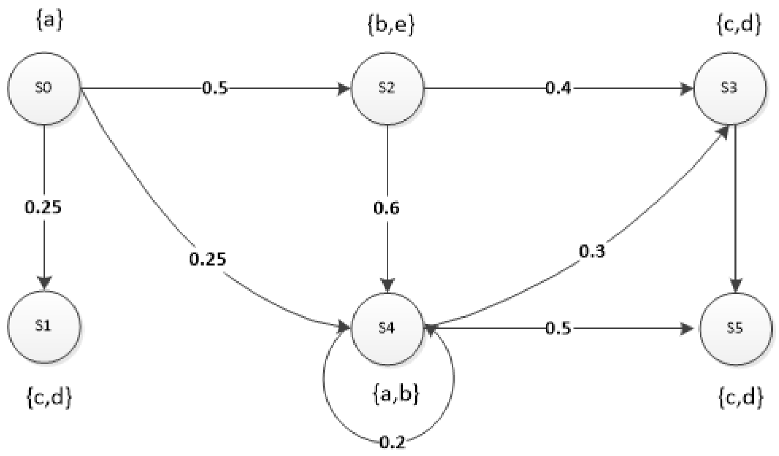


Fig. 4.1 A DTMC

$$P(CX_1) = P(\{s_0s_1, s_0s_2s_3, s_0s_2s_4s_3, s_0s_2s_4s_5, s_0s_4s_5\})$$

$$= 0.25 + 0.2 + 0.09 + 0.15 + 0.12 = 0.81$$

$$P(CX_2) = P(\{s_0s_1, s_0s_2s_4s_5, s_0s_4s_5\})$$

$$= 0.25 + 0.15 + 0.12 = 0.52$$

$$P(CX_3) = P(\{s_0s_1, s_0s_2s_3, s_0s_2s_4s_5\})$$

$$= 0.25 + 0.2 + 0.15 = 0.6$$

The last probabilistic counterexample is the most indicative since it is minimal and its probability is higher than the other minimal counterexample CX_2,$P(CX_3) = 0.6 > P(CX_2)$. The strongest path is s_0s_1, which is included in the most indicative probabilistic counterexample.

4.3 Causality and Responsibility for Probabilistic Counterexamples

For PCTL/CSL properties of the form $\phi = \mathbf{P}_{\leq p}(\varphi)$ explaining the violation reduces to the explanation of exceeding the probability bound over the DTMC/CTMC model. Therefore, the question of " what labelling and/or probability values in the counterexample cause the system to falsify a specification " reduces to the question: " what labelling and/or probability values in the counterexample cause the exceeding of probability bound over the model ".

With respect to the definition of causality by Halpern and Pearl, the causality model is defined by a set of exogenous variables U, set of endogenous variables V and set of functions F. We can adopt this definition to the most indicative counterexample as follows.

Definition 4.3.1. (**Causality Model**) A causality model for the most indicative probabilistic counterexample $MIPCX(s_0 \models \phi)$ is a tuple $M = (U,V,F)$, where the set U is represented by a context variable; its value u represents a state $s \in MIPCX(s_0 \models \phi)$. V is a set of atomic propositions and Boolean formulas. F associates with every variable $V_i \in V$ a truth function f_{V_i} that determines the value of V_i (0 or 1), given a state s and the values of other variables in V.

For example, $f_{p\wedge r}(s, p = 1, r = 1) = 1$ where p and r are atomic propositions, and s is state representing a context. The causal influence is modelled by the transitions in $MIPCX(s_0 \models \phi)$.

Let us denote by $MIPCX_{\widehat{(s,X\leftarrow x')}}(s_0 \models \phi)$ the set of finite paths resulted from $MIPCX(s_0 \models \phi)$ by switching the value of a variable $X \in V$ in a state s.

Definition 4.3.2. (Criticality) Consider a counterexample $MIPCX(s_0 \models \phi)$ for a probabilistic formula $\phi = \mathbf{P}_{\leq p}(\varphi)$, a state $s \in MIPCX(s_0 \models \phi)$ and a variable $X \in V$ has a value $x \in \{0,1\}$ in s. We say that a pair $(s,X = x)$ is critical for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$, if $MIPCX_{\widehat{(s,X\leftarrow x')}}(s_0 \models \phi)$ is not a valid counterexample for $\phi = \mathbf{P}_{\leq p}(\varphi)$.

That is, for the probabilistic counterexample $MIPCX(s_0 \models \phi)$, we say that the value of a variable X is critical for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$ in a state s, if changing the value of X in this state renders the result not a counterexample. Following the semantics of PCTL and the definition of counterexamples in the previous section, it is evident that each critical pair $(s,X = x)$ for the violation of probabilistic formula $\phi = \mathbf{P}_{\leq p}(\varphi)$ is in the first place a critical pair for the satisfaction of path formula φ in such set of paths from $MIPCX(s_0 \models \phi)$. Since $MIPCX(s_0 \models \phi)$ is minimal and following lemma 4.2.1, we can give the following lemma.

Lemma 4.3.1. For every path $\sigma \in MIPCX(s_0 \models \phi)$, each critical pair $(s,X = x)$ for the satisfaction of φ on σ is critical for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$.

Definition 4.3.3. (Actual Cause) Consider a counterexample $MIPCX(s_0 \models \phi)$ for a probabilistic formula $\phi = \mathbf{P}_{\leq p}(\varphi)$ and a variable $X \in V$. We say that $(s,X = x)$ is a cause for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$, if $(s,X = x)$ is critical , or there exists a subset of variables W of V such that switching their values in s makes $(s,X = x)$ critical.

Thus, in our setting, we can refer to a cause as a pair $(s,X = x)$ where X is an atomic proposition has the value 1 in s if $X \in L(s)$, and it has the value 0 otherwise. $(s,X = x)$ is said to be cause, if it is critical, or it can be made critical by switching the values of set of variables W in s.

Definition 4.3.4. (Responsibility) Consider a counterexample $MIPCX(s_0 \models \phi)$ for a probabilistic formula $\phi = \mathbf{P}_{\leq p}(\varphi)$ and a variable $X \in V$. The degree of responsibility of a cause $(s, X = x)$ for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$ denoted $dR(s, X = x, \phi)$is 1 if $(s, X = x)$ is critical, and otherwise is $1/(|W| + 1)$.

That is, we can think of responsibility as a quantitative measure that gives us diagnostic information on $(s, X = x)$ being a cause for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$, where the cause having the highest responsibility for the violation is the critical one. We should mention that there is a duality between PCTL property violation and fault-tolerance. The smaller the degree of responsibility of a pair $(s, X = x)$, the less responsible and relevant cause to the user. In fault-tolerance, the smaller the degree of responsibility of a component, the more tolerant the system is to its failure.

Although the responsibility gives a good measure for the violation, which is better than getting just qualitative causes, this measure is not yet suitable as diagnostic information in probabilistic model checking, because we are dealing with uncertainty where the error is quantitative (probability measure exceeds threshold P). Therefore, we have to define for each context s its probability, so that we arrive to the definition of probabilistic causality model.

Definition 4.3.5. (Probabilistic Causality Model) A probabilistic causality model for $MIPCX(s_0 \models \phi)$ is a tuple (M, Pr), where M is the causality model and Pr is a probability function defined over the states of $MIPCX(s_0 \models \phi)$. We define for each state $s \in MIPCX(s_0 \models \phi)$ its probability as follows:

$$Pr(s) = \sum_{s \in \sigma | \sigma \in MIPCX(s_0 \models \phi)} P(\sigma) \tag{4.1}$$

Thus, this probability is obtained by summing the probabilities of paths in which s is included. Since every variable in V is a function of U, we can define the probability of each cause $(s, X = x)$ in the same way :

$$Pr(s, X = x) = Pr(s) \tag{4.2}$$

That is, we have associated for each cause a probability that represents exactly the probability of a state s in which $X = x$. This probability simply measures the contribution of this cause to the violation of probabilistic formula. Given the definitions of cause responsibility and cause probability, we arrive to the following definition.

Definition 4.3.6. (Most Responsible Cause) Cause C is a most responsible cause for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$, if $dR(C)Pr(C) \geq dR(C')Pr(C')$ for any cause C'.

With respect to this definition, the most responsible cause will be often a critical cause with high probability. This definition responds to both issues for probabilistic counterexamples diagnosis, the conventional one that concerns the criticality of a state with respect to a variable, and the second concerns the quantitative contribution to the error. We should mention that the most responsible cause may not be unique. We propose an algorithm for finding the causes and computing their responsibilities and probabilities in the following section.

Lemma 4.3.2. A most responsible cause is not necessarily included in a strongest path.

Example 2 Consider the most indicative counterexample $CX_3 = \{s_0s_1, s_0s_2s_3, s_0s_2s_4s_5\}$ generated from the DTMC presented in Figure 4.1 against the property $P_{\leq 0.5}[(a \vee b)U(c \wedge d)]$.

it is possible to define a causality model for CX_3, where $u \in \{s_0, s_1, s_2, s_3, s_4, s_5\}$, and F can be defined over the variables in V as follows

$$\begin{gathered} f_b(s_2) = 1 \\ f_{c\wedge d}(s_2, c = 0, d = 0) = 0 \\ \vdots \end{gathered}$$

For instance, it is clear that in s_2, the actual cause for the violation of $P_{\leq 0.5}[(a \vee b)U(c \wedge d)]$ is $b = 1$. The responsibility of this cause is $dR(s_2, b = 1) = 1$ because it is critical, switching the value of $b = 1$ to $b = 0$ in s_2 results in falsifying φ on two paths, which makes $CX3$ no longer a valid counterexample. Whereas in s_4 $dR(s_4, b = 1) = 1/2$, because we have to switch the value of a in order to make $(s_4, b = 1)$ critical. Starting from s_0, $(s_2, b = 1)$ is the most responsible cause with a probability $Pr(s_2, b = 1) = 0.2 + 0.15 = 0.35$, nevertheless it is not included in the strongest path $\sigma = s_0s_1$.

Now, Consider the counterexample $CX1$ from the previous example . Although $(s_1, c = 1)$ is critical for the satisfaction of $\varphi = (a \vee b)U(c \wedge d)$ on the path s_0s_1, it is not critical for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$, since the set resulted from $CX1$ by removing s_0s_1 is a valid counterexample. In contrast, $(s_1, c = 1)$ is critical in $CX2$ and $CX3$ since they are minimal counterexamples.

4.4 Algorithm for Computing Causes Responsibilities

This algorithm performs on counterexample generated from the tool DiPro [7], since probabilistic model checkers do not offer the possibility to generate counterexamples. DiPro is a tool used for generating counterexamples from DTMC, CTMC and MDPs models, and can be jointly used with the model checkers PRISM [17] and MRMC [13], and can render the counterexamples in text formats as well as in graphical form.

The algorithm gets from DiPro tool the counterexample $MIPCX(s_0 \models \phi)$ and the probabilistic formula $\phi = \mathbf{P}_{\leq p}(\varphi)$ as input, and outputs the causes with their probabilities and responsibilities. The formula φ is until formula written in NNF (Negative Normal Form), which means that negation appears just at the front of atomic propositions.

The algorithm explores the counterexample and computes the causes with their responsibilities and probabilities with respect to each state s. The condition put on last state follows Lemma 4.2.3. The main function of this algorithm is FindCauses, which is based on the formula structure. It takes a state and state formula as input as well as the variable W, and returns recursively the set of causes and their responsibilities. We note that when the state formula ψ is of the form $\psi_1 \wedge \psi_2$, both sub-formulas are essentially true at state s. But When ψ is of the form $\psi_1 \vee \psi_2$, one of them could be true at s or both of them. This actually follows the causal intuition that in conjunctive scenario, both $\psi_1$and ψ_2 are required for ψ being satisfied, whereas in the disjunctive scenario, either ψ_1 or ψ_2 suffices to make ψ satisfied. In the two cases, we apply FindCauses to each sub-formula.

Computing the degree of responsibility follows the same intuition. We use the variable W to measure the number of changes that increases when the both sub-formulas of OR formula are satisfied, otherwise W remains unchanged. Finally, at the propositional level, the cause will be a pair $\langle s,a \rangle$ if $a \in L(s)$ or $\langle s,\neg a \rangle$ otherwise, with dR computed based on the value of W. From Example 1, For instance for disjunctive scenario, in s_4 the causes generated with their responsibilities are: $\{\langle s_4, a=1\rangle, 1/2\}$ and $\{\langle s_4, b=1\rangle, 1/2\}$. For conjunctive scenario, in s_1, the causes generated with their responsibilities are:$\{\langle s_1, c = 1\rangle, \langle s_1, d=1\rangle, 1\}$.

The Algorithm computes an *over-approximation* set of causes , since computing the set of causes exactly in binary causal models is NP-complete [81]. In [36], the authors introduced a polynomial algorithm that approximates the set of causes for the violation of LTL formula. The reduction from binary causal models to Boolean circuits and from Boolean circuits to model checking as introduced in [48] proved that computing the degree of responsibility for branching time formula is also NP-complete, and they provided a polynomial

Algorithm 1 . Generate Causes

```
1: Inputs: The counterexample $MIPCX(s_0 \models \phi)$, The probabilistic formula $\phi = P_{\leq p}(\varphi)$
   where $\varphi$ is of the form $\phi_1 \mathbf{U} \phi_2$ or $(\phi_1 \mathbf{U}^{\leq n} \phi_2)$
2: Outputs: Causes with their responsibilities and probabilities
3: Causes := $\emptyset$
4: for each state $s \in MIPCX(s_0 \models \phi)$ do
5:     W:= 0
6:     if $s$ is the last state in a path $\sigma$ then
7:         Causes with dR:= Causes $\cup$ FindCauses($s, \phi_2, W$)
8:         Pr(Causes)=$\sum_{s \in \sigma | \sigma \in MIPCX(s_0 \models \phi)} P(\sigma)$
9:     else
10:        Causes with dR:= Causes $\cup$ FindCauses($s, \phi_1, W$)
11:        Pr(Causes)=$\sum_{s \in \sigma | \sigma \in MIPCX(s_0 \models \phi)} P(\sigma)$
12:    end if
13: end for
   function FINDCAUSES($s$, $\psi$, $W$)
2:     if $\psi$ is of the form $a$ where $a \in AP$ and $a \in L(s)$ then
           return $(\langle s,a \rangle, dR(\langle s,a \rangle) = 1/(W+1))$
4:     end if
       if $\psi$ is of the form $\neg a$ where $a \in AP$and $a \notin L(s)$ then
6:         return $(\langle s,\neg a \rangle, dR(\langle s,\neg a \rangle) = 1/(W+1))$
       end if
8:     if $\psi$ is of the form $\psi_1 \wedge \psi_2$ then
           return FindCauses($s$, $\psi_1$, $W$) $\cup$
10:        FindCauses($s$, $\psi_2$, $W$)
       end if
12:    if $\psi$ is of the form $\psi_1 \vee \psi_2$ then
           if $s \models \psi_1$ and $s \models \psi_2$ then
14:            return FindCauses($s$, $\psi_1$, $W++$) $\cup$
               FindCauses($s$, $\psi_2$, $W++$)
16:            if $s \models \psi_1 \wedge s \not\models \psi_2$ then
                   return FindCauses($s$, $\psi_1$, $W$)
18:            end if
               if $s \not\models \psi_1 \wedge s \models \psi_2$ then
20:                return FindCauses($s$, $\psi_2$, $W$)
               end if
22:        end if
       end if
24: end function
```

algorithm for computing responsibility for read-once Boolean formulas.

It is evident that the complexity of Algorithm 1 is polynomial in the number of states of $MIPCX(s_0 \models \phi)$ and the size of φ. This follows from the fact that the left sub-formula ϕ_1 is evaluated at most once in each state except the last ones, in which the right sub-formula $\phi 2$ is evaluated at most once. This is much less hard than evaluating both sub-formulas at each state of the counterexample [36]. Moreover, if the sub-formula is AND formula, it is not necessary to be evaluated, since its satisfaction is ensured. Computing the degree of responsibility is based on the formula structure and it is performed while evaluating the sub-formulas, where the number of changes represented by W increases just with response to OR formula. Thus, computing the degree of responsibility does not bring additional complexity. In addition, computing the degree of responsibility is based only on local information at each state, regardless of other states. This is much less hard than computing the degree of responsibly in a state depending on the information in other states [16].

4.5 Probabilistic Counterexamples for MDPs

The PCTL property $\phi = \mathbf{P}_{\leq p}(\varphi)$ is violated on MDP, if there exists at least one *scheduler* d such that the probability mass of the paths satisfying φ under d exceeds the bound p. A probabilistic counterexample for the property $\phi = \mathbf{P}_{\leq p}(\varphi)$ can be formed of a set of paths from $FinitePaths_d(s_0)$ starting at state s and satisfying the path formula φ. We denote this set by $FinitePaths_d(s_0 \models \phi)$.

It is clear that given a scheduler d, it is possible to find a set of probabilistic counterexamples under d denoted $PCX_d(s_0 \models \phi)$, which is a set of any combination from $FinitePaths_d(s_0 \models \phi)$, their probability mass exceeds the bound p. Like counterexample for DTMC and CTMC, the most indicative counterexample is minimal counterexample (has the least number of paths from $FinitePaths_d(s_0 \models \phi)$) and its probability mass is the highest among all other minimal counterexamples. We denote the most indicative probabilistic counterexample of an MDP by $MIPCX_d(s_0 \models \phi)$. For finding $MIPCX_d(s_0 \models \phi)$, we have to find first the maximizing scheduler d that induces this counterexample.

Example 3 Let us consider the example of MDP shown in Figure 4.2 and the property $P_{\leq 0.5}(\varphi)$, where $\varphi = (a \vee b)U(c \wedge d)$.

This property is violated in this model ($s_0 \not\models P_{\leq 0.5}(\varphi)$), since there exists a scheduler d induces a set of finite paths satisfying φ and their probability mass is higher than the probability bound (0.5). Any combination from $FinitePaths_d(s_0 \models \phi)$ having probability mass higher than 0.5, is a valid probabilistic counterexample including the whole set. We

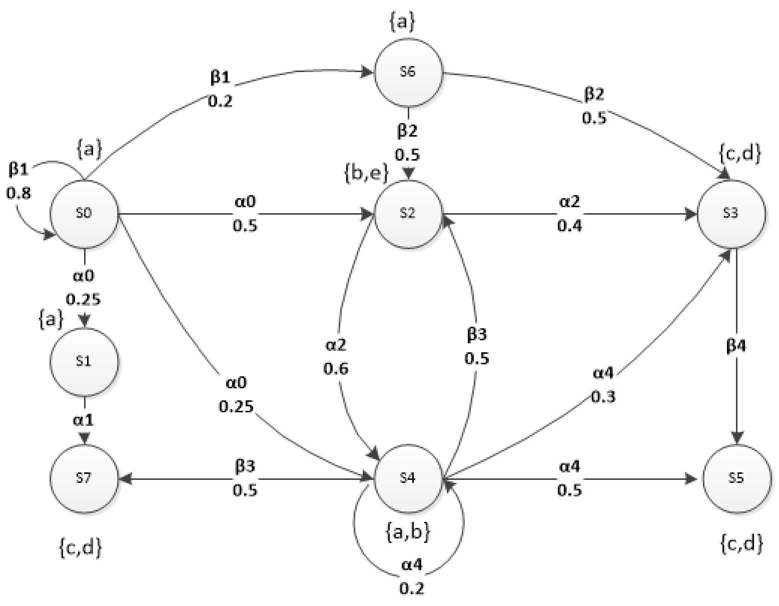


Fig. 4.2 An MDP

Example 3 Let us consider the example of MDP shown in Figure 4.2 and the property $P_{\leq 0.5}(\varphi)$, where $\varphi = (a \vee b)U(c \wedge d)$.

This property is violated in this model ($s_0 \nvDash P_{\leq 0.5}(\varphi)$), since there exists a scheduler d induces a set of finite paths satisfying φ and their probability mass is higher than the probability bound (0.5). Any combination from $FinitePaths_d(s_0 \models \phi)$ having probability mass higher than 0.5, is a valid probabilistic counterexample including the whole set. We can find three counterexamples.

$$\begin{aligned} P(CX_{d1}) = P(\{ & s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_7, s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_2} s_3, s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_2} \\ & s_4 \xrightarrow{\alpha_4} s_3, s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_2} s_4 \xrightarrow{\alpha_4} s_5, s_0 \xrightarrow{\alpha_0} s_4 \xrightarrow{\alpha_4} s_5 \ \}) \\ = {} & 0.25 + 0.2 + 0.09 + 0.15 + 0.12 = 0.81 \end{aligned}$$

$$\begin{aligned} P(CX_{d2}) &= P(\{s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_7, s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_2} s_4 \xrightarrow{\alpha_4} s_5, s_0 \xrightarrow{\alpha_0} s_4 \xrightarrow{\alpha_4} s_5\}) \\ &= 0.25 + 0.15 + 0.12 = 0.52 \end{aligned}$$

$$\begin{aligned} P(CX_{d3}) &= P(\{s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_7, s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_2} s_3, s_0 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_2} s_4 \xrightarrow{\alpha_4} s_5\}) \\ &= 0.25 + 0.2 + 0.15 = 0.60 \end{aligned}$$

The last probabilistic counterexample is the most indicative since it is minimal and its probability is higher than the other minimal counterexample CX_{d2},$P(CX_{d3})$
$> P(CX_{d2})$.

assignments but he also needs to know how such actions are involved, with respect to that, the designer could fix the action in way he get the acceptable outcome. We recall that blame considers whether an action a is to blame for an outcome φ under uncertainty [47].

Consider a state s from $MIPCX_d(s_0 \models \phi)$. For each action $\alpha \in A(s)$, we denote by $Suc(s,\alpha)$ the set of α-successors of s, where α-successor is a state $s' \in S$ such that $P(s,\alpha,s') > 0$. We should mention that for every s from $MIPCX_d(s_0 \models \phi)$, the set $A(s)$ is singleton. We associate for each transition from s to s', where $s' \in Suc(s,\alpha)$ a probability as follow

$$P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s') = \sum_{(s,\alpha,s') \in \sigma \mid \sigma \in MIPCX_d(s_0 \models \phi)} P(\sigma) \tag{4.3}$$

It is evident that not every transition enabled by an action α has the same presence in the paths forming the counterexample. So this probability measures simply the contribution of a transition to the probability of the counterexample by summing the probabilities of the paths in which it is included.

Proposition 4.6.1. Consider a transition $(s,\alpha,s') \in MIPCX_d(s_0 \models \phi)$, $Max(P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s')) = Pr(s)$ iff s' is the unique successor of s.

Proof. $\Pr(s)$ represents the sum of probabilities of paths in which s is included, $P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s'))$ represents the probabilities of the paths in which the transition is included, so it is sufficient to prove that both of them are included in the same set of paths iff s' is the unique successor of s. Let s' and s'' two successors of s, if s is included in N paths, the transition (s,s'') will be included at least in one path from this set, and thus s' will be included at most at $N-1$ of paths. Hence, $Pr(s) \neq P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s'))$ if there is another successor s'' of s. □

We should mention that every transition in a probabilistic program describes how the values of the variables evolve over time, and thus considering the transitions and their contribution to the error is very important as debugging information, which makes it a required measure for the definition of blame.

Definition 4.6.1. (Blame) Consider a counterexample $MIPCX_d(s_0 \models \phi)$ for a probabilistic formula $\phi = \mathbf{P}_{\leqslant p}(\varphi)$, a state s, an action $\alpha \in A(s)$ and a set of successors $Suc(s,\alpha)$. The degree of blame for an action α for the violation of $\phi = \mathbf{P}_{\leqslant p}(\varphi)$ in a state s denoted $dB(s,\alpha,\phi)$ is

$$\sum_{s' \in Suc(s,\alpha)} dR(s', X = x, \phi) P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s') \tag{4.4}$$

That is, the degree of blame dB informs us about the contribution of an action α in state s to the violation of the probabilistic formula $\phi = \mathbf{P}_{\leqslant p}(\varphi)$. While responsibility stands as a criticality measure for actual causes given well-defined states for the violation of $\phi = \mathbf{P}_{\leqslant p}(\varphi)$, dB describes how an action should be blamed for this violation through considering the probabilities assigned to the transitions leading to these states. So that, the action more blamed for the violation will be the one more contributing to the probability of $MIPCX_d(s_0 \models \phi)$ and leading to more critical states.

Definition 4.6.2. (Most Blame) Action $\alpha \in A(s)$ has most blame for the violation of PCTL property $\phi = \mathbf{P}_{\leq p}(\varphi)$, if $dB(s, \alpha, \phi) \geq dB(s', \alpha', \phi)$ for any $\alpha' \in A(s')$.

Proposition 4.6.2. Consider an action α enabled at a state s, $Max(dB(s, \alpha, \phi)) = P(MIPCX_d(s_0 \models \phi))$ iff $\forall \sigma \in MIPCX_d(s_0 \models \phi) \exists (s, \alpha, s') \in \sigma$, where $s' \in Suc(s, \alpha)$ and s' is critical with respect to some variable X.

That is, the maximum degree of blame of an action α enabled at a state s is equal to the probability of the counterexample if only if for every path σ from the counterexample, there exists a transition from s to a state s' under this action, and s' is critical with respect to such variable X.

Theorem 4.6.3. Let $\alpha \in A(s)$ and $s' \in Suc(s, \alpha)$, $(s', X = x)$ is most responsible cause $\not\Rightarrow \alpha$ has the most blame.

Proof. Let $s1$ and $s2$ be two states, $\alpha 1$ and $\alpha 2$ two actions enabled at these states respectively. Let $\alpha 1$ leads to a most responsible cause C and we assume that $dB(s1, \alpha 1, \phi) \geqslant dB(s2, \alpha 2, \phi)$. Then, for every cause C', $\alpha 2$ leads to, $dR(C')Pr(C') < dR(C)Pr(C)$. Let C be the unique cause $\alpha 1$ it leads to, let $Pr(s2) > Pr(s1)$ and let all the causes $\alpha 1$ and $\alpha 2$ lead to are critical. With respect to proposition 4.6.1, the probability of the transition leading to C' is $Pr(s1)$, and thus with respect to the definition of blame 4.6.2, $dB(s1, \alpha 1, \phi)$ will be at most $Pr(s1)$, which is less than $Pr(s2)$, and since the causes $\alpha 2$ leads to are critical, it contradicts $\alpha 1$ being the action with the most blame.

4.7 Algorithm for Computing Blame

The algorithm 2 explores the counterexample and computes the causes with their responsibilities and probabilities with respect to each state s, and computes the degree of blame for each action enabled at this state. The condition put on last state always follows lemma 4.2.3. Algorithm 2 uses the function FindCauses that takes a state and state formula as

Algorithm 2 . Generate Diagnoses

```
 1: Inputs: The counterexample $MIPCX(s_0 \models \phi)$, The probabilistic formula $\phi = P_{\leq p}(\varphi)$
    where $\varphi$ is of the form $\phi_1 \mathbf{U} \phi_2$ or $(\phi_1 \mathbf{U}^{\leq n} \phi_2)$
 2: Outputs: Causes with responsibilities and probabilities
 3:          Actions with blame
 4:
 5: Causes := $\emptyset$
 6: Actions := $\emptyset$
 7: for each state $s \in MIPCX(s_0 \models \phi)$ do
 8:     W:= 0
 9:     if $s$ is the last state in a path $\sigma$ then
10:         Causes with dR:= Causes $\cup$ FindCauses($s, \phi_2, W$)
11:         Pr(Causes)=$\sum_{s \in \sigma | \sigma \in MIPCX(s_0 \models \phi)} P(\sigma)$
12:     else
13:         Causes with dR:= Causes $\cup$ FindCauses($s, \phi_1, W$)
14:         Pr(Causes)=$\sum_{s \in \sigma | \sigma \in MIPCX(s_0 \models \phi)} P(\sigma)$
15:         $P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s') = \sum_{(s,\alpha,s') \in \sigma | \sigma \in MIPCX_d(s_0 \models \phi)} P(\sigma)$
16:     end if
17: end for
18: for each $s \in MIPCX(s_0 \models \phi)$ do
19:     Actions with dB:= Order(Actions $\cup$ $\alpha \in A(s)$, $dB(s,\alpha,\phi_1) = \sum_{s' \in Suc(s,\alpha)} dR(s', X =$
20:                 $x, \phi_1) P_{\alpha}^{MIPCX_d(s_0 \models \phi)}(s,s')$)
21: end for
22: OUTPUTDIAGNOSES(Causes with dR and Pr, Actions with dB)
    function OUTPUTDIAGNOSES(Causes with dR and Pr, Actions with dB)
 2:     for each action$\alpha \in A(s)$ from Actions do
            Output ($\alpha$)
 4:         for each a successor $s' \in Suc(s,\alpha)$ do
                Output Causes Ordered with respect to $dR \times Pr$
 6:             Output $(s,s')$
            end for
 8:     end for
    end function
```

input as well as the variable W, and returns recursively the set of causes and their responsibilities. This algorithm computes an over-approximate set of causes, since computing the set of causes exactly in binary causal models is NP-complete [19]. Like Algorithm 1, the algorithm 2 is linear in the size of $MIPCX_d(s_0 \models \phi)$ and the size of φ , because reconfiguring the algorithm to compute the degree of blame by adding the loop in line 18 does not bring additional complexity since it is directly based on measures already computed, which are te degree of responsibility of each cause and the probability of each transition $P_\alpha^{MIPCX_d(s_0 \models \phi)}(s,s')$. Computing $P_\alpha^{MIPCX_d(s_0 \models \phi)}(s,s')$ (line 15)is performed under the same loop for computing the probabilities of causes (line 14).

At the end we present the function OUTPUTDIAGNOSES that shows the way of presenting the diagnoses to te user. It gets the actions ordered with respect to dB and the causes with dR and Pr, and outputs the diagnoses in order. We see that the output of the diagnoses starts with the action with the most blame, and among the causes it leads to, we begin by the most responsible cause, by indicating also the transition leading to this cause. Presenting the transition enabled under this action is very important as a diagnostic information, because transitions in typical probabilistic program describe how the values of the variables evolve over time.

Example

Let us apply this algorithm on the counterexample $CX3$ from the previous example. The user gets the action α_0 first, since $dB(s_0,\alpha_0) = 0.6$ is the highest, it is equal to the probability of the counterexample. From $Succ(s_0,\alpha)$, the user gets first the cause (s_2,b) because it is the most responsible by computing the measure $P(s_2,b) \times dR(s_2,b) = 0.58 > P(s_1,a) \times dR(s_1,a,\phi) = 0.41$. The following action the user gets is α_2 with degree of blame $dB(s_2,\alpha_2,\phi) = 0.5 \times (0.15) + 1 \times (0.2) = 0.275$ with the causes it led to $\{(s_3,c),(s_3,d)\}$ and $\{(s_4,a)\}$,$\{(s_4,b)\}$ respectively, then α_1 with $dB(s_1,\alpha_1) = 0.25$ with the cause it led to $\{(s_7,c),(s_7,d)\}$, and finally the action α_4 with $dB(s_4,\alpha_4,\phi) = 0.15$ with the cause it led to $\{(s_5,c),(s_5,d)\}$.

4.8 Experimental Results

4.8.1 Algorithm 1

We have implemented the first method for DTMCs and CTMCs in Java. To evaluate our method, we use two benchmark case studies, the cyclic polling model taken from [50] and the embedded control systems taken from [144]. All the experiments were carried out on windows XP with Intel Pentium CPU 3.2 GHz speed and 512 mb of memory.

Polling Server System The system is modelled in PRISM as a CTMC [6], where the number of stations handled by the polling server is denoted by N. Each station has a single-message buffer and is cyclically attended by the server. We choose the property that measures the probability of station 1 being served $(s = 1)$ before station 2 $(s = 2)$ where $(a = 1)$ denotes serving. This property is given as follows:

$$P = ?[!(s = 2 \wedge a = 1)U(s = 1 \wedge a = 1)]$$

We test this property using PRISM for (N=3, N=5, N=7 and N=9). For all of these values, PRISM renders a probability higher than 0.5. As a result, we chose the value 0.5 as a threshold. The property can be rewritten as follows:

$$P \leq 0.5[(!(s = 2) \vee !(a = 1))U(s = 1 \wedge a = 1)]$$

We use DiPro to generate the counterexample, which in turn uses PRISM. We have to specify the model and the property to be verified, and then DiPro computes the counterexample for violating the probability threshold. The counterexample can be rendered graphically and stored in text format as well as XML format. The tool implements many algorithms. In our experiments, we used the heuristic search algorithm XBF that computes the counterexample as a diagnostic sub-graph. Our method takes the counterexample generated from DiPro and the property to be verified as parameters, and outputs the causes as well as their probabilities and responsibilities. Detailed results of the experiments are given in Table 4.1, Table 4.2 and Table 4.3.

The Table 4.1shows the size of the model, states and transitions, and the time required for its construction as well as the probability estimated for each N given. The Table 4.2 contains information concerning the counterexample generation. It is evident that the number of explored vertices and explored edges while searching the counterexample is always less than the number of states and transitions of the model. It is also evident that the number of diagnostic paths is less than the number of solution traces. While solution traces refer to all the paths of the diagnostic sub-graph found through exploring the model, diagnostic paths refer just to the paths forming the counterexample.

The Table 4.3 shows the results of the algorithm execution. The first column represents the number of causes generated from the counterexample with respect to the states, while the second column shows the number of classes of causes probabilities. It is evident that the number of probabilities classes is lower than the number of causes, because many causes could share the same probability. The results show that the time taken for computing the

Table 4.1 PRISM results: model size, time construction and probabilities.

N	States	Transitions	Time Construction(Sec)	Probability
3	36	84	0	0.52
5	240	800	0.043	0.53
7	1344	5824	0.063	0.53
9	6912	36864	0.09	0.54

Table 4.2 DiPro results for counterexample generation.

N	Explored vertices	Explored edges	Time Construc-tion(Sec)	Solution traces	Diagnostic paths
3	29	49	10.2	5	2
5	188	411	30.23	30	5
7	1094	3310	295.18	184	18
9	4722	16021	1240.62	533	55

causes is negligible comparing to the time of generating the counterexample. We notice that changing N from 3 to 9 that results in augmenting the size of the diagnostic sub-graph from 29 to 4722 vertices and from 49 to 16021 edges, and increasing of causes from 11 to 367, this results just in increasing in 1 second of execution time. The number of causes also is very low comparing to the size of the model and the diagnostic sub-graph.

The Table 4.4 shows the actual causes and compares their presence in the counterexample. We begin by the left sub-formula of the until formula $(!(s=2)\vee!(a=1)$. The results show that we faced all the possible scenarios. The first: $!(s=2)$ is the only cause, where $a=1$ means another station is served. The second: $!(a=1)$ is the only cause, where $s=2$ means station 2 is polled but is not served. The third: both of them $!(s=2)$ and $!(a=1)$ are actual causes, which means other states are polled without being served. The diagnostic information shows that the cause for $(!(s=2)\vee!(a=1)$ being fulfilled is often due to the absence of station 2, which means that the server is not polling it at all, whereas the case of polling station 2 by the server $(s=2)$ but without being served $!(a=1)$ is very rare. For the right sub-formula of the until formula, the causes are $(s=2)$ and $(a=1)$. From Table 4.3 and Table 4.4, we notice that the number of causes represents exactly the number of diagnostic paths, because the state in which $(s=1\wedge a=1)$ is the last state in each diagnostic path. Given these results, we can easily define the number of critical causes. The number of states involved in criticality is the number of states in which $dR=1$, which are the states in which $!(s=2)$, $!(a=1)$ and $(s=1\wedge a=1)$ are the only causes. For example when N=3, the number of critical causes are (3+1+2=6). The other 5 states are not involved in criticality, because in these states $!(s=2)$ and $!(a=1)$ are both fulfilled, as a result each cause has a responsibility 1/2.

Table 4.3 Execution results of our algorithm.

N	Causes	Causes Probabilities	Execution Time(Sec)
3	11	3	0.093
5	38	7	0.197
7	126	27	0.582
9	367	78	1.52

Table 4.4 Detailed results for the causes generated

N	$!(s=2)/(dR=1)$	$!(a=1)/(dR=1)$	$!(s=2)$ and $!(a=1)/(dR=1/2)$	$(s=1 \wedge a=1)/(dR=1)$
3	3	1	5	2
5	11	3	19	5
7	42	5	61	18
9	123	7	182	55

Embedded Control System

The Embedded control system consists of input processor (I) that reads incoming data from three sensors (S1, S2 and S3) and then passes it to main processor (M). The processor M processes the data and sends instructions to an output processor (O) that controls two actuators (A1 and A2) using these instructions, (see Figure 4.3). The system is modelled in PRISM as CTMC [9].

Any of the system's components M, I/O, the sensors and the actuators may fail; as a result the system is shut down. The types of failures are:

$$\begin{gathered} fail\_sensors = (i = 2 \wedge s < MIN\_SENSORS) \\ fail\_actuators = (o = 2 \wedge a < MIN\_ACTUATORS) \\ fail\_io = (count = MAX\_COUNT + 1) \\ fail\_main = (m = 0) \end{gathered}$$

The first failure occurs when the number of working sensors drops below 2 and the input processor is functioning. The second failure occurs when the number of working actuators drops below 1 and the output processor is functioning. The third failure occurs when the number of consecutive cycles skipped exceeds a limit Max_Count. The last means that main processor has filed. The down status of the system is labelled as:

$$down = fail\_sensors | fail\_actuators | fail\_io | fail\_main$$

For this model, we choose the PCTL property that estimates the long-run probability of I/O failure occurring first, which is given as follows:

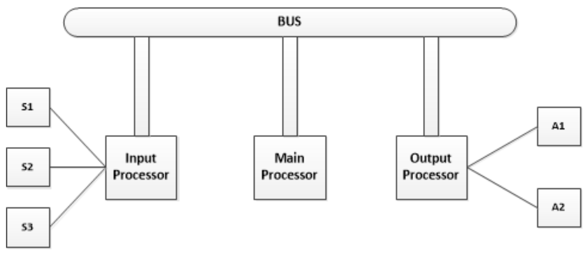


Fig. 4.3 Embedded controle system

$$P = ?[!(down) U fail\_io]$$

We test this property using PRISM for ($Max\_Count = 2$). For this value, PRISM renders a probability equal to 0.24. We chose the value 0.2 as a threshold for this property to generate the counterexample. Thus the property can be rewritten as follows:

$$P \leq 0.2[!(down) U fail\_io]$$

We follow the same steps indicated in the previous case study. The PRISM model consists of 3478 states and 14639 transitions. For generating the counterexample, DiPro Explored 225 traces resulting in 795 vertices and 1574 edges in more than 1 minute. Finally, the counterexample rendered consists just of 36 diagnostic paths. We pass this counterexample to our algorithm for generating the causes and their probabilities and responsibilities. Our algorithm takes less than 1 second as execution time. We notice that this time is negligible comparing to the size of the model and the time taken for computing the counterexample.

The causes generated for this property are as follows: For the right sub-formula ($fail\_io$), the cause generated is $C0 = (count = MAX\_COUNT + 1)$. For the left sub-formula, the set of causes for the system to not be in down state is:

$$C1 = \neg(i = 2)$$
$$C2 = \neg(s < MIN\_SENSORS)$$
$$C3 = \neg(o = 2)$$
$$C4 = \neg(a < MIN\_ACTUATORS)$$
$$C5 = \neg(count = MAX\_COUNT + 1)$$
$$C6 = \neg(m = 0)$$

$C1$and $C2$ refer the probable causes for not failure occurring on the level of sensors, whereas $C3$ and $C4$ refer to the probable causes for not failure occurring on the level of actuators. The causes for not failure occurring in I/O and M are singletons, $C5$ and $C6$ respectively.

The number of states from 36 diagnostic paths in which we found theses causes are estimated to be 296 states, this is much less than the states of the model (3478). The number of causes probabilities found are 61, ranging between 0 and 0.054. It is evident that the number of probabilities classes (61) is lower than the number of states (296), because the states included in the same set of paths share the same probability.

Due to the size of the counterexample, it is not possible to cite here all the pairs (state, variable), but we can give results description concerning $C1$ to $C6$. $C5$ and $C6$ were found to be the most responsible causes in such states, since they are singletons. For sensor and actuators failures, we are facing disjunctive scenario, which means that either $C1$ or $C2$ can be a cause for not sensors failure occurring. It is the same case for actuators failure with $C3$ and $C4$. In all states, $C2$ (all sensors are working) and $C4$ (all actuators are working) are found to be the actual causes for not sensors and actuators failures occurring respectively. Thus, they have absolutely more responsibility than the two other causes in such states,$C1$ (input processor not functioning) and $C3$ (output processor not functioning) respectively. For example, in the state having highest probability 0.054, we found that the only causes for sensors failure and actuators failure are $C2$ and $C4$ respectively, thus they have 1 as a degree of responsibility. These causes are rendered to the user as the most responsible causes, since they are critical and they have the highest probability.

To estimate the probability of I/O failure occurring first continuously , we use the following CSL property:

$$P = ?[!(down) U^{\leq T*3600*24} fail\_io]$$

We test this property using PRISM for (T=15, T=20, T=25 and T=30), where each value of T represents number of days, since one time unit is one second according to PRISM model. For these values, PRISM renders a probability Higher than 0.05. Therefore, we chose the value 0.05 as a threshold for this property to generate the counterexample. Thus, the property can be rewritten as follows:

$$P \leq 0.05[!(down) U^{\leq T*3600*24} fail\_io]$$

We follow the same steps performed before with the previous property. The execution results are given in the Table 4.5 for ($Max\_Count = 4$ and $Max\_Count = 8$). For all values of Max_Count and T, the execution time of our algorithm was estimated to be less than 0.1 second. For ($Max\_Count = 4$), we notice that within 15 days the probability of I/O failure occurring first is 0.076, as the time passes, the probability increases, until 30 days the probability will reach 0.11. From $T = 20$, the number of : diagnostic paths, causes and

Table 4.5 Execution results of our algorithm for T and N given.

T	N	Property's Probability	Diagnostic Paths	Causes	Causes Probabilities	highest probability
15	4	0.076	10	57	16	0.026
	8	0.066	8	77	13	0.026
20	4	0.091	9	49	14	0.030
	8	0.079	8	77	13	0.030
25	4	0.102	9	49	14	0.032
	8	0.088	7	65	11	0.032
30	4	0.110	9	49	14	0.033
	8	0.095	7	65	11	0.033

causes probabilities are the same, nevertheless, the probability of the most responsible cause (highest probability) is not the same, which means that the counterexamples for the values ($T = 20$,$T = 25$ and $T = 30$) are not identical. We notice also that the probability of the most responsible cause increases as the time passes.

As same as for ($Max\_Count = 4$), for ($Max\_Count = 8$) the probability of I/O failure occurring first increases as the time passes. The number of : diagnostic paths, causes and causes probabilities are the same for ($T = 15$ and $T = 20$) and are also the same for ($T = 15$ and $T = 20$). By comparing the results of $Max\_Count = 4$ and $Max\_Count = 8$, we see that the probability of I/O failure occurring first decreases with $Max\_Count = 8$. This is evident, because we gave the main processor more skipped cycles to wait before deciding that I/O have failed.

Although the number of diagnostic paths decreases, the number of causes rises up, which means that these diagnostic paths are longer than those found with $Max\_Count = 4$. Despite the accurate change in property's probability, number of diagnostic paths and the number of causes with $Max\_Count = 8$, we see that the probability of the most responsible cause is still as the same as for $Max\_Count = 4$ given each value of T. We also found that the most responsible cause for $Max\_Count = 4$ is the same for $Max\_Count = 8$, given each value of T.

4.8.2 Algorithm 2

We have implemented the second method in Java. To evaluate our method, we used two benchmark case studies, the Zeroconf Protocol [10] and CSMA/CD protocol [79]. All the experiments were carried out on windows XP with Intel Pentium CPU 3.2 GHz speed and 512 mb of memory. DiPro employees two algorithms for generating counterexamples for

Table 4.6 PRISM results for Zeroconf

Reset	K	states	transitions
true	4	9683	15727
	6	7743	11401
	8	7743	11401
false	4	59076	121265
	6	58937	120525
	8	58937	120525

MDPs, Eppstein's algorithm and K* algorithm. In our experiments we use the search algorithm K*. Our method takes the counterexample generated from DiPro and the property to be verified as input, and outputs the diagnoses.

Zeroconf

The protocol is modelled in PRISM as an MDP, where the number of abstract hosts is denoted by N, the number of probes to send is denoted by K and the probability of message loss is denoted by $loss$. Each station has a single-message buffer and is cyclically attended by the server. The buffer could store the messages that it want to send, in such cases, messages are not relevant after reconfiguring, and thus keeping these messages can slow down the network and making hosts reconfigure when they do not need to. We therefore considered two different versions of the network: one where the host does not do anything about these messages (No_Reset) and the another where the host deletes these messages when it decides to choose a new IP address (Reset).

We chose the property that measures the probability of not choosing a fresh address by time T. This property is given as follows:

$$Pmax =?[!(l = 4 \wedge ip = 2)U(t > T)]$$

We test this property using PRISM for both types of network (Reset) and (No_Reset) for the following values (T = 10; N = 1000;loss = 0.1 and K could vary from 4 to 8). For these values, PRISM renders a probability higher than 0.5. As a result, we chose the value 0.5 as a threshold. The property can be rewritten as follows:

$$P \leq 0.5[!(l = 4 \wedge ip = 2)U(t > T)]$$

The PRISM results are shown in Table 4.6. We notice that the size of the models when there is no reset is very huge comparing to reset. Despite that, DiPro renders the same counterexample for all these different configurations with the same execution time. For generating the counterexample, DiPro Explored 24 traces resulting in 121 vertices and 150 edges in 5 seconds for all the configurations. Finally, the counterexample rendered by DiPro consists just of 8 diagnostic paths. The time required for generating the counterexample is very small with respect to the model, because DiPro rapidly succeeded to find the scheduler that induces the counterexample through finding paths of high probabilities. In other cases, where we have paths of small probabilities could not be the same case.

We pass this counterexample to our algorithm for generating the diagnoses. Our algorithm takes less than 1 second as execution time. The causes generated for this property are as follows: For the right sub-formula, the cause generated is singleton $C0 = (t > T)$. For the left sub-formula, the set of causes for not choosing a fresh address: $C1 =!(l = 4)$ and $C2 =!(ip = 2)$. Notice that we are facing disjunctive scenario here, which means that either $C1$ (address not in use) is the actual cause or $C2$ (not fresh address) or both of them. Our results show that except the initial state where $ip = 1$ (IP address of an abstract host which the concrete host is currently trying to configure), the actual cause for not choosing fresh address within 10 time units is $C1$, which means that we reach states in which there is fresh ip which the concrete host is currently trying to configure but without being used.

Concerning the actions and their blame, in the model we have two main actions causing the non-determinism in such states which are: 'Reconfigure' denoted *rec* and 'defend' which is performed by sending an ARP packet and thus this action denoted in the model by *send*. For the counterexample generated given the previous property, our results show that there is no state in which the host reconfigures, which means that the only action we are dealing with is *send*. As a result, we compute the dB of *send* action in such states. We found that the action has one degree of blame $dB = 0.25$ in each state is enabled, which means that at each state, *send* has half the blame for the error, since it contributes to the half probability of the counterexample. We should mention that this measure (0.25) refers to one of the six probability classes found, and since all the states reached through taking *send* are critical (as we mentioned before that the only cause is $!(l = 4)$), with respect to proposition 4.6.2, the dB of *send* is as the same as the probability of the state in which it is enabled.

Carrier Sense, Multiple Access with Collision Detection (CSMA/CD)

CSMA/CD is a protocol for carrier transmission access in Ethernet networks that avoids collision (minimising simultaneous use of the channel) when Network Interface Card (NIC) tries to send its packet. The protocol is modelled as a probabilistic timed automata (PTA), and can be reduced to an MDP in order to be analysed against probabilistic properties by PRISM [5]. The model in PRISM consists of three main components or modules, the two senders namely station 1 and station 2 respectively and the third component is the bus or the medium. The protocol functionality is as follows: if a station has a data to send, it listens first to the medium, in case it is free, the stations send the data, otherwise (bus is busy), it repeats the process after random amount of time. If there is a collision , the station

Table 4.7 PRISM Results for CSMA/CD

K	States	Transitions
2	1083	1282
4	7958	10594

Table 4.8 DiPro Results for CSMA/CD

K	States	Transitions	Time Construction(Sec)	Diagnostic Paths
2	1037	1276	6 sec	134
4	4222	5201	16 sec	324

attempts to retransmit the packet where the scheduling of the retransmission is determined by a truncated binary exponential backoff process.

We chose the property that estimates the maximum probability of all stations sending successfully before a collision with max backoff. This property is given as follows:

$$Pmax = ?[!"collision\_max\_backoff" U "all\_delivered"]$$

We tested this property using PRISM for the following values: N=2 and K=2, N=2 and K=4 respectively, where N represents the number of stations and K represents the exponential backoff limit. For these values, PRISM renders a probability higher than 0.7. As a result, we chose the value 0.7 as a threshold. The property can be rewritten as follows:

$$P \leq 0.7[!"collision\_max\_backoff" U "all\_delivered"]$$

where $"collision\_max\_backoff"$ and $"all\_delivered"$ are defined as follows:

$"collision\_max\_backoff" = (cd1 = K\&s1 = 1\&b = 2)|(cd2 = K\&s2 = 1\&b = 2)$, $"all\_delivered" = s1 = 4\&s2 = 4$. The variables $cd1$ and $cd2$ refer to collision counters for both stations where k represents the backoff limit , $s1$ and $s2$ refer to the state of the stations where $s1 = 1, s2 = 1$ indicate that the stations are transmitting data, and finally b refers to the state of the bus where $b = 2$ indicates that there is a collision.

We use the K* algorithm employed by DiPro to generate the counterexample. Our method takes the counterexample generated from DiPro and the property to be verified as input, and outputs the diagnoses.

The Table 4.7 shows the size of the model by PRISM. Table 4.8 shows the states and transition explored while searching for the counterexample and the time required for its construction by DiPro. We notice that DiPro nearly explored all the model to generate the counterexample for K=2, whereas for K=4, DiPro explores nearly half of the model. For k=2, the counterexample generated consists of 134 diagnostic paths and for k=4, the

Table 4.9 Execution results of our algorithm on CSMA/CD

K	Causes	Causes Probabilities	Execution Time(Sec)
2	616	98	3 sec
4	923	47	8 sec

counterexample generated consists of 324 diagnostic paths. We pass both counterexamples to our algorithm for generating the diagnoses.

Table 4.9 shows the execution results of our algorithm. The second column represents the number of causes generated from the counterexample with respect to the states, while the third column shows the number of classes of causes probabilities. We notice that the number of candidate causes is small comparing to the size of the model. We also notice that the number of probabilities classes is lower than the number of causes, because many causes could share the same probability. The results show that the time taken for computing the causes is less than the time taken for generating the counterexample.

The causes generated for this property are as follows: For the right sub-formula, the cause generated is a conjunct $C0 = (s1 = 4\&s2 = 4)$. For the left sub-formula, the set of causes for not facing a collision with max backoff are: $C1 = \neg(cd1 = K)$, $C2 = \neg(s1 = 1)$, $C3 = \neg(b = 2)$,$C4 = \neg(cd2 = K)$ $C5 = \neg(s2 = 1)$.

For both values of K, our results show that there are causes that share the same probability, as we said before that the most responsible cause may not be unique. In addition, for both values of K, the most responsible causes are the same. Among all causes, we found that the most responsible causes for not facing collision are $C1$ and $C4$, where the state in which these causes are found has the highest probability and it is critical with respect to these causes. Concerning the actions and their blame, we found that the transition leading to this critical state has highest probability among all other transitions. This transition is represented by valuations $b = 2$ (bus busy - collision) and $s2 = 1$ (station 2 is transmitting) and is enabled by the action $send2$ that represents sending data by station 2. This action was found as the action having the most blame among all other actions. Given this information it would be easy to go-back to the model, which is given in PRISM guarded command language, and map the command which is considered more likely the responsible for exceeding the probability the given threshold. For this example precisely, given the previous information we should begin verifying the command $[send2](b = 1|b = 2)\&(y1 < sigma)- > (b' = 2);$ in line 40 (the complete model is available in the PRISM Benchmark Suite under MDPs section).

The other states in which $C1$ and $C4$ are not the causes, we find that $C3$ is the most responsible cause. So by defining the transition leading to it, and under which action is enabled, we could also map to the commands related and analyse all the model driven by the weight calculated (cause probability, responsibility and blame).

4.9 Conclusion

In this chapter we have shown how the notions of causality and responsibility can be interpreted in the context of probabilistic counterexamples. Due to the probabilistic nature of the causal model, we had to define for each cause its probability. Accordingly, we introduced the notion of the most responsible cause. Following that, we first introduced an algorithm for diagnoses generation for DTMCs and CTMCs that acts as a guided-method to the most responsible causes in the counterexample. The most responsible cause is considered to be the most relevant to the user. We then extended our method to counterexamples for MDPs by adopting the notion of blame where we showed that delivering the causes/actions with respect to their responsibility/blame stands as a good debugging method that guides the user through large counterexamples. The two methods were applied on many case studies, and showed good results in term of quality and execution time.

Chapter 5

Analysing Probabilistic Counterexamples using Regression

5.1 Introduction

This chapter introduces another approach for the diagnosis of probabilistic counterexamples. This approach adopts the same definition of causality by Halpern and Pearl [104] to reason formally about the causes, and then transforms the causality model into regression model using Structural Equation Modelling (SEM). SEM is a comprehensive analytical method used for testing and estimating causal relationships between variables embedded in theoretical causal model [160]. Regression analysis, path analysis and factor analysis are all special cases of SEM. The regression model generated will quantify the effect of each cause on the violation of the PCTL/CSL formula.

5.2 Regression Analysis

Regression analysis is a statistical technique enables us to reason about the relationship between variables. It describes the change in the value of a variable Y, namely dependent or endogenous in response to the variation of a variable X namely independent or exogenous. The relationship between these variables is either positive or negative. If the value of Y increases when the value of X rises, it is said that the relationship is positive. Conversely, the relationship is said to be negative. One of the relationships that we can infer and statistically reason about using regression analysis is causality. The causal model is build based on assumptions or hypothesis, and then is tested against statistical data to determine how the specified model fits the data. So when it comes to causal relationships, we can use regression analysis to estimate the causal effect instead of exploring the causal relationships. The causal relationship between dependent variable Y and independent variable X with the consideration of factors noise called disturbance term or error term and denoted ε is given by the linear equation

$$Y = \beta X + \varepsilon \tag{5.1}$$

Where X stands as a cause for Y,β stands as a coefficient or weight that quantifies the direct causal effect of X on Y. ε is an error term stands for all extraneous variables, which are assumed to be uncorrelated with X. Whereas the variables X and Y are perfectly known, the objective behind using regression analysis is to produce an estimate of the two parameters β and ε.

5.3 Diagnostic Model

Definition 5.3.1. (Cause) Consider a counterexample $MIPCX(s_0 \models \phi)$ for a probabilistic formula $\phi = \mathbf{P}_{\leq p}(\varphi)$ and a variable $X \in V$. We say that $(X = x)$ is a cause for the violation of $\phi = \mathbf{P}_{\leq p}(\varphi)$ in s, if $(X = x)$ is critical , or there exists a subset of variables W of V in s such that switching the values$\overrightarrow{w}$ of $\overrightarrow{W}$ in s makes $(X = x)$ critical.

Thus, in this setting, we can refer to a cause as $(X = x)$ where X is an atomic proposition has the value 1 in s if $X \in L(s)$, and it has the value 0 otherwise. $(X = x)$ is said to be cause in state s, if it is critical, or it can be made critical by switching the values of set of variables W in s.

Definition 5.3.2. (Diagnostic causality model) A Diagnostic causality model for $MIPCX(s_0 \models \phi)$ is a tuple $\langle M, CC^{\sigma} \rangle$, where M is a causality model, and CC^{σ} is a contribution function that assigns to each cause its contribution to the satisfaction of φ with respect to a path $\sigma \in MIPCX(s_0 \models \phi)$ as follows

$$CC^{\sigma}(X = x) = \sum_{s \in \sigma | f_X(s) = x} P(s) \tag{5.2}$$

That is, with respect to each path $\sigma \in MIPCX(s_0 \models \phi)$, we have associated to each cause a measure that represents the sum of probabilities of reaching the states in which $X = x$. We can say that $X = x$ is a cause for the satisfaction of φ with respect to a path σ with a contribution CC^{σ}. So, the cause having the highest contribution will be the one found

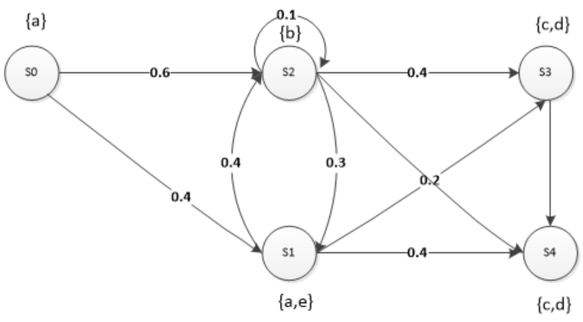


Fig. 5.1 A DTMC

the most along the path.
Example. Consider the most indicative counterexample

$$CX_3 = \{s_0s_2s_3, s_0s_1s_4, s_0s_2s_4, s_0s_1s_3, s_0s_2s_1s_4, s_0s_1s_2s_3\}$$

generated from the DTMC presented in Figure 5.1 against the property $P_{\leq 0.7}(\varphi)$, where $\varphi = (a \vee b)U(c \wedge d)$. It is possible to define a causality model for CX_3, where $u \in \{s_0, s_1, s_2, s_3, s_4, s_5\}$, and F can be defined over the variables in V as follows

$$f_a(s_1) = 1$$
$$f_{c \wedge d}(s_1, c = 0, d = 0) = 0$$
$$\dots$$

For instance, it is clear that in s_2, the actual cause for the satisfaction of $\varphi = (a \vee b)U(c \wedge d)$ is $b = 1$. The probability of the actual cause $b = 1$ in the path $\sigma_2 = s_0s_2s_3$ is $CC^{\sigma_2}(b = 1) = 0.6$.

5.4 Algorithm for Generating dataset

We aim now quantify the effect of a cause $X = x$ on the violation of ϕ. To do so, we use the regression analysis as a comprehensive technique for estimating this effect, where the path formula φ will stand as a dependent variable, whereas the formula of the form $X = x$ will stand as an independent variable. So, the regression model will describe the behaviour of the system by analysing the change of the probability of φ satisfaction with respect to variables that are considered to be causes.

While the variables are now well defined, the remaining task is to generate the data required for estimating the structural parameters. With respect to the definition 5.3.2 of diagnostic causality model , we have seen that each cause $X = x$ has a contribution $CC^{\sigma}(X =$

x) with respect to each path $\sigma \in MIPCX(s_0 \models \phi)$. So in our setting, $P(\sigma)$ will stand for the value of φ and $CC^{\sigma}(X = x)$ will stand for the value of $X = x$. As a result, the number of data rows will be exactly the number of finite paths forming the $MIPCX(s_0 \models \phi)$. We present an algorithm for generating the causes and their contributions, hence constructing the data set.

This algorithm performs on counterexample generated by the tool DiPro. The algorithm gets from DiPro tool the counterexample $MIPCX(s_0 \models \phi)$ and the probabilistic formula $\phi = \mathbf{P}_{\leqslant p}(\varphi)$ as input, and outputs the dataset.

The algorithm explores the counterexample path by path, and computes the causes that will act as data variables, and computes their probabilities that will act as data values. We notice that these two tasks are performed simultaneously. We compute $CC^{\sigma}(X = x)$ of each cause $X = x$ found in set of states. As with the previous algorithms, The main function is FindCauses, which is responsible doe evaluating every sub-formula, it takes a state and state formula as input and returns recursively a set of causes.

5.5 Illustrative Example

We modelled the DTMC presented before in Figure 5.1 in Prism. Then we generate most indicative counterexample using DiPro, which in turns uses prism for the property $\mathbf{P}_{\leq 0.7}(\varphi)$. The counterexample generated is $CX3$ where

$$P(CX_3) = P(\{s_0s_2s_3, s_0s_1s_4, s_0s_2s_4, s_0s_1s_3, s_0s_2s_1s_4, s_0s_1s_2s_3\})$$
$$= 0.24 + 0.16 + 0.12 + 0.08 + 0.072 + 0.064$$

We pass the counterexample to be analyzed to our algorithm, which is implemented in Java. It takes the counterexample generated by DiPro in XML format and returns the dataset. The set of causes as can be generated for this counterexample using our algorithm is as follows:$C_1 = a, C_2 = b, C_3 = \{c, d\}$

These causes denoted C_i stand as independent variables (X) for the regression model we are going to build, where the until formula φ denoted UF stands for the dependent variable (Y). The data values of these observed variables are given in the table 5.1.

As we see in the Table 5.1, the number of data rows represents the number of paths of $MIPCX(s_0 \models \phi)$, where the value of until formula UF refers to the probability of each path $P(\sigma)$ and the value of cause refers to $CC^{\sigma}(X = x)$.

Algorithm 3 Compute Dataset

```
**Inputs:** T̄he probabilistic formula $\phi = P_{\leq p}(\varphi)$ The most indicative counterexample $MIPCX(s_0 \models \phi)$
**Outputs:** DataSet: Variables (causes) with their values ($CC^{\sigma}$)
Causes **:=** $\emptyset$
**for each** path $\sigma \in MIPCX(s_0 \models \phi)$ **do**
    **for each** state $s \in \sigma$ **do**
        **if** $s$ is the last state in a path $\sigma$ **then**
            Causes**:=** FindCauses($s, \phi_2$)
            $CC^{\sigma}(Causes) = \sum_{s \in \sigma | f_X(s) = x} P(s)$
        **else**
            Causes**:=** FindCauses($s, \phi_1$)
            $CC^{\sigma}(Causes) = \sum_{s \in \sigma | f_X(s) = x} P(s)$
        **end if**
    **end for**
**end for**
**function** FINDCAUSES($s$, $\psi$)
    **if** $\psi$ is of the form $a$ where $a \in AP$ and $a \in L(s)$ **then**
        **return** $a$
    **end if**
    **if** $\psi$ is of the form $\neg a$ where $a \in AP$and $a \notin L(s)$ **then**
        **return** $\neg a$
    **end if**
    **if** $\psi$ is of the form $\psi_1 \wedge \psi_2$ **then**
        **return** FindCauses($s$, $\psi_1, W$) $\cup$
        FindCauses($s$, $\psi_2$)
    **end if**
    **if** $\psi$ is of the form $\psi_1 \vee \psi_2$ **then**
        **if** $s \models \psi_1$ and $s \models \psi_2$ **then**
            **return** FindCauses($s$, $\psi_1$) $\cup$
            FindCauses($s$, $\psi_2$)
            **if** $s \models \psi_1 \wedge s \not\models \psi_2$ **then**
                **return** FindCauses($s$, $\psi_1$)
            **end if**
            **if** $s \not\models \psi_1 \wedge s \models \psi_2$ **then**
                **return** FindCauses($s$, $\psi_2$)
            **end if**
        **end if**
    **end if**
**end function**
```

Table 5.1 Counterexample data set

UF	C1	C2	C3
0.24	1	0.6	0.24
0.16	1.04	0	0.16
0.12	1	0.6	0.12
0.08	1.04	0	0.08
0.072	1.03	0.6	0.072
0.064	1.04	0.4	0.064

5.6 Experimental Results

We have implemented the above method in Java. To evaluate our method, we use two benchmark case studies, the embedded control systems taken from [9] and the cyclic polling model taken from [6]. All the experiments were carried out on windows XP with Intel Pentium CPU 3.2 GHz speed And 512 mb of memory.

5.6.1 Embedded Control System

We should recall here that the system consists of input processor (I) that reads incoming data from three sensors (s1,s2,s3) and then passes it to main processor (M). The processor M processes the data and sends instructions to an output processor (O) that controls two actuators (A1 and A2) using these instructions. Any of the system's components M, I/O, the sensors and the actuators may fail; as a result the system is shut down. The types of failure are:

$fail\_sensors = (i = 2 \wedge s < MIN\_SENSORS)$
$fail\_actuators = (o = 2 \wedge a < MIN\_ACTUATORS)$
$fail\_io = (count = MAX\_COUNT + 1)$
$fail\_main = (m = 0)$

We use the variable Max_Count to refer to the maximum number of consecutive cycles skipped allowed. Thus, the I/O processor will fail if the count exceeds the limit Max_Count. The down status of the system is labelled as:

$$down = fail\_sensors|fail\_actuators|fail\_io|fail\_main$$

That is, the systems is down if any of this failures occurs. For this model, we choose the property that estimates the probability of I/O failure occurring first, which is given as follows:

$$P = ?[\neg(down) U fail\_io]$$

We test this property using prism for ($Max\_Count = 6$). For this value, prism renders a probability equal to 0.11. We chose the value 0.1 as a threshold for this property to generate the counterexample. Thus the property can be rewritten as follows:

$$P \leq 0.1[\neg(down) U fail\_io]$$

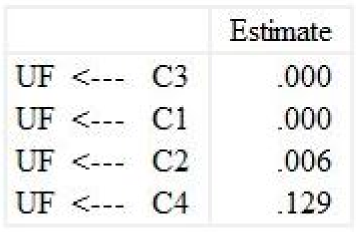
We use DiPro to generate the counterexample. We used the heuristic search algorithm XBF that computes the counterexample as a diagnostic sub-graph. Our method takes the counterexample generated from DiPro in XML format and the property to be verified , and outputs the causes and their values as a dataset in Excel file. This excel file is passed to the tool AMOS in order to generate the regression model. AMOS [1] is well-known software for SEM that enables user to specify, confirm and refine models, by incorporating many statistical methods.

The prism model consists of 6858 states and 28907 transitions. For generating the counterexample, DiPro Explored 480 traces in about 5 minutes resulting in 1685 vertices and 3291 edges. Finally, the counterexample rendered consists just of 33 diagnostic paths. It is evident that the number of explored vertices and explored edges while searching the counterexample is always less than the number of states and the transitions of the model. It is also evident that the number of diagnostic paths is less than the number of solution traces. While solution traces refer to all the paths of the diagnostic sub-graph found through exploring the model, diagnostic paths refer just to the paths forming the counterexample.

We pass this counterexample to our algorithm for generating the dataset of causes. The causes will be the basic sub-formulas causing the satisfaction of $\neg(down)$ $U fail\_io$. For the right sub-formula, the cause generated is $C0 = (count = MAX$ $\_COUNT + 1)$. For the left sub-formula, the set of causes is

$C1 = \neg(i = 2), C2 = \neg(s < MIN\_SENSORS)$
$C3 = \neg(o = 2), C4 = \neg(a < MIN\_ACTUATORS)$
$C5 = \neg(count = MAX\_COUNT + 1)$
$C6 = \neg(m = 0)$

After generating these causes with their probabilities with respect to each path as a dataset, we found the following results: Along all paths (data rows), C0 has the same probability. C2, C5 and C6 have exactly equivalent probabilities with respect to each path (data row). This means that they always occur together. As a result, before passing the data set to AMOS tool, C0 will be ignored since its value is constant along all paths.C2, C5 and C6



	Estimate
UF <--- C3	.000
UF <--- C1	.000
UF <--- C2	.006
UF <--- C4	.129

Fig. 5.2 Regression weights

will be regrouped into one variable named (C2), since they share the same values. Thus, the final data set will consist of the causes C1, C2, C3, C4 and the ‘until‘ formula φ denoted UF. For estimating the causal effect of each cause on UF, we generate the regression model based on this data set using AMOS. The results as rendered by AMOS tool are presented in Figure 5.2.

What concerns us more, is how to get the diagnostic information by interpreting these weights or coefficients. We should recall that C1(input processor is not in OK state),C2(all sensors are working) are the probable causes for not sensor failure occurring, whereas C3(output processor is not in OK state),C4 (all actuators are working) are the causes for not actuators failure occurring. Here we are facing a disjunctive scenario for both failures. In fact, the weights presented above have more importance when we face a disjunctive scenario $(\psi_1 \vee \psi_2)$, because the designer will need to know which sub-formula is more responsible for the satisfaction of φ along the paths.

The results as generated by AMOS tool shows that C4 has the highest effect where C2 has more effect than C1 and C4 has more effect than C3.By fast check on the dataset generated, these explanations are confirmed, which means that along all the paths, C2 and C4 are usually the actual causes not C1 and C3, with great notable presence for C4 comparing to C2. These results are well conform with the previous results, that showed that the causes C2 and C4 has more responsibility for not sensors and actuators failures occurring than C1 and C3 respectively, and C4 is the most responsible cause.

Taking just $MIPCX(s_0 \models \phi)$ to be the causality model instead of the whole model is due first to the nature of $MIPCX(s_0 \models \phi)$ itself; $MIPCX(s_0 \models \phi)$ by definition covers the most probable causes for the error, since it is consisting of paths with high probabilities. Second, lowest probability values will have no effect on the regression model generated since they tend to be zero. We tested this issue, by generating a counterexample for the property $P = ?[\neg(down)U fail\_io]$.

That is, the counterexample will consist of all paths satisfying $\neg(down)U$ $fail\_io$, not just the paths with high probabilities. The counterexample generated consists of 132 paths, DipRo takes more than 30 minutes for computing this counterexample. Adding the new paths (99 paths) to the old data set and analysing it using AMOS didn't bring considerable change to the previous regression model generated.

5.6.2 Polling Server System

We should recall here that the system is modelled in PRISM as a CTMC, where the number of stations handled by the polling server is denoted by N. Each station has a single-message buffer and is cyclically attended by the server. We choose the property that measures the probability of station 1 being served (s=1) before station 2 (s=2) where (a=1) denotes serving. This property is given as follows:

$$P =?[!(s = 2 \wedge a = 1)U(s = 1 \wedge a = 1)]$$

We test this property using PRISM for (N=3, N=5, N=7 and N=9). For all of these values, PRISM renders a probability higher than 0.5. As a result, we chose the value 0.5 as a threshold. The property can be rewritten as follows:

$$P \leq 0.5[(!(s = 2) \vee !(a = 1))U(s = 1 \wedge a = 1)]$$

The prism model consists of 1344 states and 5824 transitions. For generating the counterexample, DiPro Explored 184 traces in about 5 minutes resulting in 1094 vertices and 3310 edges. Finally, the counterexample rendered consists just of 18 diagnostic paths. It is evident that the number of explored vertices and explored edges while searching the counterexample is always less than the number of states and the transitions of the model.

We pass this counterexample to our algorithm for generating the dataset of causes. The causes will be the basic sub-formulas causing the satisfaction of $(!(s = 2) \vee !(a = 1))U(s = 1 \wedge a = 1)$. For the right sub-formula, the cause generated is $C0 = (s = 1 \wedge a = 1)$. For the left sub-formula, the set of causes is $C1 =!(a = 1)$ and $C2 =!(s = 2)$.

After generating these causes with their contribution with respect to each path as a dataset, we found that $C0$ has the same probability along all paths (data rows). thus before passing the data set to AMOS tool, C0 will be ignored since its value is constant along all paths. The effect of $C1$ and $C2$ on UF is as generated by AMOS is given by the following equation:

$$UF = -(0.304)C1 + (0.464)C2 \tag{5.3}$$

The results as rendered by AMOS show that C2 has higher effect than C1, where the effect of C1 is negative. This means that the effect of C1 on UF increases once the paths probabilities get lower, which means that the presence of C1 increases just with in the paths with low probabilities. This is also well conform with the previous results that showed that the most responsible cause is C2, which means that the server is not polling station 2 at all.

5.6.3 Comparison with Previous Method

Comparing to the two methods presented in the previous chapter, the method presented in this chapter considers the causes as variables that have different values along the paths of the counterexample, accordingly it studies their contribution to the violation by computing the presence of each cause with respect to each path. We can say that the two first methods are more effective for debugging, because they can guide the user directly to the source of the error through getting all the information needed (Variable, state, transition, action,...), whereas for the second method it just delivers a statistical information about the variables involved in the violation, which could not indicate the source of the error directly, but it could inform us about the behaviour of the model. For instance, finding dependence between two variables means that the values of these variables are changing together. Besides, Statistical information could carry imprecise results. As a conclusion, The two first methods are more useful and appropriate for debugging of probabilistic models, but they could benefit from the later one, for measuring the effect of variables.

5.7 Conclusion

In this chapter we proposed the use of regression analysis for error explanation by generating a regression model corresponding to the causality model. For doing so, we have proposed an algorithm for generating the causes as data set. We have seen that delivering the causes for the violation of PCTL/CSL formula as a regression model stands as a good technique for describing the effect of variables with their values on the error. Besides, we found that the experimental results of this method on the two case studies are well conform with the experimental results of the previous one. Hence, we conclude that both of the methods could be used in complementary way.

Chapter 6

Analysing Probabilistic Systems using Probabilistic Model Checking

6.1 Introduction

Probabilistic model checking has appeared as an extension of model checking for the verification of quantitative properties of stochastic systems. While this can be considered as the main aim behind using probabilistic model checking, in recent years we notice also a great attend to use probabilistic model checkers especially the probabilistic model checker PRISM for the estimation of quantitative measures that help us to understand and analyse the performance of such system. For instance in biology, many works have used probabilistic model checking for modelling and analysing complex dynamic phenomena, from biological pathways [110] and bone pathologies [35] to Codon bias [148]. In all the case studies investigated, the modelling principle is based on modelling the evolution of individual molecules, whose rates of interaction are controlled by exponential distributions [137]. Other interesting direction for using probabilistic model checking is cloud computing environments. Johnson et al. [126] delivered probabilistic pattern modelling tool for transforming such cost and resource usage queries into probabilistic properties to be analysed by probabilistic model checkers like PRISM. This approach helps the customer of cloud services to get better insight on the cost and the resources usage, since both of them are so dynamic and could vary over time in stochastic manner. Other work has used probabilistic modelling for analysing live migration of virtual machines between physical servers in a cloud data centre [131].

With the growing importance of probabilistic model checking for the verification and quantitative analysis of probabilistic systems, in this chapter we investigate its applicability to two different domains. In the first section we propose a verification approach for *Probabilistic Complex Event Processing (Probabilistic CEP)* . CEP is an Event Driven Architecture (EDA) style consists of processing different events within the distributed enterprise system or externally attempting to discover interesting information from multiple streams of events in timely manner. In real world, the streams of events are uncertain, which means that is not guaranteed that an event has actually occurred , this uncertainty is due mainly to imprecise content from the event sources (Sensors, RFID,...). As a result, Probabilistic CEP has become an important issue in complex environments that requires a real-time reaction given streams of probabilistic events.

In the second section, we will show how probabilistic model checking can serve as a comprehensive technique for modelling and analysing medical treatment problems. Physicians and patients are always facing critical situations where they have alternative actions, and they have to choose the appropriate one in order to get the best outcome. So, Medical treatment decision is a highly complex process that involves health states, preferences, the offered options (actions) and the corresponding cost. The complexity of this process is due to the multiple treatment decision and the accompanying risk factors, as well as that these decisions are made under uncertainty.

6.2 Analysing Probabilistic Complex Event Processing (CEP) Applications

Complex Event Processing (CEP) is defined by its founder Luckham as a set of tools and techniques for analysing and controlling the complex series of interrelated events that drive modern distributed information systems [67]. CEP is a style of Event Driven Architecture (EDA) that refers to generation, reaction, detection and consumption of events that represent notable changes in the state of enterprise's activities. CEP applications are based on decoupling principle, which means that the events are sent or received over publish/subscribe bus, where the event providers and event consumers are independent components.

The CEP Architecture consists of two main components, event processing engine and adapters. The event processing engine is the core of CEP architecture; it can processes and analyzes thousands of upcoming events from different external sources with low latency. By correlating these events across multiple streams, the engine generates complex events to be delivered to different destinations. Thus, the complex or high level event is generated by aggregating set of events, these events are called the members and represent basic activities in the system. The engine is based on event processing language that manipulates event data in real-time. The adapter is a software layer that interacts with stream of events enabling the CEP engine to take place in complex environments. Regarding to event sources and destinations, we distinguish two types of adapters, input and output adapters respectively.

CEP is considered as a promising complementary technique for many existing techniques. CEP was proposed to support Business Process Management (BPM), where business processes are continuously reacting with simple or complex events, CEP can give the ability to discover patterns within the events cloud providing the business manager by interesting information [29]. CEP was also proposed as a support Business Activity Monitoring (BAM) to take place in complex monitoring environments [159]. CEP can also Business Intelligence (BI) tools to extract information from continuous events not just from historical data, thus enabling real-time intelligence [138]. It has been shown that CEP can play a crucial role in many domains, sensor and RFID networks [184], security [75],health care systems [182].

Based on the academic research conducted at Cambridge University from the early 1990s onwards, many CEP vendors have arisen, although this was so late, just after the beginning of the new century. streambase , Apama , Aptsoft and Coral8 were the first vendors, CEP market was reinforced by the acquisition of Aptsoft by IBM, the acquisition of Apama by Progress software, the acquisition of Sybase by SAP, and offering Oracle and Microsoft their products, ORACLE-CEP [16] and Microsoft-Insight respectively [14].

In real world the streams of events are uncertain, this uncertainty is due mainly to imprecise content from the event sources (Sensors, RFID,...). As a result, probabilistic CEP has become an important issue to deal with in complex and uncertain environments. There are two main challenges for implementing probabilistic CEP applications, the first concerns the huge number of events incoming from different sources that we should to deal with in real-time constraints, and the second concerns the assignment of probabilities measures to complex events aggregated from probabilistic basic events. Probabilistic CEP has attracted recently a great attention. Chuanfei et al. [49] proposed an infrastructure for event detection and triggering with noisily input-data from sensors and delivered a probabilistic inference system through using Bayesian networks. Li and Ge [141] have studied the problem of windowed sub-sequences from probabilistic sequences of events, providing optimization algorithms that perform in real time. Mainly focusing on RFID networks, Re et al. [154] proposed set of algorithms and probabilistic processing engine Lahar that acts on probabilistic RFID events. The authors in [177] proposed a model for representing materialized events with baysien and sampling algorithm for correctly specifying the probabilities of complex events from events history. The issue of uncertain data Processing is not exclusive for CEP, but it has its origins in databases community [68].

There was before an attempt to analyse CEP applications using formal methods. The authors in [87] introduced an approach based on transforming CEP rules into timed automata to be verified using timed model checker UPPAAL through a proposed tool REX. A formal verification approach was also proposed among other approaches proposed by [150] for analysing CEP applications. They used Discreet Transition System (DS) as a verification model and the Property Specification Language (PSL) sequences for specifying temporal properties. So, it is evident that building probabilistic CEP applications is not a trivial task, which makes the description of these applications and the analysis of their behavior a necessary task. In this section, we propose a formal verification approach for probabilistic CEP applications based on probabilistic model checking. To this end, we use the Probabilistic Timed Automata (PTA) for describing the probabilistic CEP applications, and the Probabilistic Timed CTL (PTCTL) logic for specifying probabilistic timed properties. While analysing CEP applications has been investigated before and not in depth manner, to our knowledge this is the first attempt to analyse probabilistic CEP.

6.2.1 Preliminaries and Definitions

Clocks and Zones

We denote by $\mathbb{R}$ the domain of time (non-negative reals), and by $\mathbb{N}$ the naturals. Let X be a set of finite variables called clocks which take values from R. We denote by $v(x)$ the clock valuation function that assigns a value $v \in \mathbb{R}^X$, where $\mathbb{R}^x$ represents the set of all clock valuations of X. For any $v \in \mathbb{R}_X$ and $t \in \mathbb{R}$, $v+t$ denotes the clock valuation defined as $(v+t)(x) = v(x)+t$ for all $x \in X$.

The set of zones(clock constraints) of X, denoted $Z(X)$ is defined by the syntax

$\zeta ::= x \leq d \mid c \leq x \mid x+c \leq y+d \mid \neg\zeta \mid \zeta \vee \zeta$

where $x,y \in X$ and $c,d \in \mathbb{N}$. We say that a clock valuation v satisfies a zone ζ, denoted $v \triangleright \zeta$ if and only if ζ resolves to true after substituting each clock x with $v(x)$. Other constraints can be easily derived,for example, $x > 1 \equiv \neg(x \leq 2)$ and equality can be written as a conjunction of constraints, for example $x = 2 \equiv (x \geq 2 \wedge x \leq 3)$.

Probabilistic Timed Automata

While the formalism of clocks and zones is the same as of classical timed automata, PTA are extended with discrete probability distributions over edges.

Definition 6.2.1. (Probabilistic Timed Automata (PTA)) A *Probabilistic Timed Automata (PTA)* is a tuple $(S, S_0, X, inv, prob, L)$ where: S is a finite set of locations with s_0 is the initial location. X is a finite set of clocks. $inv : S \longrightarrow Z(x)$ maps to each location an invariant condition. $prob \subseteq S \times Z(X) \times Dis(S \times 2^x)$ is the probabilistic edge relation. L is a labelling function that assigns to each location $s \in S$ set of atomic propositions.

A state of PTA is a pair $(s, v)\ inS \times \mathbb{R}^X$ such that $\triangleright inv(s)$. An edge of PTA is $(s, g, a, p, l', Y))$ where l' is the destination location and and Y is the set of clocks to be reset and (s, g, a, p) is a probabilistic edge of PTA where s is source location, g is a guard, a is an action and p is the destination distribution. In s_0 all clocks are initialized to zero. For any state (l, v), there is a non-deterministic choice between making a discrete transition and letting time pass, the transition is enabled if $v \triangleright g$ and probability to moving to destination location l' resulting in resetting the set Y of clocks equals to $p(s', Y)$. Letting the time passes in the current location s is provided by the invariant condition $inv(l)$, which is continuously satisfied while time passes.

Probabilistic Timed CTL (PTCTL)

The *Probabilistic Timed Computation Tree Logic (PTCTL)* has appeared as an extension of CTL for the specification of probabilistic timed systems. We use the PTCTL for defining quantitative and timing properties of PTAs. As with TCTL, we use a set of clock variables for expressing timing properties, this set is denoted by Z disjoint from X, where $\xi : Z \rightarrow \mathbb{R}$ is a formula clock valuation that assigns values to such clocks. TPCTL state formulas are formed according to the following grammar:

$$\phi ::= true|a|\zeta|z.\phi|\phi_1 \wedge \phi_2|\neg\phi|\mathbf{P}_{\sim p}(\varphi)$$

Where $a \in AP$ is an atomic proposition, ζ is a zone over $X \cup Z$, $z.\phi$ is reset quantifier, φ is a path formula, $\mathbf{P}$ is a probability threshold operator, $\sim \in \{<, \leq, >, \geq\}$ is a comparison operator, and p is a probability threshold. The path formulas φ are formed according to the following grammar:

$$\varphi ::= \phi_1 \mathbf{U} \phi_2 | \phi_1 \mathbf{W} \phi_2 | \phi_1 \mathbf{U}^{\leq n} \phi_2 | \phi_1 \mathbf{W}^{\leq n} \phi_2$$

Where $\phi_1 and \phi_2$ are state formulas and $n \in N$. As in CTL, the temporal operators ($\mathbf{U}$ for strong until, $\mathbf{W}$ for weak (unless) until and their bounded variants) are required to be immediately preceded by the operator $\mathbf{P}$. The PTCTL formula is a state formula, where path formulas only occur inside the operator $\mathbf{P}$. The operator $\mathbf{P}$ can be seen as a quantification

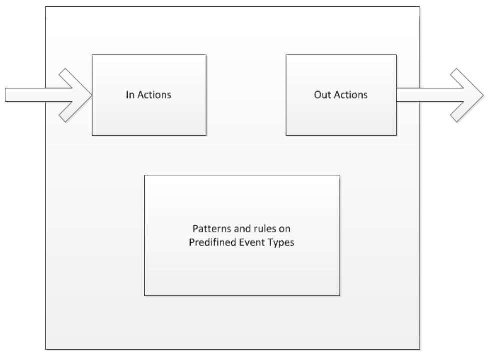


Fig. 6.1 An EPA

operator for both the operators $\forall$ (universal quantification) and $\exists$ (existential quantification), since the properties are representing quantitative requirements.

With TPCTL we can express properties such as, with a probability at least 0.95, the system clock x does not exceed 2, before 5 time units elapse, which is expressed using TPCTL as follows: $P_{\geq 0.95}[(x \leq 2)\mathbf{U}(z = 5)]$.

6.2.2 CEP Verification Approach

Event Processing Agent (EPA) and Event Processing Network(EPN)

Any CEP application regardless of the technique employed for processing events: query-based, rule-based,... can be described using the Event Processing Network (EPN) [150, 165]. EPN is a conceptual model that enables us to build a CEP application in reliable way by describing the event execution flow, from source passing by processing modules to destination. We call the modules responsible for the processing by Event Processing Agents (EPAs). EPAs are simply a set of objects that monitor event execution to detect such patterns. For each pattern found there is a set of actions to be performed. The EPN we use for modelling CEP applications consists of four main components:

event producer: the entity responsible for generating the stream of events

event consumer: the final entity that consumes the outcome of EPAs

EPA : is the component that given set of input events it generates output events to be consumed by applying such logic that must expresses time constraints.

Event Type: represents the event object structure that consists of specified attributes, where the time-stamp is a present attribute in any event type.

The output of an EPA is either consumed by event consumer, or it feeds another EPA. We notice that EPA is the central component in EPN, in way we can say that an EPN is a

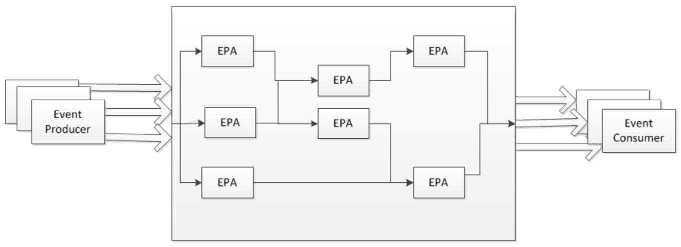


Fig. 6.2 An EPN

set of EPAs communicating between them by exchanging events asynchronously. Roughly speaking, EPA employ such rules that consist of two parts: pattern called the trigger and a set of actions, by executing these rules, output events are generated. In CEP applications, EPAs nature varies according to the engine used (rule-based, query-based,..)[67]. For example, we can express an EPA as an Event Processing Language (EPL) query, the language used by ESPER [8] engine for processing events engine as follows:

[insert into insert_into_def]
select select_list
from stream_def[as name] [, stream_def[as name] [,...]
[where search_conditions]
[group by grouping_expression_list]
[having grouping_search_conditions]
[output output_specification]
[order by order_by_expression_list]
[limit num_rows]

Using EPAs we can perform three functions that represent the main features of CEP applications:

1) Filtering: In complex and dynamic systems, we can capture thousands of events and we have to filter them to get only the interesting events in critical time. This feature is very useful in detection and alerting systems such as fraud detection and intrusion detection.

2) Aggregation and abstraction hierachy: We say that an event is complex or high level event, if it is generated by aggregating set of events. These events are called the members and represent basic activities in the system. For example, by using the timestamp of each event we can create a complex event that match specific pattern such as Event A and B occur in either order followed by event C or D. We say that an event has an abstraction hierarchy if it consists of sequence levels of activities and aggregations patterns in each level, where each activity is signified by a specific event.

3) Causality: According to D. Luckham, the father of CEP [67], the Causality represents a dependence relationship between activities in the system, “If the activity signified by the event A had to happen in order for the activity signified by event B to happen, then A caused B”. With CEP technology, we can consider an event as a piece of data having two essential parameters, which are the timestamp and the causal vector. The timestamp indicates when the event has happened, whereas the causal vector contains the identifiers of set of events that have caused this event. Placing the causal vector in the event facilitates the tracking of causality in complex systems.

The verification Approach

In real world the streams of events are uncertain, this uncertainty is due mainly to imprecise content from the event sources (Sensors, RFID,...). As a result, the EPAs will not employ just time constraints but also probabilistic thresholds. The major challenge for employing probabilities is how EPAs derive and output complex events aggregated from probabilistic basic events. This was the main subject of many works. Therefore, we do not consider here this subject, but rather it reveals the way of analysing probabilistic CEP application independently of any proposed technique in the literature.

The analysis of probabilistic CEP applications relies on the analysis of how EPAs behave as it is intended. Therefore, we need to model the EPAs and their interaction using modelling tool capable of dealing with time and probability and verified against properties specified using a logic that expresses time and probability. In our approach, we use Probabilistic timed automata (PTA) for modelling EPAs and their interaction and the PTCTL as specification logic for specifying probabilistic real-time properties.

1) Modelling Phase

In our approach, we argue that probabilistic timed automata is the best candidate for verifying CEP applications under uncertainty. To this end we have to show the correspondence between EPN that decribes the event execution flow and PTA. Using PTA, we model the EPAs as locations where the initial location is represented by an EPA that acts on events generated from event producers and the final location is represented by an EPA that delivers final events to event consumers. We associate to each location an invariant that represents in CEP the time window defined by the EPA for such type of events to be kept. Each out action from an EPA is enabled with respect to a guard which is specified by the EPA source itself, where each action leads to another EPAs with new event types. That is, a probabilistic edges from an EPA source will lead to possible EPAs with respect to the probabilistic output events types. By moving to new EPAs, previous clocks could be reset.

2) Specification Phase

After modelling probabilistic CEP application as a PTA, we can use TPCTL logic to specify temporal and probabilistic properties, to verify if the model meets the specification. The type of properties that can be specified for probabilisic CEP are not much different from the standard types defined by [136]. We present adapt the four standard types of properties to probabilisic CEP applications as follows:

Reachability: The application can produce an output event with a given probability. For example, “with probability 0.9999 or greater, a deliver alert is received by a client ”.

$$P_{\geq 0.99}[true\mathbf{U}AllertDelivered]$$

Time bounded reachability: The application can produce an output event within a certain time deadline with a given probability. For example, “with probability 0.975 or greater, a deliver alert is received by a client within 5 time units ”.

$$P_{\geq 0.99}[true\mathbf{U}(AllertDelivered \wedge (z < 5))]$$

Invariance: Certain type of events are not produced with a given probability. For example, “ with probability 0.75 or greater, error event is never generated.

$P_{\geq 0.75}[true\mathbf{U}\neg error]$

We can also specify such safety propeties using PTCTL for complex events. For example, eventC must not appear until eventA and eventB have occured with probability 0.9 or greater.

$P_{\geq 0.9}[\neg EventC\mathbf{U}EventA \wedge EventB]$

6.3 Medical Treatment Analysis

Controlling risk factors over the course of patient’s lifetime is important for preventing some common chronic diseases and improving life expectancy. Whereas treatment decisions are one time decisions, in such cases, diseases involve multiple treatment decisions. For example, patients with diabetes must carefully weigh the costs and benefits associated with treatment of multiple risk factors including blood sugar, blood pressure, and cholesterol control [77].

In addition to the complexity of treatments decisions, this complexity is compounded by uncertainty of the treatment effects. As a result, it is very difficult to take all the possibilities and choose among the offered options, the appropriate one. Therefore, we need a comprehensive framework in which we can investigate all the actions and compare the effect of each of them, and here is where decision analysis methods come.

Many mathematical and computational models and their related tools have been used to improve *medical decision* process. Whereas computational models such as decision trees and artificial neural networks have been widely used in medical and biomedical domains, the authors [147] sought to use a combination of them for the diagnosis of Neuromuscular Diseases (NMDs). Other work has proposed an approach for medical decision support based on the notion of knowledge integration through analysing clinico-genomic data [161] and another proposed a hybrid framework that merges between graph b-coloring and Markov chain models for clustering clinical pathways [82]. Markov decision processes (MDPs) are one of these methods that know a great success as an appropriate method in medical treatment decision.

Markov decision processes (MDPs) provide a mathematical framework for modelling dynamic systems under uncertainty. The goal of an MDP is to provide an optimal policy, which is a sequence of decision rules to optimize a particular criterion, such as maximizing a total discounted reward. Based on multiple actions and rewards, a decision maker can get the consequences of all policies and the appropriate one between them. To decide on the optimal policy, varied solution algorithms are used, and defer according to the finite or infinite horizon.

One of the major advantages of MDP is its flexibility, which means that at each time epoch, there are multiple possible choices. This feature is very important for modeling the patient's health states. For example, organ transplantation can be modelled as an MDP, in which the action is to either accept the organ or reject it, once a donor organ becomes available [156]. While the patient seeks to maximize life expectancy, the health states are assigned with such values known as quality-adjusted life years" (QALYs) [98].

With the growing importance of medical decision analysis using MDPs, we need to invent software solutions for solving, visualizing and cost-effective analyzing of MDPs. Although there are existing tools for modelling and cost effective analysis of MDPs, such as TreeAge, according to [78], most of medical MDPs have been built with general-purpose languages, such as C++, Matlab, or Java. In this aim, [78] developed an open source software tool, OpenMarkov that can be used to build and evaluate MDPs in addition to Bayesian networks and influence diagrams.

Due to the growing importance of software in the healthcare sector in general, the authors [145] introduced a good survey on the computer programs used in healthcare and their effect in the United Kingdom. In this section, we show how probabilistic model checking serves as a formal and logical framework for modelling and analysing medical treatment problems specified in MDPs. We will show the effectiveness of our approach for modelling and especially cost-effective analysis of MDPs through showing a case study. This case study concerns The Optimal Timing of living donor liver transplantation. We refer to the MDP introduced by [22, 23] for formulating the problem. We use PRISM for the specification and the analysis of the model.

6.3.1 MDPs for Medical Treatment Decision

For simple medical treatment decisions, the decision tree [77] could be used by medical decision maker for representing expected utility of a patient whose health progression follows that branch of the tree. In such cases, the path to a terminal node, such as dead could be very complex involving much health states. This requires a large number of nodes, thus resulting in a tree explosion [156] . Therefore, MDPs have been proved to be the appropriate alternative for dealing with medical treatment problems that involve complex, stochastic and dynamic decisions.

In standard MDPs, at each decision epoch, the sates are completely observable. However, in real world problems there are systems where the states are not entirely known or are partially observable. For dealing with this case, the partially observed Markov decision processes (POMDPs), have been proposed as an extension of the standard MDPs. The expressiveness of the POMDP over standard MDP makes it suitable for such medical treatment decisions. The POMDP model distinguishes between states defining the dynamics of the system (e.g. disease states) and observations, and thus, states can also be hidden and unobservable. In addition, it allows handling investigative actions. That is, information gathering actions or actions enabling observations (e.g. a biopsy procedure allows us to see biopsy results) [168] . Both, the standard MDPs and POMDPs have been widely used in medical treatment decision.

[162] addressed the problem of Optimal Time to Initiate HIV therapy. The decision whether to intervene and initiate therapy or delay it, is still a difficult decision, because delaying therapy from a side could bring such benefits, such as avoiding the negative side effects and toxicities associated with the drugs. On the other hand, delaying therapy brings such risks, including the possibility of irreversible damage to the immune system, development of AIDS related complications and death. The authors addressed this issue through formulating MDP model. Where states are represented by patient's health states and the

actions taken are: initiate or delay the therapy.

Similar to this work, [76] proposed an MDP model to optimize the selection of patients for statin therapy of hypercholesterolemia, for patients with Type 2 diabetes, using each of these risk models. In this model, the patient can initiate the stain therapy or delay it. If the patient chooses to initiate therapy, then the discounted value of expected rewards of all future quality adjusted life years are obtained. On the other hand, if the patient delays the decision then the patient moves to a new metabolic state in the next period with certain probability and reward.

The authors [187] proposed a POMDP model for formulating the PSA-based screening for prostate cancer problem that trades off the benefit of early detection with the cost of screening and loss of patient quality of life, because such imperfect sensitivity of PSA tests can result in harm to patients. This model consists of unobservable health states: no cancer (NC), organ-confined (OC) cancer detected, extraprostatic (EP) cancer detected, lymph node–positive (LN) cancer detected, metastases (mets) detected, and death from prostate cancer and all other causes (D), and others including a particular PSA interval, cancer detected and treated (T) or death (D). The decisions required at each decision epoch: perform a biopsy (B), defer biopsy and obtain a new PSA test result in epoch t + 1 (DB), or defer the biopsy decision and PSA testing in decision epoch t + 1 (DP). The transitions probabilities range in two kinds: the first denotes the state transition probability from health state st to st + 1 at epoch t given action at. The second denotes the probability of observing PSA state $t' \in M$ given the patient is in health state. Rewards also range in two kinds: the first for the patient's perspective, it is measured in QALYs. The second for the societal perspective, it is measured (in dollars) as the difference in (a) the product of QALYs and a willingness to pay factor and (b) the cost of PSA tests, biopsy, and treatment.

The authors [109] addressed the problem of managing patients with ischemic heart disease (IHD), through using POMDP model. For patients with this disease, physicians must choose among various diagnostic procedures (such as an angiogram or one of many varieties of stress test), which may be followed by a therapeutic intervention such as medication, surgery (such as angioplasty or bypass surgery), or nothing at all. The State variables are either observable or hidden. For example, variables representing status of the coronary artery disease and ischemia level are hidden (not observable directly), while other state variables like chest pain, rest EKG result and stress test result are perfectly observable. The actions correspond to treatment or investigative procedures. Treatment actions actively change the state of the patient to a more appropriate state; Whereas Investigative actions explore the state of the patient, especially the related hidden process state variables. The transition probabilities rang in two kinds: The first concerns the variable status and represents the distribution of a patient being alive as a result of some procedure performed in the previous state. The second represents a conditional distribution of coronary artery disease given a previous state, an action and patient being alive. Concerning the rewards, there the reward associated with a patient state only and the second stands for a cost associated with an action (e.g. cost of performing coronary bypass surgery that includes the economic cost).

6.3.2 Preliminaries and Definitions

Markov Decision process (MDP) and optimal policy

Markov decision processes (MDPs) provide a mathematical framework for modeling dynamic systems under uncertainty. Markov Decision Process is a Discrete-time Markov Chain allowing the nondeterministic choice. At each time step, the process is in some state s, and the decision maker may choose any action a available from state s. The process responds at the next time step by randomly moving into a new state s', and giving a corresponding reward $R_a(s,s')$. The probability that the process moves into its new state is influenced by the chosen action, it is given by the state transition function.

More formally, A Markov Decision Process MDP is a quadruple $(S, s_0, L, step)$ where S is a finite set of states, s_0 is the initial stat, $L : S \rightarrow 2^{AP}$ is a labelling function of atomic propositions, $step : S \rightarrow 2^{(Act \times Dist(s))}$ is a probability transition function, where Act is a set of actions and $Dist(s)$ represents the discrete probability distributions over S . We define a reward structure for MDP as a pair (r_s, s_A) where $r_s : S \rightarrow \mathbb{R}_{\geq 0}$ is a state reward function and $r_a : S \times Act \rightarrow \mathbb{R}_{\geq 0}$ is a transition reward function.

In MDPs, we denote a policy, scheduler or adversary by $\pi = \{\pi_1, \pi_2, ...\}$ where each $\pi_i : S \rightarrow Act$ is a mapping from state to action. The policy π specifies the set of actions the decision maker could choose in each state in order to get the future expected reward. An optimal policy is the policy where the decider chose the actions that maximize the utility. For computing the optimal policy many methods were proposed in the literature, the most known are the Value iteration method [37] and policy iteration [121].

PRISM is a tool used for formal modelling and analyzing systems that exhibit random or probabilistic behavior [115]. It supports several types of probabilistic models such as: Discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markov decision processes (MDPs). The analysis is performed on these models against properties specified in PCTL logic [108] for DTMCs and MDPs, and Continuous Stochastic Logic (CSL) [31, 32] for CTMCs. PRISM uses several numeric methods for model analysis such as Gauss-Seidel method, Backwards Gauss-Seidel method and Jacobi method. For MDPs and CTMCs, PRISM uses value iteration and uniformisation respectively. As additional features, PRISM offers a simulation framework for reasoning about probabilities and rewards.

A model in PRISM consists of one or several modules that interact with each other. The module is specified using PRISM language as set of guarded commands.

$$[< action >] < guard > \rightarrow < updates >$$

Where the guard is a predicate over the variables of the system, the updates describe probabilistic transitions which the module can make if the guard is true. These updates are defined as follow:

$$< prob >:< atomicupdate > + ... + < prob >:< atomicupdate >$$

PRISM also supports rewards which are real values associated with states or transitions of the model. Where State rewards can be specified as: $g : r$. The Transition rewards are represented as: $[a]g : r$. According to the definitions above, the properties for a model can be specified in PCTL using two main operators: the P operator that refers to the probability of event occurring, and the R operator that refers to the expected value of rewards.

The objective of using MDPs is to find the optimal policy. Therefore, MDPs analysis is performed over all the possible resolutions of non-determinism, which means that properties using the P operator and R operator effectively reason about the minimum or maximum probability and the minimum or maximum reward respectively over all possible resolutions of non-determinism.

For example, the properties below compute the maximum and the minimal probability, the maximum and the minimal reward over all resolutions, of reaching an error state:

$$\begin{gathered} Pmax = ?[F"error"] \\ Pmin = ?[F"error"] \\ Rmax = ?[F"error"] \\ Rmin = ?[F"error"] \end{gathered}$$

6.3.3 Case Study

To illustrate and clarify the benefits of using the probabilistic model checking for medical treatment analysis, we investigate a case study. This case study concerns the optimal timing of *living-donor liver transplantation* . We refer to the MDP introduced by [22, 23] for formulating this problem.

According to [114], more than nine million people die due to internal organ failure and one percent of this number is due to liver disease. Organ donation is the donation of an organ of the human body, from a living or dead person to a living recipient in need for a transplantation. Due to the insufficient supply of cadaveric organs and the considerable risks for transplantation, living donors have become an increasing source of livers for transplantation. The Liver transplantation refers to the replacement of a diseased liver with a healthy one. One of the most open questions investigated by researchers in this area is the optimal timing for transplantation in the aim to maximize the quality adjusted life expectancy of the patient.

The authors [22, 23] have addressed this issue by formulating an infinite-horizon MDP model for finding the policy that describes the health states in which the patient should wait, and the states in which the patient should do the transplantation. In this model, the health states of the patient are represented by the Model for End-Stage Liver Disease (MELD) scores. MELD is a chronic liver disease severity scoring system mainly developed for predicting death within three months of surgery in patients who had undergone a transjugular intrahepatic portosystemic shunt (TIPS). In this model, the patient can take two actions, either "transplant" in the current decision epoch and gets expected plant reward, which represents post-transplant life days of the patient, or "wait" for another decision epoch to make the transplantation and get a pre-transplant reward which is equal to 1 day, or he dies. The reward in this model is assigned to actions of transplantation from each particular MELD score, not to transplanted state, since post-transplant life expectancy depends on the pre-transplant MELD score.

As we see in Figure 6.3, the model consists of 18 health sates, each state represents two adjacent MELD scores, beginning with MELD [6-7] until the last state [40]. At each sate the patient can choose the action "transplant" represented by T, or "wait" for one more time period, which is represented by W. By choosing the first action, the patient moves to the state "Transplant" which is an absorbing state with probability 1 and gets the reward that represents the expected post-transplant life days of the patient. By choosing the second action, the patient stays at the same health sate with probability $P(h|h)$, or he progresses to other health sate h' with probability $P(h|h')$, or he may die at the beginning of the next decision epoch with probability $P(D|h)$. The patient will receive a reward of 1 day for each wait action without transplantation.

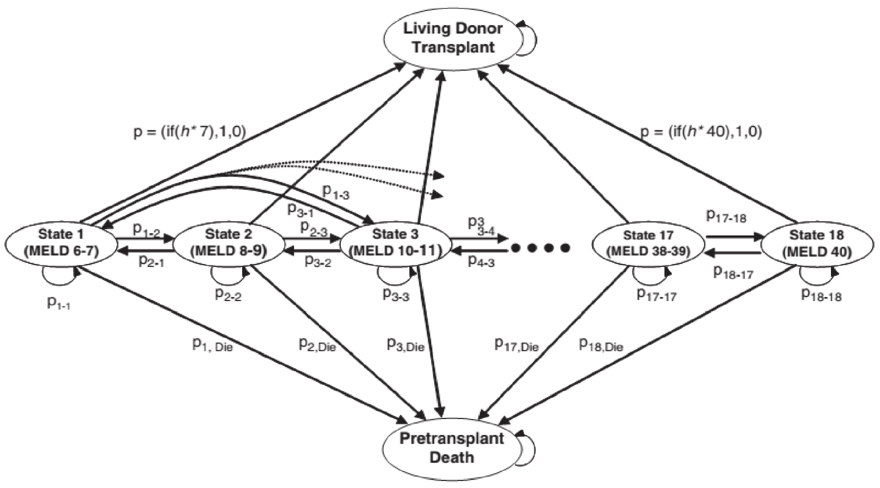
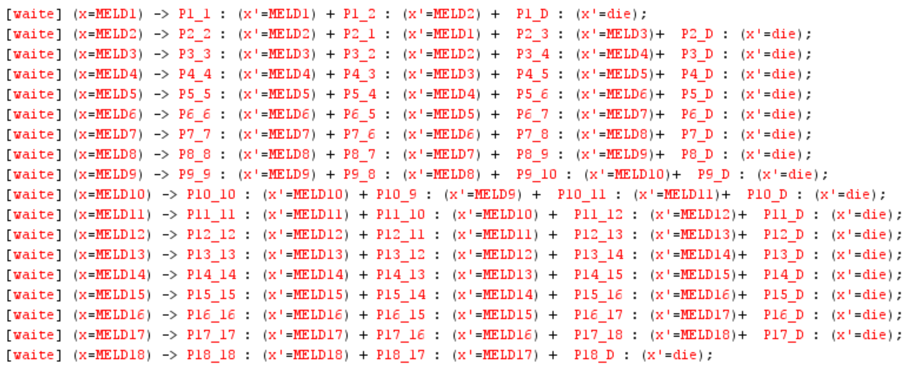


Fig. 6.3 MDP for Living Donor Transplantation [23]



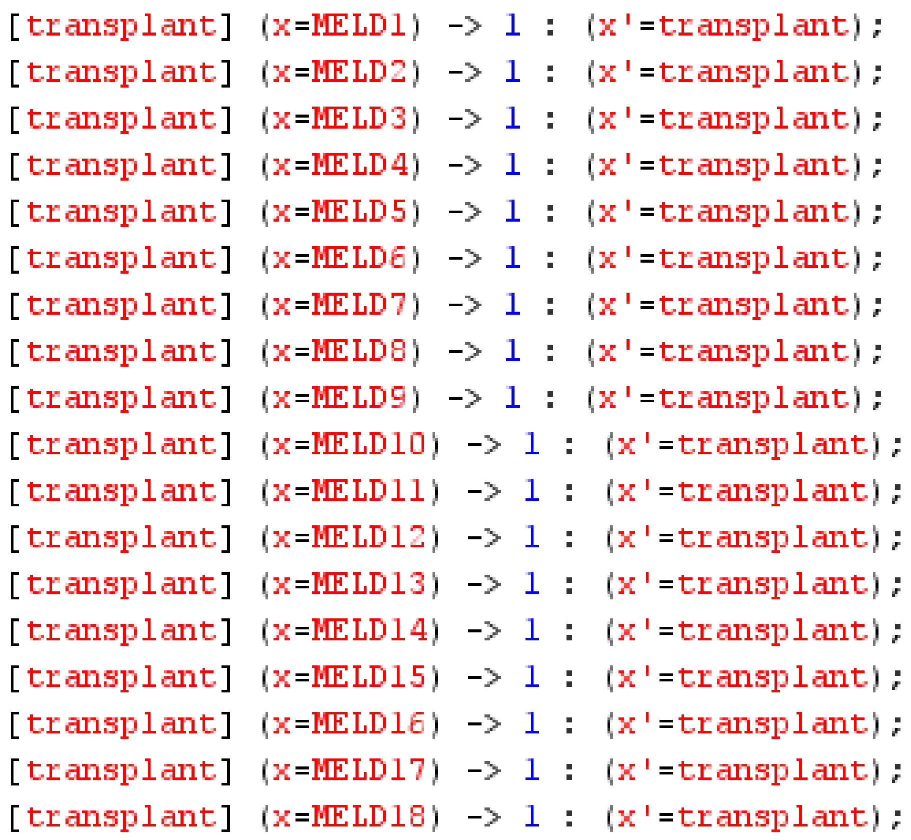
```
[waite] (x=MELD1) -> P1_1 : (x'=MELD1) + P1_2 : (x'=MELD2) +  P1_D : (x'=die);
[waite] (x=MELD2) -> P2_2 : (x'=MELD2) + P2_1 : (x'=MELD1) +  P2_3 : (x'=MELD3)+  P2_D : (x'=die);
[waite] (x=MELD3) -> P3_3 : (x'=MELD3) + P3_2 : (x'=MELD2) +  P3_4 : (x'=MELD4)+  P3_D : (x'=die);
[waite] (x=MELD4) -> P4_4 : (x'=MELD4) + P4_3 : (x'=MELD3) +  P4_5 : (x'=MELD5)+  P4_D : (x'=die);
[waite] (x=MELD5) -> P5_5 : (x'=MELD5) + P5_4 : (x'=MELD4) +  P5_6 : (x'=MELD6)+  P5_D : (x'=die);
[waite] (x=MELD6) -> P6_6 : (x'=MELD6) + P6_5 : (x'=MELD5) +  P6_7 : (x'=MELD7)+  P6_D : (x'=die);
[waite] (x=MELD7) -> P7_7 : (x'=MELD7) + P7_6 : (x'=MELD6) +  P7_8 : (x'=MELD8)+  P7_D : (x'=die);
[waite] (x=MELD8) -> P8_8 : (x'=MELD8) + P8_7 : (x'=MELD7) +  P8_9 : (x'=MELD9)+  P8_D : (x'=die);
[waite] (x=MELD9) -> P9_9 : (x'=MELD9) + P9_8 : (x'=MELD8) +  P9_10 : (x'=MELD10)+  P9_D : (x'=die);
[waite] (x=MELD10) -> P10_10 : (x'=MELD10) + P10_9 : (x'=MELD9) +  P10_11 : (x'=MELD11)+  P10_D : (x'=die);
[waite] (x=MELD11) -> P11_11 : (x'=MELD11) + P11_10 : (x'=MELD10) +  P11_12 : (x'=MELD12)+  P11_D : (x'=die);
[waite] (x=MELD12) -> P12_12 : (x'=MELD12) + P12_11 : (x'=MELD11) +  P12_13 : (x'=MELD13)+  P12_D : (x'=die);
[waite] (x=MELD13) -> P13_13 : (x'=MELD13) + P13_12 : (x'=MELD12) +  P13_14 : (x'=MELD14)+  P13_D : (x'=die);
[waite] (x=MELD14) -> P14_14 : (x'=MELD14) + P14_13 : (x'=MELD13) +  P14_15 : (x'=MELD15)+  P14_D : (x'=die);
[waite] (x=MELD15) -> P15_15 : (x'=MELD15) + P15_14 : (x'=MELD14) +  P15_16 : (x'=MELD16)+  P15_D : (x'=die);
[waite] (x=MELD16) -> P16_16 : (x'=MELD16) + P16_15 : (x'=MELD15) +  P16_17 : (x'=MELD17)+  P16_D : (x'=die);
[waite] (x=MELD17) -> P17_17 : (x'=MELD17) + P17_16 : (x'=MELD16) +  P17_18 : (x'=MELD18)+  P17_D : (x'=die);
[waite] (x=MELD18) -> P18_18 : (x'=MELD18) + P18_17 : (x'=MELD17) +  P18_D : (x'=die);
```

Fig. 6.4 "Wait" Actions

Model Construction

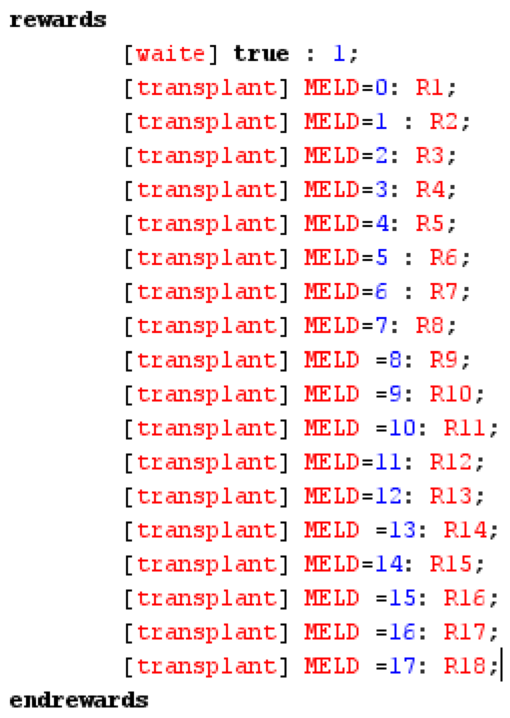
Using PRISM, we create a module for this MDP that consists of 20 states, [1-18] represent the MELD scores, and the two others represent the" transplant" and the" die" states. Concerning the type of transitions, they are divided with respect to the actions taken.

For patient to take "wait", all the transitions between health states are possible. But to facilitate the analysis, we consider just the adjacent health states and staying at the same health state, because this is the most common [13]. Therefore, at each health state, the patient could remain the same health state, get sicker or improves. In addition, at each decision epoch, patient could die from such health state, but practically, the patient could die just as from illness states. As we see in Figure 6.4, from the state MELD2, the patient can remain the same state with probability P2_2, get sicker by progressing to MELD3 with probability P2_3, improves by returning to MELD1 with probability P2_1 or he can die with



```
[transplant] (x=MELD1) -> 1 : (x'=transplant);
[transplant] (x=MELD2) -> 1 : (x'=transplant);
[transplant] (x=MELD3) -> 1 : (x'=transplant);
[transplant] (x=MELD4) -> 1 : (x'=transplant);
[transplant] (x=MELD5) -> 1 : (x'=transplant);
[transplant] (x=MELD6) -> 1 : (x'=transplant);
[transplant] (x=MELD7) -> 1 : (x'=transplant);
[transplant] (x=MELD8) -> 1 : (x'=transplant);
[transplant] (x=MELD9) -> 1 : (x'=transplant);
[transplant] (x=MELD10) -> 1 : (x'=transplant);
[transplant] (x=MELD11) -> 1 : (x'=transplant);
[transplant] (x=MELD12) -> 1 : (x'=transplant);
[transplant] (x=MELD13) -> 1 : (x'=transplant);
[transplant] (x=MELD14) -> 1 : (x'=transplant);
[transplant] (x=MELD15) -> 1 : (x'=transplant);
[transplant] (x=MELD16) -> 1 : (x'=transplant);
[transplant] (x=MELD17) -> 1 : (x'=transplant);
[transplant] (x=MELD18) -> 1 : (x'=transplant);
```

Fig. 6.5 Transplantation at each time epoch



```
rewards
        [waite] true : 1;
        [transplant] MELD=0: R1;
        [transplant] MELD=1 : R2;
        [transplant] MELD=2: R3;
        [transplant] MELD=3: R4;
        [transplant] MELD=4: R5;
        [transplant] MELD=5 : R6;
        [transplant] MELD=6 : R7;
        [transplant] MELD=7: R8;
        [transplant] MELD =8: R9;
        [transplant] MELD =9: R10;
        [transplant] MELD =10: R11;
        [transplant] MELD=11: R12;
        [transplant] MELD=12: R13;
        [transplant] MELD =13: R14;
        [transplant] MELD=14: R15;
        [transplant] MELD =15: R16;
        [transplant] MELD =16: R17;
        [transplant] MELD =17: R18;
endrewards
```

Fig. 6.6 Transplant transitions reward

probability P2_D. In this case study we use probabilities measures based on The Natural History of MELD [12] as well as [129].

The patient can move from any health state to “transplant” state, which is an absorbing state, by taking the action “transplant” with probability equals to 1 (Figure 6.5). We distinguish between the transplant actions at each time epoch, because at each state, the patient gets different post-transplant reward (R1, R2. . . .R18). This is in contrast to “wait” action that gets a constant reward of 1 day (see Figure 6.6). The post-transplant rewards adopted refer to [155].

Model Analysis

Using PRISM we can analyze MDPs, using properties with the operators P and R. By using them, we can effectively reason on the minimum-maximum probability and the minimum-maximum reward respectively over all possible resolutions of non-determinism. PRISM doesn’t offer just the maximum reward value of such policy or the maximum probability, but also offers simulation framework aiding the user to interpret the results.

For finding the optimal policy for the MDP presented, we chose the value iteration method for solving the MDP and Linear equations Jacobi provided by PRISM. For the rest

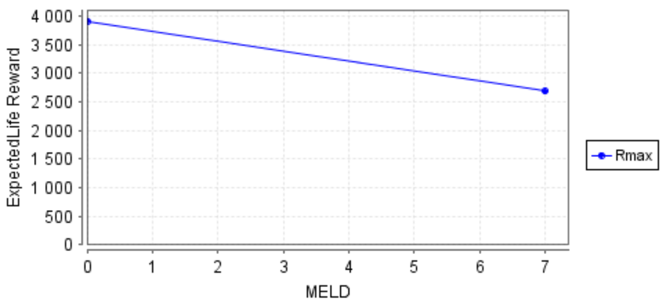


Fig. 6.7 Maximum Expected Life Rewards

of the analysis, we use the same algorithms. To find the optimal policy we use the following property:

$$Rmax = ?[F"transplant"]$$

This property computes the maximum reward over all resolutions of the model for reaching a transplant state. The time taken to accomplish the analysis was less than one second. The result was 3897, 58 as a maximum expected life reward, which is obtained following the policy of waiting until the health state MELD8 and then make the transplantation.

For further analysis, we can track the maximum reward along the policy found. We can get the maximum expected life rewards starting from each MELD to the final state “transplant”, by using the following property:

$$Rmax = ?[F"transplant"\{MELD = K\}\{max\}]$$

The graph simulating this formula for MELD states as can be generated using PRISM, is given in Figure 6.7. As we see, the maximum reward is high by starting from the first health state (MELD 1 = 0), because it’s obvious that while the patient gets sicker, his expected life reward goes down.

The User may also need to get informed about the minimum expected life reward starting from each state. Similarly to the maximum expected life, we can compute minimum expected life reward using the following property:

$$Rmin = ?[F"transplant"\{MELD = K\}\{min\}]$$

The graph simulating this property for MELD states is given in Figure 6.8. From the initial state, getting minimum reward for transplantation is nearly 2000. This minimum reward goes down when the patient gets sicker.

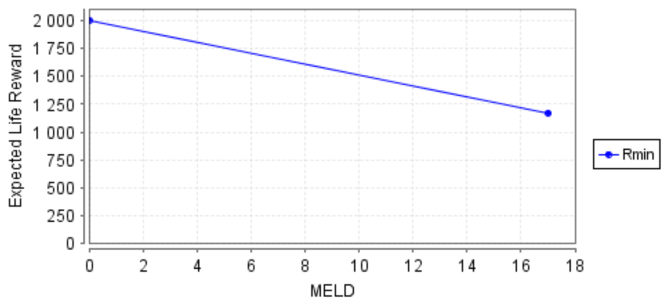


Fig. 6.8 Minimum expected life rewards

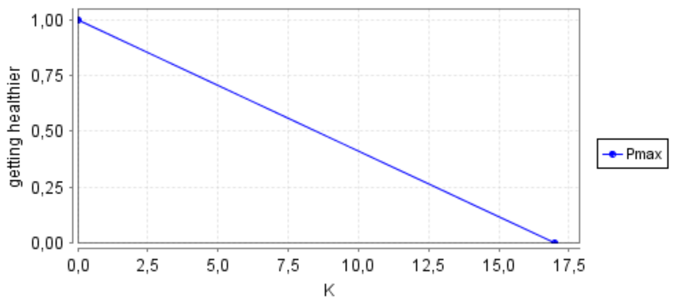


Fig. 6.9 Probability of getting healthier

More interesting about using probabilistic model checking is that we can analyse and simulate probabilistic properties. For example, the user may want to reason on the maximum probabilities getting patient from health-state to another. This can be achieved by using the following property:

$$Pmax = ?[F(MELD = N)\{MELD = K\}]$$

The graph simulating this formula is given by taking the case of estimating probabilities of returning to the first healthier state from the following sicker states (see Figure 6.9). As we see, while the patient gets sicker, the probability of returning to the first healthier state decreases.

Another useful example of probabilistic properties is when the user needs to reason on the maximum probability of death, given a period of time without transplantation. This is formulated using bounded until formula as follow:

$$Pmax = ?[trueU \leq days"die"]$$

The simulation result of this property is given in Figure 6.10. The interpretation of this graph is so simple. The days the patient spent without transplantation, the probability of death will increase.

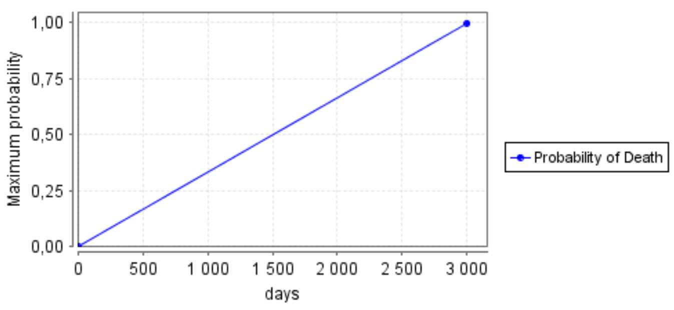


Fig. 6.10 Probability of death

6.4 Conclusion

Probabilistic CEP applications have known a great success as a technological paradigm to deal with the high processing of events in real time under uncertainty. Therefore, delivering analysis techniques to ensure the correctness of these systems has become a great challenge. In this chapter, we proposed a formal verification framework for probabilistic CEP based on timed probabilistic model checking. We showed how useful is using TPCTL logic for specifying timed probabilistic properties to analyse the behaviour of EPAs, dependently and independently.

In this chapter, we have also illustrated the use of probabilistic model checking for analysing MDPs in the context of medical treatment decision. We have investigated a case study that concerns the problem of the optimal timing of living-donor liver transplantation, and we have implemented the corresponding MDP in the model checker PRISM. We showed its effectiveness for decision making from building models to quantitative and cost-effective analysis. Due to its formal and logical foundation, probabilistic model checking can serve as a good framework for modelling and analysing medical treatment problems that involve probability and non-determinism.

Chapter 7

Conclusions

In this thesis, we surveyed the counterexamples in model checking from many aspects, but mainly generation and debugging, and we have seen the usefulness of counterexamples for other purposes like CEGAR and test cases generation. We have seen that the techniques based on counterexamples can directly benefit from any advancement and new proposition for generating small and indicative counterexamples in considerable time. It is not possible to cover all the issues related to counterexamples in model checking . However, we hope that we surveyed most important issues for counterexamples that could stand as a good starting point for new research works in this field. In the future we expect to see more works on generating counterexamples in the context of probabilistic model checking and their analysis, and we also expect to see more works on CEGAR especially in probabilistic model checking. Roughly speaking, the probabilistic model checking is an active area for counterexamples. Another challenge concerning counterexamples is their visualisation to help in debugging. In addition, we expect to see more works in other domains that adapt model checking techniques just for the seek of using counterexamples.

Then, we showed that the principle of counterexample generation in probabilistic model checking is completely different, where we showed that generating small and indicative counterexamples is not enough for understanding the error, and thus counterexample analysis is a necessary task. So, In the second and the main part of this thesis, we have presented a set of novel techniques and methods for analysing counterexamples in probabilistic model checking. The proposed methods are based on strong theoretical background of causality. In this chapter we have shown how the notions of causality and responsibility can be interpreted in the context of probabilistic counterexamples. Due to the probabilistic nature of the causal model, we had to define for each cause its probability. Accordingly, we introduced the notion of the most responsible cause. Following that, we first introduced an algorithm for diagnoses generation for DTMCs and CTMCs that acts as a guided-method to the most responsible causes in the counterexample. The most responsible cause is considered to be the most relevant to the user. We then extended our method to counterexamples for MDPs by adopting the notion of blame where we showed that delivering the causes/actions with respect to their responsibility/blame stands as a good debugging method that guides the user through large counterexamples. The two methods were applied on many case studies, and showed good results in term of quality and execution time. All the methods proposed are applied on many case studies from deterministic and non-deterministic systems to discrete and continuous systems, and show promising results. Such theoretical notions like blame is adopted for the first time in application and industrial domain.

We have also proposed another approach for analysing probabilistic counterexamples using regression analysis. The idea was about generating a regression model corresponding to the causality model. For doing so, we have proposed an algorithm for generating the causes as data set. We have seen that delivering the causes for the violation of PCTL/CSL formula as a regression model stands as a good technique for describing the effect of variables with their values on the error. Besides, we find that the experimental results of this method on the two case studies are well conform with the experimental results of the previous one. Hence, we conclude that both of the methods could be used in complementary way.

As future works, we plan to investigate the problem of incomplete data, which is well known problem in regression, as well as the problem of linear dependence between variables in the context of probabilistic counterexamples. We want also to combine between the two approaches to reach better analysis of the counterexample. As further future work, we plan to deliver a debugging tool that generates the diagnoses graphically. Furthermore, we aim to integrate our methods in the model checking process itself in order to generate the diagnoses with respect to the guarded command language used by probabilistic model checkers like PRISM.

Debugging in probabilistic model checking is still in its first stage. In the future we expect to see more works on debugging. We expect to see new debugging tools that base directly on the guarded-command language of probabilistic model checkers like PRISM, and not just use the existed tools for counterexample generation.

With the growing importance of probabilistic model checking for the verification and quantitative analysis of probabilistic systems, in this thesis, we investigated its applicability on two different domains. In the first section we proposed a verification approach for *Probabilistic Complex Event Processing (Probabilistic CEP)* . The formal verification framework for probabilistic CEP was based on timed probabilistic model checking. We showed how useful is using TPCTL logic for specifying timed probabilistic properties to analyse the behaviour of Event Processing Agents (EPAs), dependently and independently. In the second section, we showed how probabilistic model checking can serve as a comprehensive technique for modelling and analysing medical treatment problems. We have investigated a case study that concerns the problem of the optimal timing of living-donor liver transplantation, and we have implemented the corresponding MDP in the model checker PRISM. We showed its effectiveness for decision making from building models to quantitative and cost-effective analysis. Due to its formal and logical foundation, probabilistic model checking can serve as a good framework for modelling and analysing medical treatment problems that involve probability and non-determinism.

As future works, we aim to investigate in depth manner the constraints that should be put on adopting the probabilistic timed automata (PTA) as a description model for CEP application, and we aim also to investigate the extension of TPCTL for specifying more special properties of CEP applications. For medical treatment analysis, we have seen that we can get the optimal policy using probabilistic model checker PRISM, but PRISM cannot offer a rich guiding for explaining the optimal policy found. Policy explanation is one of the most open problems in literature; it was treated in general aspect as well as in medical aspect . As future works, we aim to build a software solution at the front of PRISM tool for helping the user in policy explanation with high visualization help.

References

[1] Amos. "http://amosdevelopment.com/download/", consulted on september 2014.

[2] Blast. http://goto.ucsd.edu/~rjhala/blast.html, consulted on september 2014.

[3] Cbmc. http://www.cprover.org/cbmc/, consulted on september 2014.

[4] Comics. http://www-i2.informatik.rwth-aachen.de/i2/623/, consulted on september 2014.

[5] Csma/cd protocol. http://www.prismmodelchecker.org/casestudies/csma.php, consulted on september 2014.

[6] Cyclic server polling system. http://www.prismmodelchecker.org/casestudies/polling.php, consulted on september 2014.

[7] Dipro. http://www.inf.uni-konstanz.de/soft/dipro/, consulted on september 2014.

[8] Esper.http://esper.codehaus.org/, consulted on september 2014.

[9] Embedded control system. http://www.prismmodelchecker.org/casestudies/embedded.php, consulted on september 2014.

[10] Ipv4 zeroconf protocol.http://www.prismmodelchecker.org/casestudies/zeroconf.php#time., consulted on september 2014.

[11] Javapathfinder. http://javapathfinder.sourceforge.net/, consulted on september 2014.

[12] The natural history of meld informs healthcare. http://www.users.iems.northwestern.edu, consulted on september 2014.

[13] Mrmc. http://www.mrmc-tool.org/trac/, consulted on september 2014.

[14] Microsoft streaminsight. ttp://msdn.microsoft.com/en-us/sqlserver/ee476990.aspx, consulted on september 2014.

[15] Nusmv. http://nusmv.fbk.eu/, consulted on september 2014.

[16] Oracle cep. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html, consulted on september 2014.

[17] Prism. http://www.prismmodelchecker.org/, consulted on september 2014.

[18] Spin. http://spinroot.com/spin/whatispin.html,consulted on september 2014.

[19] Uppaal. http://www.uppaal.org/, consulted on september 2014.

[20] E. Abraham, B. Becker, C. Dehnert, N. Jansen, J.P Katoen, and R. Wimmer. Counterexample generation for discrete-time markov models: An introductory survey. In *Formal Methods for Executable Software Models*, LNCS, vol. 8483, pages 65–121. Springer International Publishing Switzerland, 2014.

[21] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical report, School of Information and Computer Science, 1979.

[22] O. Alagoz, L.M. Aillart, A.J. Schaefer, and M.S Roberts. The optimal timing of livingdonor liver transplantation. *Manage Science*, 50(10):1420–1430, 2004.

[23] O. Alagoz, L.M. Aillart, A.J. Schaefer, and M.S Roberts. Markov decision processes: a tool for sequential decision making. *Medical Decision Making*, 30(04):474–483, 2010.

[24] H. Aljazzar and S. Leue. Extended directed search for probabilistic timed reachability. In *FORMATS*, LNCS, vol. 4202, pages 33–51. Springer, Berlin, Heidelberg, 2006.

[25] H. Aljazzar and S. Leue. Generation of counterexamples for model checking of markov decision processes. In *International Conference on Quantitative Evaluation of Systems (QEST)*, pages 197–206, 2009.

[26] H. Aljazzar and S. Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking. *IEEE Trans. on Software Engineering*, 36(1):37–60, 2010.

[27] H. Aljazzar, H. Hermanns, and S. Leue. Counterexamples for timed probabilistic reachability. In *FORMATS*, LNCS, vol. 3829, pages 177–195. Springer, Berlin, Heidelberg, 2005.

[28] H. Aljazzar, F. Leitner-Fischer, S. Leue, and D. Simeonov. Dipro - a tool for probabilistic counterexample generation. In *18th International SPIN Workshop*, LNCS, vol. 6823, pages 183–187. Springer, Berlin, Heidelberg, 2011.

[29] R. Ammon, C. Emmersberger, F. Springer, and C. Wolff. Event-driven business process management and its practical application taking the example of dhl. In *the 1st International workshop on Complex Event Processing for the Future Internet (iCEP)*, 2008.

[30] M. E. Andres, P.R. DArgenio, and P. van Rossum. Significant diagnostic counterexamples in probabilistic model checking. In *Haifa Verification Conference*, pages 129–148, 2008.

[31] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model-checking continuous-time markov chains. *ACM Transactions on Computational Logic*, 1(1):162–170, 2000.

[32] C. Baier, B. Haverkort, H. Hermanns, and J.-P Katoen. Model checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29 (7):524–541, 2003.

[33] T. Ball and S.K. Rajamani. The slam project: Debugging system software via static analysis. In *ACM Symposium on the Principles of Programming Languages*, pages 1–3, 2002.

[34] T. Ball, M. Naik, and S.K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *ACM Symposium on the Principles of Programming Languages*, pages 97–105, 2003.

[35] E. Bartocci, P. Lio, E.Merelli, and N. Paoletti. Multiple verification in computational modeling of bone pathologies. *Transactions on Computational Systems Biology*, 14: 53–76, 2012.

[36] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Treer. Explaining counterexamples using causality. *Formal Methods Systems Design*, 40(1):20–40, 2012.

[37] R. Bellman. Dynamic programming. *Princeton University Press.*

[38] P. Bjesse and J. Kukula. Using counterexample guided abstraction refinement to find complex bugs. In *Design, Automation and Test in European Conference and Exhibition*, pages 156–161, 2004.

[39] B. Braitling and R. Wimmer. Counterexample generation for markov chains using smt-based bounded model checking. In *Formal Techniques for Distributed Systems*, LNCS, vol. 6722, pages 75–89. Springer, Berlin, Heidelberg, 2011.

[40] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder a second generation of a java model checker. In *Workshop on Advances in Verification*, 2000.

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

[42] T.A. Budd and A.S. Gopal. Program testing by specification mutation. *Journal Computer Languages*, 10(1):63–73, 1985.

[43] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[44] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model checking. In *SPIN Workshop*, 1996.

[45] S. Chaki and A. Groce. Explaining abstract counterexamples. In *SIGSOFT04/FSE*, pages 73–82, 2004.

[46] P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, and D.Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer Aided Design(FMCAD)*, LNCS, vol. 2517, pages 33–51. Springer, Berlin, Heidelberg, 2002.

[47] H. Chockler and J. Y. Halpern. Responsibility and blame: a structural model approach. *Journal of Artificial Intelligence Research (JAIR)*, 22(1):93–115, 2004.

[48] H. Chockler, J.Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? *ACM Transactions on Computational Logic*, 9(3):1–24, 2007.

[49] X. Chuanfei, L. Shukuan, W. Lei, and Q. Jianzhong. Complex event detection in probabilistic stream. In *12th International Asia-Pacific Web Conference*, 2010.

[50] O. C.Ib and K. Trivedi. Stochastic petri net models of polling systems. *IEEEJournal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[51] E. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms and applications. In *In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking*, LNCS, pages 1–26. Springer, Berlin, Heidelberg, 2008.

[52] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proc. of the Design Automation Conference*.

[53] E. Clarke, Y. Lu, s. Jha, and H. Veith. Tree-like counterexamples in model checking. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 19–29, 2002.

[54] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50 (5):752–794, 2003.

[55] E. M. Clarke, O. Grumberg, and D.E. Andlong. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[56] Edmund Clarke. The birth of model checking. In *Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking*, LNCS, pages 1–26. Springer, Berlin, Heidelberg, 2008.

[57] Edmund Clarke and Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71. Springer-Verlag, 1982.

[58] E.M. Clarke, O.Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 1986.

[59] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[60] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT, 1999.

[61] E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. Sat based abstraction refinement using ilp and machine leraning techniques. In *Computer-Aided Verification (CAV)*, LNCS, vol. 2404, pages 137–150. Springer, Berlin, Heidelberg, 2002.

[62] H. Cleve and A. Zeller. Locating causes of program failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.

[63] F. Copty, A. Irron, O. Weissberg, N. Kropp, and K. Gila. Effcient debugging in a formal verification environment. *Int J Softw Tools Technol Transfer*, 4:335–348, 2003.

[64] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *J. ACM*, 42(4):857–904, 1995.

[65] P. COUSOT and R. COUSOT. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium of Programming Language*, pages 238–252, 2003.

[66] J.M. Couvreur. On-the-fly verification of linear temporal logic. In *FM*, LNCS, vol. 1708, pages 253–271. Springer, Heidelberg, 1999.

[67] Luckham D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[68] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 7:864–875, 2007.

[69] B. Damman, T. Han, and J.P. Katoen. Regular expressions for pctl counterexamples. In *Quantitative Evaluation of Systems(QEST)*, pages 179–188, 2008.

[70] L. de Alfaro, T.A. Henzinger, and F. Mang. Detecting errors before reaching them. In *CAV*, LNCS, vol. 2725, pages 186–201. Springer, Berlin, Heidelberg, 2000.

[71] H. Debbi and M. Bourahla. Generating diagnoses for probabilistic model checking using causality. *Journal of Computing and Information Technology*, 21(1):13–22, 2013.

[72] H. Debbi and M. Bourahla. Causal analysis of probabilistic counterexamples. In *Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign (Memocode)*, pages 77–86, 2013.

[73] H. Debbi and M. Bourahla. Verification approach for probabilistic complex event processing (cep) applications. In *The First International Symposium on Informatics and its Applications, M'sila*, 2014.

[74] H. Debbi, M. Bourahla, and A. Debbi. Medical treatment analysisusing probabilistic model checking. *International Journal of Biomedical Engineering and Technology*, 12(4):346–359, 2013.

[75] H. Debbi, B. Lounas, and A. Bentaleb. Real-time alert correlation approach based on complex event processing. In *International Conference on Systems and Procesing Information*, 2013.

[76] B.T Denton, K. Murat, N.D. Shah, S.C. Bryant, and S.A. Smith. Optimizing the start time of statin therapy for patients with diabetes. *Medical Decision Making*, 29(03): 351–367, 2009.

[77] B.T.O. Denton, A. Holder, and E.K. Lee. Medical decision making: open research challenges. *IIE Transactions on Healthcare Systems Engineering*, 1(03):161–167, 2011.

[78] F.J. Diez, M.A. Palacios, and M. Arias. Mdps in medicine: Opportunities and challenges. In *IJCAI Workshop on Decision Making in Partially Observable, Uncertain Worlds: Exploring Insights from Multiple Communities*, 2011.

[79] M. Duot, L. Fribourg, and T. Herault et al. Probabilistic model checking of the csma/cd protocol using prism and apmc. In *the Fouth International Workshop on Automated Verification of Critical Systems (AVoCS 2004)*, ENTCS, pages 195–214, 2004.

[80] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2):247–267, 2004.

[81] T. Eiter and T. Lukasiewicz. Complexity results for structure-based causality. *Artificial Intelligence*, 142(1):53–89, 2002.

[82] H. Elghazel, V. Deslandres, K. Kallel, and A. Dussauchoy. Clinical pathway clustering using graph b-colouring and markov models. *International Journal of Biomedical Engineering and Technology*, 3(1):156–172, 2010.

[83] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *STOC 82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 169–180. ACM Press, 1982.

[84] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*.

[85] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS97)*, LNCS, vol. 1217, pages 384–398. Springer, Berlin, Heidelberg, 1997.

[86] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems(TACAS)*, LNCS, vol. 1217, pages 384–398. Springer, Berlin, Heidelberg, 2010.

[87] A. Ericsson, P. Pettersson, M. Berndtsson, and M.Seirio. seamless formal verification of complex event processing applications. In *International conference on Distributed Event Based Systems (DEBS)*, pages 50–61, 2007.

[88] O. Etzion. *Event Processing in Action*. MANNING, 2010.

[89] G. Fey and R. Drechsler. Finding good counterexamples to aid design verification. In *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE03)*, pages 51–52, 2003.

[90] F. Fischer and S. Leue. Causality checking for complex system models. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, vol. 7737, pages 248–276. Springer, Berlin, Heidelberg, 2013.

[91] K. Fisler, R. Fraer, G. Kamhi, M.Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm. In *TACAS 2001*, LNCS, vol. 2031, pages 420–434. Springer, Berlin, Heidelberg, 2001.

[92] G. Fraser. *Automated Software Testing with Model Checkers*. PhD thesis, IST - Institute for Software Technology, 2007.

[93] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers. *Journal of Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[94] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements ments specifications. In *ESEC/FSE99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, LNCS, vol. 1687, pages 146–162. Springer, Berlin, Heidelberg, 1999.

[95] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from asm specifications. In *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM*, LNCS, vol. 2589, pages 263–277. Springer, Berlin, Heidelberg, 2003.

[96] P. Gastin and P. Moro. Minimal counterexample generation for spin. In *14th International SPIN Workshop 2007*, LNCS, vol. 4595, pages 24–38. Springer, Berlin, Heidelberg, 2007.

[97] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexample in spin. In *SPIN 2004*, LNCS, vol. 2989, pages 92–108. Springer, Berlin, Heidelberg, 2004.

[98] M.R. Gold, J.E. Siegel, L.B. Russell, and M.C. Weinstein. *Cost Effectiveness in Health and Medicine*. Oxford University Press, 1996.

[99] S. GRAF and H. ANDSADI. Construction of abstract state graphs with pvs. In *CAV*, LNCS, vol. 1254, pages 72–83. Springer, Berlin, Heidelberg, 1997.

[100] A. Groce. Error explanation with distance metrics. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 108–122, 2004.

[101] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[102] A.D Groce. *Error Explanation and Fault Localization with Distance Metrics*. PhD thesis, School of Computer Science Carnegie Mellon University, 2005.

[103] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *15th international conference on Compiler Construction*, LNCS, vol. 3923, pages 80–95. Springer, Berlin, Heidelberg, 2006.

[104] J. Halpern and J. Pearl. Causes and explanations: A structural-model approach part i: Causes. In *17th UAI*, pages 194–202, 2001.

[105] T. Han and J.P. Katoen. Counterexamples generation in probabilistic model checking. *IEEE Trans. on Software Engineering*, 35(2):72–86, 2009.

[106] H. Hansen and J. Geldenhuys. Cheap and small counterexamples. In *Software Engineering and Formal Methods, SEFM '08*, pages 53–62. IEEE Computer Society Press, 2008.

[107] H. Hansen and A. Kervinen. Minimal counterexamples in o(n log n) memory and o(n 2) time. In *ACDC 2006*, pages 131–142. IEEE Computer Society Press, 2006.

[108] H. Hansson and B. Jonsson. Logic for reasoning about time and reliability. *Formal aspects of Computing*, 6(5):512–535, 1994.

[109] M. Hauskrecht and H. Fraser. Planning treatment of ischemic heart disease with partially observable markov decision processes. *Artificial Intelligence in Medicine*, 18(3):221–244, 2000.

[110] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O.Tymchyshyn. Probabilistic model checking of complex biological pathways. *Theoretical Computer Science*, 391 (3):239–257, 2008.

[111] M. Heimdahl, S. Rayadurgam, and W. Visser. Specification centered testing. In *Second International Workshop on Automates Program Analysis, Testing and Verification*, 2000.

[112] T. Herault, R. Lassaigne, and S. Peyronnet. Apmc 3.0: Approximate verification of discrete and continuous time markov chains. In *Third International Conference on the Quantitative Evaluaiton of Systems (QEST 2006)*, pages 129–130, 2006.

[113] H. Hermanns, B. Wachter, and L. Zhang. Probabilistic cegar. In *Computer Aided Verification (CAV)*, LNCS, vol. 5123, pages 162–175. Springer, Berlin, Heidelberg, 2008.

[114] A. Hilal and J. Gaylor. Bioartificial liver: review of science requirements and technology. *World Review of Science, Technology and Sustainable Development*, 3(1): 80–97, 2006.

[115] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, LNCS, vol. 3920, pages 441–444. Springer, Berlin, Heidelberg, 2006.

[116] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Effcient -regular language containment. In *Computer Aided Verification*, LNCS, vol. 1708, pages 371–382. Springer, Berlin, Heidelberg, 1992.

[117] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In *Fifth Conference on Computer Aided Verification (CAV 93)*, LNCS, vol. 697, pages 41–58. Springer, Berlin, Heidelberg, 1993.

[118] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In *CAV 93*, LNCS, vol. 697, pages 41–58. Springer, Berlin, Heidelberg, 1993.

[119] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth
first search. In *SPIN'96*, 1996.

[120] H. Seok Hong and I. Lee. Automatic test generation from specifications for controlflow and data-flow coverage criteria. In *International Conference on Software Engineering (ICSE)*, 2003.

[121] R.A. Howard. *Dynamic Programming andMarkov Processes*. MIT Press, Cambridge, 1960.

[122] D. Hume. A treatise of human nature. 1739.

[123] M. Janota, R. Grigore, and J. Marques-Silva. Counterexample guided abstraction refinement algorithm for propositional circumscription. In *JELIA'10 Proceedings of the 12th European conference on Logics in artificial intelligence*, LNCS, vol. 6341, pages 195–207. Springer, Berlin, Heidelberg, 2010.

[124] N. Jansen, E. Abraham, M. Volk, R. Wilmer, J.P Katoen, and B. Becker. The comics tool - computing minimal counterexamples for dtmcs. In *ATVA*, LNCS, vol. 7561, pages 249–253. Springer, Berlin, Heidelberg, 2012.

[125] H. Jin, K. Ravi, and F.Somenzi. Fate and free will in error traces. *International Journal on Software Tools for Technology Transfer*, 6(2):102–116, 2004.

[126] K. Johnson, S. Reed, and R. Calinescu. Specification and quantitative analysis of probabilistic cloud deployment patterns. In *HVC 2011*, LNCS, pages 145–159. Springer-Verlag Berlin Heidelberg, 2011.

[127] S. Kashyap and V.K. Garg. Producing short counterexamples using crucial events. In *CAV 2008*, LNCS, vol. 5123, pages 491–503. Springer, Berlin, Heidelberg, 2008.

[128] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A markov reward model checker. In *QEST*, pages 243–244, 2005.

[129] D. Kaufman, A. Schaefer, and M. Roberts. Living-donor liver transplantation timing under ambiguous health state transition probabilities. In *MSOM Annual Conference*, 2011.

[130] Y. Kesten, A. Pnueli, and L. o. Raviv. Algorithmic verification of linear temporal logic specifications. In *International Colloquium on Automata, Languages, and Programming (ICALP-98),*, LNCS, vol. 1443, pages 1–16. Springer, Berlin, Heidelberg, 1998.

[131] S. Kikuchi and Y. Matsumoto. Performance modeling of concurrent live migration operations in cloud computing systems using prism probabilistic model checker. In *4th Intl. Conf. on Cloud Computing (2011)*, pages 49–56, 2011.

[132] D. Kroening, A. Groce, and E. Clarke. Counterexample guided abstraction refinement via program execution. In *6th International Conference on Formal Engineering Methods (ICFEM)*, LNCS, vol. 3308, pages 224–238. Springer, Berlin, Heidelberg, 2004.

[133] N. Kuma, V. Kumar, and M. Viswanathan. On the complexity of error explanation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, LNCS, vol. 3385, pages 448–464. Springer, Berlin, Heidelberg, 2005.

[134] T. Kumazawa and T. Tamai. Counterexample-based error localization of behavior models. In *NASA Formal Methods*, pages 222–236, 2011.

[135] M. Kuntz, F. Leitner-Fischer, and S. Leue. From probabilistic counterexamples via causality to fault trees. In *Computer Safety, Reliability, and Security*, LNCS, vol. 6894, pages 71–84. Springer, Berlin, Heidelberg, 2011.

[136] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science, Elsevier*, 282(1):101–150, 2002.

[137] M. Kwiatkowska, G. Norman, and D. Parker. Using probabilistic model checking in systems biology. *ACM Sigmetrics Performance Evaluation Review*, 35(4):14–21, 2008.

[138] A. Laha. Rap: A conceptual business intelligence framework. In *1st Bangalore Annual Compute Conference*, 1994.

[139] F. Leitner-Fischer and S. Leue. On the synergy of probabilistic causality computation and causality checking. In *SPIN 2013*, LNCS, vol. 7976, pages 246–263. Springer-Verlag, Berlin, Heidelberg, 2013.

[140] D. Lewis. Causation. *Journal of Philosophy*, 70:556–567, 1973.

[141] Z. Li and T.Ge. Online windowed subsequence matching over probabilistic sequences. In *ACM SIGMOD international conference on Management of data*, pages 277–288, 2012.

[142] D.E. LONG. *Model checking, abstraction and compositional verification*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2005.

[143] K.L. McMillan and L.D. Zuck. Abstract counterexamples for non-disjunctive abstractions. In *Reachability Problems*, LNCS, vol. 5797, pages 176–188. Springer, Berlin, Heidelberg, 2009.

[144] J. Muppala, G. Ciardo, and K. Trivedi. stochastic reward nets for reliability prediction. *Communications in Reliability Maintainability and Serviceability*, 1(5), 1994.

[145] T. Nguyen, H. Guo, and R. Naguib. A survey of industrial experiences for the uk healthcare software development sector. *International Journal of Biomedical Engineering and Technology*, 1(3):329–341, 2008.

[146] T. Nopper, C. Scholl, and B. Becker. Computation of minimal counterexamples by using black box techniques and symbolic methods. In *Computer-Aided Design (ICCAD)*, pages 273–280. IEEE Computer Society Press, 2007.

[147] B. Pandey and R.B. Mishra. Data-mining models for the diagnosis of emg-based neuromuscular diseases. *International Journal of Biomedical Engineering and Technology*, 6(2):109–128, 2011.

[148] T. Pronk, E. de Vink, D. Bosnacki, and T. Breit. Stochastic modeling of codon bias with prism. In *3rd Int. Workshop Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2007)*, pages 1–15, 2007.

[149] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, pages 273–276, 2003.

[150] E. Rabinovich, O. Etzion, and S. Ruah. Analyzing the behavior of event processing applications. In *the Fourth ACM International Conference on Distributed Event-Based Systems*, pages 223–234, 2010.

[151] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *TACAS*, LNCS, vol. 2988, pages 31–45. Springer, Berlin, Heidelberg, 2004.

[152] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In *Third International Conference, FMCAD 2000*, LNCS, vol. 1954, pages 162–179. Springer, Berlin, Heidelberg, 2000.

[153] K. Ravi, R. Bloem, and F. Somenzi. A note on on-the-fly verification algorithms. In *TACAS 2005*, LNCS, vol. 3440, pages 174–190. Springer, Berlin, Heidelberg, 2005.

[154] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[155] M.S. Roberts, D.C. Angus, C.L. Bryce, Z. Valenta, and L. Weissfeld. Survival after liver transplantation in the united states: a disease-specific analysis of the unos database. *Liver Transplantation*, 10(7):886–897, 2004.

[156] A.J. Schaefer, M.D. Bailey, S.M. Shechter, and M.S. Roberts. Modeling medical treatment using markov decision processes. *International Series in Operations Research and Management Science*, 70:593–612, 2005.

[157] M. Schmalz, D. Varacca, and H. Volzer. Counterexamples in probabilistic ltl model checking for markov chains. In *International Conference on Concurrency Theory (CONCUR)*, LNCS, vol. 5710, pages 787–602. Springer, Berlin, Heidelberg, 2009.

[158] V. Schuppan and A. Biere. Shortest counterexamples for symbolic model checking of ltl with past. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, vol. 3440, pages 493–509. Springer, Berlin, Heidelberg, 2005.

[159] S. Sen. Business activity monitoring based on action-ready dashboards and response loop. In *1st International workshop on Complex Event Processing for the Future Internet (iCEP)*, 2008.

[160] S. Sewall. Correlation and causation. *Journal of Agricultural Research*, 20:557, 1921.

[161] S. Sfakianakis, M. Blazantonakis, I. Dimou, M. Zervakis, M.Tsiknakis, G. Potamias, D. Kafetzopoulos, and D. Lowe. Decision support based on genomics: integration of data and knowledge-driven reasoning. *International Journal of Biomedical Engineering and Technology*, 3(3), 2010.

[162] S.M. Shechter, M.D. Bailey, A.J. Schaefer, and M.S. Roberts. The optimal time to initiate hiv therapy under ordered health states. *Operations Research*, 56(1):20–33, 2008.

[163] S. Shen and S. Li Y. Qin. Localizing errors in counterexample with iteratively witness searching. In *ATVA 2004*, LNCS, vol. 3299, pages 459–464. Springer, Berlin, Heidelberg, 2004.

[164] S. Shen, Y. Qin, and S. Li. Localizing errors in counterexample with iteratively witness searching. In *ATVA*, LNCS, vol. 3299, pages 456–469. Springer, Berlin, Heidelberg, 2004.

[165] S. Shen, Y. Qin, and S. Li. Bug localization of hardware system with control flow distance minimization. In *13th IEEE International Workshop on Logic and Synthesis (IWLS 2004)*, 2004.

[166] S. Shen, Y. Qin, and S. Li. Minimizing counterexample with unit core extraction and incremental sat. In *Verification, Model Checking, and Abstract Interpretation*, LNCS, vol. 3385, pages 298–312. Springer, Berlin, Heidelberg, 2005.

[167] S.-Y. Shen, Y. Qin, and S. Li. A fast counterexample minimization approach with refutation analysis and incremental sat. In *Conference on Asia South Pacific Design Automation*, pages 451–454, 2005.

[168] F.A. Sonnenberg and J.R. Beck. Markov models in medical decision making: a practical guide. *Medical Decision Making*, 13(4):322–339, 1993.

[169] J. Tan, G.S. Avrunin, and S. Leue. Heuristic-guided counterexample search in flavers. In *12th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 201–210, 2004.

[170] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[171] F. Tip and T.B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology1*, 10(1):5–55, 2001.

[172] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω automata using bdds. In *International Workshop on Formal Methods in VLSI Design*, pages 371–382, 1991.

[173] A. Valmari and J. Geldenhuys. Tarjans algorithm makes on-the-fly ltl verification more effcient. In *Jensen, K., Podelski, A. (eds.) TACAS*, LNCS, vol. 2988, pages 205–219. Springer, Berlin, Heidelberg, 2004.

[174] M. Y. Vardi. Verification of probabilistic concurrent finite-state programs. In *the 26th Annual Symposium on Foundations of Computer Science (FOCS 85)*, 1985.

[175] M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, 1992.

[176] C. Wang, Z. Yang, F. Ivancic, and A. Gupta. Whodunit? causal analysis for counterexamples. In *4th International Symposium, ATVA*, LNCS, vol. 4218, pages 82–95. Springer, Berlin, Heidelberg, 2006.

[177] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Complex event processing over uncertain data,. In *second international conference on Distributed event-based systems*, pages 253–264, 2008.

[178] R. Wimmer, B. Braitling, and B. Becker. Counterexample generation for discrete-time markov chains using bounded model checking. In *Verification, Model Checking, and Abstract Interpretation*, LNCS, vol. 5403, pages 366–380. Springer, Berlin, Heidelberg, 2009.

[179] R. Wimmer, N. Jansen, E. Abraham, B. Becker, and J.P. Katoen. Minimal critical subsystems for discrete-time markov models. In *TACAS*, LNCS, vol. 7214, pages 299–314. Springer, Berlin, Heidelberg, 2012.

[180] R. Wimmer, N. Jansen, and A. Vorpahl. High-level counterexamples for probabilistic automata. In *Quantitative Evaluation of Systems (QEST)*, LNCS, vol. 8054, pages 39–54. Springer, Berlin, Heidelberg, 2013.

[181] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *International Conference on ComputerAided Design*, pages 37–40, 1999.

[182] W. Yao, C. Chu, and Z. Li. ?leveraging complex event processing for smart hospitals using rfid. *Journal of Network and Computer Applications*, 34(3):799–810, 1994.

[183] H. Younes. A statistical model checker. In *7th International Conference on Computer Aided Verification (CAV'05)*, LNCS, pages 429–433. Springer, 2005.

[184] C. Zang and Y. Fan. Complex event processing in enterprise information systems based on rfid. *Journal Enterprise Information Systems*, 1(1):3–23, 2007.

[185] A. Zeller. Yesterday, my program worked. today, is does not. why? In *ACM Symposium on the Foundations of Software Engineering*, pages 253–267, 1999.

[186] A. Zeller. Isolating cause-effect chains for computer programs. In *ACM Symposium on the Foundations of Software Engineering*, pages 1–10, 2002.

[187] J. Zhang, B.T. Denton, and H. Balasubramanian. Optimization of psa screening policies: a comparison of the patient and societal perspectives. *Medical Decision Making*, 32(2):337–349, 1993.

Index