Predictive Runtime Verification of Stochastic Systems

by

Reza Babaee Cheshmeahmadrezaee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Hanan Lutfiyya Professor, Dept. of Computer Science, University of Western Ontario
Supervisor(s):	Derek Rayside Vijay Ganesh Professor, Dept. of Electrical & Computer Engineering, University of Waterloo
Internal Member(s):	Stephen L. Smith Mark Crowley Professor, Dept. of Electrical & Computer Engineering, University of Waterloo
Internal-External Member:	Joanne Atlee Professor, Dept. of Computer Science, University of Waterloo

Author's Declaration

Abstract

Runtime Verification (RV) is the formal analysis of the execution of a system against some properties at runtime. RV is particularly useful for stochastic systems that have a non-zero probability of failure at runtime. The standard RV assumes constructing a monitor that checks only the currently observed execution of the system against the given properties.

This dissertation proposes a framework for *predictive* RV, where the monitor instead checks the current execution with its finite extensions against some property. The extensions are generated using a prediction model, that is built based on execution samples randomly generated from the system. The thesis statement is that predictive RV for stochastic systems is *feasible*, *effective*, and *useful*.

The feasibility is demonstrated by providing a framework, called $\mathcal{P}revent$, that builds a predictive monitor by using trained prediction models to finitely extend an execution path, and computing the probabilities of the extensions that satisfy or violate the given property. The prediction model is trained using statistical learning techniques from independent and identically distributed samples of system executions. The prediction is the result of a quantitative bounded reachability analysis on the product of the prediction model and the automaton specifying the property. The analysis results are computed offline and stored in a lookup table. At runtime the monitor obtains the state of the system on the prediction model based on the observed execution, directly or by approximation, and uses the lookup table to retrieve the computed probability that the system at the current state will satisfy or violate the given property within some finite number of steps.

The effectiveness of $\mathcal{P}revent$ is shown by applying abstraction when constructing the prediction model. The abstraction is on the observation space based on extracting the symmetry relation between symbols that have similar probabilities to satisfy a property. The abstraction may introduce nondeterminism in the final model, which is handled by using a hidden state variable when building the prediction model. We also demonstrate that, under the convergence conditions of the learning algorithms, the prediction results from the abstract models are the same as the concrete models.

Finally, the usefulness of $\mathcal{P}revent$ is indicated in real-world applications by showing how it can be applied for predicting rare properties, properties with very low but non-zero probability of satisfaction. More specifically, we adjust the training algorithm that uses the samples generated by importance sampling to generate the prediction models for rare properties without increasing the number of samples and without having a negative impact on the prediction accuracy.

Acknowledgements

First and foremost, I sincerely thank Derek Rayside for his solid support, thoughtful mentorship, and his genuine trust in myself and my research during my PhD. I also thank Vijay Ganesh for his clear vision on my research, his great support and amazing mentorship towards the end of my studies. I appreciate Sean Sedwards for his constructive feedback on my thesis, as well as the collaboration and the discussions that we had, the result of which helped me write most of Chapter 6. I thank the committee members for their great feedback, comments, and recommendations.

I would also like to extend a special thank-you to Sebastian Fischmeister for his generous help and support until the end of my PhD, and Arie Gurfinkel for his in-depth insights on my research and helping me to develop the skills to become a better researcher.

I thank my beloved mother, who raised me, my sister and brother, single-handedly with all her capacity, and lead us to become who we are right now. I specially thank my sister for her complete emotional support during my PhD, in such a way that I feel a strong bond with her like never before. I thank my brother for his insights and the occasional deep conversations that we had about my studies.

I thank all of my amazing friends for their wholehearted support in the ways that they could, especially, Allyson Roberts, Jeremy Roberts, Reinier Torres, Mailén Fong, Amy Wallace, Melissa Melick, Ali Mostolizadeh, Pooneh Torabian, Ali Sarhadi, Behrouz Semnani, Meisam Shahrbaf, Ehsan Asadi, Anis Sharafoddini, Mansour Ataei, Nazi Pakpoor, Mehdi Akbari, Mohammad Ghaziaskar, Hanie Masjedian, Mohammad Al-Sharman. I thank all other people who came across my way during my studies and nudged me one way or another towards completing my PhD.

I have to extend a special thank-you to the current and former staff members in the department of ECE: Sarah Landy, Susan Widdifield, Cassandra Brett, Brenda McQuarrie, whose generous help never stopped during my studies. I would also like to thank the amazing janitors in our building, E5, particularly Elena and Tony whose words, wisdom, and comfort helped me stay a bit more motivated and on-the-track in those long nights of doing research. I also thank the staff members at Williams whose smiles and service, combined with their great signature coffee, definitely contributed to my PhD.

Dedication

Dedicated to Marzieh, my sister, for her most sincere, wholehearted, and tremendous love and support.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The prevalence of digital computing in every aspect of life is undeniable. From cell phones to cloud-based services, computing systems are becoming ubiquitous in various forms including Cyber-Physical Systems (CPSs). CPSs are used in the applications that require a high level of reliability (e.g., aerospace, automative, and healthcare). Due to the intertwined nature of software, hardware, and the physical components in such systems, they exhibit complex and distinctive modalities at runtime.

With an exponentially increasing complexity of computing systems, achieving such reliability, particularly in safety- and mission-critical systems, is difficult. A miscalibrated sensor input to the system may result in software to lead an airplane to nose-dive and crash [125]. An overflow can cause the software to never issue a required command to activate a hardware module to operate an actuator, and lead an entire project to a disaster [17].

Testing [88] is perhaps the most accessible method to verify the behaviour of a system, where the inputs and the desired outputs are provided and checked against the generated outputs. The coverage of testing depends on the inputs and how much they cover the state space of the system. For a system with limited information about its internal mechanism, i.e., gray- or black-box systems, the coverage of testing is difficult to be measured, which makes it an incomplete approach to verify such systems.

Model checking [8] provides an exhaustive verification of a mathematical model of the system, typically in the form of a finite transition machine. Model checking provides a higher level of guarantee when verifying a system; however, it suffers from the state-explosion problem, where the number of states in the model becomes so large that the state space can not be stored in the memory or effectively explored. Further, in many cases, a model of the system that represents all possible states of the system is infeasible to achieve.

Runtime Verification (RV) [75] is proposed as a middle ground that compromises between the practicality of testing and the thoroughness of model checking. RV is the study and analysis of the execution artifacts of software [77], where instead of the entire model of the system, only the current execution is verified at runtime against some properties. RV has been applied in a variety of systems, from CPSs [84] to biological systems [33].

Amongst all, RV [75] has become a particularly important element in monitoring and analysing stochastic systems [77, 117], where they represent a wide range of systems with randomness involved in one or more of their constituent components. Although the more anticipated failures are detected using design-time verification techniques (e.g., model checking); in practice there is almost surely a non-zero probability of failure at runtime.

In RV, a monitor checks the execution against a given property and produces a verdict at runtime. The execution is considered the finite prefix of an infinite execution path. The given property, typically expressed in Linear Temporal Logic (LTL) [97], represents a set of acceptable infinite paths. If all the infinite extensions of the prefix belong (do not belong) to the set of acceptable infinite paths, i.e., they satisfy the property, the monitor accepts (respectively rejects) the prefix. However, if the monitor is not able to reach a verdict with the given prefix because it can be extended to both infinite paths that satisfy and violate the property, the monitor outputs *unknown* [13].

As an example consider the property *eventually an error occurs* (*always there is no error*), captured in LTL with $\Diamond e$ (respectively $\Box\neg e$). This property is satisfied (respectively not satisfied) on any infinite paths with the prefix $u_1 = \neg e \neg e e$, in which case the monitor outputs *satisfied* (respectively *violated*). On the other hand, the prefix $u_2 = \neg e \neg e$ can be extended to both a path that satisfies the formula (e.g., any extension of u_1) and a path that violates it (e.g., $(\neg e)^\omega$), in which case the monitor outputs *unknown*.

The challenge of inconclusive results to evaluate LTL properties over finite paths is initially discussed in [79], and several suggestions based on modifying the semantics of LTL are proposed, such as FLTL [79], LTL$^{\mp}$ [41], LTL$_3$ [12], RV-LTL (LTL$_4$) [14], and RV$^\infty$-LTL [86]. In all of the aforementioned semantics, the evaluation of the LTL property is assumed to be happening only on the current execution (the finite prefix). In [135] the authors introduce a predictive semantics definition based on LTL$_3$ such that, besides the current execution u, a finite suffix v is predicted from the program's Control Flow Graph (CFG), and the evaluation occurs on the path uv.

In this dissertation, we propose a *predictive runtime verification* framework, called $\mathcal{P}$*revent*, in which the monitor calculates the probability of all the finite extensions of the prefix, i.e., the current execution, that satisfy or violate a *monitorable* property using a prediction model. A property is non-monitorable [43] if and only if the monitor always produces *unknown* regardless of the prefix. Therefore, if the property is monitorable, the monitor is able to reach a verdict with a finite extension of the prefix. In our framework the LTL semantics does not change, and the prediction model is built from independent and identically distributed (*iid*) samples of executions of the system using statistical learning methods, which makes $\mathcal{P}$*revent* more suitable for black- or gray-box systems.

In contrast to the traditional RV frameworks where only the violations are detected at the time of occurrence, in our proposed framework, the violation of a property is predicted in advance, hence, there is a wider range of actions to be taken to correct the behavior of the system. Although, devising and taking a corrective action is not in the scope of this thesis, in the next section we roughly explain how $\mathcal{P}$*revent* can be used in a scenario adopted from real-world to avoid a failure and increase the reliability of a CPS.

1.1 Motivating Example

In October 2018 and March 2019 two Boeing 737 MAX airplanes crashed shortly after departure [125]. Further analysis revealed a series of design decisions that led the airplanes into those fatal incidents. One key issue was within the Maneuvering Characteristics Augmentation System (MCAS), a control system developed to compensate for the risk of stalling due to the larger engines used in this model. The MCAS task is to read the direction of the nose measured by a sensor, and change the tail flaps accordingly to adjust the nose and prevent the airplane from stalling (see Figure 1.1).

One of the factors that played a key role in causing the crashes was that the MCAS upon receiving wrong data from a faulty sensor kept issuing the commands to change the tail flaps and led the airplane into a nose-diving situation. Since the MCAS was implemented in the flight management system its commands could not be overridden by the pilot, which made the situation worse. This is a perfect example of how even a highly reliable system can still malfunction due to incorrect inputs (in this case due to a faulty sensor).

Although the cause of the crashes was multi-factor and complicated; there are various proposals to resolve it [124, 3]; incorporating a runtime monitoring framework with predictive capacity might reduce the risk of such disasters. Notice that such a runtime monitoring system verifies the behaviour of a controller (e.g., an MCAS), which demonstrably is a crucial difference between our approach, a verification mechanism, and other seemingly similar approaches such as Model Predictive Control [103], a controlling mecha-

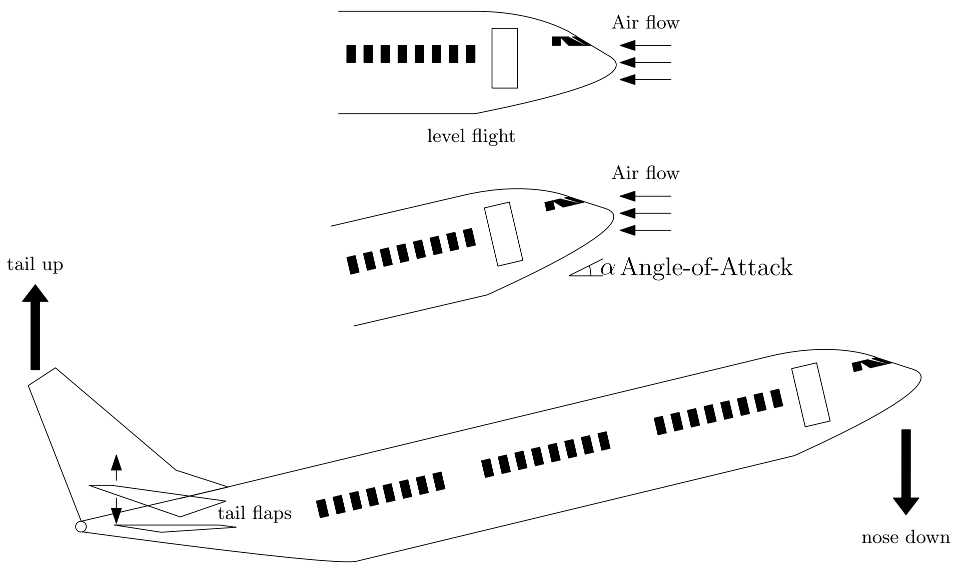


Figure 1.1: The MCAS mechanism to prevent the airplane from stalling.

nism. In other words, our framework is a verification layer that can be inserted on top of any controller to cross-check its outputs at runtime based on some specification properties.

As an example, a predictive monitor for MCAS can be implemented to observe three indicators and decides if the controller's output is going to deviate from the desired behaviour. If a deviation is deemed to be likely, a corrective action can be taken. For instance, we can simply disengage the MCAS and alert the pilots so that they can take over the control of the tail flaps.

The monitor can combine the data from other sensors (by means of sensor fusion [47, 3]), and measure the change of the altitude or the vertical speed to determine if the vertical fluctuations of the airplane are within a safe envelope. In addition, the monitor is able to observe the output commands of the MCAS and predicts if the sequence of the commands issued by the system is likely to lead the airplane into a dangerous nose-dive situation, by checking the execution of the MCAS against the following property:

$$\Box\neg(\alpha < \mathit{threshold} \land \texttt{down}) \tag{1.1}$$

where α is the angle of the airflow over the wings, so-called Angle-of-Attack, which is obtained from various sensors, *threshold* indicates the threshold for which the nose is allowed to be negative (down) and the plane is not considered in a risky position, and `down` is the command issued by MCAS to drive the nose down. Property (1.1) is a safety property. We can similarly develop a guarantee property that specifies that the MCAS will eventually drive the airplane in a level flight and prevents the plane from stalling.

In this thesis we introduce $\mathcal{P}$*revent*, a framework in which a predictive monitor is constructed for each property such that the monitor is able to predict if the sequence of the output commands by a system such as the MCAS is going to lead to the violation of a safety or the satisfaction of a guarantee property. Using the result of the prediction the monitoring system can detect a hazardous situation and issue a corrective action based on the severity of the risk. In our example, the corrective action is simply disengaging the MCAS and alerting the pilots, but a more sophisticated adaptive system could be implemented in the case of an likely crash or stalling. The mechanism to use the prediction results for devising and taking the corrective actions is beyond the scope of this dissertation.

1.2 Related Work

In this section we provide the related work to the prediction runtime verification in general, and leave the specific research related to each part of the framework to the chapter dedicated to it.

With a growing interest in RV, assorted tools such as JPAX [50], MOP [26], Java-MaC [66], Copilot [95], are developed to apply RV within various industrial applications. These tools, however, focus more on verifying non-probabilistic properties on non-stochastic systems. Consequently, the output of the monitor is a discrete value (typically from some Boolean lattice) that shows given the current execution if the property is satisfied or not.

Modifications to such tools such as [109, 110, 129], or brand-new tools such as Eagle [10], Temporal Rover [37], ProMon [46, 136] propose RV tools that allow for a probabilistic system to be verified against probabilistic properties. The interpretation of the probability varies in the literature. In [129], for instance, using a probabilistic model with hidden states (similar to a prediction model in $\mathcal{P}$*revent*) is justified by the presence of faults and degrading hardware. $\mathcal{P}$*revent* uses the probabilistic models for prediction on the stochastic black- or gray-box systems, which is not offered in any of these works.

A few predictive RV methods are proposed for white-box systems [133, 135, 74] using the Control Flow Graph (CFG) of a program to generate the extensions of an execution. While some of the ideas in these approaches can be also applied to our framework, for example, using event-triggered monitors or considering only the calculations of local variables in each method call [135]; $\mathcal{P}$*revent* is specifically developed for black- and gray-box systems, where the future behaviour of the system must be approximated by only observing the sampled executions. Also, even for a white-box system, although using CFG provides additional information for the monitor at runtime and potentially makes its prediction more accurate; the analysis of the CFG involves graph analysis algorithms which impose space and time overhead at runtime and are not practically useful for embedded systems [133]. A key insight in the development of $\mathcal{P}revent$ is to minimize the time and space overhead of the monitor at runtime.

Srinivas Pinisetty *et al.* [96] propose a predictive RV approach that fits the black- or gray-box system better: a prior knowledge of the system is provided in the form of a timed property, which the monitor uses to predict the satisfaction or violation of a timed property, and the minimum time instant that it will happen. The prior knowledge in $\mathcal{P}revent$ is obtained by training a prediction model from the execution samples, which can also be used to extract timed properties that specify the behaviour of the system [30, 5].

RV with state estimation (RVSE) [120] uses an HMM to fill the gaps that are caused by an imperfect sampling in the traces. The gaps are added to the traces during instrumentation of the program, and with a distribution on the length of the gap, RVSE gives an estimation of how probable an LTL property is satisfied over the incomplete trace. The idea of incomplete or noisy traces is orthogonal to the idea of $\mathcal{P}revent$; however, the assumption in the current thesis is that the sampling is perfect, i.e., there is no data loss or noise in the collected training samples.

To the best of our knowledge, prediction of the future behaviour of a stochastic system and verifying a property with respect to the execution extensions at runtime is not explored in any of the discussed works. More specifically, the main novelty of this thesis is the development of a predictive runtime verification framework for a black- or gray-box stochastic system, which is based on predicting the extensions of an execution path using a trained prediction model.

1.3 Formulating the Research Problems

In building the predictive runtime verification framework, we address three research problems:

1.3.1 Predictive Monitor

Probabilistic language of finite strings is used to model the executions of a stochastic system. More specifically, the probabilistic language model is a probability space over all the finite executions and the probability measure function that maps each execution to its probability. The probabilistic language model is used by the monitor as the prediction model to extend the execution paths at runtime.

Within the context of black- or gray-box systems, the probabilistic language model is not available. The first research problem that is addressed in this dissertation is *how to construct a monitor that uses the trained model to predict the satisfaction or violation of a monitorable property.*

The predictive monitor is constructed with two key insights that are important to consider for systems with limited resources at runtime: (1) the overhead of the prediction must be as low as possible at runtime, (2) using the prediction model should not have a significant impact on the size of the monitor. Since the monitor construction occurs offline, the cost of construction itself is not crucial.

Addressing these two challenges leads us to the second research problem.

1.3.2 The Size of the Prediction Model

Both the time and space overhead of the monitor relies on the size of the prediction model.

Monitor construction in $\mathcal{P}$*revent* specifically relies on building a lookup table that contains the probability of satisfying or violating a property from each state of the prediction model. Hence, the size of the table is directly determined by the size of the prediction model, which needs to be stored with the monitor, which in turn, directly impacts the space usage of the monitor.

The state of the system is determined at runtime by estimating the state of the prediction model. The state estimation techniques at best have a linear complexity in the number of the states. Hence, having a smaller prediction model decreases the execution time of the state estimation, and consequently, the runtime overhead of the monitor.

Reducing the size of the prediction model should not significantly affect the accuracy of the predictions. Therefore, the second research question is *how to obtain a small prediction model, i.e., with fewer states, from a set of execution samples without significantly compromising the prediction accuracy.*

1.3.3 Predicting Rare-Events

Given that real-world software systems are thoroughly verified at design-time, the properties that are checked at runtime have very low probability of violation, what we call *rare properties*. Training the prediction models for rare properties requires a large number of samples. In the context of machine learning, the size of training sample to successfully train a target model is referred to as the *sample complexity* [108]. Having a large sample complexity leads to a slow training process and an excessive memory usage, which is impractical to handle during training a model.

Therefore, in the third research problem we address *how to build a prediction model for rare properties without increasing the sample complexity.*

1.4 Summary of Contributions

The current dissertation makes the following contributions in an attempt to address each research question raised in Section 1.3.

1.4.1 Constructing Predictive Monitor using Statistical Learning

First, we formulate the notion of predictability of an LTL property within the context of stochastic systems, and generalize some related theories in the literature to discuss the theoretical limits of predictive monitoring. The remainder of the dissertation assumes that these limits hold.

Second, we use statistical learning methods to construct a prediction model from the execution samples. My focus here is only on Markovian models, with a discussion on the convergence of the learning algorithms in the limit, i.e., with large enough number of samples.

Third, we show how to use the prediction model to construct a predictive monitor that extracts the probability of the extensions that satisfy or violate a given specification property. Depending on the property, the predicted extensions may specify the prefixes that satisfy the property (*good extensions*) or violate it (*bad extensions*). Since the property is assumed to be monitorable [43], only the finite extensions of the execution path are considered. In addition, the length of the finite extensions is bounded with a maximum length, which demonstrates the horizon of the prediction.

Finally, we discuss the monitoring process given the prediction model, the state estimation algorithms, and two approaches to evaluate the prediction of a monitor: one with knowing the true model of the system, and the other without any knowledge about the system and just via hypothesis testing. we show the feasibility of $\mathcal{P}revent$ on two case studies.

1.4.2 Trace-Level Abstraction

To reduce the size of the prediction model, we define a trace-level abstraction using projection. The projection relation maps symmetrical symbols with respect to the probability of satisfying a given property to a single symbol, and thus reducing the size of the observation space.

We apply the k-gapped pair model and hypothesis testing to infer the symmetry relation in the observation space. The prediction model is then trained on the abstract traces, that contain fewer symbols, and hopefully has a smaller size.

The prediction model trained from abstract traces, called an abstract model, can be deterministic or nondeterministic depending on the abstraction. If the abstract model is deterministic that means that the state of the model can be exactly determined by just using the observation. In this case, we show that in the limit the prediction by the abstract model is identical to the prediction by the concrete model. If the model is nondeterministic, i.e., the system could be at more than one states, we resolve the nondeterminism by using a hidden state variable during the training, which also in the limit converges to the actual model and produces the same prediction results. Training the nondeterministic models takes the number of hidden states as an input, which can be chosen in a way that reduces the size of the prediction model without significantly impacting the accuracy of the predictions.

We show the effectiveness of $\mathcal{P}revent$ on a stochastic distributed system in which the size of the prediction model is reduced without any impact on the prediction accuracy.

1.4.3 Training with Importance Sampling

We use the samples generated by Importance Sampling (IS) to train the prediction model for a rare property, i.e., a property with very low satisfaction or violation probability. The samples generated using IS are not necessarily *iid* but follow a distribution that increases the likelihood of the events that satisfy or violate a property, hence, requires fewer samples compared to the original distribution.

Since the distribution of the samples obtained from the IS is skewed, the training algorithm needs to be modified so that the final probabilities reflect the actual probability distributions. We achieve this goal by introducing a weight for each sample that is calculated based on the likelihood ratio of the original probability distribution and the IS distribution.

The usefulness of $\mathcal{P}$*revent* is demonstrated through a series of experiments, in which we show how training with IS reduces the sample complexity of training a prediction model for a rare property.

1.4.4 Thesis Statement

In summary, in this dissertation we provide evidence for the following statement:

Thesis. *Predictive runtime verification of a stochastic system with respect to a monitorable property is feasible, effective, and useful.*

1.5 Overview of the Dissertation

The remaining of this dissertation is organized as follows. Chapter 2 provides the necessary definitions and notations that are used throughout the dissertation. Chapter 3 defines the notion of predictability, predictive monitoring, and the theoretical circumstances under which predictive monitoring is possible. Then the formal definition of the two types of Markovian models, the Discrete-Time Markov Chain (DTMC) and the Hidden Markov Model (HMM) used as prediction models, are provided. The remainder of the chapter lays out a schematic description of the framework architecture and its building blocks, each of which is fully described in the subsequent chapters. We conclude the chapter with a running example that is used in all the following chapters, and reviewing the related work to predictive RV.

Chapters 4, 5 and 6 are each dedicated to one of the claims in the thesis.

Chapter 4 supports the *feasibility* claim by explaining the learning techniques to construct a prediction model. The prediction is described as a quantitative bounded reachability analysis, and how the results are used to construct a monitor as a lookup table. The runtime verification process, which involves state estimation and retrieving the prediction probability from the lookup table, is then discussed, and at the end a measure to evaluate the prediction accuracy is defined with some experimental results.

Chapter 5 establishes the *effectiveness* claim by introducing the trace-level abstraction and its inference using k-gapped pair model. Statistical t-test is used to extract the symbols that have the same prediction power, and HMMs are applied to handle the nondeterminism introduced by the abstraction. The inferred abstraction is then used to generate abstract traces that are used to construct an abstract prediction model.

Chapter 6 provides support for the *usefulness* claim by applying importance sampling to generate more samples for a rare property. A modification of the state-merging algorithm used to train a DTMC is introduced to compensate for the perturbation of the sample distribution generated by IS. A comparison between the results of IS samples and simple Monte-Carlo is also provided in the experiments.

At the end, Chapter 7 summarizes the dissertation with some future directions.

Notice that for clarity purposes, the related work is provided within each chapter, as the context of each chapter is widely different from that of the other ones.

Chapter 2

Preliminaries

In this chapter, we briefly introduce definitions and notations that are necessary for the following chapters in the dissertation.

A probability distribution over a finite set S is a function $Pr : S \to [0,1]$ such that $\sum_{s \in S} Pr(s) = 1$. We use u and w to denote a finite and an infinite sequence, respectively. The i^{th} element in the sequence w is shown as $w^{(i)}$. We use $u^{(1:n)}$ to abbreviate $u^{(1)}u^{(2)} \dots u^{(n)}$. The set of all the finite and infinite paths over the alphabet Σ is shown as Σ^* and Σ^ω, respectively ($\Sigma^+ = \Sigma^* \cdot \Sigma$). Notice that we use the terms *sequence* and *path* interchangeably.

2.1 The Syntax and Semantics of LTL

Linear Temporal Logic (LTL) [97] is used to specify properties with modalities referring to linear time. Definition 1 describes the syntax of LTL:

Definition 1 (The Syntax of LTL [79]). *The set of LTL formulas, shown by* LTL*, over a finite set of atomic propositions denoted by* Σ *is defined by the following grammar:*

$$\varphi = \Sigma \,|\, \neg\varphi \,|\, \varphi \vee \varphi \,|\, \mathsf{X}\,\varphi \,|\, \varphi\,\mathcal{U}\,\varphi$$

In addition, we use $\varphi \wedge \psi$, $\Diamond\varphi$, $\Box\varphi$, and $\varphi \to \psi$ as abbreviations for, $\neg(\neg\varphi \vee \neg\psi)$, *true* $\mathcal{U}$ φ, $\neg\Diamond\neg\varphi$, and $\neg\varphi \vee \psi$, respectively.

The semantics of LTL over infinite paths is defined as follows:

Definition 2 (The Semantics of LTL [79]). *Let w be an infinite path in Σ^ω for a given set of atomic propositions, Σ. The semantics of an LTL formula is given by the function $[\_ \models \_] : \Sigma^\omega \times \mathsf{LTL} \to \mathbb{B}_2 = \{\top, \bot\}$ as follows:*

$$
\begin{aligned}
[w \models p] &= \begin{cases} \top & \textit{if } p \in w^{(1)} \\ \bot & \textit{else} \end{cases} \quad p \in \Sigma \\
[w \models \neg\varphi] &= \overline{[w \models \varphi]} \\
[w \models \varphi \land \psi] &= [w \models \varphi] \textit{ and } [w \models \psi] \\
[w \models \varphi \lor \psi] &= [w \models \varphi] \textit{ or } [w \models \psi] \\
[w \models \mathsf{X}\varphi] &= [w^{(2)} \models \varphi] \\
[w \models \varphi\, \mathcal{U}\, \psi] &= \exists t \geq 1, [w^{(t)} \models \psi], \textit{ and, } \forall 1 \leq t' < t, [w^{(t')} \models \varphi]
\end{aligned}
$$

We use $L_\varphi = \{w \in \Sigma^\omega : [w \models \varphi] = \top\}$ to denote the set of strings that satisfy φ according to the LTL semantics in Definition 2.

As an example, with $\Sigma = \{r, s, t\}$ the evaluations of $r \lor \mathsf{X}\, t$ and $\Box\, t$ are both $\top$ on $w = (t, t \ldots)$; whereas $\Diamond\, s$ is not satisfied on w, i.e., $[w \models \Diamond\, s] = \bot$.

2.2 Monitorability

We use the notions introduced in [15] to define monitorability.

Definition 3 (Good, Bad, and Ugly Prefixes [13]). *Let φ be an LTL property. A finite prefix $u \in \Sigma^*$ is defined a good (bad) prefix w.r.t. φ iff $\forall w \in \Sigma^\omega, uw \models \varphi = \top$ ($\bot$ respectively). A prefix that is neither bad or good is ugly.*

Using the notion of an ugly prefix we define the notion of monitorability:

Definition 4 (Monitorability [15]). *An LTL property φ is defined to be monitorable iff $\forall u \in \Sigma^*, u$ is not an ugly prefix for φ.*

Checking the monitorability of a property is EXPSPACE [91]. Nevertheless, it is known that *guarantee* and *safety* properties are monitorable [44]. A guarantee (safety) property is always accepted (respectively, refuted) by a *good* (respectively, *bad*) prefix [15]. Throughout this dissertation we consider only guarantee and safety LTL properties.

We use a Deterministic-Finite Automata (DFA) to specify the prefixes that specify the pattern of guarantee and safety properties [44].

Definition 5 (DFA)**.** *A Deterministic Finite Automaton (DFA) is a tuple* $\mathcal{A} : (Q, \Sigma, \delta, q_I, F)$*, where* Q *is a set of finite states,* Σ *is a finite alphabet,* $\delta : Q \times \Sigma \to Q$ *is a transition function determining the next state for a given state and symbol in the alphabet,* $q_I \in Q$ *is the initial state, and* $F \subseteq Q$ *is the set of final states.*

We use $L_{\mathcal{A}} \subseteq \Sigma^*$ to denote the set of all the finite sequences accepted by the DFA $\mathcal{A}$. In the remainder of this work we refer to the specification as an LTL property or a DFA, interchangeably. Notice that there are safety or guarantee properties that are not recognizable by a DFA, the so-called *non-regular* properties [8] (e.g., require a push-down automaton to specify the prefixes), but we do not consider such properties in this dissertation.

2.3 Stochastic System

A stochastic system is defined by the notion of probabilistic languages, where there is a set of sequences and a probability associated with each sequence.

Definition 6 (Stochastic System)**.** *A stochastic system* $\mathcal{S}$ *is defined as the tuple* $(\mathcal{L}, \mathcal{F}, \mathcal{P})$*, which represents a probabilistic space on the set of sequences, such that* $\mathcal{L} \subseteq \Sigma^* \cup \Sigma^\omega$ *(*Σ *is the output alphabet),* $\mathcal{F} \subseteq 2^{\mathcal{L}}$ *is a* $\sigma-$*algebra, and* $\mathcal{P} : \mathcal{F} \to [0, 1] \in \mathbb{R}$ *is a probability measure.*

We assume that $\mathcal{P}$ is computable. We also use the symbol S to denote the set of all the stochastic systems.

The sequence $w^{(1)}w^{(2)} \dots$ is an execution path of the stochastic system $\mathcal{S}$ iff $\mathcal{P}(w^{(1)}w^{(2)} \dots) > 0$. An execution path at runtime is a prefix, i.e., a finite sequence, of a possibly infinite path. We denote by $Path_k(o)$ the set of all finite execution paths of length k that start with the symbol o. The set of all the finite execution paths of the system $\mathcal{S}$ of an arbitrary length is denoted by $Path(\mathcal{S})$.

Chapter 3

An Overview of Predictive Runtime Verification

In this chapter we formally define the notion of *predictability* and the conditions under which we consider a system predictable (Section 3.1). We also formally state the notion of a *predictive monitor* given that the system is predictable (Section 3.2), and define the Markovian models that are used as the prediction models (Section 3.3). We then propose a framework, called $\mathcal{P}revent$, that realizes the notion of predictive monitoring, and briefly describe its components (Section 3.4). At the end, we present the running example, that is used in the subsequent chapters (Section 3.5), and the literature review (Section 3.6).

3.1 Predictability

In this section, we formalize the notion of predictability. We first define the finite extension of a path that needs to be predicted.

Definition 7 (Finite Extensions of a Path)**.** *Let $u = u^{(1..n)}$ be a finite execution path of a stochastic system. The set of finite extensions of u is denoted by Ext_u and defined as follows:*

$$Ext_u = \{u \in Path_k(u^{(n)}) | k \geq 0\}$$

We use the notation $Ext_u^{\leq h}$, if $k \leq h$ in Definition 7.

In predictive runtime verification, we extend the notions of a *good*, *bad*, and *ugly* prefix (Definition 8) to the extensions of a prefix.

Definition 8 (Good, Bad, and Ugly Extensions)**.** *Let φ be an LTL property, and $v \in Ext_u$ be an extension to the finite path $u \in \Sigma^*$. Extension v is defined a good (bad) extension of u w.r.t. φ iff $\forall w \in \Sigma^\omega, uvw \models \varphi = \top$ ($\bot$ respectively). If v is neither bad nor good, it is an ugly extension.*

Trivially, any extension of a good (bad) prefix is a good (bad respectively) extension.

Lemma 1. *Let u be a good (bad) prefix for the LTL property φ. Then $\forall v \in Ext_u$, v is a good (bad respectively) extension of u w.r.t. φ.*

To make the prediction, we need to distinguish the good or the bad extensions, and compute the probability of them.

Definition 9 (Set of Satisfying or Refuting Extensions)**.** *We define $\mathcal{C}_{(u,\varphi)}$ the set of satisfying (refuting) extensions of u w.r.t. the LTL property φ, such that $\forall v \in \mathcal{C}_{(u,\varphi)}$ iff v is a good (bad, respectively) extension.*

In other words, $\mathcal{C}_{(u,\varphi)}$ in Definition 9 is strictly the set of all the extensions of u generated by the system that satisfy (violate) property φ. We denote $\mathcal{C}^{\leq h}_{(u,\varphi)}$ iff the length of the extensions is bounded by some non-zero positive integer h, i.e., the extensions are selected from $Ext^{\leq h}_u$.

Next we define the notion of predictability.

Definition 10 (Predictability)**.** *A stochastic system $\mathcal{S}$ is called predictable w.r.t. an LTL formula φ iff $\forall u \in Path(\mathcal{S}), \mathcal{C}_{(u,\varphi)} \in \mathcal{F}$ exists and is measurable.*

Clearly a property that is non-monitorable (Definition 4) is also non-predictable, regardless of the behaviour of the stochastic system.

Lemma 2. *A stochastic system $\mathcal{S}$ is non-predictable w.r.t. an LTL property if it is non-monitorable.*

Suppose that $\mathcal{S}$ is the stochastic system such that $\mathcal{F}$ is defined based on the *cylinder sets*, the set of sequences that have a common finite prefix [18]. Notice that the cylinder sets are measurable by defining a probability measure $\mathcal{P}$ such that the probability of each sequence in the set is defined based on the probability of the common prefix. It is straightforward to demonstrate that any bounded finite extension of an execution path of $\mathcal{S}$ is also measurable.

Theorem 1. *Let $\mathcal{S} = (\mathcal{L}, \mathcal{F}, \mathcal{P})$ be a stochastic system, where $\mathcal{F}$ is defined using some cylinder sets measurable by $\mathcal{P}$. Then, for any integer $h > 0$, $\mathcal{C}^{\leq h}_{(u,\varphi)} \in \mathcal{F}$, is also measurable by $\mathcal{P}$.*

Proof. The proof directly follows from the fact that $\mathcal{C}^{\leq h}_{(u,\varphi)}$ is a finite set, hence, its measure can be simply defined using the measures of the cylinder sets that comprise the finite sequences in $\mathcal{C}^{\leq h}_{(u,\varphi)}$. □

If a stochastic system $\mathcal{S}$ is representable by a probabilistic finite-state machine, we can show that it is predictable with respect to any guarantee or safety properties.

Theorem 2. *Let $\mathcal{S} = (\mathcal{L}, \mathcal{F}, \mathcal{P})$ be a stochastic system, that can be represented by a probabilistic finite-state machine, and φ be a guarantee or safety property. Then $\mathcal{S}$ is predictable with respect to φ.*

Proof. It suffices to find an integer $h > 0$ such that $\mathcal{C}^{\leq h}_{(u,\varphi)} \in \mathcal{F}$ (Theorem 1). Since $\mathcal{S}$ is a finite-state machine, h exists and is bounded to the number of states. □

Notice that according to the non-universality problem of probabilistic finite state automata determining that a stochastic system $\mathcal{S}$ is predictable with respect to an LTL formula in general is undecidable [117].

In the remainder of the dissertation we consider predicting guarantee and safety properties, which are monitorable, and use probabilistic finite-state machines to model stochastic systems, thus, holding the predictability conditions.

3.2 Predictive Monitoring

In this work, we realize predictive runtime verification through constructing a predictive monitor. Given that the system is predictable, we define the output of a predictive monitor to be a real value in $[0, 1]$ that demonstrates the probability of $\mathcal{C}^{\leq h}_{(u,\varphi)}$, i.e., the set of satisfying or refuting extensions of u, and of length at most h, with respect to the guarantee or safety property φ.

Definition 11 (Predictive Monitor)**.** *Let $u \in \Sigma^*$ be an execution path of a predictable stochastic system $\mathcal{S}$ w.r.t. φ, a guarantee or safety LTL formula. A predictive monitor is a function $\mathcal{PM}(\_,\_,\_)$: $\Sigma^* \times \mathsf{S} \times \mathsf{LTL} \to [0, 1]$:*

$$\mathcal{PM}(u, \mathcal{S}, \varphi) = \mathcal{P}(\mathcal{C}_{(u,\varphi)}) \tag{3.1}$$

As an example, consider the following stochastic language models for the executions of three hypothetical stochastic systems: $\mathcal{S}_1 = (\Sigma^*, \{\Sigma^+\}, 2^{-|u|})$, $\mathcal{S}_2 = (\Sigma^*, \{p^+\}, 2^{-|u|})$, $\mathcal{S}_3 = (\Sigma^*, \{pq^+\}, 2^{-|u|})$, where, $\Sigma = \{p, q\}$; $\{\Sigma^+\}, \{p^+\}, \{pq^+\}$ are the *trivial partition* of Σ^+, p^+, and pq^+, respectively; and $|u|$ is the length of the string u in each partition. Suppose $\varphi_1 = \Diamond q$, and that the system has generated $u_1 = (p)$. According to Definition 11, the output of the predictive monitor for φ_1 on u_1 using $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ as prediction models are $1/2 + 1/4 + \cdots = 1$, 0 and $1/4 + 1/8 + \cdots = 1/2$, respectively.

For a probabilistic finite-state machine, (3.1) is defined over bounded finite extensions (see Theorem 2). In practice, however, this upper bound is unknown, e.g., for a black-box system, and needs to be provided as an input by the engineer. Therefore, the predictive monitor takes an extra input, an integer $h > 0$, we call the *prediction horizon*, and, (3.1) changes to $\mathcal{PM}(u, \mathcal{S}, \varphi, h) = \mathcal{P}(\mathcal{C}^{\leq h}_{(u,\varphi)})$.

3.3 Prediction Model

The full model of the stochastic system within the context of RV is often unknown. However, we typically are able to collect execution traces from the system. As a result, we can generate an arbitrary number of execution samples and build a prediction model that describes the behaviour of the system precise enough for prediction purposes.

The prediction models in the current dissertation are finite probabilistic state machines, represented as Markovian models. We assume that the state space is not directly observable, but an execution path consists of the symbols from a known finite alphabet, we call the observation space. We also assume that the observation space is known and is somehow related to the state space. We focus on two specific models that represent the relation between the state space and the observation space in two distinct ways: Discrete-Time Markov Chain (DTMC) and Hidden Markov Model (HMM).

3.3.1 Discrete-Time Markov Chain

A DTMC is a simple probabilistic finite-state transition system that describes a wide range of systems. The observation space in a DTMC is related to the state space via a function.

Definition 12 (DTMC)**.** *A Discrete-Time Markov Chain (DTMC) is a tuple $\mathcal{M} = (S, \Sigma, \pi, \mathbf{P}, L)$, where S is a non-empty finite set of states, Σ is a non-empty finite alphabet, $\pi : S \to [0, 1]$ is the initial probability distribution over S, $\mathbf{P} : S \times S \to [0, 1]$ is the* *transition probability, such that for any $s \in S$, $P(s, \cdot)$ is a probability distribution, and $L : S \to \Sigma$ is the labelling function.*

We denote $[s \models \varphi] = \top$ $(= \bot)$ for some $s \in S$ and LTL property φ iff $u = u^{(0)}u^{(1)} \cdots u^{(n)} \in L_\varphi$ (respectively $\notin L_\varphi$) for some $n > 0$ such that $L(s) = u^{(0)}$. Notice that the property is defined on the observation space, i.e., Σ, not the state space. We use $Path_k(u^{(i)})$ to denote the execution paths of length k that start with $u^{(i)}$.

Given a DTMC, the sequence of states are derivable from the execution path, i.e., the sequence of labels, if the DTMC is deterministic. A deterministic DTMC is a DTMC that can be in at most one state after an observation sequence u.

Definition 13 (Deterministic DTMC [80])**.** *A DTMC is deterministic iff:*

- *There exists $s_{init} \in S$ such that $\pi(s_{init}) = 1$.*
- *For all $s \in S$ and $\sigma \in \Sigma$, there exists at most one $s' \in S$ such that $\mathbf{P}(s, s') > 0$ and $L(s') = \sigma$.*

Therefore, a deterministic DTMC requires no state estimation, as the state is determined by the observation path u.

3.3.2 Hidden Markov Model

HMMs are similar to DTMCs except they enable us to model nondeterminism (See Chapter 5) by relating the observation space to the state space via a probability distribution (Definition 14). More particularly, an HMM describes the distribution of a sequence as the joint distribution of two random variables: the hidden state variable X and the observation variable Y. The joint distribution is such that $Pr(Y^{(i)} = o \mid X^{(1:i)}, Y^{(1:i)}) = Pr(Y^{(i)} = o \mid X^{(i)} = s)$ for $i \in [1, n]$, i.e., the probability of the current observation o is conditioned only on the current state s, and $Pr(X^{(i)} = s \mid X^{(1:i-1)}, Y^{(1:i-1)}) = Pr(X^{(i)} = s \mid X^{(i-1)})$ for $i \in [1, n]$ i.e., the probability of the current state s is only conditioned on the previous hidden state. We use π to denote the initial probability distribution over the state space, i.e., $Pr(X^{(1)}) = \pi(X^{(1)})$. As a result, an HMM can be defined with three distributions:

Definition 14. *A finite discrete Hidden Markov Model (HMM) is a tuple $\mathcal{H} : (S, \Sigma, \pi, T, O)$, where S is the non-empty finite set of states, Σ is the non-empty finite set of observations, $\pi : S \to [0, 1]$ is the initial probability distribution over S, $T : S \times S \to [0, 1]$ is the transition probability, and $O : S \times \Sigma \to [0, 1]$ is the observation probability. We use Θ to denote (π, T, O).*

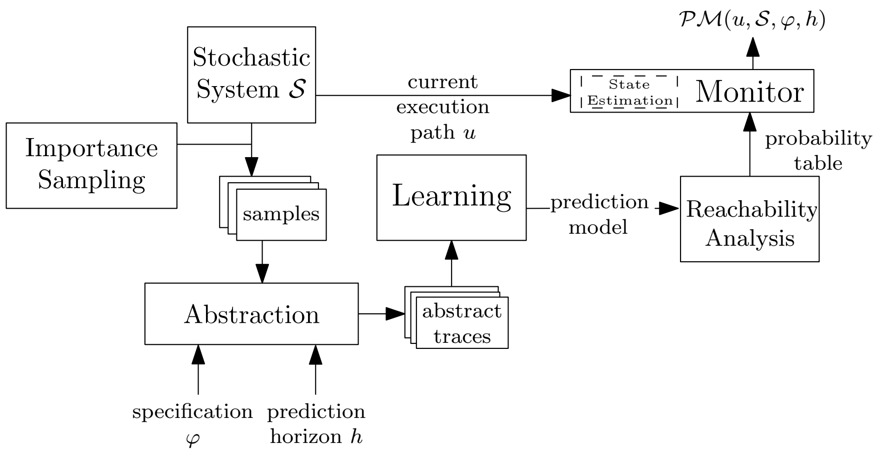


Figure 3.1: Overview of predictive RV with abstraction.

Similar to a DTMC we use $Path_k(o)$ to denote the execution paths of length k that start with the symbol o. The corresponding state sequence to an execution path needs to be estimated in an HMM.

3.4 The $\mathcal{P}$*revent* Framework

In this section we provide the details of the generic framework for predictive runtime verification, called $\mathcal{P}$*revent*. The goal of $\mathcal{P}$*revent* is to construct a predictive monitor according to the Definition 11 for a discrete-time black- or gray-box stochastic system, where only the observation space is known. The framework is implemented and is available online[1], with part of the code displayed in Appendix B. An overview of $\mathcal{P}$*revent* is depicted in Figure 3.1.

$\mathcal{P}$*revent* takes the following inputs:

Inputs.

- *Samples:* Sample execution paths are collected from the system for learning purposes. We assume that the samples are independent and identically distributed unless importance sampling is used for rare properties. The length of the samples are drawn

[1]https://bitbucket.org/rbabaeecar/prevent/

from a geometric distribution with parameter $\lambda > 0$ [81]. We also assume that the number and the maximum length of the samples are set so that the state space is sufficiently covered, hence can represent the stochastic system $\mathcal{S}$.

- *Specification φ:* The specification is a guarantee or safety LTL property, where the satisfying or violating prefixes are specifiable by a DFA. In the remainder of the dissertation we suppose that the LTL property is converted to the corresponding DFA, for which the details are not discussed but can be found at [15].

- *Prediction Horizon h:* The length of the finite extensions predicted in $\mathcal{P}$*revent* is limited by an upper bound given as a non-zero positive integer value h, which we call the prediction horizon. Having this upper bound gives the user the flexibility to predict satisfaction or violation of a property within a certain number of steps, and avoid obtaining the vacuous value 1 in an unbounded case (e.g., *some failure with non-zero probability will eventually happen*).

- *Current Execution Path u:* At runtime, the monitor receives the current execution path u generated by the system. The path u is assumed to be a prefix of an infinite execution path, and is used by the monitor to estimate the state of the system and consequently calculate the probability of the satisfaction or the violation of the given specification at runtime.

The output of $\mathcal{P}$*revent* is as follows:

Output. As formulated in Definition 11, the output of the monitor is the probability of the extensions that satisfy a guarantee property or violate a safety property. Under the condition of the convergence of the learning algorithm computing this probability using the trained prediction model gives the same result as the actual model.

$\mathcal{P}$*revent* comprises the following components:

Learning. Constructing the prediction model is performed via statistical learning from *iid* samples. Under the assumption that the learned model converges to the actual model in the limit, the output of a predictive monitor, i.e., (3.1) using the learned model is equivalent to that of a predictive monitor using the actual model. We show that in the case of a DTMC, the state-merging algorithms, such as Alergia [34, 80], suffice to produce a prediction model that in the limit is as accurate as the actual model. In the case where there is nondeterminism (for example due to abstraction), an HMM is trained from the samples, using the Baum-Welch algorithm [101], which is a form of the Expectation-Maximization (EM) algorithm, and provides an approximation to find the parameters of the HMM that maximizes the likelihood of the training data.

Reachability Analysis & Monitor Construction. The monitor is the result of a quantitative bounded reachability analysis on the product of the prediction model and the specification. We use Prism [90] to compute the probability results of the quantitative bounded reachability analysis. The monitor is constructed as a lookup table, where each entry is a combination of three elements: an integer variable t, a state of the prediction model, and the probability that from that state the system reaches the states that satisfy or violate the given property within t steps. The value of t is bounded by the prediction horizon h.

State Estimation. Given that the original DTMC is deterministic, the system state can be determined by the prefix u. However, if the prediction model is nondeterministic, the state of the system needs to be estimated based on u [120]. Although we use the Forward-Backward algorithm [101]; any *filtering techniques* [105] can be exploited to estimate the state at runtime. If the size of the model is large, approximate techniques such as the Viterbi algorithm [126, 7] is applied.

The three aforementioned components of $\mathcal{P}$*revent* are fully discussed in Chapter 4.

Abstraction. The size of the prediction model significantly impacts both the size and the performance of the monitor. We employ abstraction to decrease the size of the observation space by extracting symbols that have similar *transient probability* [70] to reach the states that satisfy a reachability property. The abstraction technique is in fact a form of symmetry reduction [69] on the observation space that is implemented at the trace level [89]. More specifically, we employ the *k-gapped pair model* [36] to detect the symbols that have similar empirical probability to reach the states satisfying the property within k steps. The symbols that have probability *zero* are considered irrelevant in the prediction and are lumped together. Other symbols with non-zero and symmetrical probabilities are lumped into equivalence partitions over the observation space to create a smaller observation space, we call the abstract alphabet. The abstract alphabet is then used to convert the traces of the training set into abstract traces. Training a prediction model from the abstract traces typically produces a model with smaller size. The abstraction is fully described in Chapter 5.

Importance Sampling. The rareness of a given property poses a challenge for statistical learning in our framework in terms of the number of samples. We use Importance Sampling (IS)[107, 106] to generate samples that have more instances of the events related to a rare property. Using IS we train a prediction model for a rare property without having to increase the number of training samples, where the relationship between the IS and the original distributions is well defined by the *likelihood ratio* (LR). We use the LR in the training algorithm to compensate the distribution of the samples, that are drawn using IS, and build an accurate prediction model of the rare property with fewer samples. Building a prediction model from the rare-event samples is explained in Chapter 6.

3.5 Running Example

We use the die example of [90] as the running example in the following chapters. This example demonstrates the simulation of throwing a fair 6-sided die by flipping a fair coin [67]. The model is shown as a DTMC in Figure 3.2 (The PRISM description is provided in Appendix A.1).

Let C be the output of the flipped coin ($C \in \{ii, hh, tt\}$), where hh, tt signify *head* and *tail*, respectively, and ii is a special symbol to indicate the initial state of the coin. Also, let $D \in \{0, \dots, 6\}$ be the output of the simulated die, where $1, \dots, 6$ is the simulated output of the die, and 0 shows that the output of the die is not determined yet and the coin needs to be flipped again. The coin needs to be flipped at least three times to simulate observing a number on the die. We define $\Sigma_{die} = C \times D$ as the observation space, that denotes the output of the coin and the die in the process.

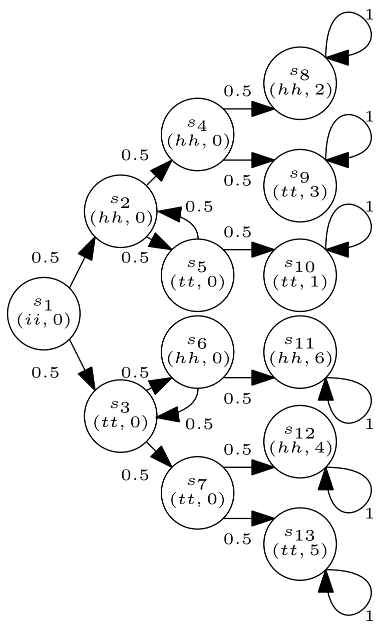
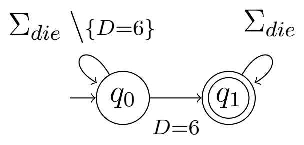
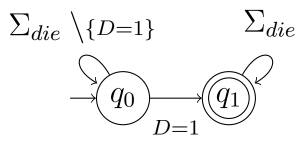


Figure 3.2: Simulating rolling a fair die with flipping a fair coin.

We consider a guarantee and a safety property (Figure 3.3). Suppose checking the guarantee property *eventually the outcome of the die is "6"*, at runtime, which translates to the LTL property $\varphi_{\mathsf{F}} = \Diamond(D = 6)$, the DFA of which is depicted in Figure 3.3a. Any (infinite) execution paths with the prefix $u_{\mathsf{F}} = (ii, 0)(tt, 0)(hh, 0)(hh, 6)$ satisfies φ_{F}. However, the result on the prefix $u'_{\mathsf{F}} = (ii, 0)(tt, 0)(hh, 0)(tt, 0)$ is *unknown* [14], as it can be extended to an infinite path that



(a) The DFA of the property $\varphi_{\mathsf{F}} = \Diamond(D = 6)$.



(b) The DFA of the property $\varphi_{\mathsf{G}} = \Box\neg(D = 1)$.

Figure 3.3: The corresponding DFAs of the satisfying or violating extensions for φ_{F} and φ_{G}, respectively.

satisfies φ_{F} (e.g., $(ii, 0)(tt, 0)(hh, 0)$ $(tt, 0)(hh, 0)(hh, 6)^{\omega}$), or an infinite path that violates φ_{F} (e.g., $(ii, 0)(tt, 0)(hh, 0)(tt, 0)(tt, 0)(tt, 5)^{\omega}$).

Similarly consider the safety property *always never the outcome of the die is "1"*, which translates to the LTL property $\varphi_{\mathsf{G}} = \Box\neg(D = 1)$, the DFA of which is depicted in Figure 3.3b. Any (infinite) execution paths with the prefix $u_{\mathsf{G}} = (ii, 0)(hh, 0)$ $(tt, 0)(tt, 1)$ violates φ_{G}. However, the result on the prefix $u'_{\mathsf{G}} = (ii, 0)(hh, 0)(tt, 0)(hh, 0)$ is *unknown*, as it can be extended to an infinite path that satisfies φ (e.g., $(ii, 0)(hh, 0)(tt, 0)$ $(hh, 0)(hh, 0)(tt, 3)^{\omega}$), or an infinite path that violates φ_{G} (e.g., $(ii, 0)(hh, 0)(tt, 0)(hh, 0)$ $(tt, 0)(tt, 1)^{\omega}$).

To deal with the inconclusive results due to unknown extensions [13], we provide the monitor with a *prediction model* to extend the prefix and generate the results based on the probability of the extensions that satisfy φ_{F} and violate φ_{G}.

3.6 Related Work

In this section we review the work related to the predictive runtime verification framework, and contrast it to $\mathcal{P}$*revent*. We group the literature into three sections: the semantic-based, the probabilistic, and the statistical approaches.

In the semantic-based approaches, a modified definition of the LTL semantics provides a logic-based method to interpret the extensions of a finite prefix, and evaluate an LTL property. These approaches typically use the semantic information (e.g., the CFG) of the program execution for prediction [135]. Hence, the semantic-based approaches are considered within the context of white-box systems.

In the probabilistic approaches, a probabilistic model of the system execution is provided and this is verified against given properties using numerical methods [70]. Since the probabilistic model is given, the probabilistic approaches are applicable to white-box systems.

In the statistical approaches, a given property is tested against a series of executions using statistical methods and hence are useful in the context of black-box systems.

$\mathcal{P}$*revent* can be seen as a fusion of the probabilistic and statistical approaches: instead of checking the execution samples, a stochastic model is built from them [89, 80, 31] and then the model is checked against a given property. In the proposed framework the probability is calculated when constructing the monitor, and the property itself remains non-probabilistic. The calculated probability is merely used as a quantitative measure to reflect how likely the given property is going to be satisfied or violated.

In the following sections, we describe the work related to each of these methods in the literature.

3.6.1 Semantics-based Approaches

The semantics-based approaches interpret the evaluation of an LTL property with respect to a finite prefix, by developing new semantics and the truth-value domain. The monitor uses the new semantics to issue a verdict based on either only the current path (non-predictive semantics) or different possible extensions of the paths, regardless of what the system is able to generate (predictive semantics) [74]. In the following, we review the related work in each category as well as some related work in the literature that use quantitative semantics and evaluation for purposes other than prediction.

Non-predictive Semantics. The semantics of LTL over finite paths was initially discussed in [79], with following modifications proposed later on, such as FLTL [79], $\text{LTL}^{\mp}$ [41], LTL_3 [12], RV-LTL (LTL_4) [14, 24], and RV^{∞}-LTL [86]. In FLTL and $\text{LTL}^{\mp}$, the semantics of the *next* operator is redefined to deal with non-existent future state, often considered an *empty* path.

Introducing separate semantics for the *empty* path results in vacuous evaluation for certain LTL formulas [86, 41]. For example, the evaluation of $\Box(\mathsf{X}p)$ in $\text{LTL}^{\mp}$ is $\bot$ at any temporal step on any sequence, if the *next* operator is interpreted *weak* (denoted by $\underline{\mathsf{X}}$ [14]). The only way to make a distinction on using different *next* operators is to consider the context of the formula. In the previous example, we must interpret *next* strong, if the formula contains $\Box$, and *weak*, if the formula contains $\Diamond$. This leads to RV^{∞}-LTL [86], in which each LTL formula has different semantics and a truth-value domain depending on the class they belong to in the LTL hierarchy [111].

LTL_3 is essentially an extension of the original LTL with an extra truth value, i.e., *unknown* [12]. However, despite the original LTL, the semantics of LTL_3 is non-inductive [14]. If the truth value of an LTL formula can not be yielded over a finite path, it will be *unknown*. It is extensively discussed in the literature that the evaluation of the so-called *non-monitorable* properties over any finite paths, is always *unknown* in LTL_3 [14, 13]. RV-LTL (LTL_4) combines the semantics definitions of both FLTL and LTL_3 [14] to provide a more conclusive result for the properties with unknown evaluation in LTL_3.

In $\mathcal{P}$*revent*, we only focus on the monitorable properties with unknown evaluation in LTL_3. However, despite the semantics-based approaches, $\mathcal{P}$*revent* uses a prediction model trained from the execution samples of the system to evaluate the LTL formula on only the extensions that the system could generate. Therefore, our approach provides a predictive evaluation of the LTL formula without changing the LTL semantics and based on the observed behaviour of the system.

Predictive Semantics. Some previous works define predictive semantics of LTL over finite paths based on LTL_3 such that, besides the current execution u, a finite suffix v is predicted from the program's CFG, and the evaluation occurs on the path uv [135, 133]. Nonetheless, the predictive semantics of LTL is different from LTL_3 and is more similar to $\mathcal{P}$*revent* in the sense that it detects the violation of a formula before it occurs. Their approach can be seen as the semantics-based predictive RV for white-box non-stochastic systems.

Quantitative Semantics & Evaluation. In [78] a quantitative semantics definition is proposed for specifying properties over continuous signals with real values. The logic is based on a subset of Metric Interval Temporal Logic (MITL) [4], and is called Signal Temporal Logic (STL). An atomic proposition in STL is defined over the value of a signal that is in the time domain. Since a STL formula is an $\text{MITL}_{[a,b]}$ formula, the semantics is inherently bounded in time and defined on finite traces [78]. Fainekos & Pappas [42] propose a metric to measure the robustness of the satisfaction of an STL formula. The quantified evaluation in $\mathcal{P}$*revent* is the result of a probability measure that reflects how likely the formula will be satisfied in the future. Although we do not discuss the details, it is not difficult to extend $\mathcal{P}$*revent* to predict the satisfaction of a STL property with respect to the robustness measure.

3.6.2 Probabilistic Approaches

In probabilistic model checking (PMC) [70, 68], an explicit probabilistic model, such as a Markov chain, is given, and a probabilistic property needs to be verified on the model. In PMC and SMC, the probability can be embedded into the specification language, e.g., Probabilistic CTL (PCTL) [49, 70]. In both cases, the probability of the formula can also be calculated quantitatively [89].

3.6.3 Statistical Approaches

Statistical Model Checking. Statistical model checking [132, 113, 52] (SMC) refers to verifying a property on a sampled set of execution paths, and generalizing it to the entire system by hypothesis testing. In SMC, the model of the system is statistical and based on Bernoulli distribution of the evaluation of the property on each simulated path. The goal of SMC is to compute the statistical confidence for $Pr(\mathcal{S} \models \varphi)$, given an LTL formula φ and a stochastic system $\mathcal{S}$.

In [31], a fast SMC approach is introduced where unbounded LTL properties are verified on a Markov chain whose only minimum transition probability is known. The topology or even the number of states of the Markov chain is not required to be known in advance, because the model is constructed on the fly when the hypothesis is tested on a trace. They use a technique that combines detecting Bottom Strongly Connected Components (BSCCs) and fast termination of the executions using the observed prefix from the system. The algorithm then gives confidence bounds of whether the property is satisfied or not.

In our context, the prediction is only required for the properties that can not be evaluated on the current execution path, hence, fast termination is irrelevant to our framework. In the current implementation of $\mathcal{P}revent$, we only consider bounded properties and the model is constructed offline. However, we propose constructing or refining the prediction model online as one of the future directions to the proposed framework (see Chapter 7).

Statistical Runtime Verification. Statistical runtime verification (SRV) [53] is an approach resembling SMC, in which an LTL-based formula is verified against sampled traces generated via simulation. The main difference is that in SRV instead of a set of traces, the statistical reasoning about the system takes place with observing a single execution of the system [110]. The *confidence* and *accuracy* of the verification is bound to the sample size, or the number of subtraces that are extracted from the execution. One can use the *sliding window* technique [110] to evaluate a probabilistic formula on the subtraces of an execution. The idea of evaluation with a sliding window can be used in $\mathcal{P}$*revent* to obtain an optimal value for the prediction horizon without hindering the prediction accuracy.

Despite [110], which is based on the hypothesis testing, the SRV method introduced in [53] is based on the notion of approximate probabilistic model checking (APMC) [52]. The monitoring algorithm in [53] is based on *temporal testers* [65] which evaluates a formula with the semantics similar to LTL_3 [12].

The online nature of SRV, similar to SMC [31], creates an *optimal* verification framework in the sense that checking a property terminates as soon as the evaluation result is yielded. As mentioned before, $\mathcal{P}$*revent* only predicts the satisfaction or violation of a formula whose evaluation can not be determined using the current prefix, hence the optimal termination is not applicable in our context. However, it is worth mentioning that the termination of predicting extensions is based on the prediction horizon, the maximum length that the extensions can take. In addition, the prediction model, that is trained from several executions of the system covers more of the state space compared to a single trace.

Chapter 4

Constructing a Predictive Monitor

A monitor is a finite-state machine that consumes the output of the system execution sequentially, and produces the evaluation of a given property at each step, typically as a Boolean value [14]. The monitor in $\mathcal{P}revent$ takes the form of a lookup table, that instead of Boolean values produces a value in $[0, 1]$. The value indicates the probability of the extensions that satisfy or violate the property, assuming that it is currently not satisfied or violated. These probability values are the result of a bounded reachability analysis on the product of the trained prediction model and the specification.

In this chapter, we describe the procedure to construct the predictive monitor using two types of prediction models, a DTMC and an an HMM. First, we discuss learning the prediction models from *iid* samples (Section 4.1), and describe the realization of prediction by performing a quantitative bounded reachability analysis on the product of the prediction model and the specification (Section 4.2). We use the results of the reachability analysis to construct the monitor as a lookup table (Section 4.3), and describe how to use the lookup table with state estimation to predict the satisfaction or violation of a property at runtime (Sections 4.4 and 4.5). To assess the accuracy of the predictions we formally define two error functions (Section 4.6), and two case studies to demonstrate the feasibility of our approach (Section 4.8). We conclude this chapter by reviewing some of the related works (Section 4.9).

4.1 Learning Models

Inferring the probabilistic languages of infinite strings from a set of samples is defined as obtaining the probability distribution over Σ^ω. In this section, we review the learning algorithms of the two types of prediction models from the literature that are used in $\mathcal{P}revent$: Discrete-Time Markov Models (DTMCs) and Hidden Markov Models (HMMs).

4.1.1 Training a DTMC from *iid* Samples

The state-merging algorithms [23, 80] are shown to be effective in learning probabilistic finite automata (PFA) [34], which in turn can be converted to a DTMC [80]. We choose the ALERGIA algorithm [34] as the representative of the state-merging learning algorithms. In ALERGIA first the training data is converted into a Frequency Prefix Tree Acceptor (FPTA), which simply is another representation of the execution samples based on the prefixes and their frequencies:

Definition 15 (FPTA). *A Frequency Prefix Tree Acceptor (FPTA) is a tuple $\mathcal{A} : (Q, \Sigma, Fr_I, \delta, Fr_F, Fr_T)$, where Q is a non-empty finite set of states, Σ is a non-empty finite alphabet, $Fr_I : Q \to \mathbb{N}$ is the initial frequency of the state(s), $\delta : Q \times \Sigma \to Q$ is the transition function, $Fr_F : Q \to \mathbb{N}$ is the final frequency of the state(s), and $Fr_T : Q \times \Sigma \times Q \to \mathbb{N}$ is the transition frequency function between two states.*

The states of the FPTA are then merged into one another based on a compatibility test, e.g., based on Hoeffding bound [81] or Angluin's bound [80], until some convergence criteria are met [82].

Algorithm 1 demonstrates the main steps of ALERGIA, adapted from [80] and [23]. The algorithm starts by building the FPTA (line **3**). Each node of the tree represents a prefix, with its frequency, i.e., the number of times the prefix appeared in the entire sample dataset (see Figure 4.1).

After building the FPTA, the algorithm initializes and maintains two sets of nodes: RED and BLUE. The *red* nodes are those that have already been merged and are part of the final model. The *blue* nodes are the candidates to be merged with a *red* node. The set RED is initialized with q_ϵ (line **4**), that is, the initial node in the FPTA that represents the empty string, i.e., no prefix. The set BLUE is initialized with all the nodes connected to q_ϵ, which represent prefixes of length one (line **5**). We use PREF($\mathcal{D}$) to denote the set of all the prefixes in the sample dataset $\mathcal{D}$.

The main *while* loop (lines **6-16**) selects a *blue* node based on the lexicographical order (the function SELECTBLUESTATE in line **7**), and tests it against all the *red* nodes (lines **9-13**) for compatibility (the function in COMPATIBILITYTEST in line **10**). To be faithful to the training data, similar to [80], we assume that the compatibility test is performed

```
1  ALERGIA(Sample dataset $\mathcal{D}, \alpha > 0$)
   output: PFA $\mathcal{A}$
2  begin
3      $\mathcal{A} \leftarrow$ BUILDFPTA($\mathcal{D}$)
4      RED $\leftarrow \{q_\epsilon\}$
5      BLUE $\leftarrow \{q_a : a \in \Sigma \cap$ PREF($\mathcal{D}$)$\}$
6      while BLUE $\neq \emptyset$ do
7          $q_b \leftarrow$ SELECTBLUESTATE
8          merge $\leftarrow$ false
9          foreach $q_r \in$ RED do
10             if COMPATIBILITYTEST($q_b, q_r, \alpha$) then
11                 $\mathcal{A} \leftarrow$ STOCHASTICMERGE($\mathcal{A}, q_r, q_b$)
12                 merge $\leftarrow$ true
13                 break
14         if $\neg$merge then
15             RED $\leftarrow$ RED $\cup \{q_b\}$
16         BLUE $\leftarrow \{q_{ua} \mid q_u \in$ RED$, a \in \Sigma, ua \in$ PREF($\mathcal{D}$)$\} \setminus$ RED
17     $\mathcal{A} \leftarrow$ NORMALIZEFPTA($\mathcal{A}$)
18     return $\mathcal{A}$
```

Algorithm 1: Generating a PFA from a set of *iid* samples.

on the original FPTA, rather than the intermediate automaton. The parameter α is used for the compatibility criterion, which is in $(0, 2]$ for the Hoeffding bound [81] and > 0 for Angluin's bound [80, 6]. If the two nodes are compatible, they are merged, with all the frequencies of the *blue* node and its descendants recursively added to the *red* nodes (the procedure StochasticMerge in line **11**).

If there is no compatible *red* node with a given *blue* node, then the *blue* node is promoted to a *red* node (line **15**). In either case, Blue is updated in line **16** according to its declarative definition: all the successors of *red* nodes that are not themselves *red*.

Example. Table 4.1 demonstrates 1000 samples that are randomly generated from the DTMC in Figure 3.2. For simplicity, the maximum length of the samples is 3 which covers all the transitions in the model except for the self-loops. Figure 4.1 depicts the FPTA obtained from these samples, with Table 4.2 showing the prefix and final frequency (*fin.*

Table 4.1: A training set containing 1000 randomly generated samples from the model in Figure 3.2.

trace	*freq.*	*trace*	*freq.*
$(ii, 0)$	300	$(ii, 0)(hh, 0)(hh, 0)(hh, 2)$	49
$(ii, 0)(hh, 0)$	51	$(ii, 0)(hh, 0)(hh, 0)(tt, 3)$	51
$(ii, 0)(tt, 0)$	49	$(ii, 0)(hh, 0)(tt, 0)(hh, 0)$	52
$(ii, 0)(hh, 0)(hh, 0)$	48	$(ii, 0)(hh, 0)(tt, 0)(tt, 1)$	48
$(ii, 0)(hh, 0)(tt, 0)$	52	$(ii, 0)(tt, 0)(hh, 0)(tt, 0)$	51
$(ii, 0)(tt, 0)(hh, 0)$	52	$(ii, 0)(tt, 0)(hh, 0)(hh, 6)$	49
$(ii, 0)(tt, 0)(tt, 0)$	48	$(ii, 0)(tt, 0)(tt, 0)(hh, 4)$	52
		$(ii, 0)(tt, 0)(tt, 0)(tt, 5)$	48

freq.) associated with each state of the FPTA. The final frequency of each state is the frequency of the traces that are equal to the prefix corresponding to the state. □

The output of Algorithm 1 is a PFA, which is essentially the transformation of the merged FPTA by normalizing all the frequencies to obtain a probability distribution for each transition (line **12**).

Definition 16 (PFA). *A Probabilistic Finite Automaton (PFA) is a tuple $\mathcal{A} = (Q, \Sigma, \pi, \delta, \mathbf{P}_T, \mathbf{P}_F)$, where Q is a non-empty finite set of states, Σ is a non-empty finite alphabet, $\pi : Q \to [0, 1]$ is the initial probability distribution over Q, $\delta : Q \times \Sigma \to Q$ is the transition function that maps the state-symbol pair to another state, $\mathbf{P}_T : Q \times \Sigma \to [0, 1]$ is the transition probability distribution, and $\mathbf{P}_F : Q \to [0, 1]$ is the final probability distribution, such that for any $q \in Q$, $\mathbf{P}_T(q, \cdot) + \mathbf{P}_F(q)$ is a probability distribution.*

Algorithm 2 constructs a DTMC from the trained PFA. Constructing the states of the DTMC begins by iterating over all the successor states of q_0, i.e., the state representing the empty string in the PFA. Notice that we assume the length of the samples is non-zero, i.e., there is no empty string (ϵ) in the training data. The *for* loop in lines **4**-**8** sets the initial probability distribution of the underlying DTMC, which is obtained from the transition from q_0 to the states using an alphabet symbol a (line **7**). Those states are added to the set of states of the DTMC (line **6**), and labelled with a (line **8**).

The for loop in lines **9**-**13** computes the transition probability distribution of the DTMC for each pair (q, a). Notice that each state in the DTMC is also in the PFA, hence $q \in S$ and $\delta(q, a)$ is well defined. In line **12**, we normalize the transition probability by dividing it by the complement of the final probability distribution in the PFA (see Definition 16). To

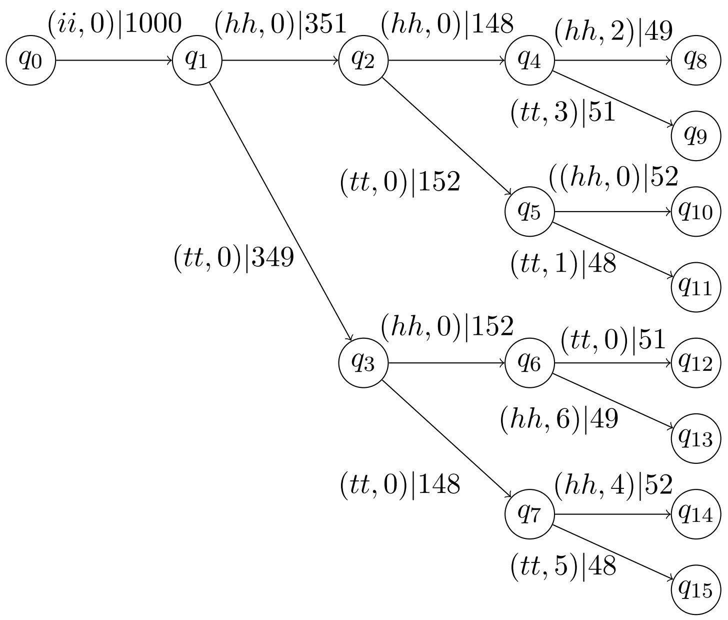


Figure 4.1: FPTA constructed from the samples of Table 4.1.

Table 4.2: The prefixes and their final frequencies associated with each state of the FPTA in Figure 4.1.

state	*prefix*	*fin. freq.*	*state*	*prefix*	*fin. freq.*
q_0	ϵ	0	q_1	$(ii,0)$	300
q_2	$(ii,0)(hh,0)$	51	q_3	$(ii,0)(tt,0)$	49
q_4	$(ii,0)(hh,0)(hh,0)$	48	q_5	$(ii,0)(hh,0)(tt,0)$	52
q_6	$(ii,0)(tt,0)(hh,0)$	52	q_7	$(ii,0)(tt,0)(tt,0)$	48
q_8	$(ii,0)(hh,0)(hh,0)(hh,2)$	49	q_9	$(ii,0)(hh,0)(hh,0)(tt,3)$	51
q_{10}	$(ii,0)(hh,0)(tt,0)(hh,0)$	52	q_{11}	$(ii,0)(hh,0)(tt,0)(tt,1)$	48
q_{12}	$(ii,0)(tt,0)(hh,0)(tt,0)$	51	q_{13}	$(ii,0)(tt,0)(hh,0)(hh,6)$	49
q_{14}	$(ii,0)(tt,0)(tt,0)(hh,4)$	52	q_{15}	$(ii,0)(tt,0)(tt,0)(tt,5)$	48

generate a distribution over Σ^ω, if a state in the PFA does not have any outgoing transition, we turn it into an absorbing state in the DTMC by adding a self loop (lines **14**-**15**).

```
1  Pfa2Dtmc($\mathcal{A} = (Q, \Sigma, \pi_{\mathcal{A}}, \delta, \mathbf{P}_T, \mathbf{P}_F)$)
   output: $\mathcal{M} = (S, \Sigma, \pi_{\mathcal{M}}, \mathbf{P}, L)$
2  begin
3      $S \leftarrow \{\}$
4      foreach $a \in \Sigma$ where $\delta(q_0, a)$ exists do
5          $q \leftarrow \delta(q_0, a)$
6          $S \leftarrow S \cup \{q\}$
7          $\pi_{\mathcal{M}}(q) \leftarrow \mathbf{P}_T(q_0, a)$
8          $L(q) \leftarrow a$
9      foreach $a \in \Sigma$ and $q \in S$, where $\delta(q, a)$ exists do
10         $q' \leftarrow \delta(q, a)$
11         $S \leftarrow S \cup \{q'\}$
12         $\mathbf{P}(q, q') \leftarrow \mathbf{P}_T(q, a)/(1 - \mathbf{P}_F(q, a))$
13         $L(q') \leftarrow a$
14     if $\delta(q, a)$ does not exist $\forall a \in \Sigma$ then
15         $\mathbf{P}(q, q) \leftarrow 1$
16     return $\mathcal{M}$
```

Algorithm 2: Constructing a DTMC from a PFA.

Example. Figure 4.2 shows the final DTMC after merging the states of the FPTA and turning the resulting PFA into a DTMC. □

Convergence. Under the assumption that the length of each sample is drawn from a geometric distribution [81], the model learned by the Alergia algorithm converges to the actual model with probability one:

Theorem 3 ([81] [80]). *Let φ be an LTL property, $\mathcal{M}_{true}$ be the generating model for the training data containing N samples used in Algorithm 1, and $\mathcal{M} = (S, \Sigma, \pi_{\mathcal{M}}, \mathbf{P}, L)$ be the output of Algorithm 2. Then we have:*

$$\lim_{N \to \infty} Pr(\mathcal{M}_{true} \models \varphi = \mathcal{M} \models \varphi) = 1$$

The proof of Theorem 3 follows from showing that a bisimulation relationship exists between the states of $\mathcal{M}_{true}$ and $\mathcal{M}$ as $N \to \infty$ [81].

Complexity. The complexity of the Alergia algorithm is cubic to the number of training samples at worst [23], but in practice the learning terminates in linear time with respect to the number of samples [80].

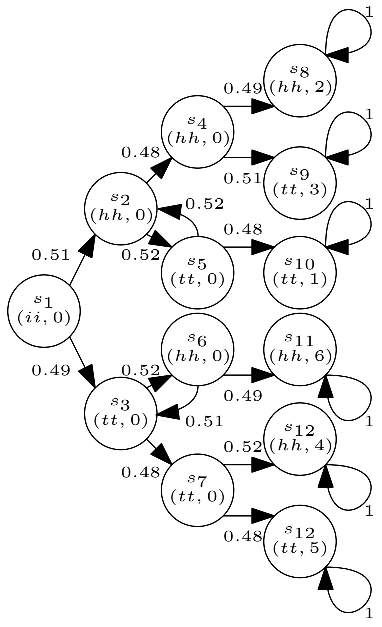


Figure 4.2: The final learned DTMC after merging the states of the FPTA in Figure 4.1.

4.1.2 Training HMM

We use the *Maximum Likelihood Estimation* (MLE) technique [114] to train an HMM. The log-likelihood function $L(\Theta)$ of the HMM with parameters $\Theta = (\pi, T, O)$ over the observation and the state sequence $Y^{(1:n)}$ and $X^{(1:n)}$, respectively, is defined as:

$$L(\Theta) = \log\left(\sum_{X^{(1:n)}} Pr(X^{(1:n)}, Y^{(1:n)} \mid \Theta)\right) \tag{4.1}$$

Since the probability distribution over the state sequence $X^{(1:n)}$ is unknown, $L(\Theta)$ does not have a closed form [123], leaving the training techniques to heuristics such as EM. One well-known EM technique for training an HMM is the Baum-Welch algorithm [101] (BWA), where the training alternates between estimating the distribution over the hidden state variable, $Q : X \to [0, 1]$, with some fixed choice for Θ (*Expectation*), and maximizing the log-likelihood to estimate the values of Θ by fixing Q (*Maximization*) [105].

The *Expectation* phase in BWA computes $Pr(X^{(t)} = s \mid Y, \Theta)$ and $Pr(X^{(t)} = s, X^{(t+1)} = s' \mid Y, \Theta)$ for $s, s' \in S$ through the Forward-Backward algorithm [101] (S is the set of hidden states in the HMM). Given Θ, the Forward algorithm (Algorithm 3) calculates the probability of being at state $i \in S$ at time τ and observing the sequence $Y^{(1:\tau)}$. Given Θ, the Backward algorithm (Algorithm 4) computes the probability of observing the partial path $Y^{(\tau+1:n)}$ given that the HMM is at state $i \in S$ at time τ. Given Θ, the Forward-Backward algorithm (Algorithm 5) uses the Forward and the Backward algorithms to compute the probability of being at state $i \in S$ given that we observed $Y^{(1:n)}$. We later also use the Forward-Backward algorithm to estimate the posterior probability distribution of the state space during runtime monitoring (see Section 4.4.1).

Maximization is performed on a lower bound of $L(\Theta)$ in (4.1) using Jensen's inequality:

```
1 FORWARD(Sample dataset $Y^{(1:\tau)}, \Theta, i$)
  output: $\alpha(i,\tau) = Pr(Y^{(1:\tau)}, X^{(\tau)} = i|\Theta)$
2 begin
3     $\alpha(i,1) \leftarrow \pi(i)O(i, Y^{(1)})$
4     for $t \leftarrow 1$ to $\tau - 1$ do
5         $\alpha(i,t+1) \leftarrow O(i, Y^{(t+1)}) \sum_{j=1}^{M} \alpha(j,t)T(i,j)$
          /* $M$ is the number of hidden states */
6
7     return $\alpha(i,\tau)$
```

Algorithm 3: The FORWARD algorithm to compute $Pr(Y^{(1:\tau)}, X^{(\tau)} = i|\Theta)$.

```
1 BACKWARD(Sample dataset $Y^{(1:\tau)}, \Theta, i$)
  output: $\beta(i,1) = Pr(Y^{(2:\tau)}|X^{(1)} = i, \Theta)$
2 begin
3     $\beta(i,\tau) \leftarrow 1$
4     for $t \leftarrow \tau - 1$ down to $1$ do
5         $\beta(i,t) \leftarrow \sum_{j=1}^{M} \beta(j,t+1)T(i,j)O(j, Y^{(t+1)})$
          /* $M$ is the number of hidden states */
6
7     return $\beta(i,1)$
```

Algorithm 4: The BACKWARD algorithm to compute $Pr(Y^{(2:\tau)}|X^{(1)} = i, \Theta)$.

```
1 $\textsc{Forward-Backward}(\text{Sample dataset } Y^{(1:n)}, \Theta, i)$
  output: $\gamma(i, n) = Pr(X^{(n)} = i | Y^{(1:n)}, \Theta)$
2 begin
3     $Pr(X^{(n)} = i, Y^{(1:n)} | \Theta) = \textsc{Forward}(Y^{(1:n)}, \Theta, i) \cdot \textsc{Backward}(Y^{(1:n)}, \Theta, i)$
4     $Pr(Y^{(1:n)} | \Theta) = \sum_{j=1}^{M} \textsc{Forward}(Y^{(1:n)}, \Theta, j) \cdot \textsc{Backward}(Y^{(1:n)}, \Theta, j)$
      /* Marginal probability                                        */
5
6     $\gamma(i, n) \leftarrow \frac{Pr(X^{(n)}=i, Y^{(1:n)} | \Theta)}{Pr(Y^{(1:n)} | \Theta)}$
7     return $\gamma(i, n)$
```

Algorithm 5: The Forward-Backward algorithm to compute $Pr(X^{(n)} = i | Y^{(1:n)}, \Theta)$ using Bayes' rule.

$$L(\Theta) \geq Q(X) \log Pr(X^{(1:n)}, Y^{(1:n)} \mid \Theta) - Q(X) \log Q(X) \tag{4.2}$$

Since the second term in (4.2) is independent of Θ [105], only the first term is maximized in each iteration:

$$\Theta^{(k)} = \operatorname*{argmax}_{\Theta} Q(X) \log Pr(X^{(1:n)}, Y^{(1:n)} \mid \Theta^{(k-1)}) \tag{4.3}$$

The BWA uses (4.3) to update Θ in an iterative manner. Algorithm 6 demonstrates the BWA with the *Expectation* and *Maximization* steps. The training starts with random initial values for Θ (denoted by $\Theta^{(0)}$—line **3**). The loop in lines **7-14** shows the *Maximization* step where $\gamma(i, t)$ is computed for all $t \in [1, n]$ and $i \in S$ using the Forward-Backward algorithm. In addition, given $Y^{(1:t)}$ and Θ, the probability of transitioning between states i and j at time t, denoted by $\xi(i, j, t)$, is computed for all $t \in [1, n]$ and $i, j \in S$ using the Forward and Backward algorithms as well as Bayes' theorem.

The *Maximization* step (lines **15-17**) updates the new parameters using γ and ξ, divided by the marginal probabilities to obtain a probability distribution. The function $\mathbb{1}_{o=Y^{(t)}}$ is the indicator function that returns 1 if the tth observation in the sequence Y is the symbol o; and 0 otherwise.

```
1  BAUM-WELCH(Sample dataset $Y^{(1:n)}$, Number of states $M$)
   output: $\Theta = (\pi, T, O)$
2  begin
3      $\Theta^{(0)} \leftarrow$ RANDOMINITIALIZATION()
4      $k \leftarrow 0$
5      repeat
6          COMPUTE($L(\Theta^{(k)})$)
           /* The Expectation step                                          */
7          foreach $t \in [1, n], i, j \in S$ /* $S$ is the set of hidden states   */
8           do
9              $\gamma(i, t) \leftarrow$ FORWARD-BACKWARD($Y^{(1:t)}, \Theta^{(k)}, i$)
10             $Pr(X^{(t)} = i, X^{(t+1)} = j, Y|\Theta) =$
                 FORWARD$(i, t) \cdot T^{(k)}(i, j) \cdot$ BACKWARD$(j, t+1) \cdot O^{(k)}(j, Y^{(t+1)})$
11             $Pr(Y|\Theta) = \sum_{i=1}^{M} \sum_{j=1}^{M}$ FORWARD$(Y^{(1:t)}, \Theta^{(k)}, i) \cdot T^{(k)}(i, j) \cdot$
                 BACKWARD$(Y^{(t+1:n)}, \Theta, j) \cdot O^{(k)}(j, Y^{(t+1)})$
12             $Pr(X^{(t)} = i, X^{(t+1)} = j|Y^{(1:t)}, \Theta) = \frac{Pr(X^{(t)}=i, X^{(t+1)}=j, Y|\Theta)}{Pr(Y|\Theta)}$ /* Bayes'
                 theorem                                                     */
13
14             $\xi(i, j, t) \leftarrow Pr(X^{(t)} = i, X^{(t+1)} = j|Y^{(1:t)}, \Theta)$
           /* The Maximization step                                         */
15         $\pi^{(k+1)}(i) \leftarrow \gamma(i, 1)$
16         $T^{(k+1)}(i, j) \leftarrow \frac{\sum_{t=1}^{n-1} \xi(i,j,t)}{\sum_{t=1}^{n-1} \gamma(i,t)}$
17         $O^{(k+1)}(i, o) \leftarrow \frac{\sum_{t=1}^{n} \mathbb{1}_{o=Y^{(t)}} \cdot \gamma(i,t)}{\sum_{t=1}^{n} \gamma(i,t)}$
18         $k \leftarrow k + 1$
19         COMPUTE($L(\Theta^{(k)})$)
20     until $L(\Theta^{k}) \leq L(\Theta^{k-1})$
21     return $\Theta^{(k-1)}$
```

Algorithm 6: The BAUM-WELCH algorithm to find the parameters of an HMM, denoted by Θ.

Number of Hidden States. The BWE requires the number of hidden states (size of S denoted by M) in addition to the training sample as input. We apply the Bayesian Information Criterion (BIC) [28] to choose the number of hidden states. BIC assigns a score to a model according to its likelihood, but also penalizes models with more parameters to avoid overfitting:

$$BIC(\mathcal{H}) = \log(N)|\Theta| - 2L(\Theta), \tag{4.4}$$

where $|\Theta| = M^2 + M|\Sigma|$, and N is the number of training samples.

Convergence. In each iteration of the *loop* in lines 4-20, the likelihood of the updated Θ is compared to the likelihood of the previous value of Θ (denoted by $\Theta^{(k)}$ and $\Theta^{(k-1)}$, respectively), and if the previous value has a larger likelihood then the algorithm terminates. Therefore, the BWA is essentially a gradient-descent approach, and its outcome is guaranteed to converge to a local maximum [123].

Complexity. Since BWE uses dynamic programming, its complexity is polynomial with a slight sacrifice in space. The Forward-Backward algorithm is of $O(M^2 \cdot n)$ where M is the number of hidden states and n is the length of the sample. The BWE runs the Forward-Backward algorithm for each iteration and sample, hence the total complexity of BWE is of $O(\#\text{ of iterations } \cdot M^2 \cdot N \cdot n_{max})$ where n_{max} is the maximum length of the samples and N is the number of training samples. The space complexity of the BWE is of $O(M \cdot n_{max})$.

Example. Figure 4.3 depicts the trained HMM using the 1000 samples shown in Table 4.1. The optimal number of hidden states, 11, is calculated by finding the minimum BIC score among HMMs with 1 to 20 hidden states. □

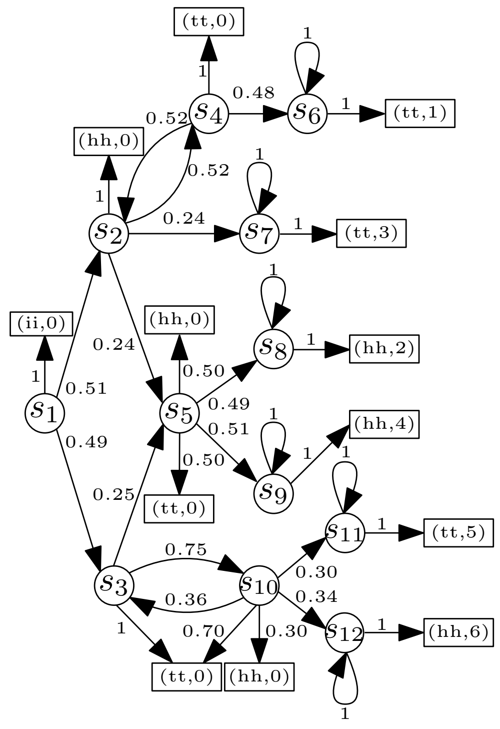


Figure 4.3: The trained HMM using 1000 samples and 11 hidden states.

4.2 Prediction as Quantitative Bounded Reachability Analysis on a DTMC

In this section, we explain how to build the product model of the prediction model and the DFA, as a DTMC. A quantitative bounded reachability analysis is performed on the product model to construct the lookup table that creates our predictive monitor (Definition 11).

4.2.1 Building the Product of the Prediction Model and the DFA

The monitor needs to expand the observed execution, u, and predict the expected probability of the extensions that satisfy or violate the given property. The expansion of u is based on a DFA that specifies good or bad extensions with respect to the given guarantee or safety property. To maintain the configurations of both the DFA and the prediction model we adapt a similar technique in the literature [129, 134] and create the product of the two models.

If the prediction model is a DTMC, the product model is simply built by combining the states and transitions of the DTMC and the DFA, and relabelling the states of the DTMC according to the final states of the DFA.

Definition 17 (The Product of a DTMC and a DFA). *Let $\mathcal{M} = (S, \Sigma, \pi, \mathbf{P}, L)$ and $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$ respectively be a DTMC and a DFA. We define the product DTMC of $\mathcal{M}$ and $\mathcal{A}$, $\mathcal{M}_{\mathcal{M}\times\mathcal{A}} = (S' = S \times Q \times \Sigma, \{\text{`Accept'}\}, \pi', \mathbf{P}', L')$ as follows:*

$$\pi'(s,q,o) = \begin{cases} \pi(s) & \text{if } q \in q_I \\ 0 & \text{otherwise} \end{cases} \qquad L(s,q,o) = \begin{cases} \{\text{`Accept'}\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

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

A similar approach is taken in an HMM, except that the transition probabilities and observation probabilities are multiplied together:

Definition 18 (The Product of an HMM and a DFA). *Let $\mathcal{H} = (S, \Sigma, \pi, T, O)$ and $\mathcal{A} = (Q, \Sigma, \delta, q_I, F)$ respectively be an HMM and a DFA. We define the DTMC $\mathcal{M}_{\mathcal{H}\times\mathcal{A}} =$*

$(S' = S \times Q \times \Sigma, \{\text{‘}Accept\text{’}\}, \pi', \mathbf{P}, L)$ *as follows:*

$$\pi'(s,q,o) = \begin{cases} \pi(s) & \text{if } q \in q_I \\ 0 & \text{otherwise} \end{cases} \qquad L(s,q,o) = \begin{cases} \{\text{‘}Accept\text{’}\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathbf{P}((s,q,o),(s',q',o')) = \begin{cases} T(s,s') \cdot O(s',o') & \text{if } \delta(q',o) = q \\ 0 & \text{otherwise} \end{cases}$$

Notice that the joint probability distribution represented by $\mathcal{H}$ remains intact at $\mathcal{M}_{\mathcal{H}\times\mathcal{A}}$ in Definition 18.

4.2.2 Quantitative Bounded Reachability Analysis

Let $\mathcal{M}$ be a DTMC that is the product of the prediction model and the DFA. The purpose of the monitor is to estimate the probability of all the extensions of length at most h that satisfy or violate φ. In a discrete-time setting, the variable h is a positive integer, which we call the *prediction horizon*. Computing the probability of the satisfying or refuting extensions is achieved by performing a bounded reachability analysis on the product model.

Let $u \notin L_\varphi$ be the execution on $\mathcal{M}$ observed so far, and Ext_u be the set of finite extensions of u. Recall that the output of the monitor is the probability measure of the set of the finite paths that satisfy or violate φ (Definition 11).

In the product DTMC, suppose that s is the state corresponding to the last label in u, which is obtained by some state estimation technique [120, 7, 63]. As u extends, the new state is consequently estimated at runtime (see Section 4.4).

The probability in Definition 11 can be obtained by recursively computing the *transient probability* of the states in the product DTMC [70]: starting from the composite state $\sigma = (s,q,o)$, the probability of being at state $\sigma' = (s',q',o')$ after h steps, such that $L(s',q',o') = $ *‘Accept’*, that is, all the paths that end at s' and satisfy or violate φ. We can effectively turn computing the transient probability into checking the following quantitative PCTL property [49, 70]:

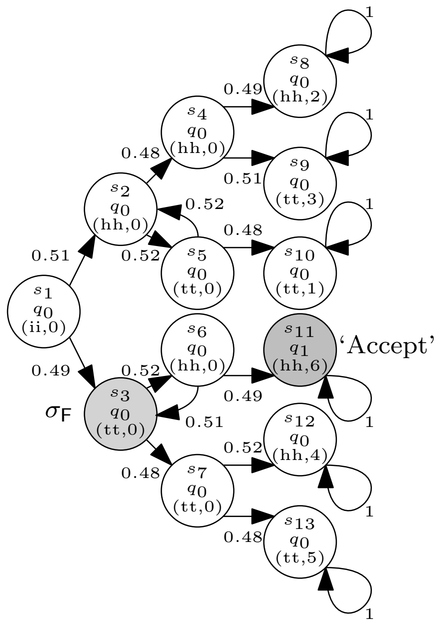


Figure 4.4: The product of the learned DTMC in Figure 4.2 and the DFA in Figure 3.3a for the die example.

$$Pr(\sigma \models \Diamond^{\leq h} \text{`}Accept\text{'} = \top) \tag{4.5}$$

Example. Figure 4.4 demonstrates the product model $\mathcal{M}$ of the learned DTMC for the die example (Figure 4.2) and the DFA of the property φ_F (Figure 3.3a). The light gray node demonstrates the corresponding state after observing $u'_\mathsf{F} = (ii,0)(tt,0)(hh,0)(tt,0)$, shown by σ_F, and the dark gray node displays state $(s_{11}, q_1, (hh, 6))$ that is labelled *'Accept'*.

Figure 4.5 demonstrates the probability (4.5) for σ_F and different values of h on the product model in Figure 4.4. Notice that the prediction results for φ_G and u'_G are similar, except that $\sigma_\mathsf{G} = (s_2, q_0, (hh, 0))$ and the *'Accept'* state is $(s_{10}, q_1, (tt, 1))$.

Since the DTMC in Figure 4.2 is deterministic σ_F and σ_G are directly derivable from u'_F and u'_G. □

h	The Result of (4.5) for σ_F
1	0.0
2	0.26
3	0.26
4	0.33
5	0.33
6	0.34
7	0.34
8	0.35
9	0.35
10	0.35

Figure 4.5: The probability that property φ_F will be satisfied within h steps, given that the model $\mathcal{M}$ in Figure 4.4 is at σ_F.

4.3 Constructing the Monitor as a Lookup Table

In the systems where RV is used, the time and space overhead of the monitor must be minimal, as the computational and memory capacities are limited [77]. Due to multiplications of large and typically sparse matrices, the calculation of (4.5) is not practical during runtime [70]. Hence we suggest implementing the monitor as a lookup hash table that has $O(1)$ retrieval time. We compensate the memory usage of the lookup table by reducing the size of the prediction model. Fewer number of states in the prediction model also expedites the state estimation process at runtime (see Section 4.4). We discuss the details of reducing the size of the prediction model in Chapter 5.

Given a final path that ends at the state $\sigma^{(n)}$, let $\sigma^{(n)}\sigma^{(n+1)} \cdots \sigma^{(n+t)}$ be the extension, such that $L(\sigma^{(n+t)}) =$ *'Accept'* in the product model $\mathcal{M}$, where $\sigma^{(n+i)} = (s_i, q_i, o_i)$ is the composite state of the model, for all $0 \leq i \leq t$. The monitor output is

$$Pr(\sigma^{(n+1)}\sigma^{(n+2)} \cdots \sigma^{(n+t)}), t \leq h. \tag{4.6}$$

In order to compute this probability we adopt the transformation from [70]:

$$\mathbf{P}_{Acc}(\sigma,\sigma') = \begin{cases} 0 & \text{if } L(\sigma) = \textit{`Accept'} \text{ and } \sigma \neq \sigma' \\ 1 & \text{if } L(\sigma) = \textit{`Accept'} \text{ and } \sigma = \sigma' \\ \mathbf{P}(\sigma,\sigma') & \text{otherwise} \end{cases} \tag{4.7}$$

The transformation (4.7) allows us to recursively compute (4.5) as follows:

$$Pr(\sigma \models \Diamond^{\leq h} \textit{`Accept'}) = \sum_{\sigma' \in S'} \mathbf{P}_{Acc}(\sigma,\sigma') Pr(\sigma' \models \Diamond^{\leq h-1} \textit{`Accept'}) \tag{4.8}$$

This is the *transient probability* for $\sigma^{(n)}\sigma^{(n+1)}\cdots\sigma^{(n+t)}w$, that is, starting from $\sigma^{(n)}$ the probability of being at state $\sigma^{(n+t)}$ (i.e., after t steps), such that $L(\sigma^{(n+t)}) = \textit{`Accept'}$, where $w \in \Sigma^\omega$ is any infinite extension of the path. The probability measure of $\sigma^{(n)}\sigma^{(n+1)}\cdots\sigma^{(n+t)}w$ is based on the cylinder set defined on the prefix $\sigma^{(n)}\sigma^{(n+1)}\cdots\sigma^{(n+t)}$[70].

Algorithm 7 demonstrates the monitor construction procedure. For all $t \leq h$ and $\sigma \in S'$, we compute the probabilities offline and store them in a table $MT(\sigma,t)$, where $MT(\sigma,t) = Pr(\sigma \models \Diamond^{\leq t} \textit{`Accept'})$. The monitor is thus a lookup table with the size of $O(|S'| \times h)$, where $|S'| \in O(|S| \cdot |Q| \cdot |\Sigma|)$. Algorithm 7 has the time complexity $O(|S'|^2 + h)$.

Example. Table 4.3 shows the MT table for φ_{F} and φ_{G} and for $h = 5$, obtained from the product model in Figure 4.4. The probabilities are only shown for the states that can reach the *'Accept'* state. □

4.4 State Estimation

Recall that $\sigma = (s, q, o)$ is the composite state of the product model, where o is the observation obtained from the execution path u, and q is the state of the DFA that is derivable by running the DFA over u. The only remaining element is the (hidden) state s that the monitor needs to obtain at runtime.

If the prediction model is deterministic s is determined by the execution path u (see Definition 13). However, if the prediction model is nondeterministic, s is not directly derivable from u, hence needs to be estimated.

To have consistent notations for both types of prediction model, i.e., DTMC and HMM, we define the notations $Obs(s, a)$ and $Trans(s, s')$ as follows ($s, s' \in S, a \in \Sigma$):

```
1 ConstructMonitor(Product Model $\mathcal{M} = (S' = S \times Q \times \Sigma, \{\textit{`Accept'}\}, \pi', \mathbf{P}, L)$, Prediction Horizon $h$)
  output: Table $MT(\sigma, t)$ for all $\sigma \in S'$ and $t \in [1, h]$
2 begin
3     forall $\sigma, \sigma' \in S'$ do
4         if $L(\sigma) = \textit{`Accept'}$ then
5             if $\sigma = \sigma'$ then
6                 $\mathbf{P}_{Acc}(\sigma, \sigma) \leftarrow 1$
7             else
8                 $\mathbf{P}_{Acc}(\sigma, \sigma') \leftarrow 0$
9         else
10            $\mathbf{P}_{Acc}(\sigma, \sigma') \leftarrow \mathbf{P}(\sigma, \sigma')$
11    for $\sigma \in S'$ do
12        $MT(\sigma, 1) \leftarrow \sum_{\sigma' \in S'} \mathbf{P}_{Acc}(\sigma, \sigma')$
13    for $t \leftarrow 2$ to $h$ do
14        $MT(\sigma, t) \leftarrow \sum_{\sigma' \in S'} \mathbf{P}_{Acc}(\sigma, \sigma') MT(\sigma', t-1)$
15    return $MT$
```

Algorithm 7: Constructing the monitor as a lookup table.

If the prediction model is a DTMC:

$$\begin{aligned} Obs(s, a) &= \begin{cases} 1 & \text{if } L(s) = a \\ 0 & \text{otherwise.} \end{cases} \\ Trans(s, s') &= \mathbf{P}(s, s') \end{aligned} \tag{4.9}$$

If the prediction model is an HMM:

$$\begin{aligned} Obs(s, a) &= O(s, a) \\ Trans(s, s') &= T(s, s') \end{aligned} \tag{4.10}$$

Table 4.3: The monitor table for φ_{F} and φ_{G}, from the product model in Figure 4.4.

φ_{F}			φ_{G}		
$\sigma = (\mathbf{s}, \mathbf{q}, \mathbf{o})$	$\mathbf{t}$	$\mathbf{MT}(\sigma, \mathbf{t})$	$\sigma = (\mathbf{s}, \mathbf{q}, \mathbf{o})$	$\mathbf{t}$	$\mathbf{MT}(\sigma, \mathbf{t})$
	1	0		1	0
	2	0		2	0
$(s_1, q_0, (ii, 0))$	3	0.12	$(s_1, q_0, (ii, 0))$	3	0.13
	4	0.12		4	0.13
	5	0.16		5	0.16
	1	0		1	0
	2	0.26		2	0.25
$(s_3, q_0, (tt, 0))$	3	0.26	$(s_2, q_0, (hh, 0))$	3	0.25
	4	0.32		4	0.32
	5	0.32		5	0.32
	1	0.50		1	0.48
	2	0.50		2	0.48
$(s_6, q_0, (hh, 0))$	3	0.62	$(s_5, q_0, (tt, 0))$	3	0.61
	4	0.62		4	0.61
	5	0.66		5	0.65

4.4.1 Computing the Posterior State Probability Distribution

The Bayesian-based approach to derive the state of the system computes the posterior probability distribution over the state space, given the execution path u. The Forward-Backward algorithm (Algorithm 5) employs dynamic programming to compute the posterior probability distribution using the prediction model and u:

$$Pr(X^{(n)} = s \mid u, \Theta) \tag{4.11}$$

where n is the length of u.

Algorithm 8 demonstrates computing (4.11) for all the states of the prediction model using the Forward-Backward algorithm. Since the algorithm is defined for an HMM, if the prediction model is a DTMC we turn the observation function into a probability distribution ($Pr_{\mathcal{D}}(S)$ in line **5**). Notice that this algorithm can also be used for a deterministic DTMC, where the returned probability distribution will only contain one state. The *for* loop in line **4** calls the Forward-Backward function in Algorithm 5 to compute the posterior probability for each state. The algorithm normalizes the probability in line **5** and returns the posterior probability distribution denoted by $Pr_{\mathcal{D}}(S)$.

```
1 PosteriorState($\mathcal{M}, u$)
  inputs : Execution observation $u$, Prediction Model $\mathcal{M}$ with parameters
           $\Theta = (\pi, Trans, Obs)$ and state space $S$
  output: Posterior probability distribution over $S$
2 begin
3     foreach $s \in S$ do
4         $Pr(s) \leftarrow$ Forward-Backward($u, \Theta, s$)
5     $Pr_{\mathcal{D}}(S) \leftarrow \frac{Pr(s)}{\sum_{s \in S} Pr(s)}$
6     return $Pr_{\mathcal{D}}(S)$
```

Algorithm 8: Computing the posterior state probability distribution using the FORWARD-BACKWARD algorithm.

Complexity. Algorithm 8 calls the FORWARD-BACKWARD algorithm M times, where M is the number of states in the prediction model. Therefore, in total Algorithm 8 is of $O(M^3 \cdot n)$. For an average sized prediction model the CPU time of the monitor might still be very limited for this complexity. We can modify Algorithm 8 so that the most probable state is returned, instead of the distribution. This approximation improves the performance of obtaining the state to a quasilinear time complexity, and explained in the next section.

4.4.2 Viterbi Approximation

The VITERBI algorithm [126] provides an approximation procedure, where instead of calculating the posterior probability distribution, only the most probable state is determined.

For an observation sequence $Y = Y^{(1:n)}$, the VITERBI algorithm [126, 39] derives $X^{*(1:n)} = \text{argmax}_{X^{(1:n)}} Pr(X^{(1:n)}|Y, \Theta)$, the so-called *Viterbi path*. Let $v_t(s)$ be the probability of the Viterbi path ending with state s at time t:

$$v_{t+1}(s) = O(s, Y^{(t+1)}) \max_{s' \in S}(v_t(s')T(s', s)) \tag{4.12}$$

To find $X^{*(t+1)}$ the monitor only requires $v_t(s')$ for all $s' \in S$. Therefore, we can obtain $X^{*(t+1)}$ by using only two vectors that maintain the values of $v_{t+1}(s)$ and $v_t(s)$: we call the *current* and the *next Viterbi vectors*, respectively.

Algorithm 9 demonstrates state estimation using the VITERBI algorithm. Line **3** initializes the current Viterbi vector. Each iteration of the *for* loop in lines **6-13** is over

```
1 $\textsc{Viterbi}(\mathcal{M}, u)$
  inputs : Observation sequence $Y$, Prediction Model $\mathcal{M}$ with state space $S$
  output: $s \in S$
2 begin
3     foreach $s \in S$ do $v(s) \leftarrow Obs(s, Y^{(1)})\pi(s)$ // Initialize the Viterbi vector
4     forall $i \leftarrow 1$ to $|Y|$ do
5         $s \leftarrow \text{argmax}_s\, v(s)$
6         forall $s \in S$ do // Updating the next Viterbi vector
7             $v_{next}(s) \leftarrow Obs(s, Y^{(i+1)}) \max_{s'}(v(s')Trans(s', s))$
8         $v \leftarrow v_{next}$, $i \leftarrow i + 1$
9     return $\textsc{Vectorize}(s)$
```

Algorithm 9: State estimation using the VITERBI approximation.

one observation in the sequence Y. For each observation $Y^{(i)}$, the next Viterbi vector is obtained in lines **6-8**, according to (4.12). The algorithm returns a vector of the states with the most probable state having the value 1 and the rest having the value 0 by calling the VECTORIZE function in line **9**, so that the returned value has a similar structure to that of Algorithm 8. This will enable us to use either Algorithm 8 or 9 in the monitoring procedure without having to change their outputs.

Complexity. Each iteration of the loop in lines **6-13** is done at the same time as observing a new element, i.e., when the system emits a new observation symbol. For a prediction model with M (hidden) states, updating the Viterbi vector requires $O(M)$ operations of finding maximums, which can be improved to $\log(M)$ using a Max-Heap. Therefore, each monitoring iteration is of $O(M \cdot \log M)$ in execution time, which is significantly better than Algorithm 8.

Example. Table 4.4 demonstrates the estimated states of the HMM in Figure 4.3 for each step of u'_{F} and u'_{G}. Notice that the posterior state probabilities are 1 for the states shown on the table and 0 for all other states. The VITERBI estimation has the same results. $\square$

Table 4.4: The estimated state of the HMM in Figure 4.3 for each step of u'_{F} and u'_{G}.

u'_{F} State	$(ii,0)$ s_1	$(tt,0)$ s_3	$(hh,0)$ s_{10}	$(tt,0)$ s_3
u'_{G} State	$(ii,0)$ s_1	$(hh,0)$ s_2	$(tt,0)$ s_4	$(hh,0)$ s_2

```
1  MONITORING($u, MT, \mathcal{M}, h$)
   inputs : Execution observation $u$, The probability table $MT$, Prediction model
            $\mathcal{M}$, Prediction horizon $h$
   output: $\mathcal{PM}(\mathcal{C}(u, \mathcal{M}, \varphi, h))$ (see (3.1))
2  begin
3      $t \leftarrow h$
4      $q \leftarrow q_I$
5      $n \leftarrow |u|$
6      forall $i \leftarrow 1$ to $n$ do
7          $StateProbVector \leftarrow$ STATEESTM($\mathcal{M}, u$)
8          $q \leftarrow \delta(q, u^{(i)})$
9          PRINT($MT(\sigma = (s, q, u^{(i)}), t) \times StateProbVector$)
10         if $q \in F$ or $t = 0$ then
11             $t \leftarrow h$
12         else
13             $t \leftarrow t - 1$
```

Algorithm 10: Predictive monitoring of software execution.

4.5 Monitoring

Algorithm 10 demonstrates the monitoring procedure using the monitor table constructed in Algorithm 7. The algorithm takes the entire execution path as an argument, thus, can be seen as an offline monitoring procedure. It is straightforward to call Algorithm 10 upon the occurrence of an event or with a certain frequency to respectively create event- or time-triggered online monitoring procedures [19].

The variable t is the horizon index that is initialized to h (line **3**), i.e., the prediction horizon. The horizon index is reset to h, if $q \in F$, i.e., the property is satisfied, or once $t = 0$, i.e., we have reached the prediction horizon and the property is still not satisfied.

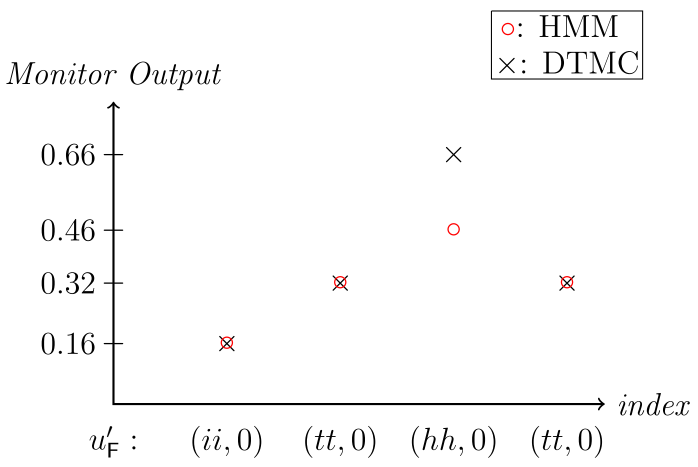


Figure 4.6: The output of the predictive monitor for u'_{F} at different time indices and using DTMC and HMM as prediction models.

Otherwise, t is decremented (lines **10-13**).

The state of the DFA is derived on the fly, by starting from the initial state q_I (line **4**), and updating the state by using the transition function on the output symbol $u^{(i)}$ (line **8**).

The function StateEstm in line **7** calls one of the Algorithms 8 or 9 which return a row vector that has M' rows, where M' is the size of the product model. Each entry of the vector contains the probability of the hidden state after observing $u^{(1:i)}$. If there is only one state (e.g., in the case of the Viterbi algorithm) the entry has the value 1 for that state and 0 for the rest of the states. The monitor outputs the cross product of the vector $StateProbVector$ by the vector $MT(\sigma, t)$ in line **9** to obtain the probability described in (3.1).

Complexity. For each iteration of the *for* loop in line **6**, i.e., each observation, the complexity of Algorithm 10 is $O(M')$ for the multiplication in line **9**, times the complexity of the state estimation.

Example. Figure 4.6 displays the result of the predictive monitor for u'_{F} at different indices using Algorithm 10, assuming that the monitor is invoked for each index. The prediction results are shown using DTMC and HMM (Figures 4.2 and 4.3). Notice that the $StateProbVector$ has only one element with value 1, and the rest are 0 (see Table 4.4). □

4.6 Prediction Evaluation

In this section, we define two measures that we use to evaluate the accuracy of the predictions made by the monitor. One measure is defined using the actual model of the system, and the other measure is defined for when the actual model is unavailable. In this case, we assume that we are still able to run the system an arbitrary number of times to collect additional samples.

4.6.1 Evaluation Using the True Model (White-box)

Using a true model of the system execution, we are able to obtain the true values of (4.5), and use Mean-Square Prediction Error (MSPE) [45] to measure the performance of the predictions by the trained models. The evaluation is conducted on a separate set of *iid* samples, where the following is computed for each instance i that the prediction is made:

$$\varepsilon_i^2 = (Pr(s^{(i)} \models \Diamond^{\leq h}\varphi) - Pr(\sigma^{(i)} \models \Diamond^{\leq h} \text{`}Accept\text{'})^2, \tag{4.13}$$

where $s^{(i)}$ is the state of the actual model and $\sigma^{(i)}$ is the state of the product model at index i (see Definition 18). The value of ε_i^2 reflects both the error associated with training the prediction model as well as the state estimation.

For a single sample execution, we define MSPE as the average of (4.13), i.e., $\frac{1}{t}\sum_{i=1}^{t} \varepsilon_i^2$, where t indicates the number of points on the sample where a prediction is made.

Example. The MSPE of the monitor with the output shown in Figure 4.6 using the DTMC as prediction model is 5×10^{-5} and using the HMM as prediction model is 10^{-2}. The MSPE of the HMM monitor is high due to the error of the training algorithm that did not converge to a global optimum, and consequently, the model in Figure 4.3 has the hidden states with labels $(tt, 0)$ and $(hh, 0)$ that are not fully distinguished, e.g., states s_3 and s_{10} emit the same symbol, i.e., $(tt, 0)$. □

4.6.2 Evaluation using Hypothesis Testing (Black-box)

In the case where the true model of the system is not available, we use hypothesis testing on a set of execution samples from the system to evaluate the accuracy of the prediction. We call these samples the *test samples*, which are different from the *training samples*.

Given the execution path u, let $u^{(i)..(i+\lambda_i(\mathcal{A}))}$ be its extension of length $\lambda_i(\mathcal{A})$ at point i that is accepted by a given DFA $\mathcal{A}$, i.e., $(u^{(i)..(i+\lambda_i)}) \in L_{\mathcal{A}}$. For brevity, we use λ_i instead of $\lambda_i(\mathcal{A})$. Recall that the monitor output at point i is the probability of all the extensions of length at most h that are accepted by $\mathcal{A}$ (4.5). For any $\lambda_i \leq h$ we have:

$$Pr(\sigma^{(i)} \models \Diamond^{\leq h} \textit{`Accept'}) \geq Pr(\sigma^{(i)..(i+\lambda_i)} \models \textit{`Accept'})$$
$$\lambda_i \times Pr(\sigma^{(i)} \models \Diamond^{\leq h} \textit{`Accept'}) \geq \lambda_i \times Pr(\sigma^{(i)..(i+\lambda_i)} \models \textit{`Accept'})$$

We define $\hat{\lambda}_i = \lambda_i \cdot Pr(\sigma^{(i)} \models \Diamond^{\leq h} \textit{`Accept'})$ as the expected value of λ_i estimated by the monitor. Therefore, one can obtain the following minimum error of the prediction at point i:

$$\varepsilon_i^{min} = \lambda_i - \hat{\lambda}_i \tag{4.14}$$

Notice that since $\lambda_i \geq \hat{\lambda}_i$, ε_i^{min} is always positive. If there is no k, $i < k < \lambda_i$, such that $(u^{(i..i+k)}) \in L_{\mathcal{A}}$, i.e., $u^{(i..i+\lambda_i)}$ is the minimal extension that is accepted by $\mathcal{A}$, then $\varepsilon_{i+t}^{min} = (\lambda_i - t) - \hat{\lambda}_{\lambda_i - t}, 0 \leq t < \lambda_i \leq h$, where t is the horizon index. As a result, the value of ε_i^{min} can be computed on the fly.

In our implementation, we assume that there exists at least one point $k \leq h$ such that $(u^{(i..i+k)}) \in L_{\mathcal{A}}$; otherwise, ε_i^{min} is not well-defined, and the prediction accuracy can not be calculated. If such a point does not exist, we can extend the prediction horizon by increasing h such that there is at least one accepting extension in the trace. The rest of the path after the last point in which the trace is accepted by $\mathcal{A}$ is discarded, as there is no observation to compare the prediction and compute the error.

To assess the performance of the prediction, we use hypothesis testing on a set of test samples. Given the expected value of a prediction λ_i, let $\Lambda = \frac{1}{\tau}\sum_{i=1}^{\tau} \lambda_i$ be the random variable that represents the mean of all λ_i values, for $1 \leq i \leq \tau$. Notice that for *iid* samples, the value of Λ for a trace is independent of that value for other traces.

Let $\bar{\lambda}_M$ be the estimation of Λ by the monitor over a set of monitored traces, and $\bar{\lambda}$ be the mean of Λ on a separate set of n *iid* test samples with variance ν. We test the accuracy of the prediction using the following two-sided hypothesis $H_0 : \bar{\lambda}_M = \bar{\lambda}$. Using confidence α, we employ the t-distribution to test H_0:

$$\frac{\bar{\lambda} - \bar{\lambda}_M}{\frac{\sqrt{\nu}}{n}} \leq t_{n-1,\alpha} \tag{4.15}$$

4.7 Limitations

In this section we discuss some of the limitations and trade-offs that need to be made in constructing a predictive monitor. First notice that we assume that the observable behaviour of the system is learnable by a Markovian model. This implies that the system possesses the Markovian property, i.e., the future behaviour of the system can be estimated based on a static probabilistic model of the past behaviour. To a certain extent, modeling the real systems that exhibit extreme dynamic behaviour violates the Markovian assumption. With adding more states it is possible to model a system with non-Markovian property using a Markovian model but again that poses the risk of state explosion in the learned model.

In addition, adding more states to the prediction model although may increase the prediction accuracy; it also impacts the state estimation performance which may increase the monitoring overhead at runtime. Therefore it is important to consider the trade-off between the prediction accuracy and the overhead of the monitor, which may be different depending on the nature of the system.

4.8 Case Studies

We evaluate $\mathcal{P}revent$ on two case studies: (1) the randomized dining philosophers from the PRISM case studies [38], which includes the original algorithm, and a modified version that we introduce specifically for evaluating $\mathcal{P}revent$; (2) the QNX Neutrino kernel traces collected from the flight control software of a hexacopter [104]. We show the estimation of *good* and *bad* extensions in the randomized dining philosophers and hexacopter traces to demonstrate the predictive monitoring of *guarantee* and *safety* properties, respectively. In addition, these properties represent the most commonly used property patterns in Dwyer *et al.*'s survey [40]: *response* pattern in the randomized dining philosophers algorithm, and the *absence* pattern for monitoring a regular safety property [8] in the flight control of a hexacopter. The implementation of monitoring in both experiments is conducted offline.

4.8.1 Randomized Dining Philosopher

We adapt Rabin & Lehmann's solution [100] to the dining philosophers problem that has the characteristics of a stochastic system to be trained using HMM. We also present a modification of their algorithm, which represents a generic form of decentralized online resource allocation [122], where our monitoring approach can be seen as a component of the *liveness enforcement supervisory* [85].

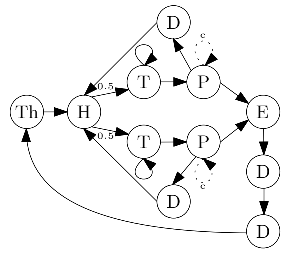
We consider the classic setting of the problem, where philosophers are in a ring topology and are selected for execution by a fair scheduler. Figure 4.7a demonstrates a state diagram of one philosopher, with Th, H, T, P, D, and E representing the philosopher to be, respectively, *thinking*, *hungry*, *trying*, *picking* a fork, *dropping* a fork, and *eating*. A philosopher starts at Th, and immediately transitions to H[1]. Based on the outcome of a fair coin, the philosopher then chooses to pick the left or the right fork if they are available, and moves to T. If the fork is not available, the philosopher remains at T until it is granted access to the fork. The philosopher moves to E, if the other fork is available; otherwise, the philosopher drops the obtained fork, moves to D, and eventually transitions back to H. After the philosopher finishes eating, it drops the forks in an arbitrary order D, and moves back to Th. The algorithm is deadlock-free, but lockouts are still possible [100]. The PRISM model of the algorithm for 3 philosophers is provided in Appendix A.2.

Our modification of the algorithm is to add a self-transition at P (shown with dotted lines in Figure 4.7a): a philosopher does not drop the first obtained fork with probability c and it stays at P. The transition from P to D has the probability $1 - c$ (not shown in the figure). This modification enables the philosopher to control its *waiting time*: the period between when it becomes hungry for the first time after thinking and when it eats. A higher value of c means that, instead of going back to H, the philosopher is more likely to stay at P so that as soon as the other fork is available it starts eating. In a distributed real-time system, where each philosopher represents a process with unfixed deadlines, changing the value of c enables the processes to dynamically adjust their waiting time according to their deadlines. It is not difficult to observe that as long as there is at least one philosopher with $c \neq 1$, the symmetry that causes the deadlock [100] eventually breaks, and our modification to the algorithm maintains it deadlock-free.

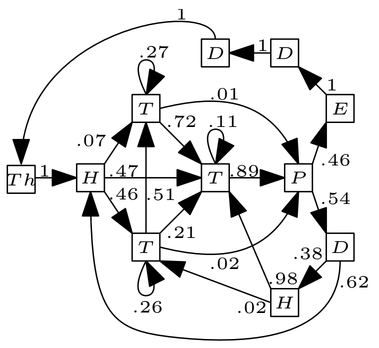
The purpose of the experiments is to implement a monitor that observes the outputs of a single philosopher, and predicts a potential starvation (lockout) by estimating the extensions that leads to *eating*.

Predicting Starvation at Runtime. We use the Matlab HMM toolbox to train HMMs, with 100 *iid* samples collected from the implementation of the modified version, with $c = 1$ for all philosophers except the one that is being monitored. The trained model presents

[1]For simplicity, we remove a self-transition to Th; however, unlike [38] we do not merge the states Th and H because we want to distinguish between the incoming transitions to Th and H in computing the waiting time.



(a) The states of one philosopher. The dotted self-transitions display our modification.



(b) The trained HMM of one philosopher, in a system with three philosophers.

Figure 4.7: Training an HMM for the monitored philosopher in a program with 3 philosophers.

the behavioural signature of the system when a *longer waiting time* is likely. The size of HMM (i.e., the number of hidden states) is chosen based on the BIC score of each model with different sizes (see Section 4.1.2). Figure 4.7b demonstrates the trained HMM of one philosopher that is constructed from the traces of a 5-second execution of three philosophers, which contains in average 1500 state transitions for each philosopher. The trained model reflects the distribution of the prefixes in the training sample, which in turn is determined by how the scheduler as well as other philosophers, behaved during training, i.e., how the nondeterminism of the model is resolved during training. For instance, multiple consecutive *try*s in the training sample create several states in the trained HMM, each emitting the symbol T, but only one has a high probability to transition to P and the others model the state where the philosopher can not pick a fork. The finite extensions that we consider for prediction are based on the following regular expression: $(\neg H)^*(H(\neg E)^*E(\neg H)^*)^*$.

Figure 4.8 gives a comparison between the prediction results of two trained models for three philosophers, one trained using the samples from the original implementation (LR) and the other one trained from the samples of the modified version (LR-sap). The monitor observes the behaviour of one philosopher and at each index (horizontal axis) outputs the probability (vertical axis) that the philosopher can eat within maximum 33 steps ($h = 33$). The output probability can be interpreted as *the probability of that the monitored philosopher does not starve*. This monitored trace is synthesized in a way that it does not contain any *eat*, and up to point 33 the philosopher is only at state T. After that, the philosopher frequently picks and drops a fork.

According to the results of Figure 4.6, when the last event of a prefix is *pick*, compared to when it ends with any other observations, the philosopher has a higher chance to reach *eat* (e.g., with probability 0.98 at index 35 using the LR-sap model); however, since the HMM maintains the history of the trace, a prefix with frequent *pick*s and *drop*s shows a decline in the probability of observing *eat* and a potential starvation (e.g., with probability of 0.8 at index 57 using LR-sap model).

In addition, the results in Figure 4.8 demonstrate that the model trained on the *bad* extensions (LR-sap) provides an under-approximation of the model that is trained on the *good* traces (LR), thus, tends to produce more false positives. More specifically, the monitor that uses the LR-sap prediction model predicts that the philosopher is going to starve with a higher probability than the monitor that uses the LR model. The reason is that the LR-sap model is learned from the traces where the philosophers stay at the state P hence increasing the chances of starvation for their neighbours.

The summary of the results is displayed in Table 4.5. We use PRISM to perform the reachability analysis on the product of the trained HMM and the DFA, and build the monitor lookup table. The size of the product model is equal to the size of the HMM, as each state in the trained HMMs emits exactly one observation. The *minimum prediction horizon* (denoted by h^{min}) is the mean of the length of the shortest good or bad finite extensions in the test sample, and obtained empirically from 100 test samples. We choose the prediction horizon to be three times as large as h^{min} during monitoring. The average of the estimated length of the acceptable extensions by the monitor is shown as $\bar{\lambda}_M$, and the mean of the error on the entire testing set is denoted by $mean(\varepsilon^{min})$. On average, the monitor predicts the next *eat* (within the prediction horizon) with one step error. Therefore, the monitor is not able to detect the waiting periods that are longer than $3 \times h^{min} \pm 1$. Increasing the prediction horizon decreases the error, with the cost of a larger monitor table (MT). The value of $\bar{\lambda}_M$ is influenced by the total number of discrete events produced by the monitored philosopher. With more philosophers $\bar{\lambda}_M$ decreases because the monitored philosopher, and hence the monitor, are scheduled less often.

4.8.2 Hexacopter Flight Control

In this section, we apply $\mathcal{P}revent$ to detect some injected faults in the QNX Neutrino's kernel calls [99]. The traces are obtained using QNX `tracelogger` during the flight of a

Table 4.5: Prediction results on 100 test samples.

N	# of hidden states	BIC ($\times 10^3$)	h^{min} # of steps	# of rows in MT	$\bar{\lambda}_M$ # of steps	$mean(\varepsilon^{min})$ # of steps
3	17	25.1	9.94	360	9.30	1.75
4	14	11.9	5.49	180	5.30	1.28
5	10	10.1	6.36	154	6.16	0.80
6	14	7.69	5.61	180	5.17	1.05
7	16	6.09	4.28	170	3.84	1.06
8	10	5.42	4.94	110	4.32	1.33
9	14	4.83	3.15	120	2.77	0.92
10	10	4.40	4.31	110	3.84	0.97

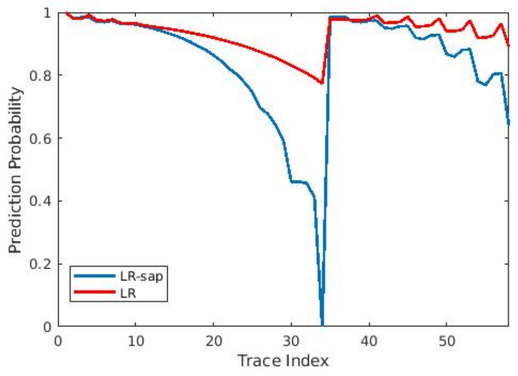


Figure 4.8: The comparison of the prediction results from two trained models.

hexacopter[2]. The vehicle is equipped with an autopilot, but can be controlled manually using a remote transmitter. The autopilot system uses a cascaded PID controller. QNX's microkernel uses message-passing architecture, where almost all the processes (even the kernel processes) communicate via sending and receiving messages that are handled by the kernel calls `MSG-SENDV`, `MSG-RECEIVEV`, and `MSG-REPLY`. Figure 4.9a shows a sub-trace of the kernel call sample from the hexacopter flight control system.

In this case study, the faults are injected by introducing an interference process, with the same priority as the autopilot process, that simply runs a while-loop to consume CPU time. The interference process interrupts message-passing between the processes of the same or lower priorities, causing a kernel call to handle the error, typically due to a timeout, and to unblock the sender (shown as event `MSG_ERROR` in Figure 4.9a). The purpose of the monitor is to predict the existence of an interference process by monitoring the kernel calls.

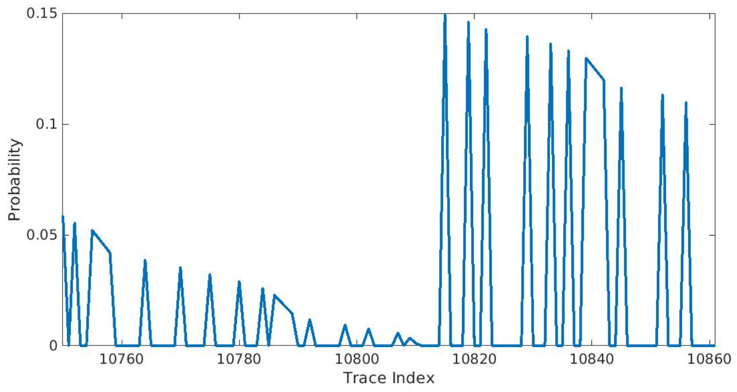
In this experiment we use SFIHMM [35] to be able to train HMMs on multiple cores. We train several HMMs from 1-second of the auto-pilot execution, with the intervening process in full effect, on an Intel Xeon 2.40GHz 128GB RAM machine with Debian 9.3. The HMM with the minimum BIC has 19 states. The regular expression $(\neg\texttt{MSG\_ERROR})^*$ $(\texttt{MSG\_ERROR})\Sigma^*$ is used to generate the finite extensions that contain the bad prefixes of the property $\Box\neg\texttt{MSG\_ERROR}$.

The monitoring is performed on part of the trace that is generated from another scenario, where the interference process is partially in effect and started executing in the mid-

[2]Full system description is available at https://wiki.uwaterloo.ca/display/ESGDAT/QNX+Hexacopter+Flight+Control+Dataset

```
⋮
10850 : MSG-SENDV
10851 : CONNECT-CLIENT-INFO
10852 : MSG-REPLYV
10853 : MSG-RECEIVEV
10854 : MSG-SENDV
10855 : CONNECT-CLIENT-INFO
10856 : MSG-REPLYV
10857 : MSG-RECEIVEV
10858 : MSG-RECEIVEV
10859 : MSG-SENDV
10860 : CONNECT-CLIENT-INFO
10861 : MSG-ERROR
⋮
```

(a) A sub-trace of the kernel calls, 20 steps before the event `MSG_ERROR`.



(b) The monitor prediction 50 steps before the event `MSG_ERROR`.

Figure 4.9: The monitoring of $\square\neg$`MSG_ERROR` on the flight control trace with the interference process.

dle of the flight. The prediction results are depicted in Figure 4.9. The points where the probability is *zero* arise because the monitor was not able to correctly estimate the hidden state of the model. In this case for example, three consecutive instances of `MSG_RECEIVEV` have not appeared in the training sample, hence the prefix can not be associated to any state of the model by the monitor. More training samples are required to enable the monitor to estimate the correct state of the model.

The event `MSG_ERROR` is emitted at index $10,861$, and the probability of the prefix that contains `MSG_ERROR` within the next 50 steps is 0.15 at index $10,815$. This is the highest probability that the monitor produces for this trace. The decline that we see after this point is due to the decrement of the prediction horizon as the execution proceeds (see Algorithm 10). This result notably shows that even though the monitor was not trained exactly on this scenario; it still can predict that the message error will occur with 15% chance, almost 45 steps before its occurrence.

4.9 Related Work

To the best of our knowledge [7] is the first approach in verifying finite paths based on the extensions obtained from a trained model.

Learning Markov Chain models for verification purposes is introduced in [89, 80]. HMMs are trained in [64] for statistical model checking. Our work focuses on predictive monitors using a similar technique. We also provide assessments for evaluating the learned model and inferring its size.

HMMs have been used in runtime monitoring of CPSs [120, 63, 117, 118, 131, 116, 11]. Sistla *et al.* [117] propose an *internal* monitoring approach, i.e., the property is specified over the state space instead of the observation space, using specification automata and HMMs with infinite states. Learning an infinite-state HMM is a harder problem than a finite HMM, but does not require inferring the size of the model [16].

The notion of *acceptance accuracy* and *rejection accuracy* in [116] are the complement to our notion of prediction error. According to their definition, the VITERBI approximation generates a *conservative* monitor for any regular safety property and regular finite horizon. The analytical method to find an upper bound as a threshold for the timeliness of a monitor [118] can be applied to find an upper bound for the prediction horizon h.

Several works focus on efficiently estimating the internal states of an HMM at runtime using particle filtering [120, 63]. Particle filtering uses weights based on the number of particles in each state, and updates the weights in each observation. The VITERBI algorithm provides the most likely state, as an overapproximation. Adaptive Runtime Verification [11] couples state estimation [120] with a feedback control loop to generate several monitors that run on different frequencies. These works are orthogonal to $\mathcal{P}revent$ and can be combined to improve the performance of our framework.

Chapter 5

Abstraction of the Prediction Model

The size of the prediction model significantly impacts both the space and the time overhead of the monitor. To reduce the size of the prediction model, we apply abstraction to the observation space by reducing the number of symbols in the alphabet and converting the training samples to abstract traces based on the new alphabet. More specifically, we use a finite partitioning of the observation space, Σ, and assign a new symbol to each partition, therefore, our abstraction is realized as a *projection*.

Typically, training the prediction model from abstract samples has shorter training time and produces a smaller model. Moreover, since the size of the product model is directly influenced by the size of the alphabet, the monitor table will have a smaller size too (see Section 4.2.1). Since the abstraction is inferred from execution samples, we only focus on the discrete-time reachability properties that specify reaching some *target* symbols on the observation space (e.g., φ_{F}). Notice that these properties are a subset of guarantee and safety properties.

In this chapter, first we describe our abstraction as a form of symmetry reduction [69] on the observation space that is implemented at the trace-level [89] (Section 5.1). Second we explain how to recognize symbols that have similar *transient probability* [70] to reach one or more target symbols by using the *k-gapped pair model* [36] (Section 5.2). Third we show how the symbols that have similar empirical probability to reach the target symbols within k steps are lumped into equivalence partitions over the observation space using hypothesis testing (Section 5.3). The *symmetrical* symbols are then used to convert the traces of the training set into *abstract* traces, which in turn, are used to train a prediction model. Fourth we discuss how the prediction of the trained models from abstract traces are comparable to the prediction of the models trained from the concrete traces (Section 5.4).

We show the effectiveness of our approach on a case study (Section 5.6) and conclude the chapter with reviewing some related work (Section 5.7).

5.1 Abstraction via Projecting Symmetrical Symbols to a Single Symbol

A straightforward abstraction is to divide the observation space into two partitions based on the symbols that appear in the property, denoted by $\mathcal{G}$ and we call the target symbols, and the symbols that do not appear in the property, i.e., $\Sigma \setminus \mathcal{G}$, which we denote by $\Sigma_{\bar{\mathcal{G}}}$ and call non-target symbols [89]. We define the projection $R_{\mathcal{G}} : \Sigma \to \{gg, nn\}$ such that a symbol in Σ is mapped to the symbol gg if it is in the property, i.e., $R(\sigma) = gg$ iff $\sigma \in \mathcal{G}$; it is mapped to the symbol nn otherwise.

Example. Consider $\mathcal{G}_{\varphi_{\mathsf{F}}} = \{(hh, 6)\}$ and the traces u_{F} and u'_{F}. The projection $R_{\mathcal{G}} : \Sigma_{die} \to \{gg, nn\}$ maps $(hh, 6)$ to gg and all other symbols in Σ_{die} to nn. The projection of u_{F} and u'_{F} using the projection relation $R_{\mathcal{G}_{\varphi_{\mathsf{F}}}}$ are, respectively, $\tilde{u} = (nn)(nn)(nn)(gg)$ and $\tilde{u'} = (nn)(nn)(nn)(nn)$. □

The projection $R_{\mathcal{G}}$, however, may merge the non-target symbols, which have a non-zero probability to reach a target symbol within some bounded steps, with symbols that never reach the target symbols, i.e., have probability *zero*. The symbols with probability zero of reaching the target symbols can be discarded as they are irrelevant to the prediction. As a result, the information that some symbols may provide for prediction might be lost.

Example. Compare the symbols $(hh, 0)$ and $(hh, 4)$. The former often appears immediately before the target symbol $(hh, 6)$ in a sample path; whereas the latter has no appearance before any of $(hh, 6)$ (see Figure 3.2). Both are replaced with nn in $R_{\mathcal{G}}$, thus the predictive information provided by $(hh, 0)$ and $(hh, 4)$ are combined. □

The key insight in the proposed abstraction method is to not only detect the symbols that do not reach the target symbols, but also recognize the ones that have similar empirical probability within a fixed number of steps, and merge them together. As a result, we exploit the notion of *symmetry* [69] on the observation space to recognize the symbols with similar prediction power and lump them into the same partition. We define the symmetry relation with respect to reaching the target symbols. More specifically, we say two symbols are symmetrical *iff* the probability measure of a fixed length path, that starts from some non-target symbol and ends with a target symbol, is equal.

Let $\mathcal{M}$ be a deterministic DTMC. Let $P_k : \Sigma_{\bar{\mathcal{G}}} \to \Sigma_{\bar{\mathcal{G}}}$ be a permutation on the set containing only non-target labels, such that $Pr(Path_k^{\mathcal{G}}(P_k(\sigma))) = Pr(Path_k^{\mathcal{G}}(\sigma))$ for all $\sigma \in \Sigma_{\bar{\mathcal{G}}}$, and some fixed integer $k > 0$, where $Path_k^{\mathcal{G}}(\cdot) = Path_k(\cdot) \cap L_{\mathcal{G}}$. A group of permutations on $\Sigma_{\bar{\mathcal{G}}}$, such as P_k, provides an equivalence relation (so-called the *orbits*) on $\Sigma_{\bar{\mathcal{G}}}$ that together with $\mathcal{G}$ define the equivalence classes over the observation space. We denote by Σ_k the abstract alphabet set that contains a unique representative symbol for each partition, and by $R_k : \Sigma \to \Sigma_k$ the corresponding projection that maps each symbol to its respective symbol in the abstract alphabet.

In the remainder, we use the k-gapped pair model combined with hypothesis testing to infer R_k, and consequently, Σ_k.

5.2 k-gapped Pair Model

The k-gapped pair model [36] has been successfully applied in mining biological sequences [57] as well as context-dependent text prediction [25]. We use the k-gapped pair model to extract the symmetrical symbols with respect to reaching some target symbols in the some execution samples.

A k-gapped pair model is a triplet (σ, σ', k), where $\sigma, \sigma' \in \Sigma$, and $k \geq 0$ is an integer that indicates the number of steps (gaps) between σ and σ'. If $k = 0$ the k-gapped pair is equivalent to a *bigram* [83].

The *k-gapped occurrence frequency* of the symbols σ and σ' is the frequency that σ appeared within exactly k steps before σ' over the sample path. Assuming that $\sigma' \in \mathcal{G}$, we use the sum of k-gapped occurrence frequency of a given symbol in the sample set, and define it as the *k-prediction support*.

Algorithm 11 shows computing k-prediction support of symbol σ. Symbols $\sigma^{(j)}$ and $\sigma^{(j+k+1)}$ are the j^{th} and $(j+k+1)^{th}$ symbols of the sample path u_i in each iteration of the loop in line **3**, and $\mathbb{1}_{(\sigma^{(i)}=\sigma \wedge \sigma^{(i+k+1)}=\sigma')}(u_i)$ in line **5** is the indicator function that returns 1 if $\sigma^{(i)} = \sigma$ and $\sigma^{(i+k+1)} = \sigma'$; and 0 otherwise. The output of Algorithm 11 is the vector $[F_{u_1} \ldots F_{u_m}]$, the k-prediction support values of each sample path for symbol σ.

Complexity. The complexity of Algorithm 11 is quadratic in the length of each sample path. Assuming that there are m samples with the size of $O(L)$, the time complexity of the algorithm is of $O(L^2 \cdot m)$.

Table 5.1: The k-prediction support of all the symbols except the target labels, $(hh, 6)$, with respect to φ_{F} for $k = 0, 1, 2$, obtained from samples in Table 4.1 (scale $\times 10^{-3}$).

	$k=0$	$k=1$	$k=2$
$(ii, 0)$	0	0	11.01
$(hh, 0)$	4.63	10.03	11.6
$(tt, 0)$	5.11	10.53	11.61
$(tt, 1)$	0	0	0
$(hh, 2)$	0	0	0
$(tt, 3)$	0	0	0
$(hh, 4)$	0	0	0
$(tt, 5)$	0	0	0

```
1 ComputePredictionSupport($S, \sigma, \mathcal{G}, k$)
  inputs : The iid sample set $S = [u_1 \dots u_m]$, $\sigma \in \Sigma$, the set of target labels $\mathcal{G}$, and
           an integer $k \geq 0$
  output: $[F_{u_1} \dots F_{u_m}]$
2 begin
3     foreach $u_i \in S$ do
4         $n \leftarrow length(u_i)$
5         $F_{u_i} \leftarrow \frac{1}{n-k-1} \sum_{j=1}^{n-k-1} \mathbb{1}_{(\sigma^{(j)}=\sigma \wedge \sigma^{(j+k+1)} \in \mathcal{G})}(u_i)$
```

Algorithm 11: Computing the k-prediction support of σ over the sample set.

Example. Table 5.1 demonstrates the k-prediction support of the symbols $(tt, 0)$ and $(hh, 0)$, for $k = 0, 1, 2$ with respect to φ_{F} ($\mathcal{G}_{\varphi_{\mathsf{F}}} = \{(hh, 6)\}$), over the 1000 samples shown in Table 4.1. $\square$

The k-prediction support of σ is essentially the empirical estimation of $Pr(\ Path_k^{\mathcal{G}}(\sigma))$. Under the assumption that $F_{u_1}, \dots, F_{u_m}$ is *covariance-stationary* [48], i.e., the mean is time-invariant and the autocovarinace function depends only on the distance k, both of which hold if the samples are *iid* and the underlying generating model is a deterministic DTMC, we are able to use hypothesis testing to extract R_k.

```
1  ExtractAbstractAlphabet(Sample set $S, \Sigma, \mathcal{G}, k$)
   output: Partition $[\mathcal{G} \cup V_1 \cup \cdots \cup V_t \cup \mathcal{R}]$ over $\Sigma$
2  begin
3      $t \leftarrow 1$
4      $\mathcal{R} \leftarrow \Sigma \setminus \mathcal{G}$
5      while $\mathcal{R} \neq \emptyset$ do
6          $[\sigma_{max}\ F_{max}] \leftarrow \max_{\sigma \in R} \sum_{i=1}^{m} F_{u_i}$
7          if $F_{max} = 0$ then break
8          $V_t \leftarrow \{\sigma_{max}\}$
9          for $\sigma \in \mathcal{R} \setminus \{\sigma_{max}\}$ do
10             $F_\sigma \leftarrow$ ComputePredictionSupport($S, \sigma, \mathcal{G}, k$)
11             if HypothesisTesting($F_{max} - F_\sigma$) then
12                 $V_t \leftarrow V_t \cup \{\sigma'\}$
13         $\mathcal{R} \leftarrow \mathcal{R} \setminus V_t$
14         $t \leftarrow t + 1$
```

Algorithm 12: Extracting the equivalence classes on the alphabet set.

5.3 Extracting Symbols with Similar Prediction Support

Algorithm 12 demonstrates the procedure of extracting the abstract alphabet set, based on the symmetry between the k-prediction support of the symbols. The algorithm receives the sample set, the alphabet set, the set of target symbols, and k, as inputs, and infers R_k, by generating the partitions $V_1, \ldots, V_t$.

The algorithm iterates over the symbols not considered in any equivalence classes, which are stored in $\mathcal{R}$ (the loop in lines **5**-**14**). In each iteration, the symbol with the maximum k-prediction support score is found in $\mathcal{R}$ and stored in σ_{max} with its score in F_{max} (line **6**). The score 0 for a symbol indicates that there is no path of length k to any target symbols from that symbol and we can end the procedure (line **7**); otherwise, the symbols with statistically similar k-prediction support score to σ_{max} are extracted from $\mathcal{R}$, and inserted in V_t (*for* loop in **9**-**12**).

The statistical testing is conducted via the function HypothesisTesting, which performs a two-sided hypothesis t-test to check $H_0 : F_{max} - F_\sigma = 0$. Depending on the chosen confidence, an appropriate number of samples is required to test H_0.

Algorithm 12 terminates, if there are no more symbols to classify, i.e., $\mathcal{R} = \emptyset$, or if all the remaining symbols in $\mathcal{R}$ have the k-prediction support equal to zero. We dedicate a representative symbol for each extracted partition, including $\mathcal{G}$ and $\mathcal{R}$ if it is not empty, and define R_k accordingly.

Complexity. At worst, a total number of $O(|\Sigma|^2)$ comparisons is required to extract the abstract alphabet. Given that the size of the actual model is at least as large as $|\Sigma|$, storing the entire vector of k-prediction support scores for all symbols is impractical for large models. In fact, to make use of the memory independence of the size of the alphabet, the computation of F in Algorithm 11 can be performed on the fly, which only depends on the size of the sample set. The space complexity of the algorithm depends on the size of the abstract alphabet, which at worst is $O(|\Sigma|)$, i.e., when there is no symmetry between symbols. However, for a model that has symbols with symmetry relation, our approach is more memory-efficient compared to inferring the abstract alphabet from a model that is trained from the concrete traces. In addition, training a model from abstract traces is in general faster than training a model from the concrete traces.

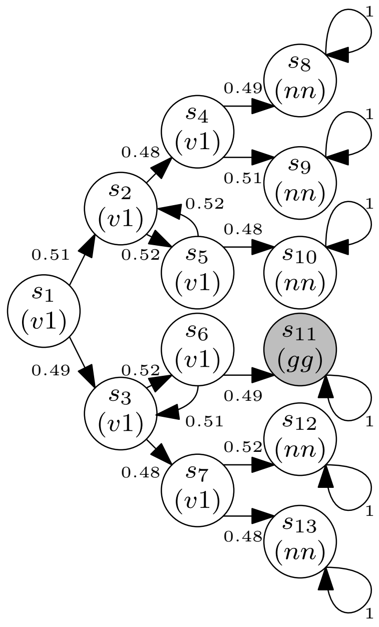


Figure 5.1: The abstract nondeterministic DTMC obtained by relabelling the states of the DTMC in Figure 3.2 using R_2.

Example. The equivalence classes obtained by Algorithm 12 for $k = 2$ are as follows: $R_2(\sigma_g) = gg, \forall \sigma_g \in \mathcal{G}_{\varphi_\mathsf{F}}, R_2(\sigma_v) = v1, \forall \sigma_v \in V_1 = \{(ii, 0), (hh, 0), (tt, 0)\}$, $R_2(\sigma_n) = nn, \forall \sigma_n \in \mathcal{R} = \Sigma_{die} - \mathcal{G}_\varphi - V_1$. Notice that according to the original model in Figure 3.2 the probability of reaching $(hh, 6)$ from $(ii, 0), (hh, 0), (tt, 0)$ in 3 steps (within 2 gaps) is equal. □

5.4 Prediction with Abstract Models

Let $\mathcal{M}_k$ be the quotient of $\mathcal{M}$, where Σ is replaced with Σ_k, and $L_k : S \to \Sigma_k$ such that $L_k(s) = R_k(L(s))$. We call $\mathcal{M}_k$ the abstract model, and it may be nondeterministic due to the applied abstraction.

Example. Figure 5.1 demonstrates the abstract model of the model in Figure 4.2 relabelled with R_2. Relabeling the model creates nondeterminism as the sequence $(v1v1v1)$ corresponds to several state sequences in the model. □

Observe that learning and abstraction are commutative [89], that is, we can apply abstraction to the traces and learn a model from abstract traces as described in Section 5.1, or learn a model from the actual traces and then apply abstraction. As we show in the following sections, the prediction results in both cases are similar; however, as mentioned before applying abstraction to the traces and learning from abstract traces leads to a faster learning process and is more efficient in space.

Learning the prediction model from abstract traces is similar to the algorithms described in Section 4.1. Given that $\mathcal{M}_k$ is deterministic, in the following we show that $\mathcal{M}_k$ is bisimilar to $\mathcal{M}$. Therefore bounded predictions from any state in both models are equal (Theorem 4). In the case that $\mathcal{M}_k$ is nondeterministic, we use a hidden state variable to infer the nondeterminism imposed by the abstraction.

5.4.1 Prediction via Deterministic Abstract DTMC

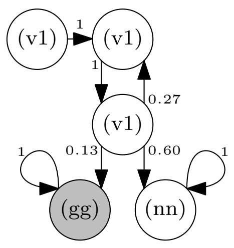
The following theorem is an extension of Theorem 1 in [89], which shows that, given that $\mathcal{M}_k$ is deterministic, in the limit the predictions are equivalent to the predictions made via the true model.

Theorem 4. *Let $\mathcal{M}_k : (S, \Sigma_k, \pi, \mathbf{P}, L_k)$ be the representation of $\mathcal{M}$, where the states are relabelled based on the symbols in Σ_k. Suppose $\mathcal{G} \subseteq \Sigma$ is the set of target symbols, and gg is their representative symbol in Σ_k. Also let $\mathcal{M}^{\#} : (\tilde{S}, \Sigma_k, \tilde{\pi}, \tilde{\mathbf{P}}, \tilde{L})$ be the learned model from the samples of $\mathcal{M}_k$, using any PAC learning algorithm. Then, under the assumptions of the convergence of the learning algorithm,*

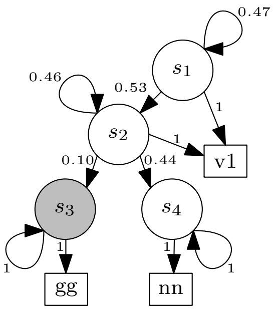
$$\Pr(\tilde{s} \models \Diamond^{\leq h}(gg)) = Pr(s \models \Diamond^{\leq h}\mathcal{G}), \forall \tilde{s} \in \tilde{S}, \forall s \in S. \tag{5.1}$$

Proof. First observe that the prediction in our setting is a *bounded* LTL property, therefore (5.1) is valid for the initial state (*Case i* in Theorem 1 in [89]). Given that $\mathcal{M}_k$ is a deterministic DTMC, the states of the trained model $\mathcal{M}^{\#}$, almost surely bisimulates the states of $\mathcal{M}_k$ in the limit (see Theorem 1 in [80]). As a result, under the assumptions of the convergence of the learning algorithm, (5.1) follows. □

Notice that the state-merging algorithms described in Section 4.1.1 train a deterministic DTMC regardless of the generating model. Obtaining the actual nondeterministic model



(a) The learned deterministic DTMC from the abstract traces.



(b) The learned HMM from the abstract traces with 4 hidden states.

Figure 5.2: Two abstract prediction models obtained from using 2-prediction support, which gives R_2 and the alphabet $\{gg, v1, nn\}$.

means retrieving the entire model, which defies the purpose of abstraction. Since the number of states in an HMM is given as a parameter, we can achieve a trade-off between the size of the model and the prediction accuracy by allowing to adjust the number of hidden states to obtain a small prediction model with a reasonable accuracy.

Example. Figure 5.2a demonstrates the learned deterministic DTMC model from the abstract traces generated from the nondeterministic model in Figure 5.1. □

5.4.2 Prediction via Nondeterministic Abstract DTMC

If the generating model becomes a nondeterministic abstract DTMC due to abstraction, we can use HMM to resolve the nondeterminism. In the abstraction setting, observations are the symbols of the abstract alphabet, Σ_k, and hidden states are the states of the generating model, i.e., the nondeterministic DTMC. The random hidden state variable creates an extra degree of freedom which allows to distinguish states that emit the same symbol but have different joint probability distributions.

A similar result to Theorem 4 can be achieved for nondeterministic models (an extension of *Case ii* of Theorem 1 in [89] to all the states). However, that relies on the convergence of the learning algorithm, which as we discuss in Section 4.1.2 is a hard problem for HMMs, and approximate methods are applied instead. In addition, even if the convergence of the trained nondeterministic model to the generating nondeterministic abstract DTMC is guaranteed, the trained model is as large as the generating model, and applying abstraction becomes useless. As a result, we may choose the number of hidden states in an HMM based on the prediction accuracy measures defined in Section 4.6, i.e., compute the prediction error of HMMs with different sizes on a set of testing samples and select the one with minimum error.

Example. Figure 5.2b displays the trained HMM over the abstract traces obtained by R_2 from Table 5.1. The HMM has 4 hidden states, similar to the size of the trained DTMC in Figure 5.2a. Each hidden state corresponds to the set of states in the DTMC in Figure 5.1 with the same labels. There are two hidden states associated with the label $v1$ to distinguish between the states s_3 and s_6 in Figure 5.1 that reach the target state, labelled gg, in 2 steps with different joint probabilities. $\square$

5.5 Limitations

Our abstraction technique relies on the assumption that there is a symmetry relation between the observable symbols. The symmetry relation is defined in terms of the probability of reaching some target symbols. As we show in the next section, in some applications this symmetry exists which enables us to reduce the size of the prediction model. However, in some applications it is difficult to extract a symmetry relation between the emitted events by the system. In particular, in systems that emanate the events with complex dynamic relation it is more difficult to find a symmetry relation. For instance, in a vehicle the speed and the steering angle may have some relation that are not observable from the values directly. One solution is to apply a transformation on the values of the observable variables such that the existing symmetries unravel. Finding and applying such transformations are beyond the scope of this thesis, but are interesting directions to follow in the future.

5.6 Experiments

We use a model of the Herman's self-stabilising algorithm [55] to demonstrate the abstraction component in $\mathcal{P}revent$. The algorithm provides a protocol for N identical processes (N is odd) in a token ring network, with unidirectional synchronous communication. Starting from an arbitrary configuration, the network will eventually converge to a defined *stable* state within a finite number of steps. The token is infinitely circulated in the ring amongst the processes in a fair manner. The stable state is defined such that there is exactly one

Table 5.2: The prediction results of different models on 100 random samples.

		Learned DTMC *conc.*			**Learned DTMC** *abst.*			**Learned HMM**		
N	**Orig. Alph.**	**Size**	**Training Time**(s)	**Abst. Alph.**	**Size**	**Training Time**(s)	**MSPE** 10^{-02}	**Size**	**Training Time**(s)	**MSPE** 10^{-02}
5	32	18	1047.12	5	5	16.12	27.57	4(7)	9.95	0.70
7	128	1319	19605.47	3	3	866.27	32.28	3(5)	61.13	1.39
9	512	7914	135004.38	2	2	275.54	79.36	3(4)	47.16	1.79
11	2048	O/M	–	2	2	2696.73	87.52	2(3)	2496.20	1.35

process that has the token. The process i has a local Boolean variable x_i. If $x_i = x_{i-1}$ there is a token with process i, in which case process i randomly chooses to set x_i to the next value or leave it unchanged, i.e., equal to x_{i-1}. See Appendix A.3 for a Prism model of 5 processes.

The observation space for a ring with N processes has 2^N symbols, each representing a different combination of the values of the local Boolean variable in each process. The observation space maps one-to-one to the state space of the corresponding DTMC. The monitoring is performed for the property $\varphi_{stable} = \lozenge^{\leq h}$ *"stable"* which translates into the target symbols in which only one process has the same label as its left neighbour, i.e., there exists only one i such that $x_i = x_{i-1}$. The monitoring procedure throughout the experiments is performed offline; however, in principle the online monitoring procedure is the same, except that the execution path keeps expanding as the system continues running.

We collected 1000 *iid* samples from the DTMC using the Prism simulation tool [90]. The length of the samples is uniformly distributed and constrained by an upper bound. We first run Algorithm 12 to extract the predictive symbols for the target symbols specified by φ_{stable}. We replaced the symbols of the sampled traces based on the found partitioning, and performed the training algorithms to learn a deterministic DTMC as well as an HMM.

Table 5.2 summarizes the results of three different prediction models compared to the prediction results obtained from the original model for $N \in \{5, 7, 9, 11\}$, $k = 0$ (a bigram model), and prediction horizon equal to one step. The evaluation of the prediction is based on the results from the true model (see Section 4.6.1). The size of the original model is identical to the size of the alphabet, as there is exactly one state corresponding to the valuation of the local variables in each process. We used Aalergia [1] to train deterministic DTMCs from both the concrete and abstract traces, and Matlab HMM toolbox to train the HMMs. The training was performed on an Ubuntu 17.10 machine with 24GB RAM.

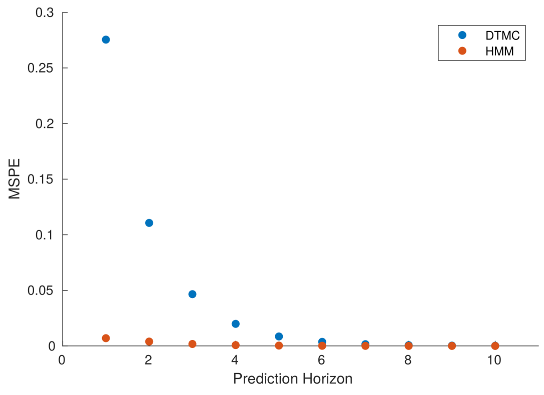
Training a DTMC from concrete traces was aborted for N=11 due to lack of memory, as the size of the FPTAs grows exponentially with the size of the alphabet. The trained DTMCs from abstract traces have significantly smaller size in direct relation to the size of the inferred abstract alphabet, and consequently a shorter training time. This result is consistent with the fact that the actual model is highly symmetrical with respect to the stable states, i.e., the probability of reaching the stable states from the states within an equivalence class in one step is equal (see Appendix A.3).

The sizes of the trained HMMs, shown in parentheses, are comparable to the size of their equivalent abstract DTMCs. As we can see HMMs with comparable size are substantially more accurate in making predictions than the abstract DTMCs. The state estimation also benefits from the small size of the HMM with virtually no computational overhead. Since the prediction horizon is formulated as an upper bound, the probability of an accepting extension increases as the prediction horizon increases, which in turn results in a lower MSPE. However, as depicted in Figure 5.3a a trained HMM has smaller error for short-range predictions. For example, for $h = 5$ the MSPE of the HMM is 0.03×10^{-2} as opposed to 0.85×10^{-2} for the abstract DTMC.

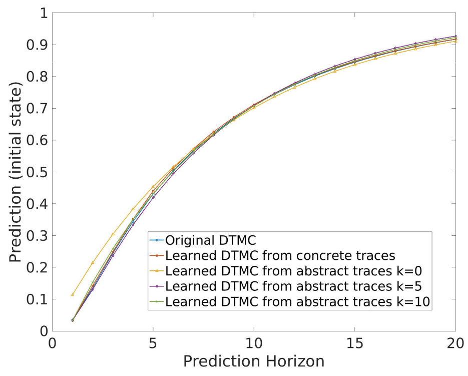
Figure 5.3b demonstrates the prediction results of the DTMC trained from concrete traces, and the traces abstracted by k-prediction support, using $k = 0, 5, 10$. The prediction results are from the initial state, and as we can see, the models learned from abstract traces almost perfectly follow the values of the actual model. The best result belongs to $k = 10$, which echos the maximum expected time to reach a stable state, i.e., that the path to the target labels from the initial state is of length at most about 9.

5.7 Related Work

Our approach is novel in terms of applying learning and abstraction at the trace level to predictive RV, and using HMM to handle nondeterminism. In [74] using an abstraction of a program is suggested to develop a predictive RV framework. The abstraction model is similar to the notion of prediction model in $\mathcal{P}revent$, that is neither extremely large to have the state-explosion problem similar to model checking, nor is non-existent similar to the standard RV, which provides no information about the future behaviour of the program. The approach in [74], however, assumes that the abstraction is given, and the system is white-box; whereas, in our framework the abstraction is inferred from the sample executions of a black-box system. The idea of using abstraction for a black-box system during learning is originally discussed in the AALERGIA paper [80]. However, no detail is provided on how to obtain a good abstraction.



(a) MSPE of the trained DTMC and HMM on abstract traces with $k = 0$, $N = 5$ (prediction horizon $[1, 10]$).



(b) The prediction results from the initial state of different prediction models compared to the original model for $N = 9$.

Figure 5.3: The experiment results for abstract models in the Herman algorithm.

Nouri *et al.* [89] use abstraction and learning to expedite statistical model checking [73]. Their approach is the probabilistic variant of *black box checking* [92] in which the inferred model, in the form of a DFA, is checked against some properties. In our case, we use abstraction to obtain a smaller prediction model for predictive RV. In [89] the atomic propositions in the property are used for abstracting the traces. We perform a statistical analysis on the traces to obtain partitions that leave the prediction probabilities intact. We also use HMM to handle the nondeterminism induced by abstraction.

Aichernig & Tappler [2] employ black-box checking in the context of reachability analysis of stochastic systems with controllable inputs. Hence, they use a Markov Decision Process (MDP), an extension of a Markov chain with nondeterministic choices, as a model that is trained from random samples. They use the inferred MDP to obtain an *adversary* with which they collect new samples and incrementally train new MDPs. LAR [127] is a combination of probabilistic model learning and counterexample guided abstraction refinement (CEGAR) [56]. These approaches are orthogonal to our technique and it is straightforward to extend $\mathcal{P}$*revent* to training other models such as MDP, and applying probabilistic CEGAR to obtain a model that guarantees checking affirmative properties.

jPredictor [27] is a runtime analysis tool for concurrent programs that creates a Lamport's *happens-before* causality abstract model [72] by dynamic slicing [26] and capturing the sliced causality with vector clocks [112]. The model can then be exhaustively explored and verified against *monitorable* safety properties. Similar to the abstraction proposed in this chapter, slicing traces in [27] is a way to abstract away the events that are not contained in the property, without hindering the soundness of the verification. Nonetheless, the implementation of jPredictor slightly deviates from the sound implementation of slicing, without raising any false alarms in practice [27], as sound slicing imposes a high computational cost for large traces.

Chapter 6

Learning Prediction Model for Rare Properties

This chapter addresses the challenge to predict a property when the occurrence of the property is *rare*, i.e., when the property is satisfied or violated with very low probability (e.g., *eventually "error"*). In short, we call such properties *rare properties*. Verifying the rare properties at runtime is common in real applications for two reasons. First, the violation or satisfaction of such properties are not typically caught by testing due to its incomplete nature. Second, although more thorough verification techniques, such as model checking, may be able to verify a rare property, the cost-benefit of fixing the system to satisfy a rare property may lead the engineers not to change the system, and instead have a monitoring mechanism to predict the improbable but possible violations at runtime.

Constructing the prediction model for rare properties, however, poses two main challenges. First, in order to construct an accurate prediction model for rare properties, it is necessary to observe the system for an unreasonably long time to collect at least one sample that has one occurrence of the rare properties. Second, the total number of training samples also needs to be large because we need to observe most of the prefixes that lead to the satisfaction or violation of the rare properties to be able to build an accurate prediction model. A large training set of execution samples typically guarantees having observed most of the prefixes leading to the rare properties, but leads to a slow learning process.

Our proposed solution adopts notions from rare event simulation techniques [107, 106] to accelerate the learning of an accurate model of rare properties. More specifically, we use Importance Sampling (IS, see, e.g., [107, Chap. 5]), which is a standard technique by which a measure of an *inconvenient* probabilistic distribution is estimated by sampling from a *convenient* distribution over the same sample space. Typically, as in the present context, the inconvenience arises because the measure of interest is a rare event and the convenient distribution makes the rare event more likely. Our approach assumes that the monitored systems have accessible parameters that allow its behaviour to be modified, and that there is a well-defined *likelihood ratio*. The likelihood ratio defines the relationship between the original distribution and the modified distribution for rare properties. The sample set, which is drawn from the modified distribution and is compensated on the fly when learning, gives an accurate estimate of the rare property with fewer samples.

Finding an optimized set of parameters of the system to create the modified distribution with respect to the rare properties is not discussed in this chapter, but for example, starting with randomly chosen parameters and how the parameters affect the likelihood ratio, without having access to a complete and explicit model of the system, it is possible to find the optimal parameters using an iterative algorithm based on cross-entropy minimization [58, 62].

In this chapter, we first introduce IS and its related concepts (Section 6.1). We use the accessible parameters of a stochastic system to alter its behaviour to create IS distributions with known relationship to the original system distribution. We assemble a training set by combining samples from arbitrarily many distributions so that the training set covers a range of prefixes leading to the rare event. We construct a *weighted* FPTA from the training set by replacing integer frequency counts with fractional weights based on the likelihood ratios of the true distribution and the IS distributions (Section 6.2). We show how to use the weighted FPTA from a single distribution to build a DTMC as a prediction model in $\mathcal{P}revent$, that produces the prediction results according to the original distribution (Section 6.3). Finally, we demonstrate the usefulness of our approach on a file transmission protocol [51], predicting the rare failure of a sender to report a successful transmission (Section 6.5), and discuss the related work (Section 6.6).

6.1 Importance Sampling

Suppose a stochastic system with distribution $\mathcal{D}\colon \Sigma^* \to [0,1]$, from which we draw random samples $u \in \Sigma^*$ according to $\mathcal{D}$, denoted $u \sim \mathcal{D}$. The indicator function $\mathbb{1}_\varphi\colon \Sigma^* \to \{0,1\}$ returns 1 iff u satisfies some property φ, then the probability of satisfying φ under $\mathcal{D}$, denoted $Pr_{\mathcal{D}}(\varphi)$, can be estimated using the standard Monte Carlo (MC) estimator,

$$Pr_{\mathcal{D}}(\varphi) \approx \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{\varphi}(u_i), \text{ with } u_i \sim \mathcal{D}, \tag{6.1}$$

where $N \in \mathbb{N}$ *iid* samples, $u_1, \ldots, u_N$, are drawn at random according to $\mathcal{D}$ and evaluated with respect to φ. The proportion of samples that satisfy φ is an unbiased estimate of $Pr_{\mathcal{D}}(\varphi)$.

If satisfying φ is a rare event under $\mathcal{D}$, i.e., $Pr_{\mathcal{D}}(\varphi) \ll 1$, the number of samples must be very large to estimate $Pr_{\mathcal{D}}(\varphi)$ with low *relative* error [107, Chap. 1]. The intuition of this is given by the fact that N must be greater than or equal to $1/Pr_{\mathcal{D}}(\varphi)$ to expect to see at least one sample that satisfies or violates φ.

Given another distribution, $\mathcal{D}'\colon \Sigma^* \to [0,1]$, such that $\mathbb{1}_{\varphi}\mathcal{D}$ is *absolutely continuous* with respect to $\mathcal{D}'$, $Pr_{\mathcal{D}}(\varphi)$ can be estimated using the importance sampling estimator,

$$Pr_{\mathcal{D}}(\varphi) \approx \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{\varphi}(u_i)\frac{\mathcal{D}(u_i)}{\mathcal{D}'(u_i)}, \text{ with } u_i \sim \mathcal{D}'. \tag{6.2}$$

N samples are drawn according to $\mathcal{D}'$ and the proportion that satisfy φ is compensated by the *likelihood ratio*, $\mathcal{D}/\mathcal{D}'$. Absolute continuity of $\mathbb{1}_{\varphi}\mathcal{D}$ with respect to $\mathcal{D}'$ requires that for all $u \in \Sigma^*$, $\mathcal{D}'(u) = 0 \implies \mathbb{1}_{\varphi}(u)\mathcal{D}(u) = 0$. This guarantees that (6.2) is always well defined. If φ is not a rare event under $\mathcal{D}'$, (6.2) will typically converge faster than (6.1). Under these circumstances, the IS estimator (6.2) is said to have lower variance than the standard MC estimator (6.1). Equation (6.2) is the basis of Statistical Model Checking (SMC) tools that use IS [59, 20].

We call $\mathcal{D}$ the original distribution and $\mathcal{D}'$ the IS distribution, using the symbols as synonyms for the explicit terms in the remainder of this chapter. Later, we also use the terms MC and IS to distinguish simulations or models generated from the original distribution and from an importance sampling distribution, respectively.

In the present context, we assume that $\mathcal{D}$ and $\mathcal{D}'$ are members of a family of distributions generated by two DTMCs, that are identified by different sets of transition probabilities over a finite set of states. Notice that the model can be described in a succinct way without having to explicitly build the entire state space. Hence by knowing some parameters of the model description, we can change the distribution generated by the underlying DTMC. More specifically, we use the Prism *guarded commands* format to describe a model [70]. In the guarded command description of the model we provide a set of commands that are guarded via predicates over the variables of the model. If the guard is satisfied the command is executed, where the values of some variables are updated based on a probability distribution (see the models in Appendix A). Knowing the update probabilities, we can infer the transition probabilities without building the state space of the model. Consequently, without access to the explicit representation of the model, we are able to calculate an individual transition as well as the likelihood ratio of a trace defined as the ratio of the transition's probability under $\mathcal{D}$ divided by its probability under $\mathcal{D}'$. Absolute continuity implies that every zero probability transition in $\mathcal{D}'$ corresponds to a zero probability transition in $\mathcal{D}$ or a transition that does not occur in a trace that satisfies φ.

6.2 Training on Rare-Event Samples

In this section, we present our modification of the Alergia algorithm (Section 4.1.1) in combination with importance sampling such that, without increasing the sample complexity [108], i.e., the size of the training set, a DTMC is trained to predict the satisfaction or violation of a rare property. Although we use the Alergia algorithm [34] to explain our approach; in principle, it is applicable to any learning algorithm that uses an FPTA in its learning process (e.g., [82]).

We modify building the FPTA (the function BuildFPTA in Algorithm 1), to adjust the probabilities of the final PTA with respect to the sample distribution generated by the importance sampling.

To achieve this goal, first we use the notion of Likelihood Ratio (LR) that is obtained by the importance sampling for each sample. The LR is effectively the inverse of the bias introduced in the transition probabilities of the model by the importance sampling to increase the probability of the rare events (see Section 6.1).

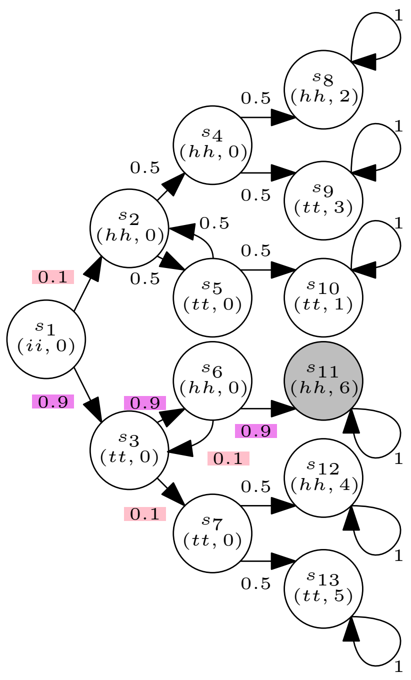


Figure 6.1: The DTMC of Figure 3.2 modified to produce more events $(hh, 6)$.

Let $\mathcal{D}(u)$ be the probability measure of the path u under the original distribution $\mathcal{D}$, and $\mathcal{D}'(u)$ be the probability measure of the same path under the importance sampling distribution, $\mathcal{D}'$. Then the likelihood ratio of u is defined as follows:

trace	*freq.*	*LR*	*trace*	*freq.*	*LR*
$(ii,0)$	200	1	$(ii,0)(hh,0)(hh,0)(hh,2)$	4	5
$(ii,0)(hh,0)$	40	5	$(ii,0)(hh,0)(hh,0)(tt,3)$	4	5
$(ii,0)(tt,0)$	360	5/9	$(ii,0)(hh,0)(tt,0)(hh,0)$	4	5
$(ii,0)(hh,0)(hh,0)$	12	5	$(ii,0)(hh,0)(tt,0)(tt,1)$	4	5
$(ii,0)(hh,0)(tt,0)$	12	5	$(ii,0)(tt,0)(hh,0)(tt,0)$	28	125/81
$(ii,0)(tt,0)(hh,0)$	44	25/81	$(ii,0)(tt,0)(hh,0)(hh,6)$	252	125/729
$(ii,0)(tt,0)(tt,0)$	12	25/9	$(ii,0)(tt,0)(tt,0)(hh,4)$	12	25/9
			$(ii,0)(tt,0)(tt,0)(tt,5)$	12	25/9

Table 6.1: 1000 IS traces generated from the DTMC in Figure 6.1.

$$LR(u) = \frac{\mathcal{D}(u)}{\mathcal{D}'(u)} \tag{6.3}$$

Example. Figure 6.1 displays the modified version of the model in Figure 3.2 obtained by an importance sampling distribution that increases the probability of φ_{F} to ≈ 0.8 (the changed transition probabilities are shaded). The samples randomly generated from this model, with the likelihood ratio for each sample, are shown in Table 6.1. The LR for each training sample is computed using (6.3). For example, $LR((ii,0)(tt,0)(hh,0))$ is $\frac{0.5\times 0.5}{0.9\times 0.9} = 25/81$. $\square$

6.2.1 Weighted Prefix Tree Acceptor

We define the notion of a Weighted Prefix Tree Acceptor (WPTA) to obtain an automaton as the representation of the samples with different likelihood ratios. A WPTA is similar to an FPTA except that instead of the integer frequency counts, fractional numbers are used as weights.

Definition 19 (WPTA). *A Weighted Prefix Tree Acceptor (WPTA) is a tuple $\mathcal{A} : (Q, \Sigma, W_I, \delta, W_F, W_T)$, where Q is a non-empty finite set of states, Σ is a non-empty finite alphabet, $Fr_I : Q \to \mathbb{R}$ is the initial weighted frequency of the state(s), $\delta : Q \times \Sigma \to Q$ is the transition function, $Fr_F : Q \to \mathbb{R}$ is the final weighted frequency of the state(s), and $Fr_T : Q \times \Sigma \times Q \to \mathbb{R}$ is the transition weighted frequency function between two states.*

Let original distribution $\mathcal{D}\colon \Sigma^* \to [0,1]$ be absolutely continuous with respect to importance sampling distributions $\mathcal{D}'_1, \ldots, \mathcal{D}'_M$, with $\mathcal{D}'_i\colon \Sigma^* \to [0,1]$ for $i \in \{1, \ldots, M\}$. Then let $N_1, \dots, N_M$ be the number of samples drawn at random using $\mathcal{D}'_1, \dots, \mathcal{D}'_M$, respectively.

When constructing our WPTA, we assign a weight to each frequency count along the trace. If p_i denotes the probability measure of some property under $\mathcal{D}$ and $\overline{W}_i$ is the expected weight applied to traces drawn from $\mathcal{D}'_i$, then we require that in the limit of large $N_1, \dots, N_M$,

$$p_i = \frac{N_i \overline{W}_i}{\sum_{j=1}^{M} N_j \overline{W}_j} \tag{6.4}$$

That is, we expect the normalized total frequency to equal the total measure of probability, which follows from the probability axioms and the law of large numbers. The distribution $\mathcal{D}'_i$ is obtained such that the likelihood of the rare event is increased (e.g., the probabilities of the transitions that lead to an error state are increased). Given that the properties of interest in the present context are rare, in practice the values of p_i typically are estimated using rare event SMC. In order to derive a formula to calculate the weight applied to an individual simulation trace, in what follows we do not consider the potential statistical errors arising from finite sampling.

Re-arranging (6.4) for $\overline{W}_i$ gives:

$$\overline{W}_i = \frac{p_i \sum_{j \in \{1,\dots,M\} \setminus \{i\}} N_j \overline{W}_j}{N_i(1 - p_i)},$$

however there is no unique solution because the numerator on the right hand side contains all the other unknown weights. To sufficiently constrain (6.4), for convenience, we set $\sum_{j=1}^{M} N_j \overline{W}_j$ to be equal to $\sum_{j=1}^{M} N_j$. Hence, the expected weight for samples from $\mathcal{D}'_i$ is given by

$$\overline{W}_i = \frac{p_i \sum_{j=1}^{M} N_j}{N_i}$$

The actual weight used for simulation trace $u \sim \mathcal{D}'_i$ is dependent on its likelihood ratio and is thus given by

$$W_i = \frac{\mathcal{D}(u)}{\mathcal{D}_i(u)} \frac{\sum_{j=1}^{M} N_j}{N_i} \tag{6.5}$$

The relative values of the set of weights calculated by (6.5) are unique up to a positive scaling factor. This scaling factor may be important when deciding which nodes are

```
1  BuildWPTA(Sample dataset $S, LR$)
   output: WPTA $\mathcal{A}$
2  begin
3      $\mathcal{A} \leftarrow (Q, \Sigma, W_I, \delta, W_F, W_T)$
4      $Q \leftarrow \{q_u | u \in \textsc{Pref}(S)\} \cup \{q_\epsilon\}$
5      $W_I(q_\epsilon) \leftarrow \sum_{\forall u \in S}(u.freq) \times LR(u)$
6      forall $ua \in \textsc{Pref}(S)$ do
7          $\delta(q_u, a) \leftarrow q_{ua}$
8          $W_T(q_u, a, q_{ua}) \leftarrow ua.freq \times LR(ua)$
9      forall $u \in S$ do
10         $W_F(q_u) \leftarrow u.freq \times LR(u)$
11     return $\mathcal{A}$
```

Algorithm 13: Building WPTA from the samples obtained by importance sampling.

compatible for merging, since metrics such as the Hoeffding bound [23] or the Angluin bound [80], are sensitive to the absolute values. We adhere to the Hoeffding bound in our experiments.

6.3 WPTA Construction from a Single Distribution

In this section we describe the steps of the algorithm to build a WPTA. To simplify the algorithm description, in the remainder of this section we assume that the samples are drawn from a single distribution, therefore, the weights become essentially the likelihood ratio for each sample defined in (6.3). In the case of multiple distributions, it is straightforward to adjust the weights according to the number of samples generated from each distribution and use (6.5) instead.

Algorithm 13 demonstrates the algorithm to build the WPTA from the IS samples, where the frequencies are multiplied by the likelihood ratio of each sample. The input is a dataset which contains samples, denoted by u, their frequency, denoted by $freq$, and their likelihood ratio, denoted by LR, provided by importance sampling. Line 4 initializes the states of the WPTA with all the prefixes that exist in the dataset, with an additional state q_ϵ that represents the empty prefix, and used as the initial state whose initial weighted frequency is equal to the sum of the weights of the entire dataset (line 5). The remaining states have the initial weighted frequency equal to zero. For each $ua \in \textsc{Pref}(S)$, where

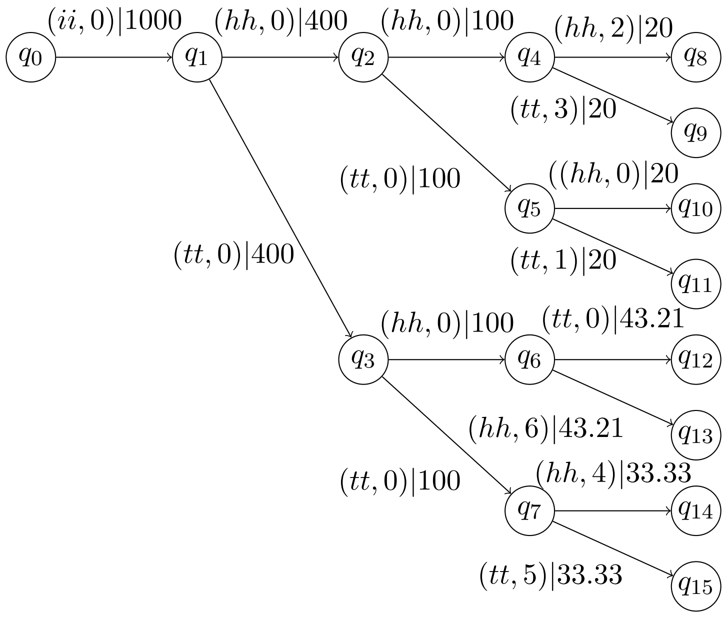


Figure 6.2: WPTA constructed from the samples of Table 6.1.

$a \in \Sigma$, the for loop in lines **6-8** sets the transition function between the states q_u and q_{ua}, and its transition weighted frequency by multiplying the frequency of ua in the dataset to its LR (lines **7-8**). The final weighted frequency is obtained as the product of the frequency of each sample and its likelihood ratio (line **10**).

Example. Figure 6.2 illustrates the WPTA obtained from the IS samples shown in Table 6.1. Table 6.2 shows the final weighted frequencies of each state of the WPTA, which are used to test the compatibility of two states when we are trying to merge them to obtain the final PFA (see Algorithm 1). □

Correctness & Complexity. Note that the weights in a WPTA are $LR(u) \times \tilde{\mathcal{D}}'(u)$, where $\tilde{\mathcal{D}}'(u)$ is the empirical probability of u according to the distribution $\mathcal{D}'$. Using (6.3), it is trivial to observe that the weights are essentially giving the same empirical probability distribution that the FPTA provides, i.e., $\mathcal{D}$. Therefore, the correctness of our approach for predicting the rare properties is implied by the convergence analysis of the compatibility

Table 6.2: The prefixes and their final weighted frequencies associated with each state of the WPTA in Figure 6.2.

state	*prefix*	*fin. wei. freq.*	*state*	*prefix*	*fin. wei. freq.*
q_0	ϵ	0	q_1	$(ii, 0)$	200
q_2	$(ii, 0)(hh, 0)$	200	q_3	$(ii, 0)(tt, 0)$	200
q_4	$(ii, 0)(hh, 0)(hh, 0)$	60	q_5	$(ii, 0)(hh, 0)(tt, 0)$	60
q_6	$(ii, 0)(tt, 0)(hh, 0)$	13.58	q_7	$(ii, 0)(tt, 0)(tt, 0)$	33.33
q_8	$(ii, 0)(hh, 0)(hh, 0)(hh, 2)$	20	q_9	$(ii, 0)(hh, 0)(hh, 0)(tt, 3)$	20
q_{10}	$(ii, 0)(hh, 0)(tt, 0)(hh, 0)$	20	q_{11}	$(ii, 0)(hh, 0)(tt, 0)(tt, 1)$	20
q_{12}	$(ii, 0)(tt, 0)(hh, 0)(tt, 0)$	43.21	q_{13}	$(ii, 0)(tt, 0)(hh, 0)(hh, 6)$	43.21
q_{14}	$(ii, 0)(tt, 0)(tt, 0)(hh, 4)$	33.33	q_{15}	$(ii, 0)(tt, 0)(tt, 0)(tt, 5)$	33.33

test on an FPTA in the large sample limit (see Theorem 3). The time complexity of building a WPTA has the additional multiplication operations to compute the weights which is in the size of the WPTA. The order of merging and training remains cubic with the size of the training set [23].

6.4 Limitations

As described in Section 6.1, the learning technique described in this chapter is based on the availability of the likelihood ratio, $\frac{\mathcal{D}}{\mathcal{D}'}$ in (6.2), without needing to build the underlying generators for the distributions $\mathcal{D}$ and $\mathcal{D}'$. In some applications finding the likelihood ratio without constructing the model is possible. For example, in the applications that the model of the system is based on the samples generated from simulations, such as in SMC, the likelihood ratio can be calculated while a sample path is being simulated. In some real applications, however, finding a well-defined likelihood ratio is much more challenging, particularly if the samples are generated from the real execution of the entire system.

6.5 Experiments

We performed experiments to demonstrate the predictive performance of our IS approach applied to the bounded retransmission protocol (BRP) model of [32]. The BRP model describes a sender, a receiver, and a channel between them, which the sender uses to send a file in chunks with a maximum number of retransmissions for each chunk defined as a parameter. We use 64 chunks with at most 2 retransmissions. See Appendix A.4 for the PRISM model description.

The rare property that we consider is *the sender does not report a successful transmission* [32], which we express in LTL as $\Diamond$ *"error"*. The probability of this property, expressed as $Pr(\Diamond$ *"error"*$)$, is approximately 1.7×10^{-4}. We also consider the time-bounded property that *the sender does not report a successful transmission within k steps*, expressed as $\Diamond^{\leq k}$ *"error"*.

The learned models of the BRP using standard MC and IS were constructed from simulation traces generated by PLASMA [59], according to the algorithms defined in Sections 4.1.1 and 6.3, respectively. The resulting models were then checked with respect to the above properties using PRISM [71]. The results are illustrated in Figures 6.3 and 6.4.

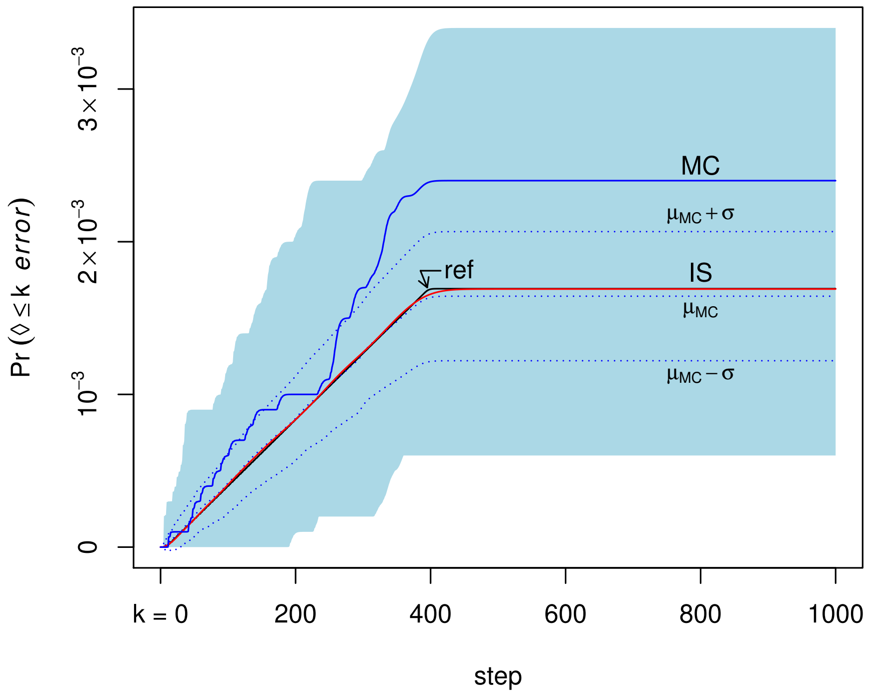
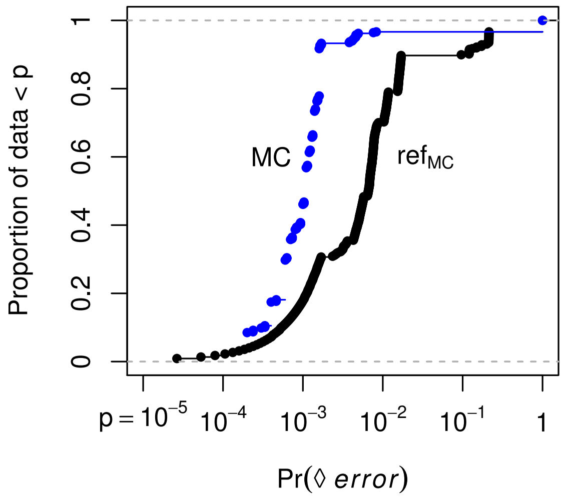
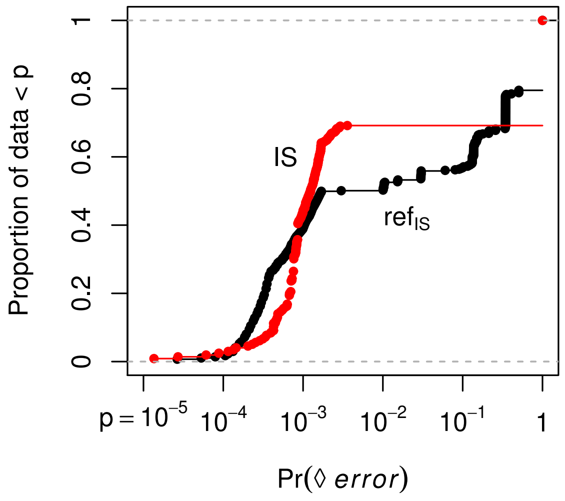


Figure 6.3: Predictive performance of a typical 10^3-trace IS model of BRP (labelled IS) vs. that of 1000 10^4-trace MC models (shaded area).

Using standard MC simulation of the original distribution of the BRP model, we constructed 1000 learned models, each using a training set of 10^4 independently sampled



(a) 10^4-trace MC model.



(b) 10^3-trace IS model.

Figure 6.4: Prediction distributions of learned models of BRP.

traces. These training sets included between 6 and 34 traces that satisfy the property. Since the performances of different models generated by IS are visually indistinguishable, for comparison we trained just a single IS model, using a training set of only 10^3 sampled traces. The IS training set comprised 500 traces that satisfy the property, selected from simulations of an IS distribution over the original BRP model, and 500 traces that do not satisfy the property, selected from simulations of the original distribution of the BRP model. By combining these traces in accordance with (6.5), the resulting IS distribution adequately covers the parts of the state space related to both satisfying and not satisfying the property.

The results for the MC models are represented as a shaded area that encloses the minimum and maximum probabilities recorded for each value of k. To give a better intuition of the distribution of these results, we plot lines representing the empirical mean (μ_{MC}) and empirical mean ± 1 standard deviation ($\mu_{\mathsf{MC}} \pm \sigma$). We also plot the performance of a typical MC model (labelled MC), where each apparent step in the curve corresponds to one of the 24 traces that satisfy the property in its training data.

The results for the IS model are almost coincident with the reference curve (ref), calculated by evaluating the property with respect to the original model of the system. The small difference, occurring at around $k = 400$ steps, arises due to the learned models using an abstraction based on only two variables in the original system.

The results presented in Figure 6.3 are with respect to a property in a single initial state. To assess the performance of our approach in constructing a predictive monitor, we consider the predictive accuracy of *all* the states in the learned models using MC and IS, with respect to probability $Pr(\Diamond \text{“}error\text{”})$. As in the previous experiments, we constructed an MC model using 10^4 traces and an IS model using 10^3 traces. To eliminate an obvious source of discrepancy, we generated multiple sets of 10^4 MC traces and selected a set with exactly 17 traces that satisfy the property. This ensures that the MC model is based on approximately the same probability as the IS model.

Each training set contains a different randomly selected set of concrete states in the original model. Hence, even though we use the same merging compatibility parameter for both sets ($\alpha = 10$), the two learned models are different. Also, since we consider the labels of the states, each learned model state maps to a different set of concrete states in the original model. Hence, we use the evaluation measure described in Section 4.6, and for every state in each learned model, we calculate the probability of the property in the learned model and, for comparison, calculate the probability of the property with respect to its associated set of concrete states in the original model.

The results of our calculations are two pairs of sets of probabilities, which we visualize as cumulative distributions in Figures 6.4a and 6.4b. In this form, the maximum vertical distance between the curves in each pair is a measure of similarity of the two distributions (the Kolmogorov-Smirnov (K-S) statistic [76]). The K-S statistic for the MC model is 0.63, while that of the IS model is 0.19 As expected, the IS model outperforms the MC model, despite using an order of magnitude fewer training samples for IS and hand-selecting a good MC training set.

6.6 Related Work

In the context of SMC, importance sampling [58, 62] and importance *splitting* [60, 61, 21] have already been used to mitigate the joint problems of intractable state space and rare events. Since SMC is essentially runtime verification of an accurate existing model that is deliberately not constructed in its entirety (see Section 3.6.2), rare event SMC is not inherently predictive, so not immediately applicable in the current context. In [21] a biased automaton is used to increase the probability of finding a rare error in randomized testing.

The focus of this chapter is on training a prediction model using rare-event samples. Importance sampling is used in the related context of reinforcement learning [115] to estimate the optimal policy in an unknown partially observable MDP [94, 98, 93]. Our technique uses the state-merging method as a form of supervised learning to build a partially observable DTMC for a rare event. The authors of [128] introduce a genetic algorithm to predict rare events in event sequences. Their approach is based on identifying temporal and sequential patterns through the mutation and crossover operators. Our approach instead uses importance sampling to identify the rare-event samples, and diversifies them by adding other (importance) sampling distributions. Our purpose is to construct the underlying model that captures the probabilities of a rare event.

Chapter 7

Conclusion

7.1 Summary

This dissertation introduces predictive runtime verification for stochastic systems. The thesis statement is that predictive runtime verification of a stochastic system is *feasible*, *effective*, and *useful*.

We showed the *feasibility* by introducing $\mathcal{P}$*revent*, a predictive runtime verification framework for monitorable properties. The core part of $\mathcal{P}$*revent* involves constructing a predictive monitor. The predictive monitor is essentially a lookup table that consists of the reachability analysis results for each state of the prediction model. The prediction model is trained from sample execution paths, and is in the form of a DTMC or an HMM. The monitor produces a quantitative output that represents the probability that from the current state, the system satisfies or violates a property within a finite horizon. The state of the system is either obtained directly by the FORWARD-BACKWARD algorithm, or estimated using the VITERBI algorithm. Two measures are introduced for empirical evaluation of the prediction. One measure is defined based on the prediction result from the true model of the system, if it is available. If the true model is not available, another measure is introduced based on the statistical comparison between the monitor prediction and that of a set of execution path samples with known prediction results. $\mathcal{P}$*revent* is evaluated on two case studies: the randomised dining philosophers problem, and the flight control of a hexacopter, where the monitor predicted the satisfaction or violation of the given properties beforehand. In both cases, the trained models are extracted from *bad* traces, thus, the monitor has a tendency to produce false positives. To reduce the number of false positives, we involve a mixture of trained models based on good and bad traces.

The *effectiveness* of the framework is demonstrated through reducing the size of the prediction model. We applied abstraction at the observation space to achieve this goal. We proposed inferring a projection relation from a random set of samples, that maps the concrete alphabet to an abstract alphabet of a smaller size. The abstract alphabet is then used to abstract traces that are used in building the prediction model. Our inference technique is based on finding a symmetrical relation between the symbols of the alphabet, using a k-gapped pair model, and lumping them into equivalence classes. We use the abstract traces to train deterministic DTMC as well as HMM to handle a potential nondeterminism induced by abstraction. We show that the prediction results remain intact with the model trained from the abstract traces, under the constraints of the learning algorithms. We evaluate our abstraction approach on the distributed randomized algorithm, i.e., a model of the Herman's algorithm, and demonstrated that in general the trained HMM from the abstract traces is more accurate than the trained DTMC. Our abstraction technique is most effective on the systems with large observation space, and where the model benefits from symmetry in the probability of satisfying or violating a given property. Our experiments confirmed this observation, where the model of the Herman's algorithm has a large observation space, exponential to the number of processes, and is highly symmetrical with regard to satisfying the given property.

Monitoring rare properties at runtime is essential in real-world applications. The statistical analysis of the samples requires a sufficient number of execution samples, both for learning and abstraction purposes. For a rare property, the number of samples might be so large that the introduced training algorithm to learn a prediction model is not as effective. Therefore, to make our framework *useful*, we use importance sampling to generate samples that contain more instances of the rare property without having to increase the number of samples. We use samples that are generated by importance sampling to train a prediction model. The importance sampling covers the state space responsible for the rare property, and the Monte Carlo simulation provides the coverage for the rest of the model. We then use likelihood ratios for each sample to construct the prediction model that represents the true distribution of the samples. We applied our approach on a file transmission protocol as a case study, demonstrating our technique's benefit in using fewer samples and achieving greater accuracy compared to the approach using only the Monte Carlo samples.

7.2 Future Directions

We believe predictive runtime verification is an original idea that brings with itself many interesting directions to the field of RV and analysis of software traces in general.

Theoretically, it is interesting to classify the monitorable properties based on how they can be predicted given a certain predictability characteristic of a stochastic system. For example, it is interesting to investigate the predictability of the safety-liveness properties based on the classification presented in [91], i.e., comparing the predictability of *always finitely refutable* properties to *sometimes finitely refutable* properties.

A practically useful, and theoretically interesting, direction is to use online learning algorithms in order to refine the prediction model on the fly. The cost of learning must be minimized so that the monitor overhead does not influence the execution of the system. Bayesian state-merging or -splitting techniques [119, 87] are effective learning algorithm candidates for an online learning framework.

The online learning framework can be combined with more statistically expressive models, such as Conditional Random Fields [121], in which the Markovian assumption is relaxed, hence, providing a more accurate prediction model for more dynamic systems. However, the more expressive models imply longer and more complicated training algorithms, which need to be adjusted if used at runtime.

In line with the abstraction techniques, an interesting future direction is to explore combining the importance sampling with abstraction, which is necessary when the state space of the system is intractable. We have already demonstrated in our experiments that this combination is effective, but obtaining a good abstraction of a complex system may be challenging. Drawing on experience with parameterized importance sampling [58, 62], we speculate that it may be possible to exploit cross entropy minimization or the coupling method [9] to find both good importance sampling distributions and good abstractions.

Practically, applying $\mathcal{P}revent$ to much larger case studies with more real-world scenarios is also a necessary direction that needs to be taken in the future. Particularly, we believe a focus needs to be directed towards *hybrid* systems, where the real-valued and discrete signals are mixed [78]. For example, for a hybrid system it is necessary to generalize $\mathcal{P}revent$ to include the robustness of an STL property evaluation in the predictions [42].

References

[1] Aalergia. http://mi.cs.aau.dk/code/aalergia/. Accessed: 2018-03-15.

[2] Bernhard K. Aichernig and Martin Tappler. Probabilistic black-box reachability checking. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, volume 10548 of *Lecture Notes in Computer Science*, pages 50–67. Springer, 2017.

[3] M. K. S. Al-Sharman, M. F. Abdel-Hafez, and M. A. R. I. Al-Omari. Attitude and flapping angles estimation for a small-scale flybarless helicopter using a kalman filter. *IEEE Sensors Journal*, 15(4):2114–2122, April 2015.

[4] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, 1996.

[5] César Andrés, Mercedes G. Merayo, and Manuel Núñez. Supporting the extraction of timed properties for passive testing by using probabilistic user models. In *Proceedings of the Ninth International Conference on Quality Software, QSIC 2009, Jeju, Korea, August 24-25, 2009*, pages 145–154, 2009.

[6] Dana Angluin. Identifying languages from stochastic examples. Technical Report YALEU/ DCS/RR-614, Yale University, Department of Computer Science, New Haven, CT, 1988.

[7] Reza Babaee, Arie Gurfinkel, and Sebastian Fischmeister. $Prevent$: A predictive run-time verification framework using statistical learning. In *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pages 205–220, 2018.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[9] Benoît Barbot, Serge Haddad, and Claudine Picaronny. Coupling and importance sampling for statistical model checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 331–346, 2012.

[10] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, pages 44–57, 2004.

[11] Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. Adaptive runtime verification. In *RV, Third International Conference*, pages 168–182, 2012.

[12] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, pages 260–272, 2006.

[13] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *RV, 7th International Workshop*, pages 126–138, 2007.

[14] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL semantics for runtime verification. *J. Log. Comput.*, 20(3):651–674, 2010.

[15] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[16] Matthew J. Beal, Zoubin Ghahramani, and Carl Edward Rasmussen. The infinite hidden markov model. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, pages 577–584, Cambridge, MA, USA, 2001. MIT Press.

[17] Mordechai Ben-Ari. The bug that destroyed a rocket. *SIGCSE Bulletin*, 33(2):58–59, 2001.

[18] P. Billingsley. *Probability and Measure*. Wiley Series in Probability and Statistics. Wiley, 2012.

[19] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, 43(1):29–60, 2013.

[20] Benoît Boyer, Kevin Corre, Axel Legay, and Sean Sedwards. Plasma-lab: A flexible, distributable statistical model checking library. In Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *Quantitative Evaluation of Systems*, pages 160–164. Springer, 2013.

[21] Lei Bu, Doron Peled, Dashuan Shen, and Yael Tzirulnikov. Chasing errors using biasing automata. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, pages 271–286, 2018.

[22] Radu Calinescu, Carlo Ghezzi, Marta Z. Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012.

[23] Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications*, pages 139–152, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[24] Ming Chai and Bernd-Holger Schlingloff. Online monitoring of distributed systems with a five-valued LTL. In *IEEE 44th International Symposium on Multiple-Valued Logic, ISMVL 2014, Bremen, Germany, May 19-21, 2014*, pages 226–231, 2014.

[25] Samuel W. K. Chan and James Franklin. A text-based decision support system for financial sequence prediction. *Decision Support Systems*, 52(1):189–198, 2011.

[26] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 569–588, 2007.

[27] Feng Chen, Traian-Florin Serbanuta, and Grigore Rosu. jpredictor: a predictive runtime analysis tool for java. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 221–230, 2008.

[28] Gerda Claeskens and Nils Lid Hjort. *Model Selection and Model Averaging.* Number 9780521852258 in Cambridge Books. Cambridge University Press, December 2008.

[29] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *25 Years of Model Checking - History, Achievements, Perspectives*, pages 196–215, 2008.

[30] Greta Cutulenco, Yogi Joshi, Apurva Narayan, and Sebastian Fischmeister. Mining timed regular expressions from system traces. In *Proceedings of the 5th International Workshop on Software Mining, SoftwareMining@ASE 2016, Singapore, Singapore, September 3, 2016*, pages 3–10, 2016.

[31] Przemyslaw Daca, Thomas A. Henzinger, Jan Kretínský, and Tatjana Petrov. Faster statistical model checking for unbounded temporal properties. *ACM Trans. Comput. Log.*, 18(2):12:1–12:25, 2017.

[32] Pedro D'Argenio, B. Jeannet, H. Jensen, and Kim Larsen. Reachability analysis of probabilistic systems by successive refinements. In L. de Alfaro and S. Gilmore, editors, *Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modelling and Verification (PAPM/PROBMIV'01)*, volume 2165 of *LNCS*, pages 39–56. Springer, 2001.

[33] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, and Sean Sedwards. Runtime verification of biological systems. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, pages 388–404, 2012.

[34] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, New York, NY, USA, 2010.

[35] Simon DeDeo. Conflict and computation on Wikipedia: A finite-state machine analysis of editor interactions. *Future Internet*, 8(3):31, 2016.

[36] Guozhu Dong and Jian Pei. *Sequence Data Mining*, volume 33 of *Advances in Database Systems.* Kluwer, 2007.

[37] Doron Drusinsky. Monitoring temporal rules combined with time series. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 114–117, 2003.

[38] Marie Duflot, Laurent Fribourg, and Claudine Picaronny. Randomized dining philosophers without fairness assumption. *Distributed Computing*, 17(1):65–76, 2004.

[39] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme J. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[40] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420, 1999.

[41] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 27–39, 2003.

[42] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theor. Comput. Sci.*, 410(42):4262–4291, 2009.

[43] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *RV, 9th International Workshop*, pages 40–59, 2009.

[44] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[45] S. Geisser. *Predictive Inference.* Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1993.

[46] Lars Grunske and Pengcheng Zhang. Monitoring probabilistic properties. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 183–192, 2009.

[47] Fredrik Gustafsson. *Statistical sensor fusion.* Lund: Studentlitteratur, 1 edition, 2010.

[48] James D. Hamilton. *Time Series Analysis.* Princeton University Press, Princeton, NJ, 1994.

[49] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[50] Klaus Havelund and Grigore Rosu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[51] Leen Helmink, M. P. A. Sellink, and Frits W. Vaandrager. Proof-checking a data link protocol. In *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, pages 127–165, 1993.

[52] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, pages 73–84, 2004.

[53] Benjamin Herd. *Statistical runtime verification of agent-based simulations.* PhD thesis, King's College London, UK, 2015.

[54] Benjamin Herd, Simon Miles, Peter McBurney, and Michael Luck. Quantitative analysis of multiagent systems through statistical model checking. In *Engineering Multi-Agent Systems - Third International Workshop, EMAS 2015, Istanbul, Turkey, May 5, 2015, Revised, Selected, and Invited Papers*, pages 109–130, 2015.

[55] Ted Herman. Probabilistic self-stabilization. *Inf. Process. Lett.*, 35(2):63–67, 1990.

[56] Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic CEGAR. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2008.

[57] Ssu-Hua Huang, Ru-Sheng Liu, Chien-Yu Chen, Ya-Ting Chao, and Shu-Yuan Chen. Prediction of outer membrane proteins by support vector machines using combinations of gapped amino acid pair compositions. In *Fifth IEEE International Symposium on Bioinformatic and Bioengineering (BIBE 2005), 19-21 October 2005, Minneapolis, MN, USA*, pages 113–120. IEEE Computer Society, 2005.

[58] Cyrille Jégourel, Axel Legay, and Sean Sedwards. Cross-entropy optimisation of importance sampling parameters for statistical model checking. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 327–342, 2012.

[59] Cyrille Jegourel, Axel Legay, and Sean Sedwards. A platform for high performance statistical model checking – PLASMA. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 498–503. Springer, 2012.

[60] Cyrille Jégourel, Axel Legay, and Sean Sedwards. Importance splitting for statistical model checking rare properties. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 576–591, 2013.

[61] Cyrille Jégourel, Axel Legay, and Sean Sedwards. An effective heuristic for adaptive importance splitting in statistical model checking. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, pages 143–159, 2014.

[62] Cyrille Jégourel, Axel Legay, and Sean Sedwards. Command-based importance sampling for statistical model checking. *Theor. Comput. Sci.*, 649:1–24, 2016.

[63] Kenan Kalajdzic, Ezio Bartocci, Scott A. Smolka, Scott D. Stoller, and Radu Grosu. Runtime verification with particle filtering. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 149–166, 2013.

[64] Kenan Kalajdzic, Cyrille Jégourel, Anna Lukina, Ezio Bartocci, Axel Legay, Scott A. Smolka, and Radu Grosu. Feedback control for statistical model checking of cyber-physical systems. In *ISoLA - 7th International Symposium*, pages 46–61, 2016.

[65] Yonit Kesten, Amir Pnueli, and Li-on Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, pages 1–16, 1998.

[66] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[67] Donald Knuth. The complexity of nonuniform random number generation. *Algorithm and Complexity, New Directions and Results*, pages 357–428, 1976.

[68] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 2.0: A tool for probabilistic model checking. In *1st International Conference on Quantitative Evaluation of Systems (QEST 2004), 27-30 September 2004, Enschede, The Netherlands*, pages 322–323, 2004.

[69] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Symmetry reduction for probabilistic model checking. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2006.

[70] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*, pages 220–270, 2007.

[71] Martha Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[72] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[73] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2010.

[74] Martin Leucker. Sliding between model checking and runtime verification. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, pages 82–87, 2012.

[75] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[76] Hubert W. Lilliefors. On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American Statistical Association*, 62(318):399–402, 1967.

[77] Oded Maler. Some thoughts on runtime verification. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016,*

Madrid, Spain, September 23-30, 2016, Proceedings, volume 10012 of *Lecture Notes in Computer Science*, pages 3–14. Springer, 2016.

[78] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, pages 152–166, 2004.

[79] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety.* Springer New York, 2012.

[80] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning probabilistic automata for model checking. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*, pages 111–120, 2011.

[81] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016.

[82] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, and Saddek Bensalem. Improved learning for stochastic timed models by state-merging algorithms. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 178–193, 2017.

[83] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[84] Stefan Mitsch and André Platzer. Modelplex: verified runtime validation of verified cyber-physical system models. *Formal Methods in System Design*, 49(1-2):33–74, 2016.

[85] J.O. Moody and P.J. Antsaklis. *Supervisory Control of Discrete Event Systems Using Petri Nets.* The International Series on Discrete Event Dynamic Systems. Springer US, 2012.

[86] Andreas Morgenstern, Manuel Gesell, and Klaus Schneider. An asymptotically correct finite path semantics for LTL. In *LPAIR - 18th International Conference*, pages 304–319, 2012.

[87] Kushal Mukherjee and Asok Ray. State splitting and merging in probabilistic finite state automata for signal representation and analysis. *Signal Processing*, 104:105–119, 2014.

[88] Glenford J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.

[89] Ayoub Nouri, Balaji Raman, Marius Bozga, Axel Legay, and Saddek Bensalem. Faster statistical model checking by means of abstraction and learning. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 340–355, 2014.

[90] Dave Parker, Gethin Norman, and Marta Kwiatkowska. Prism model checker. `http://www.prismmodelchecker.org/`. Accessed: 2017-08-14.

[91] Doron Peled and Klaus Havelund. Refining the safety–liveness classification of temporal properties according to monitorability. *LNCS (2018, submitted)*, 2018.

[92] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[93] Leonid Peshkin, Nicolas Meuleau, and Leslie Pack Kaelbling. Learning policies with external memory. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999), Bled, Slovenia, June 27 - 30, 1999*, pages 307–314, 1999.

[94] Leonid Peshkin and Christian R. Shelton. Learning from scarce experience. In *Machine Learning, Proceedings of the Nineteenth International Conference (ICML 2002), University of New South Wales, Sydney, Australia, July 8-12, 2002*, pages 498–505, 2002.

[95] Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: monitoring embedded systems. *ISSE*, 9(4):235–255, 2013.

[96] Srinivas Pinisetty, Thierry Jéron, Stavros Tripakis, Yliès Falcone, Hervé Marchand, and Viorel Preoteasa. Predictive runtime verification of timed properties. *Journal of Systems and Software*, 132:353–365, 2017.

[97] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[98] Doina Precup, Richard S. Sutton, and Sanjoy Dasgupta. Off-policy temporal difference learning with function approximation. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 417–424, 2001.

[99] Qnx neutrino rtos. http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos. Accessed: 2017-08-14.

[100] Michael O. Rabin and Daniel Lehmann. The advantages of free choice: A symmetric and fully distributed solution for the dining philosophers problem. In A. W. Roscoe, editor, *A Classical Mind*, pages 333–352. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[101] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[102] Radomised dining philosophers case study. http://www.prismmodelchecker.org/casestudies/phil.php. Accessed: 2018-01-24.

[103] James Blake Rawlings and David Q Mayne. *Model predictive control: Theory and design*. Nob Hill Pub. Madison, Wisconsin, 2009.

[104] Rizwan, Yassir. Towards high speed aerial tracking of agile targets. Master's thesis, University of Waterloo, 2012.

[105] Sam T. Roweis and Zoubin Ghahramani. A unifying review of linear gaussian models. *Neural Computation*, 11(2):305–345, 1999.

[106] G. Rubino and B. Tuffin. *Rare Event Simulation using Monte Carlo Methods*. Wiley, 2009.

[107] R.Y. Rubinstein and D.P. Kroese. *Simulation and the Monte Carlo Method*. Wiley Series in Probability and Statistics. Wiley, 2016.

[108] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[109] Usa Sammapun, Insup Lee, and Oleg Sokolsky. Rt-mac: Runtime monitoring and checking of quantitative and probabilistic properties. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005), 17-19 August 2005, Hong Kong, China*, pages 147–153, 2005.

[110] Usa Sammapun, Insup Lee, Oleg Sokolsky, and John Regehr. Statistical runtime checking of probabilistic properties. In *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, pages 164–175, 2007.

[111] Klaus Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, pages 39–54, 2001.

[112] Koushik Sen, Grigore Rosu, and Gul Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003, Helsinki, Finland, September 1-5, 2003*, pages 337–346, 2003.

[113] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pages 202–215, 2004.

[114] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, 2014.

[115] Christian R. Shelton. *Importance sampling for reinforcement learning with multiple objectives*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[116] A. Prasad Sistla and Abhigna R. Srinivas. Monitoring temporal properties of stochastic systems. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference*, pages 294–308, 2008.

[117] A. Prasad Sistla, Milos Zefran, and Yao Feng. Monitorability of stochastic dynamical systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 720–736, 2011.

[118] A. Prasad Sistla, Milos Zefran, Yao Feng, and Yue Ben. Timely monitoring of partially observable stochastic systems. In *HSCC, 17th International Conference (part of CPS Week)*, pages 61–70, 2014.

[119] Andreas Stolcke and Stephen M. Omohundro. Best-first model merging for hidden markov model induction. *CoRR*, abs/cmp-lg/9405017, 1994.

[120] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *RV - Second International Conference*, pages 193–207, 2011.

[121] Charles A. Sutton, Andrew McCallum, and Khashayar Rohanimanesh. Dynamic conditional random fields: Factorized probabilistic models for labeling and segmenting sequence data. *Journal of Machine Learning Research*, 8:693–723, 2007.

[122] Andrew S. Tanenbaum and Maarten van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.

[123] Sebastiaan Terwijn. On the learnability of hidden markov models. In *ICGI, 6th International Colloquium*, pages 261–268, 2002.

[124] Gregory Travis. How the boeing 737 max disaster looks to a software developer. *IEEE Spectrum*, Apri 2019.

[125] Ralph Vartabedian. How a 50-year-old design came back to haunt boeing with its troubled 737 max jet. *LATimes*, Mar 2019.

[126] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Information Theory*, 13(2):260–269, 1967.

[127] Jingyi Wang, Jun Sun, and Shengchao Qin. Verifying complex systems probabilistically through learning, abstraction and refinement. *CoRR*, abs/1610.06371, 2016.

[128] Gary M. Weiss and Haym Hirsh. Learning to predict rare events in event sequences. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98), New York City, New York, USA, August 27-31, 1998*, pages 359–363, 1998.

[129] Cristina M. Wilcox and Brian C. Williams. Runtime verification of stochastic, faulty systems. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 452–459, 2010.

[130] Fanny Yang, Sivaraman Balakrishnan, and Martin J. Wainwright. Statistical and computational guarantees for the baum-welch algorithm. *Journal of Machine Learning Research*, 18:125:1–125:53, 2017.

[131] Andrey Yavolovsky, Milos Zefran, and A. Prasad Sistla. Decision-theoretic monitoring of cyber-physical systems. In *RV - 16th International Conference*, pages 404–419, 2016.

[132] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 223–235, 2002.

[133] Kang Yu, Zhenbang Chen, and Wei Dong. A predictive runtime verification framework for cyber-physical systems. In *IEEE Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, CA, USA, June 30 - July 2, 2014 - Companion Volume*, pages 223–227, 2014.

[134] Lijun Zhang, Holger Hermanns, and David N. Jansen. Logic and model checking for hidden markov models. In *FORTE, 25th IFIP WG 6.1 International Conference*, pages 98–112, 2005.

[135] Xian Zhang, Martin Leucker, and Wei Dong. Runtime verification with predictive semantics. In *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, pages 418–432, 2012.

[136] Yuelong Zhu, Meijun Xu, Pengcheng Zhang, Wenrui Li, and Hareton Leung. Bayesian probabilistic monitor: A new and efficient probabilistic monitoring approach based on bayesian statistics. In *2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013*, pages 45–54, 2013.

APPENDICES

Appendix A

The Prism Models

The description of the running example and some of the case studies are provided using the Prism guarded command language.

A.1 The Die Example

```
dtmc

module die

// local state
s : [0..7] init 0;
// value of the die
d : [0..6] init 0;

[] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
[] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
[] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
[] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
[] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
[] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
[] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
[] s=7 -> (s'=7);
```

```
endmodule

label "ii0" = s=0;
label "hh0" = s=1 | s=3 | s=5;
label "tt0" = s=2 | s=4 | s=6;
label "tt1" = d=1;
label "hh2" = d=2;
label "tt3" = d=3;
label "hh4" = d=4;
label "tt5" = d=5;
label "hh6" = d=6;
```

A.2 The Randomized Dining Philosopher for 3 Philosophers

```
// randomized dining philosophers [LR81]
// atomic formulae
// left fork free and right fork free resp.

dtmc

formula lfree = (p2>=0&p2<=4)|p2=6|p2=10;
formula rfree = (p3>=0&p3<=3)|p3=5|p3=7|p3=11;

module phil1

p1: [0..11];

[] p1=0 -> (p1'=1); // hungry
[] p1=1 -> 0.5 : (p1'=2) + 0.5 : (p1'=3); // draw randomly - trying
[] p1=2 &  lfree -> (p1'=4); // pick up left
[] p1=2 &  !lfree -> (p1'=2); // left not free
[] p1=3 &  rfree -> (p1'=5); // pick up right
[] p1=3 &  !rfree -> (p1'=3); // right not free
[] p1=4 &  rfree -> (p1'=8); // pick up right (got left)
[] p1=4 & !rfree -> (p1'=6); // right not free (got left)
[] p1=5 &  lfree -> (p1'=8); // pick up left (got right)
[] p1=5 & !lfree -> (p1'=7); // left not free (got right)
[] p1=6  -> (p1'=1); // put down left
[] p1=7  -> (p1'=1); // put down right
[] p1=8  -> (p1'=9); // move to eating (got forks)
[] p1=9  -> (p1'=10); // finished eating and put down left
[] p1=10 -> (p1'=11); // put down right and return to think
[] p1=11 -> (p1'=0); // put down left and return to think

endmodule

// construct further modules through renaming
module phil2 = phil1 [ p1=p2, p2=p3, p3=p1 ] endmodule
module phil3 = phil1 [ p1=p3, p2=p1, p3=p2 ] endmodule

rewards "num_steps"
[] true: 1;
endrewards

// labels for philosopher 1
label "think" = p1=0;
label "hungry" = p1=1;
label "try" = p1=2 | p1=3;
label "pick" = p1=4 | p1=5;
label "drop" = p1=6 | p1=7;
label "release" = p1=10 | p1=11;
label "eat" = p1=8 | p1=9;
```

A.3 The Herman's Algorithm with 5 Processes

```
// herman's self stabilising algorithm [Her90]

// the procotol is synchronous with no nondeterminism (a DTMC)
dtmc
```

```
const double p = 0.5;

// module for process 1
module process1

// Boolean variable for process 1
x1 : [0..1];

[step]  (x1=x5) -> p : (x1'=0) + 1-p : (x1'=1);
[step] !(x1=x5) -> (x1'=x5);

endmodule

// add further processes through renaming
module process2 = process1 [ x1=x2, x5=x1 ] endmodule
module process3 = process1 [ x1=x3, x5=x2 ] endmodule
module process4 = process1 [ x1=x4, x5=x3 ] endmodule
module process5 = process1 [ x1=x5, x5=x4 ] endmodule

// cost - 1 in each state (expected number of steps)
rewards "steps"
true : 1;
endrewards

// formula, for use in properties: number of tokens
// (i.e. number of processes that have the same value as the process to their left)
formula num_tokens = (x1=x2?1:0)+(x2=x3?1:0)+(x3=x4?1:0)+(x4=x5?1:0)+(x5=x1?1:0);

// label - stable configurations (1 token)
label "stable" = num_tokens=1;
```

A.4 The Bounded Retransmission Protocol

```
// bounded retransmission protocol [D'AJJL01]
// gxn/dxp 23/05/2001

dtmc

// number of chunks
const int N=64;
// maximum number of retransmissions
const int MAX=2;

module sender

s : [0..6];
// 0 idle
// 1 next_frame
// 2 wait_ack
// 3 retransmit
// 4 success
// 5 error
// 6 wait sync
srep : [0..3];
// 0 bottom
// 1 not ok (nok)
// 2 do not know (dk)
// 3 ok (ok)
nrtr : [0..MAX];
i : [0..N];
bs : bool;
s_ab : bool;
fs : bool;
ls : bool;

// idle
[NewFile] (s=0) -> (s'=1) & (i'=1) & (srep'=0);
// next_frame
[aF] (s=1) -> (s'=2) & (fs'=(i=1)) & (ls'=(i=N)) & (bs'=s_ab) & (nrtr'=0);
// wait_ack
[aB] (s=2) -> (s'=4) & (s_ab'=!s_ab);
[TO_Msg] (s=2) -> (s'=3);
```

```
[TO_Ack] (s=2) -> (s'=3);
// retransmit
[aF] (s=3) & (nrtr<MAX) -> (s'=2) & (fs'=(i=1)) & (ls'=(i=N)) & (bs'=s_ab) & (nrtr'=nrt
[] (s=3) & (nrtr=MAX) & (i<N) -> (s'=5) & (srep'=1);
[] (s=3) & (nrtr=MAX) & (i=N) -> (s'=5) & (srep'=2);
// success
[] (s=4) & (i<N) -> (s'=1) & (i'=i+1);
[] (s=4) & (i=N) -> (s'=0) & (srep'=3);
// error
[SyncWait] (s=5) -> (s'=6);
// wait sync
[SyncWait] (s=6) -> (s'=0) & (s_ab'=false);

endmodule

module receiver

r : [0..5];
// 0 new_file
// 1 fst_safe
// 2 frame_received
// 3 frame_reported
// 4 idle
// 5 resync
rrep : [0..4];
// 0 bottom
// 1 fst
// 2 inc
// 3 ok
// 4 nok
fr : bool;
lr : bool;
br : bool;
r_ab : bool;
recv : bool;


// new_file
```

```
[SyncWait] (r=0) -> (r'=0);
[aG] (r=0) -> (r'=1) & (fr'=fs) & (lr'=ls) & (br'=bs) & (recv'=T);
// fst_safe_frame
[] (r=1) -> (r'=2) & (r_ab'=br);
// frame_received
[] (r=2) & (r_ab=br) & (fr=true) & (lr=false)  -> (r'=3) & (rrep'=1);
[] (r=2) & (r_ab=br) & (fr=false) & (lr=false) -> (r'=3) & (rrep'=2);
[] (r=2) & (r_ab=br) & (fr=false) & (lr=true)  -> (r'=3) & (rrep'=3);
[aA] (r=2) & !(r_ab=br) -> (r'=4);
// frame_reported
[aA] (r=3) -> (r'=4) & (r_ab'=!r_ab);
// idle
[aG] (r=4) -> (r'=2) & (fr'=fs) & (lr'=ls) & (br'=bs) & (recv'=T);
[SyncWait] (r=4) & (ls=true) -> (r'=5);
[SyncWait] (r=4) & (ls=false) -> (r'=5) & (rrep'=4);
// resync
[SyncWait] (r=5) -> (r'=0) & (rrep'=0);

endmodule

// prevents more than one file being sent
module tester

T : bool;

[NewFile] (T=false) -> (T'=true);

endmodule

module channelK

k : [0..2];

// idle
[aF] (k=0) -> 0.98 : (k'=1) + 0.02 : (k'=2);
// sending
[aG] (k=1) -> (k'=0);
// lost
```

```
[TO_Msg] (k=2) -> (k'=0);

endmodule

module channelL

l : [0..2];

// idle
[aA] (l=0) -> 0.99 : (l'=1) + 0.01 : (l'=2);
// sending
[aB] (l=1) -> (l'=0);
// lost
[TO_Ack] (l=2) -> (l'=0);

endmodule
```

Appendix B

The Source Code of Some Functions in *𝒫revent*

In this appendix we provide the source code in Matlab for some important functions in *𝒫revent*.

B.1 Training HMM

```
% Alphabet for randomized dining philosopher
Alphabet={'think','hungry','try','pick','drop','eat'};

% Read the training samples
dir = "randomized-dining-philosophers/Traces/Training/";
filename="LR_3Phils.csv";

Data = readCSVData(dir+filename);

Tolerance = 1e-06;
Iterations = 1000;

% Model folders
modelFolder =  "randomized-dining-philosophers/Trained-Models/3-Phils/";
```

```
% The maximum number of states to the trained HMM can have
n = 20;
% The log-likelihood of the trained HMM
logLH = zeros(1,n);
BIC = zeros(1,n);

% Training models with k states, starting from 1 to n
for k=1:n
    % Random initialization of trans and emis from a uniform dist.
    trans=ones(k,k);
    emis=ones(k,length(Alphabet));
    for i=1:k
        rv = randfixedsum(k,1,1,0,1);
        trans(i,:)=rv';
        rv = randfixedsum(length(Alphabet),1,1,0,1);
        emis(i,:)=rv';
    end

    % applying the initial probability,
    rv = randfixedsum(k,1,1,0,1);
    p = rv';
    trans_hat=[0 p; zeros(size(trans,1),1) trans];
    emis_hat=[zeros(1,size(emis,2)); emis];

    % Running the training algorithm
    % Size of Training set
    SizeTraining=length(Data);
    [estTrans,estEmis]=hmmtrain(Data(1:SizeTraining),trans_hat,emis_hat,
                      'Symbols',Alphabet, 'Tolerance', Tolerance,
                      'Maxiteration', iterations);
    disp('-----------------');
    output=sprintf('Number of states:%d\n Estimated HMM:',k);
    display(output);
    display(estTrans);
    display(estEmis);

    % Calculating the log-likelihood and BIC score
    for i=1:SizeTraining
```

```
        [PSTATEs,logpseq]=hmmdecode(Data{i},estTrans,estEmis, 'Symbols', Alphabet);
        logLH(1,k) = logLH(1,k)+logpseq;
    end
    BIC(1,k) = log(SizeTraining)*(k*k+k*length(Alphabet))-2*logLH(1,k);
```

B.2 Offline Monitoring with the Viterbi State Estimation

```
% Monitoring

% we assume the monitor table,  'monitorV' is already populated with the
% results from Prism.

% Reading monitoring trace -- can be changed to any other trace
dir = "randomized-dining-philosophers/Traces/Monitoring/";
execFile = "LR-sap_3Phils_B0.csv";

% The set of execution traces that the monitoring is performed on
execTrace = readCSVData(dir+execFile);
nObs = length(execTrace);

% Maximum horizon for the prediction -- currently obtained from the monitor
% vector (the maximum possible value of MAX_T)
MAX_T = size(monitorV, 3);
% The probability prediction computed by the monitor for each step of the
% trace
probVector={};
% The monitor estimated length of the extension that is accepted by the DFA
estLambda={};
% The actual length of the extension that is accepted by the DFA
lambda={};
% The for iterates through each trace in the moniotring set, if there is
% one (usual case), this loop only executes one
for j=1:nObs
    % Get a sample execution and simulate running it
```

```
Exec=execTrace{j};
% The first state of the DFA
q=dfa.InitialState;
% Initializing vectors
probVector{j}=[];
estLambda{j}=[];
lambda{j}=[];
% The previous index that the prefix was accepted by the DFA. It's used
% to determine the length of the new accepted extension
prevAccIdx = 0;
% the horizon index
T=MAX_T;
% Viterbi vectors
Vit = estEmis(:,find(strcmp(Exec(1),Alphabet)))'.*estTrans(1,:);
VitNext = Vit;
% The monitoring for each symbol in the execution trace
for i=1:length(Exec)
    % finding the most probable state using Viterbi vector
    [value,s]=max(Vit);
    % finding the state of the DFA
    q=GetTransitionState(dfa,q,Exec(i));
    % The emitted symbol
    o=find(strcmp(Exec(i),Alphabet));
    % CState is the index of the composite state at the DTMC
    % (the prediction model)
    CState=(s-1)*length(Alphabet)*length(dfa.FiniteSetOfStates)
             +(o-1)*length(dfa.FiniteSetOfStates)
             +find(strcmp(q,dfa.FiniteSetOfStates));
    % Finding the state in the monitor vector
    index=find(monitorV(:,1,T)==CState);
    % If there is a proaability defined for the composite state
    if ~isempty(index)
        probVector{j}(i)=monitorV(index,2,T);
    end
    % if the prefix is accepted by the DFA
    if (strcmp(q,dfa.FinalAcceptStates))
        % computing the value of lambda
        for k=prevAccIdx+1:i-1
```

```
                lambda{j}(k) = i-k;
            end
            % computing the estimation of lambda
            for k=prevAccIdx+1:i-1
                if(k-prevAccIdx < MAX_T)
                    estLambda{j}(k) = probVector{j}(k)*(i-k);
                else
                    estLambda{j}(k) = 0;
                end
            end
            prevAccIdx = i;
            % Reseting the horizon index
            T = MAX_T;
        % if the prefix is not accepted by the DFA
        else
            % Decrease the horizon
            T = T-1;
        end
        % There is no accepting extension of length MAX_T
        if( T <= 0 )
            % Reseting the horizon index
            T = MAX_T;
        end
        % Updating the Viterbi Vector for the next iteration
        if i < length(Exec)
            for k=1:size(estEmis,1)
                VitNext(k)=estEmis(k,find(strcmp(Exec(i+1),Alphabet)))
                              *max(Vit.*estTrans(:,k)');
            end
            Vit = VitNext;
        end
    end
end

% Calculating the mean of estLambda for each trace
estMeanVector=[];
for i=1:nObs
    estMeanVector(i)=mean(estLambda{i});
    end
```

B.3 Computing the Prediction Support of Symbols

```
function [F] = ComputePredictionSupport(Data, sigma, G, k)
% Computes the k-prediction support of sigma w.r.t. G over Data

% computing the sum of F
    F = zeros(length(Data),1);
    % using the parfor from tha parallel toolbox to speed up
    % computation of F
    parfor n=1:length(Data)
        % if k is larger than the length of the data then skip
        if k >= length(Data{n})-1
            continue;
        end
        % Vectorization
        % Finding the position of symbols in R over Data{n}
        Rpos=ismember(Data{n},sigma);
        % Finding the position of symbols in G (target labels) over Data{n}
        Gpos=ismember(Data{n},G);
        % Shifting Gpos k steps to the right
        shiftedGpos=zeros(size(Gpos));
        shiftedGpos(1:end-(k+1))=Gpos(k+2:end);
        % Computing the number of times that Data{n}(i)=sigma in R and
        % Data{n}(i+k)=sigma' in G
        prod=Rpos.*shiftedGpos;
        % Computing F as the sum
        F(n,1)=sum(prod(1:length(Data{n})-k-1));
        % Computing the prediction support
        F(n,1) = F(n,1) / (length(Data{n}) - k - 1);
    end
end
```