# Online Markov Chain Learning for Quality of Service Engineering in Adaptive Computer Systems

Yasmeen Rafiq

Doctor of Philosophy
University of York
Computer Science
January 2015

**Abstract**

Computer systems are increasingly used in applications where the consequences of failure vary from financial loss to loss of human life. As a result, significant research has focused on the model-based analysis and verification of the compliance of business-critical and security-critical computer systems with their requirements. Many of the formalisms proposed by this research target the analysis of quality-of-service (QoS) computer system properties such as reliability, performance and cost. However, the effectiveness of such analysis or verification depends on the accuracy of the QoS models they rely upon. Building accurate mathematical models for critical computer systems is a great challenge. This is particularly true for systems used in applications affected by frequent changes in workload, requirements and environment. In these scenarios, QoS models become obsolete unless they are continually updated to reflect the evolving behaviour of the analysed systems.

This thesis introduces new techniques for learning the parameters and the structure of discrete-time Markov chains, a class of models that is widely used to establish key reliability, performance and other QoS properties of real-world systems. The new learning techniques use as input run-time observations of system events associated with costs/rewards and transitions between the states of a model. When the model structure is known, they continually update its state transition probabilities and costs/rewards in line with the observed variations in the behaviour of the system. In scenarios when the model structure is unknown, a Markov chain is synthesised from sequences of such observations. The two categories of learning techniques underpin the operation of a new toolset for the engineering of self-adaptive service-based systems, which was developed as part of this research. The thesis introduces this software engineering toolset, and demonstrates its effectiveness in a case study that involves the development of a prototype telehealth service-based system capable of continual self-verification.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I would like to express my deepest gratitude to Dr Radu Calinescu, my supervisor, for his invaluable support, advice, and for his encouragement throughout the duration of this thesis and for introducing me to the world of research. I am also indebted to Prof. Richard Paige and to Prof. Vittorio Cortellessa for their invaluable advice and support.

Thank you to Dr Kenneth Johnson for his help and support; and to my office colleagues Simos Gerasimou, Gabriel Costa Silva, Babajide Ogunyomi, Dr Thomas Richardson and Dr Colin Paterson for their fruitful discussion and encouragement. I wish you all the very best in your PhDs and careers.

Thank you to all the members of the Cloud Computing for Large-Scale Complex IT Systems project for an invaluable experience. A warm thank you to the current and past members in the Enterprise Systems group, and to all friends in the YCCSA group, it has been a pleasure knowing you all and I wish you all every success. Special thanks are due to Sarah Christmas, Jayne Fitzgerald and Judith Pink for their help and support, and to Alison Watson for her advice and encouragement. I am grateful to all members in the Department of Computer Science for making my experience in York wonderful, and a special thanks to Alex Cooper who always made the time to help me overcome many technical related issues that were encountered. I am grateful to my sponsors EPSRC who funded this PhD project, it has been a great privilege.

I am deeply grateful to my parents, Mukhtiar Rafiq and Mohammad Rafiq, for their love, support and understanding, and to my brothers Tahir Rafiq and Tariq Rafiq for always being there when I needed you the most, and to rest of my family. Finally, I am grateful to all my wonderful friends, for their patience and for their unreserved support I received throughout my PhD, thank you.

# Declaration

I declare that the contents of this thesis has derived from my own research that was carried out between October 2010 and January 2015, during the period I was registered for the degree of Doctor of Philosophy. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Contributions from this thesis have been published in the following papers:

- R. Calinescu, K. Johnson, and Y. Rafiq. Using observation ageing to improve Markovian model learning in QoS engineering. In *Proceeding of the 2nd Joint WOSP/SIPEW International Conference on Performance Engineering, (ICPE 2011)*, pages 505–510, New York, NY, USA, 2011. ACM. Based on research described in Section 3.2.1 of this thesis.

- R. Calinescu, K. Johnson, and Y. Rafiq. Developing self-verifying service-based systems. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, pages 734–737, 2013. Based on research described in Chapter 5 of this thesis.

- R. Calinescu and Y. Rafiq. Using intelligent proxies to develop self-adaptive service-based systems. In *Proceedings of the 7th International Symposium on Theoretical Aspects of Software Engineering (TASE 2013)*, pages 131–134, July 2013. Based on research described in Sections 5.4 and 5.5.2 of this thesis.

- R. Calinescu, Y. Rafiq, K. Johnson, and M. E. Bakir. Adaptive model learning for continual verification of non-functional properties. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, pages 87–98. Based on research described in Section 3.3 of this thesis.

The above list of publications represents my original work, with the exception of: (i) the development of the COVE autonomic manger in [31, 36] and of the PRISM analysis of the bioinformatics workflow in [30], both of which were carried out by Dr Kenneth Johnson, the post-doctoral research associate on the EPSRC project that funded this PhD; and (ii) the web-based COVE GUI in [36], which was developed by Mehmet Bakir as part of his MSc project at the University of York. This last component of COVE is not presented in the thesis, as it is an extension of the PhD work.

*"If I am going to have a true memory, there are a thousand things that must first be forgotten."*

Thomas Merton

# Chapter 1

# Introduction

## 1.1 Motivation

As software use increases in business-critical and safety-critical applications [28, 107], so too does the adverse effect of unreliability or unpredictability in software [9, 57, 66, 169]. This may lead to failures that endanger human life, cause substantial economic loss or trigger extensive environmental damage [58, 72, 78, 107, 171]. To address this concern, significant research has focused on monitoring, modelling, analysing and verifying the compliance of computer systems [28, 51, 54, 65, 154, 151, 113] and of their components with functional and non-functional requirements [8, 101, 156, 158]. The formalisms proposed by this research effort focus on the analysis of reliability and performance quality-of-service (QoS) properties of computer systems. The definitions introduced in [12, 142] are adopted for these concepts. Thus, by quality of service we mean the operation cost of a web-service or probabilistic quality attribute of the service such as availability, reliability, and reputation" [29].

Building accurate formal models of computer systems is a great challenge for two reasons. First, the necessary knowledge may not be available until very late in the lifecycle of the system [6, 15, 30, 51, 52, 138, 159]. Second, Business-critical and safety-critical applications are increasingly being deployed in dynamic and unpredictable environments [16, 35, 37, 108] in which they must learn to self-adapt to change in order to achieve their objectives successfully. Therefore, it is important that software is both adaptable and dependable in the presence of changing conditions that may occur in the running environment [25, 29, 30, 66]. To this end, models should be developed not only to enable practitioners to reason about requirements and to identify possible conflicts in the system, but also to guide self-adaptation within the implemented system [9, 15, 17, 30, 66, 169]. To use models for the latter purpose, they must be kept in sync with the actual system behaviour at run-time, so that the necessary adaptations are driven by the analysis of an accurate model [14, 16, 30, 46, 66].

Markovian models are increasingly used to model and analyse QoS properties of technical systems [26, 29, 42, 66, 74, 75, 86, 113, 112]. The explanation for this trend is twofold. Firstly, as stated above, the use of technical systems in safety-critical [136, 144] and business-critical [81, 122] applications is on the rise, and so is the importance of ensuring that these systems comply with strict performance and reliability requirements [4, 28, 45, 153]. Secondly, the last decade has brought the advent of powerful software tools that assist system developers in building Markovian models, and automate model analysis. These tools are termed *probabilistic model checkers*. They take as input a formal system description expressed in a high-level modelling language, and convert

it into a Markovian chain that they then use to analyse user-specified QoS properties [28, 43, 84]. Examples of such properties are the probability that a fault occurs within a specified time period, and the expected response time of a web service under a given workload. Widely used probabilistic model checkers include PRISM [96], MRMC [104] and Ymer [162].

The types of Markovian models used to analyse the QoS properties of computer systems include discrete-time Markov chains, which are frequently used in analysing reliability-related QoS properties [30, 66], and continuous-time Markov chains, traditionally employed in analysing performance-related QoS properties such as response time and throughput [15]. Furthermore, variants of these models augmented with costs/rewards are increasingly used to express and analyse the costs associated with different configurations of critical system [72, 110]. These parameters often vary during the lifetime of a system, as a result of changes in system workload, environment[1] and internal state. In this research, we developed new techniques for learning the transition probabilities, the costs/rewards, and the structure of (discrete-time) Markov chains from run-time observations of the modelled systems.

Probabilistic model checking can be employed effectively at several stages in the life cycle of a technical system [28, 70, 84]. During system design, the technique can be used to analyse alternative solutions and identify those that satisfy the envisaged QoS requirements without having to build and test potentially expensive system prototypes [84]. For existing systems, probabilistic modelling and analysis can be used to verify whether QoS requirements are achieved or remedial action is needed to ensure compliance [43]. More recently, probabilistic model checking has been used to guide self-optimisation in autonomic IT systems during their execution stage [28, 29, 34, 32]. One of the primary focuses of this research has been to provide software practitioners with a set of tools for the model-driven development of dependable and self-adaptable software. To this end, the research presented in the thesis combines quantitative verification techniques with model learning techniques. These techniques are used at runtime, to predict and identify requirement violations, to plan the adaptation steps necessary to prevent or recover from violations, and to obtain evidence that the reconfigured software complies with its requirements [28].

Alternative techniques for analysing the QoS properties of technical systems include simulation and testing [1, 21, 134, 147]. However, these techniques can only examine a finite (and often small) number of scenarios that the system may operate in. As many systems are associated with an extremely large number of scenarios, the results produced by simulation and testing are approximate and cannot guarantee compliance with QoS requirements. In contrast, probabilistic model checking performs an exhaustive analysis of the considered QoS properties, producing precise results that guarantee or disprove each analysed property irrefutably [96, 103, 111].

## 1.2 Contributions

The main hypothesis of this project has been that it is possible to use observations of a running service-based system's behaviour to continually update Markovian models of the system, and to use these updated models (a) to verify the system's compliance with non-functional requirements at runtime; and (b) to dynamically reconfigure the system in order to recover from violations of these requirements.

[1] In requirements engineering the environment is the part of the world in which a computer system is operating[163]. For example, the Web Services are deployed in dynamic and unpredictable environment of the Internet in a Service-Oriented Architecture.

In this PhD project we have developed new techniques and software engineering tools that address the major challenge of basing adaptation decisions in QoS engineering on accurate models of the underlying systems. In particular, the project has developed a set of rigorous approaches for the on-line learning of the parameters and structure of (discrete-time) Markov chains used in the analysis of QoS properties of service-based systems. The techniques take as the input runtime observations of events that are associated with cost/rewards and transitions between states of the model.

These approaches and their advantages and novel characteristics are detailed in the following list of the main contributions of the thesis:

1. A parameterised on-line learning method that infers the state transition probabilities of a Markov model of a system from observations of the system behaviour, and adjusts its parameters dynamically depending on the frequency of these observations. This *adaptive learning* leads to a faster and more accurate inference of the transition probabilities than that provided by existing methods. Rigorous theoretical results link the parameters chosen dynamically by our learning method to the expected error in the accuracy of the learnt state transition probabilities. This allows the configuration of the adaptive learning method so that it yields results within an acceptable expected error range.

2. A novel approach that uses a Kalman filter and a recursive weighted least-squares filter to establish cost/reward structures for Markov chains. These structures are associated with, and support the analysis of, performance-related QoS properties of component-based systems whose instrumentation is not possible or not desirable (e.g., embedded and real-time systems). The approach supports the identification of under-performing components, and the dynamic reconfiguration of the modelled systems.

3. A model learning technique for synthesising Markov chains with the desired degree of accuracy for a class of *observable* black-box component-based systems [2]. These models are used in our continual runtime verification framework, to detect changes in QoS models (e.g., to reflect component failures in running computer systems), enabling new approaches to self-configuration, and detecting components that leave or join the system at runtime.

4. A tool-supported framework for engineering service-based systems (SBSs) capable of self-verifying their compliance with reliability requirements. The self-verifying systems developed using this framework employ continual formal verification to select the service combination that guarantees the realisation of their reliability requirements with minimal cost. The underlying model is updated online to reflect changes in the service reliability and in the frequency with which the SBS operations are invoked. The continual verification, model updating and service selection capabilities are fully automated, and are provided by a combination of reusable and automatically generated software components. The development process supported by the new framework resembles the traditional SBS development process, so practitioners can use it with little learning effort.

5. Case studies that evaluate the the efficiency (i.e., time to detect underperforming service(s)) and accuracy (i.e., ability to detect system violations). These case studies will serve as exemplars for other researchers and practitioners to gain insight into the engineering of self-adaptive service-based software systems.

[2] This is in contrast to a *white-box* (or a glass-box) system [140, 141], whose internal components and component relationships are known.

## 1.3 Overview

The remainder of the thesis is organised as follows. Chapter 2 introduces the key concepts, notations and mathematical models underpinning the research carried out by the PhD project. This includes the Markov chains whose parameters, cost/reward structures, and model structures are learnt by the techniques introduced in the thesis, the temporal logic used to specify the properties of these models, and the optimal filters employed in the synthesis of cost/reward structures for Markov chains.

The next three chapters describe the research contributions of the thesis. Chapter 3 presents the online learning of QoS model parameters, introducing the new algorithms for learning the state transition probabilities from runtime observations of the behaviour of the modelled system; and use the optimal filters to establish unknown costs/rewards of (discrete-time) Markov chains. These techniques are usable in scenarios where state transition probabilities and response times of system components are prone to changing dynamically. Chapter 4 focuses on the online learning of the structure of Markov chains, and the use of this technique within the software engineering framework mentioned in the next chapter. Chapter 5 presents an automated model-driven engineering framework that exploits the theoretical results from Chapters 3 and 4, which is used to develop service-based systems with learning-driven adaptation capabilities. Each of Chapter 3 to 5 begins with a review of related work.

Finally, Chapter 6 summarises the insights gained from the PhD project, its contributions and their envisaged applications. Also in this chapter, we suggest a range of areas for further research that can extend or build on the results of the PhD project.

# Chapter 2

# Background

This chapter introduces the key concepts, notations and mathematical models underpinning the research carried out as part of this PhD project and described in the rest of the thesis. These include the quality-of-service models learnt by the techniques presented in the thesis, the temporal logic used to specify properties of these models, and the Kalman filter and recursive weighted least-square filter used to learn cost/reward structures associated with such models.

## 2.1 Markovian models

Markovian models are state-transition systems comprising a finite set of *states* that correspond to different configurations of a real system, and *state transitions* associated with the transitions that are possible between these states. State transitions are typically annotated with probabilities that reflect the probabilities with which the transition occurs in the real system.

### 2.1.1 The Markov property

Markov models make the simplifying assumption that, given the current state of the model, the next state is independent of past model states. In other words, the current state of the modelled system encodes all the knowledge required to know the next system state. Formally, a state-transition process is called a Markov process if [106]

$$P[X_{n+1} = j | X_0 = i_0, ..., X_n = i_n] = P[X_{n+1} = j | X_n = i_n], \tag{2.1}$$

where the random variables $X_0, X_1, \ldots, X_{n+1}$ represent the state of the process at time step $t = 0, 1, \ldots, n+1$, and $i_0, ..., i_n$ and $j$ represent states of the process. Because the distribution of $X_{n+1}$ depends only on the current state $X_n = i_n$ and not on the whole history $\{X_0 = i_0, \cdots, X_n = i_n\}$, this is referred to as the *memorless property* of a Markov model, or simply the Markov property. As a consequence, the past $\{X_0, ..., X_{n-1}\}$ and the future $\{X_{n+1}, ..., X_{n+m}\}$ are conditionally independent given the present $X_n = i_n$, i.e.,

$$\begin{aligned} P[X_0 =& i_0, ..., X_{n-1} = i_{n-1}, X_{n+1} = i_{n+1}, ..., X_{n+m} = i_{n+m} | X_n = i_n] = \\ &= P[X_0 = i_0, ..., X_{n-1} = i_{n-1} | X_n = i_n] \times \\ &\quad P[X_{n+1} = i_{n+1}, ..., X_{n+m} = i_{n+m} | X_n = i_n] \end{aligned} \tag{2.2}$$

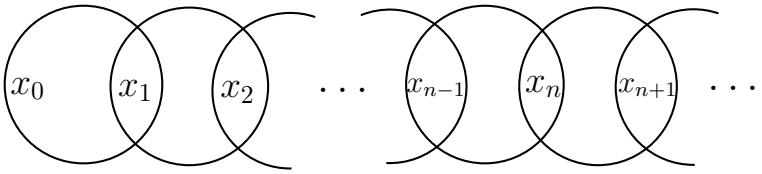


Figure 2.1: Chain dependence in a Markov process (adapted from [106])

This does not imply that the past lacks information about the future state. On the contrary, the past does affect the future through the present state, as illustrated in Figure 2.1, which shows how the random variables $X_0, X_1, \ldots, X_{n+1}, ...$ are connected by a chain of events.

### 2.1.2 Markov chains

The following formal definition is adapted from [13].

**Definition 2.1.** A (discrete-time) Markov Chain (MC) is a tuple

$$\mathcal{M} = (S, s_0, \mathrm{P}, L), \tag{2.3}$$

where

- $S = \{s_0, s_1, ..., s_{n-1}\}$ is a finite set of $n \geq 1$ states;
- $s_0 \in S$ is the initial state;
- $\mathrm{P} : S \times S \to [0, 1]$ is the state transition matrix; element $p_{ij}$ from P represents the probability of transitioning to state $s_j$ from state $s_i, 1 \leq i, j \leq n$, and $\sum_{j=1}^{n} p_{ij} = 1$ for all states $s_i \in S$;
- $L : S \to 2^{AP}$ is a labelling function which assigns a set of atomic propositions from $AP$ to each state in $S$.

The model $\mathcal{M}$ is said to be finite if $S$ and $AP$ are finite, and the size of $\mathcal{M}$ is denoted by $size(\mathcal{M}) = |S|$. The states $s' \in S$ for which $P(s, s') > 0$ are the possible successors of $s$. Therefore, the probability of moving from state $s$ to a state belonging to a subset of $T \subseteq S$, denoted $P(s, T)$ denote the probability of moving from $s$ to some state $t \in T$ in a single step. This probability is given by

$$P(s, T) = \sum_{t \in T} P(s, t). \tag{2.4}$$

Equation (2.4) can be expressed as a matrix $(P(s, t))_{s,t \in S}$, where the row $P(s, .)$ for state $s$ holds the probabilities of moving from $s$ to its successors, while the column $P(., s)$ for state $s$ includes the probabilities of entering state $s$ from any other state.

**Example 1.** Consider a web service whose invocation can be affected by two types of errors. Assume that the attempt to invoke the web service fails with probability 0.1 due to connectivity problems, and the requests that reach the web service can further fail with a probability of 0.25 due to problems with the web service implementation. If the web service invocation fails, then the system continues to retry until a successful invocation

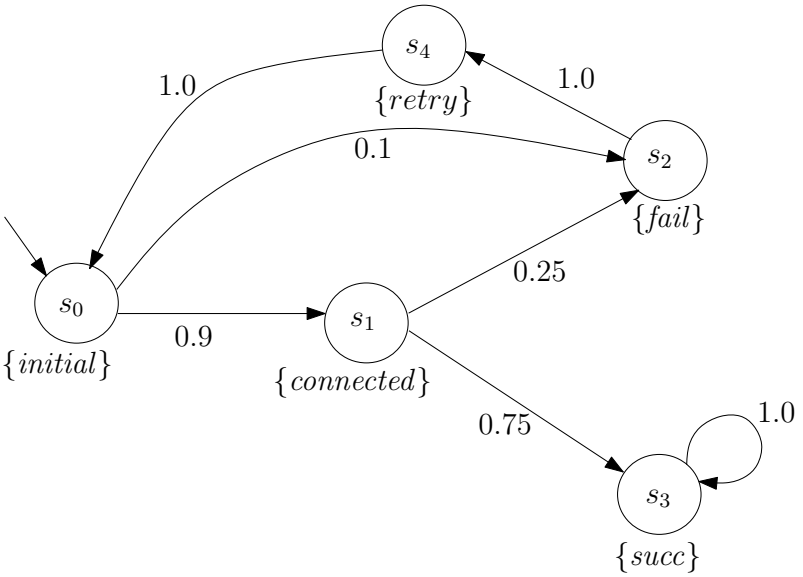


Figure 2.2: Example of a Markov chain model

is completed. Figure 2.2 shows the Markov chain model $\mathcal{M}$ for an invocation of this web service, with states depicted as vertices and state transitions depicted as directed edges that join pairs of states. The MC $\mathcal{M}$ has five states: $S = \{s_0, s_1, s_2, s_3, s_4\}$, with the initial state $s_0$ corresponding to an invocation of the web service being initiated. State $s_1$ corresponds to a successful connection with the web service having been established, and state $s_3$ corresponds to the successful invocation of the service. Finally, state $s_2$ corresponds to a failed connection or execution of the web service, and state $s_4$ to a retry being initiated.

The transition probability matrix $P$ for this Markov model is given by:

$$\mathrm{P} = \begin{pmatrix} 0 & 0.9 & 0.1 & 0 & 0 \\ 0 & 0 & 0.25 & 0.75 & 0 \\ 0 & 0 & 0 & 0 & 1.0 \\ 0 & 0 & 0 & 1.0 & 0 \\ 1.0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The atomic propositions used to label the states are taken from the set $AP = \{initial, connected, fail, succ, retry\}$. The labelling function enables the association of meaningful names to states of the MC:

$$L(s_0) = \{initial\}, L(s_1) = \{connected\}, L(s_2) = \{fail\}\ , L(s_3) = \{succ\} \text{ and}$$
$$L(s_4) = \{retry\}.$$

### 2.1.3 Extension of Markov chains with costs/rewards

To support the evaluation of a broader range of quantitative properties, MCs are augmented with reward (or cost) information. This involves associating positive real-

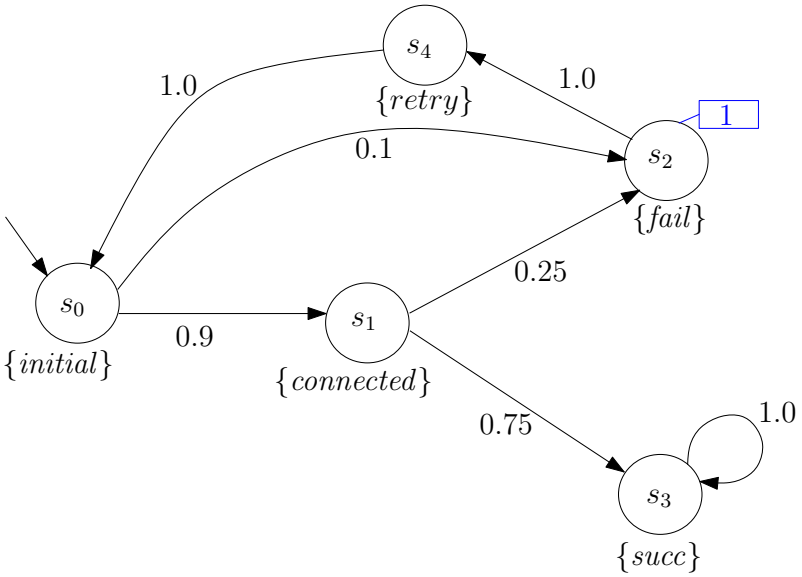


Figure 2.3: Example of a Markov chain with costs/rewards

valued quantities to the Markov model states and/or transitions. Semantically, these quantities correspond to costs that we aim to minimise, or to rewards that we want to maximise. However, the two are equivalent from a mathematical standpoint, as formally defined below.

**Definition 2.2.** Given a MC $\mathcal{M} = (S, s_0, P, L)$, a reward structure for $\mathcal{M}$ is a pair of functions $(\underline{\rho}, \iota)$, where:

- $\underline{\rho} : S \to R_{\geq 0}$ is the state reward function (a vector);
- $\iota : S \times S \to R_{\geq 0}$ is the transition reward function (a matrix).

**Example 2.** Figure 2.3 depicts a cost-annotated MC model $\mathcal{M} = (S, s_0, P, L)$ for the simple web service invocation from Example 1, where $\rho : S \to \mathbb{R}_+$ associates a cost $\rho(s_2) = 1$ to the "failed" state. For any other state $s \in S \setminus \{s_2\}$, $\rho(s) = 0$, and $\iota$ is not used.

## 2.2 Probabilistic computation tree logic

An execution of an MC $\mathcal{M} = (S, s_0, P, L)$ is represented by a *path*. Formally, a path $\pi$ is a non-empty sequence of states $s_0, s_1, s_2 ... s_n$ where $s_i \in S$ and $P(s_i, s_{i+1}) > 0$, for $0 \leq i \leq n$. A path can either be a finite or infinite. The $i$th state of a path $\pi$ is denoted by $\pi(i)$, the length of a given $\pi$ (number of transitions) is $|\pi|$ and for a finite path $\pi_{fin}$, the last state is $last(\pi_{fin})$. We say that a finite path $\pi_{fin}$ of length $n$ is a *prefix* of the infinite path $\pi_{inf}$ if $\pi_{fin}(i) = \pi_{inf}(i)$ for $0 \leq i < n$.

In order to reason about the probabilistic behaviour of the MC, we need to determine the probability that certain paths are taken. We capture non-functional QoS properties relevant to the modelled system using formal specification languages such as probabilistic computation tree logic (PCTL). The following definition, introduced in [13], define the syntax of PCTL.

**Definition 2.3.** Let $AP$ be a set of atomic propositions and $a \in AP$, $p \in [0,1]$, $k \in \mathbb{N}, r \in \mathbb{R}$ and $\bowtie \in \{\geq, >, <, \leq\}$. Then a state-formula $\Phi$ and a path formula $\Psi$ in PCTL are defined by the following grammar:

$$\begin{aligned} \Phi &::= true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid P \bowtie_p (\Psi) \\ \Psi &::= X\Phi \mid \Phi U \Phi \mid \Phi U^{\leq k} \Phi \end{aligned} \tag{2.5}$$

PCTL distinguishes between state formulae $\Phi$ and path formulae $\Psi$ that are interpreted over the states of an MC model $\mathcal{M}$. The state formulae includes the standard logical operators $\wedge$ and $\neg$, which also allow the formulation of other usual logical operators, such as disjunction ($\vee$) and implication ($\Rightarrow$), and $false$. The main extension of the state formulae, compared to non-probabilistic logics such as CTL, is the replacement of the traditional path quantifiers $E$ and $A$ with the probabilistic operator $\mathcal{P}$. This operator can be used to define upper and lower bounds on the probability of the system evolution. As an example, the formula $\mathcal{P}_{\geq p}(\Psi)$ is true at a given state if the probability of the future evolution of the system satisfying $\Psi$ is at least $p$. The semantics of PCTL is defined formally as follows.

**Definition 2.4.** Let $\mathcal{M} = (S, s_0, P, L)$ be a MC. For any state $s \in S$, the satisfaction relation $\models$ is defined inductively by:

- $s \models true$ for all $s \in S$
- $s \models a \Leftrightarrow a \in L(s)$
- $s \models^{\neg} \Phi \Leftrightarrow s \not\models \Phi$
- $s \models \Phi \wedge \Psi \Leftrightarrow s \models \Phi \wedge s \models \Psi$
- $s \models P_{\bowtie p}[\Psi] \Leftrightarrow Prob^M(s, \Psi) \bowtie p$

The path formulae that can be used with the probabilistic path operators are:

- the "next" formula $X\Phi$, which holds if $\Phi$ is true in the next state of a path;
- the time bounded "until" formula $\Phi_1 U^{\leq k} \Phi_2$, which requires that $\Phi_1$ holds continuously up to some time step $x < k$ and $\Phi_2$ becomes true at time step $x + 1$;
- the unbounded "until" formula $\Phi_1 U \Phi_2$, whose semantics is identical to that of the bounded "until", but the time-step bound is set to infinity $t = \infty$.

The standard PCTL operator "until" can be used to express additional properties encountered in other formalisms, including "eventually" $F$, "always" $G$, and "weak until" $\mathcal{W}$. For example, the "eventually" formula $P_{\bowtie p}(F\Phi)$ is semantically equivalent to $P_{\bowtie p}(true\, U \Phi)$.

**Example 3.** Below are some typical examples of PCTL formulae:

- $P_{\geq 0.4}[X \text{ delivered}]$ — The probability that a message has been delivered after one time-step is at least 0.4;

- $init \rightarrow P_{\leq 0}[F \text{ error}]$ — from any initial configuration, the probability that the system reaches an error state is 0;

- $P_{\geq 0.9}[\neg \text{down } U \text{ served}]$ — the probability the system does not go down until after the request has been served is at least 0.9;

- $P_{<0.1}[\neg \text{done } U^{\leq 10} \text{ fault}]$ — the probability that a fault occurs before the protocol finishes and within 10 time-steps is strictly less than 0.1.

To enable the verification of additional properties of MCs, PCTL is extended with costs/rewards as defined below.

**Definition 2.5.** The cost/reward augmented PCTL state formulae are defined by the grammar:

$$\Phi ::= R_{\bowtie r}[I^{=k}] \mid R_{\bowtie r}[C^{\leq k}] \mid R_{\bowtie r}[F\Phi] \tag{2.6}$$

where $\bowtie \in \{<, \leq, \geq, >\}, r \in \mathbb{R}_{\geq 0}, k \in \mathbb{N}$ and $\Phi$ is a PCTL state formula.

The cost/reward operator $R$ can be used to analyse the expected cost at time step $k$ ($R_{\bowtie r}[I^{=k}]$), the expected cumulative cost up to time step $k$ ($R_{\bowtie r}[C^{\leq k}]$), and the expected cumulative cost to reach a future state that satisfies a property $\Phi$ ($R_{\bowtie r}[F\Phi]$). Further details about the formal semantics of PCTL are available from [50, 93].

**Example 4.** For the Markov model and cost/reward structure from Example 2, the PCTL formula $R_{\leq 4}[Fsucc]$ is true iff the expected number of failures before reaching the success state does not exceed 4.

## 2.3 Probabilistic model checkers

PProbabilistic model checkers are formal verification software tools that assist software developers in modelling and analysing systems that exhibit stochastic behaviour [109]. A probabilistic model checker takes as its input a probabilistic model and a property specification. The former is expressed in a high-level modelling language (Figure 2.4), and is then transformed into a Markovian chain that is used to analyse user-specified quantitative properties. The latter is expressed in probabilistic temporal logics. In the case of a DTMC, properties are expressed in probabilistic computational tree logic, using probabilistic operators with additional features such as time bounds and costs. Efficient model checking algorithms are then performed on the model by induction over the syntax of the analysed property. These algorithms can produce two types of outputs, either true/false to indicate whether the property holds given the current model, or a numerical value, for example, the probability that the property holds or the expected cost to reach a goal state. Widely used probabilistic model checkers include PRISM [96], MRMC [104] and Ymer [162].

```
dtmc

label "initial"   = a=0;
label "connected" = a=1;
label "fail"      = a=2;
label "succ"      = a=3;
label "stop"      = a=4;
label "retry"     = a=5;

//rewards structure that associates costs to the failed invocation
rewards
(a=2) : 1.0;
endrewards

module workflow

a : [0..4] init 0;

[initial]     (a=0) -> 0.9  : (a'=1) + 0.1  : (a'=2);
[connected]   (a=1) -> 0.25 : (a'=2) + 0.75 : (a'=3);
[fail]        (a=2) -> 1.0  : (a'=4);
[succ]        (a=3) -> 1.0  : (a'=3);
[retry]       (a=4) -> 1.0  : (a'=0);

endmodule
```

Figure 2.4: PRISM model for the Markov chain with costs/rewards in Example 2

## 2.4 The Kalman filter

The Kalman filter is a widely used algorithm for estimating the parameters of real-world systems from indirect, inaccurate measurements affected by noise. The Kalman filter is *optimal* in the sense it minimises the mean square error of the estimated parameters if the noise is normally distributed. The filter has been successfully used in application domains ranging from autonomous underwater navigation [67, 114, 119] and aerospace [24, 3, 87] to electronics [126, 131] and trust-based models [161, 97]. In this section we describe the Kalman filter used in a linear system to estimate the state $x \in \mathbb{R}^n$ of a discrete-time process at time $t$, where this process has evolved from the prior state $x_{t-1}$ according to the equation

$$x_t = Ax_{t-1} + Bu_{t-1} + w_{k-1}, \tag{2.7}$$

where

- $x_t$ is an $n \times 1$ column vector that holds $n \geq 1$ state parameters of the system;
- $u_t$ is an optional control input $(l \times 1)$ column vector;
- $A$ is an $n \times n$ matrix that relates the state of the previous time step $t-1$ to the current time step $t$;
- $B$ is an $n \times l$ matrix that relates the control input $u_{t-1}$ to the current state $x_t$;
- $w_t$ is the vector containing the state transition noise for each parameter in the state vector. The process noise is assumed to have a mean of zero and a normal distribution with covariance given by the covariance matrix $Q_t$.

The Kalman filter recursively makes a prediction based on the current state of the linear system, and corrects the prediction using actual measurements $z \in \mathbb{R}^m$ taken from runtime observations, according to the measurement equation

$$z_t = Hx_t + v_t, \tag{2.8}$$

where

- $z_t$ is an $m \times 1$ column vector containing the measurements;
- $H$ is the $m \times n$ transformation matrix that maps the state vector parameters into the measurement domain;
- $v_t$ is the $m \times 1$ column vector containing measurement noise for each observation in the measurement vector. Like the process noise, the measurement noise is assumed to be zero-mean Gaussian white noise with covariance given by the covariance matrix $R_t$.

### 2.4.1 Observability of system state

The accuracy of the prediction is determined by how close the system model (2.8) is to the actual system (2.7), and the performance of the estimation is determined by the prediction. Therefore, the performance of the Kalman filter is determined by the system model. Furthermore, a system is deemed observable if from the model the properties of the real system are observable, i.e., all the state parameters in $x_t$ are linearly independent from the transformation matrix $H$. Observability can be characterised by the rank of the matrix.

$$M = \begin{bmatrix} H^{\mathrm{T}} & A^{\mathrm{T}}H\mathrm{T} & \cdots (A^T)^{n-1}H^T \end{bmatrix}. \tag{2.9}$$

The system (2.7) is observable if the rank $(M) = n$.

**Example 5.** Consider the following linear system:

$$x_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} x_{t-1} + \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} w_{t-1} \tag{2.10}$$

$$z_k = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} x_t \tag{2.11}$$

The observability matrix, according to (2.9), is

$$M = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \text{ rank of } M = 2 \tag{2.12}$$

The rank is less than the dimension of $x_t$ (i.e., 3), so the system is not observable.

### 2.4.2 Using the Kalman filter

Given a system described by the state equation (2.7), we start with an initial parameter estimate $x_0$ and with an $n \times n$ error covariance matrix $P_0$ which is the measurement of the estimated accuracy of the parameter $x_0$ (these initial values are provided by system expert), and recursively apply the sequence of steps described below.

1. **Predict:** The Kalman filter only takes estimated parameters from the previous time step (t-1), so the first step is to predict the parameter $x_t$ and the error covariance matrix $P_t$:

$$\hat{x}_t = Ax_{t-1} + w_{t-1} \tag{2.13}$$

$$\hat{P}_t = AP_{t-1}A^T + Q \tag{2.14}$$

2. **Update** : The following calculations are carried out:

   - From the current observation $z_t$, we update the *measurement residual* (also called the *measurement innovation*). The residual reflects the difference between the predicted measurement, $H\hat{x}_t$, and the actual measurement, $z_t$:

$$e_t = z_t - H\hat{x}_t \tag{2.15}$$

     A residual of zero means that the two are synchronised.

   - We then calculate the optimal Kalman gain as

$$K_t = \hat{P}_t H^T (H\hat{P}_t H^T + R_t)^{-1} \tag{2.16}$$

   - Next, we update (a *posterion*) parameter estimate $x_t$ that is used in the next iteration of the Kalman filter:

$$x_t = \hat{x}_t + K_t e_t \tag{2.17}$$

   - Finally, we update (a *posterion*) estimate covariance $P_t$ that will be used in the next iteration:

$$P_t = \hat{P}_t - K_t H\hat{P}_t \tag{2.18}$$

Intuitively, if we are more confident in the measurement $z_t$, then the error covariance (R) can decrease to zero and the Kalman gain will weight the residual more heavily than the prediction. In contrast, if more confidence is placed in the prediction, then the prediction error covariance $\hat{P}_t$ will decrease to zero and $K_t$ will increase and weight the prediction more heavily than the residual.

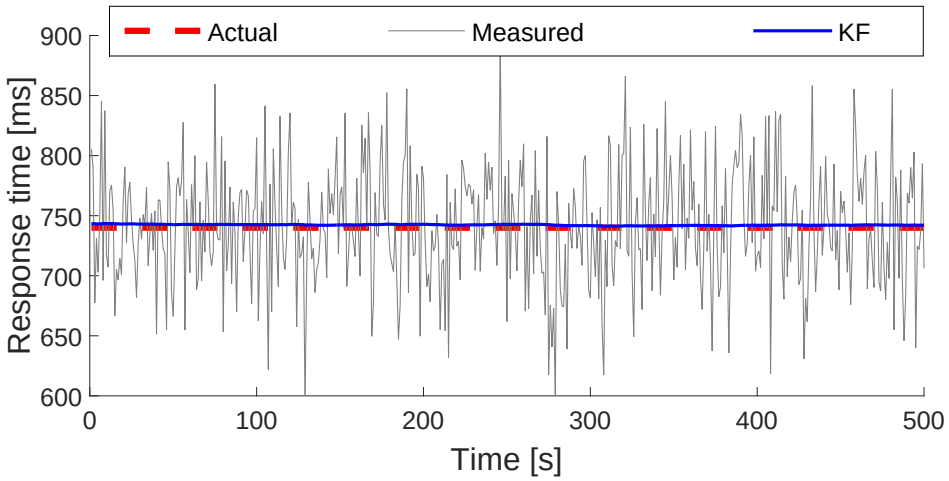


Figure 2.5: Response time estimate of a third-party service using the Kalman filter

**Example 6.** Suppose we need to use the Kalman filter to estimate the execution time of a third-party service, where the actual response time is different from that advertised as the prior value $x_0$. Furthermore, assume that the measured response time is constant but subject to white noise due to network disruptions. To implement the Kalman filter, we set the following values for A, H, Q, and R which compose the system model:

$$\begin{aligned} A &= 1 \\ H &= 1 \\ Q &= 0 \\ R &= 5 \end{aligned} \tag{2.19}$$

Note that the error covariance matrix Q is zero because $w_k = 0$. Because $x$ is a scalar, $A$ and $H$ are both 1, and thus the two equations for the system model are:

$$\begin{aligned} x_t &= x_{t-1} \\ z_t &= x_t + v_t \end{aligned} \tag{2.20}$$

Finally, the initial estimate are:

$$\begin{aligned} x_0 &= 600 \\ P_0 &= 50 \end{aligned}$$

If the initial condition of the system is unknown, then it is best to set a large covariance.

Figure 2.5 shows the results for our example using a simulator of the third-party service, with 500 measurements of the response time (RT) observed over 500 seconds of simulated time. We used the Kalman filter to estimate the RT of the third-party service.

As shown in this diagram, the measured RT is widely dispersed around the actual RT value due to the measurement noise. On the other hand, the Kalman estimate quickly converges to the actual value by the removal of the noise through the Kalman filter. See Appendix A for the Matlab code for the Kalman filter function and Appendix B for the corresponding test program.

## 2.5 Recursive weighted least-squares filter

The recursive weighted least-squares (RWLS) filter is an adaptive filter that is widely used in applications such as beamforming [11, 60, 166], blind multi-user detection in code-division multiple-access systems [47, 48] and system identification [2, 157, 167]. The RWLS filter uses a *forgetting function* $\lambda$ that controls the way in which each measurement is incorporated relative to other measurements. The choice should be such that measurements that are relevant to current system properties are included. The weighing function is referred to as the *exponential weighting into the past* (EWP) and is described in [63]. It should be noted that a *forgetting function* $\lambda$ is used to shape the estimator's memory. The filter is used in linear systems, to estimate the state $x_t \in \mathbb{R}^n$ of a discrete-time process at time $t$, where this process has evolved from the prior state $x_{t-1}$. While the Kalman filter requires a complete prior knowledge of the state-space model (2.7) and its parameters, the RWLS filter does not require a dynamic model for the time-varying parameters, and relies instead on the forgetting function $\lambda$.

The RWLS filter recursively makes a prediction on the current condition of the model $z \in \mathbb{R}^m$, and corrects the prediction using actual measurements taken from runtime observations, according to the measurement equation

$$z_t = Hx_t + v_t, \tag{2.21}$$

where

- $z_t$ is an $m \times 1$ column vector containing the measurements;
- $H$ is an $m \times n$ transformation matrix that maps the state vector parameters into the measurement domain;
- $v_t$ is an $m \times 1$ column vector containing measurement noise for each observation in the measurement vector. Like the process noise, the measurement noise is assumed to be zero-mean Gaussian white noise, with covariance given by a covariance matrix $R$.

The RWLS filter varies the covariance matrix $R$ over time with the *forgetting factor* $\lambda_t$. The following steps describe the RWLS algorithm.

1. **Predict:** The RWLS filter uses only the estimated parameters from the previous time step $(t-1)$, so the first step is to predict the parameter $x_t$ and the error covariance matrix $P_t$

$$\hat{x_t} = x_{t-1}, \tag{2.22}$$

$$\hat{P_t} = P_{t-1}, \tag{2.23}$$

2. **Update:** Using the current observations, the following calculations are carried out:

- Update the measurement residual

$$e_t = z_t - H\hat{x}_t. \tag{2.24}$$

- Update the measurement prediction covariance matrix

$$S_t = HP_tH^T + \lambda_t R. \tag{2.25}$$

- Calculate the gain

$$K_t = \hat{P}_t H^T S_t^{-1}. \tag{2.26}$$

- Update the estimate $x_t$ that will be used in the next iteration

$$x_t = \hat{x}_t + K_t e_t. \tag{2.27}$$

- Update the error covariance estimate $P_t$ that will be used in the next iteration

$$P_t = \frac{1}{\lambda}\hat{P}_t - K_t S_t K_t^T. \tag{2.28}$$

### 2.5.1 Using the recursive weighted least-squares filter

**Example 7.** Suppose we want to use the RWLS filter to estimate the response time of a third-party service, where the actual response time is different from that advertised as the prior value $x_0$. Furthermore, the measured response time is constant but subject to white noise due to network disruptions. To use the RWLS filter in this scenario, we set the following values for the system parameters:

$$\begin{aligned} H &= 1 \\ R &= 5 \\ x_0 &= 600 \\ P_0 &= 50 \\ \lambda &= 0.9 \end{aligned} \tag{2.29}$$

We used the same simulator of the third-party service as in Example 6, and collected 500 measurements of the response time (RT) of the simulated service, observed over 500 seconds of simulated time. Figure 2.6 shows the results obtained using the RWLS Filter to estimate the RT of the third-party service. Like in Example 6, the measurement is widely dispersed due to the measurement noise, yet the RWLS estimate converges rapidly to the actual RT value. Note that although the behaviours of the Kalman filter and RWLS filter for the simple system from this example are very similar, the two have different advantages and limitations when used as part of the learning techniques we introduce later in Chapter 3. See Appendix C for the Matlab code for the RWLS filter function and Appendix D for the corresponding test program.

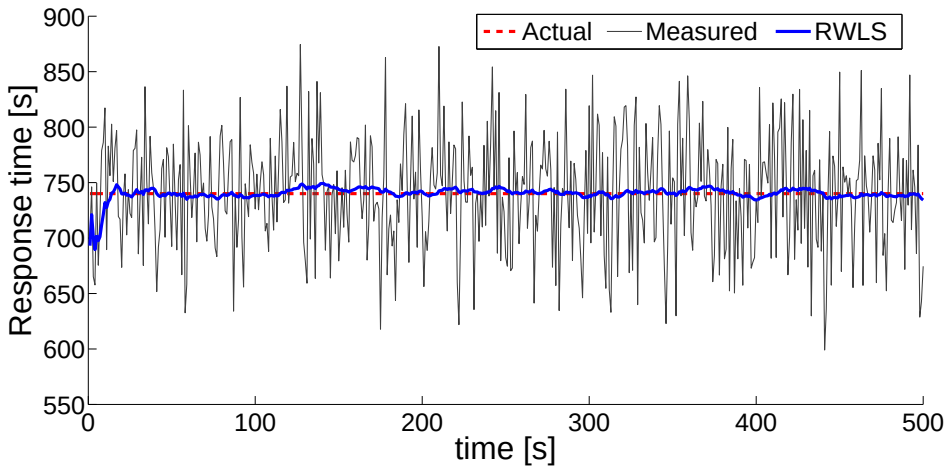


Figure 2.6: Response time estimate of a third-party service using the RWLS filter

# Chapter 3

# Online learning of Markov chain parameters

This chapter introduces new techniques for learning the transition probabilities of (discrete-time) Markov chain (MC) models and their costs/reward structures from run-time observations of the systems modelled by this class of Markov chains. The chapter begins with a critical analysis of existing techniques for learning parameters of Markovian models used in quality-of-service (QoS) engineering. The analysis emphasises the need for learning the parameters of these models when the modelled system changes over time; and discusses the advantages and limitations of existing approaches that address this need.

This review of related work is followed by the description of the new Bayesian-style learning algorithms for estimating the state transition probabilities of MC. The sensitivity of the new algorithms depends on two parameters, and we provide a detailed analysis of the learning algorithms and of the roles and options for customising their parameters. The final part of the chapter describes a new learning algorithm for estimating the unknown costs/rewards of MC models associated with component-based systems such as workflows and embedded systems. The technique is applicable to systems that execute multiple sequences of operations such that the cumulative costs/rewards associated with each sequence can be measured, but it is not possible to measure the cost/reward for individual operations and/or transitions.

## 3.1 Related work

Significant research has focused on monitoring the performance and reliability properties of technical systems [39, 64, 135, 139, 132, 133, 160], and on modelling and analysing these properties formally [15, 155, 92]. The spread of evolving critical system in multiple application domains [27, 59, 94, 148] has led to a growing need for combining techniques from these two research areas in a runtime context, in order to achieve a continual verification of the non-functional properties of these systems [28, 7, 79, 100].

The approaches presented in [18, 38, 52, 53, 88, 117, 121, 164, 165, 168, 118] rely on the definition of simple aggregate QoS functions to derive the QoS properties of a system from the properties of its components. Functions such as sum for response time and throughput [18, 52, 53, 117, 121], product for reliability [88, 117, 121], max, and average are easy to define and manage. However, due to dependencies between different services or between services and resources, these aggregation functions could

lead to quality estimation that represents optimistic (or pessimistic) bounds rather than a realistic estimate. To address this limitation Esfahani *et al.* [68, 69] use fuzzy mathematical methods to model the expected impact of an architectural alternative on system's properties. Fuzzy logic is suitable to support early architectural decisions, but when rough estimates in the early stages give way to precise estimates in the later stages, then it becomes unnecessary. Mosincat *et al.* [129, 130] carry out statistical hypothesis testing using sample based monitoring from runtime observations. Metzger *et al.* [128] discuss various prediction techniques to predict the QoS values, including time series prediction models and online testing for SBS systems. However, these approaches do not use the predicted values to learn and update system model parameters that enable later run-time analysis.

In [19, 20] Bencomo *et al.* present a formal dynamic decision networks model for soft goals (non-functional requirements) and revise the model when empirical evidence becomes available from system deployment. They use a table of conditional probabilities for satisfying the effect of the revision over the non-functional requirements. Letier *et al.* [115] represent uncertainty in early requirements and architecture decisions as a probability. They use Monte Carlo simulation to evaluate the point-based models on many different possible parameter values. In contrast to these approaches, the techniques presented in this chapter use point-based estimate to predict QoS parameters of performance and reliability models. In addition, we analyse the updated models at run-time using probabilistic model checker PRISM [96], to enable continuous verification.

The research addressing the challenges of continual verification has so far focused primarily on reducing the overheads of runtime analysis of formal models [33, 71, 73, 76, 102]. Relatively little effort has been dedicated to ensuring that formal models being verfied are updated in line with the changes in the analysed system. The work in [66] represents an exception in this respect, as it proposes a simple Bayesian technique for the online learning of the state transition probabilities of MC models. The technique is applicable when the analysed system is operational, and its state transitions are monitored. Suppose that, as a result of this monitoring, we observe $N_i > 0$ transitions from state $s_i$ to other states in S, for each $1 \leq i \leq n$. Also, assume that the $k$-th observation of a transition from state $s_i$ to another state, $1 \leq k \leq N_i$, is a transition to state $s_{j_k}$, and let

$$x_{ij}^k = \begin{cases} 1 & \text{if } j = j_k, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Note that $\sum_{j=1}^{n} x_{ij}^k = 1$ since, for each observation $k$, the system will transition from state $s_i$ to precisely one state from $S$. Given this notation, the technique uses theoretical results from [149] and the Bayes' rule to derive the *updating rule* for estimating the probability $p_{ij}$ after the observation of the $k$-th transition from state $s_i$ to another state in $S$ as

$$p_{ij}^k = \frac{c_i^0}{c_i^0 + k} p_{ij}^0 + \frac{k}{c_i^0 + k} \frac{\sum_{l=1}^{k} x_{ij}^l}{k}, \tag{3.2}$$

for $1 \leq k \leq N_i$. The *smoothing parameter* $c_i^0 \geq 1$ quantifies the confidence in the accuracy of the *a priori* estimates $p_{ij}^0$, $1 \leq i, j \leq n$. For a description of the steps involved in deriving this result the reader is referred to [66]. In addition, [28] presents the use of the technique within a framework for dynamic QoS management and optimization in service-based systems.

As shown in Sections 3.2.1-3.3, our enhanced techniques for learning the transition probabilities of discrete-time Markov chains address a key limitation of the solution in [66]. In particular, our adaptive learning is better suited for all scenarios in which

Table 3.1: Approaches to learning model parameters in QoS engineering

| **Authors & citation** | **Model type** | **Param. learnt** | **Technique** | **Analysed QoS req.(s)** | **Domain** |
|---|---|---|---|---|---|
| Ghezzi *et al.* 2009 [66] | MC | transition prob. | Bayesian estimation | Reliability | Service-based systems |
| Ghezzi *et al.* 2009 [86] | QN | service rates | Bayesian estimation | Performance, Workload, Resource allocation | Service providers |
| Calinescu *et al.* 2011 [28] | MC | transition prob. | Bayesian estimation | Reliability, Workload, Resource allocation | Service-based systems |
| Zheng *et al.* 2005–2011 [83, 169, 170] | QN | service rates | Kalman filter | Performance | Web app. deployed in a data centre |
| Section 3.2.1 | MC | transition prob. | Bayesian with observation ageing | Reliability | Service-based systems |
| Section 3.3 | MC | trans. prob. | Adaptive-parameter Bayesian | Reliability | Service-based systems |
| Section 3.4 | MC | trans. rate | Kalman filter blackbox | Performance | Telematics |

the learnt MC transition probabilities undergo multiple-changes[1] over time, which is a frequently encountered situation in real-world systems.

The approach introduced in [169, 170] uses Kalman filter estimators to update the parameters of queueing-network (QN) performance models. The approach adapts the Kalman filter estimator for performance model parameters, to track changes in performance parameters of a web application deployed in a data centre. The tracking filter is based on a layered queuing model of this system, with parameters representing CPU demands and the user load intensity. In Section 3.4 we use the Kalman filter to estimate the costs/reward structures of MC models, and thus to detect performance degradation for individual components of a black-box component-based system. Our approach complements the work in [83, 169, 170], as it monitors the analysed system from a user perspective (i.e., as a black-box component-based system), while [83, 169, 170] require that monitoring the behavior of individual system components is possible. Table 3.1 summarises the characteristics of the techniques described above, and compares them to the new techniques proposed in this chapter.

[1]Not one step change from $p$ to $p^{'} \neq p$ at a point in time

## 3.2 Markov chain transition probability learning with observation ageing

### 3.2.1 Description

The project has developed a new Bayesian derived technique for learning the state transition probabilities of MCs based on observations of the modelled system. Unlike the existing approaches, our technique weighs observations based on their age, to account for the fact that older observations are less relevant than more recent ones.

The updating rule (3.2) was shown in [29, 66] to be effective in scenarios where the actual probability $p_{ij}$ differs from the *a priori* estimate $p_{ij}^0$, but is a constant. However, in many scenarios involving real-world systems, $p_{ij}$ is prone to changing dynamically.

For such scenarios, rule (3.2) is slow to detect requirement violations or, as shown later in Section 3.2.3.3, may even fail to detect them when they are short lived.

Our extended MC parameter learning technique overcomes this limitation by weighting the $k > 0$ observations from the updating rule (3.2) based on their "age". To achieve this, the extended technique timestamps each observation $x_{ij}^k$ from eqs. (3.1) and (3.2) with the time instant $t_k$ when the observation was made.[2] We assume that the updating rule (3.2) is applied as soon as the $k$-th observation is made, i.e., at time moment $t_k$. Therefore, when the updating rule is applied, the age of a generic observation $l$, $1 \leq l \leq k$, is precisely

$$age_l = t_k - t_l. \tag{3.3}$$

To reflect the decreasing importance of observations as they become older in a dynamic scenario, we associate a weight

$$w_i^l = \alpha_i^{-age_l} = \frac{1}{\alpha^{t_k - t_l}} \tag{3.4}$$

with each observation $l$, $1 \leq l \leq k$, where $\alpha \geq 1$ represents the observation *ageing coefficient* (Figure 3.1). As shown later in this section, the choice of a negative exponential function as the ageing function is motivated by the ease with which it allows the application of the new updating rule (i.e., with O(1) time and memory complexity, cf. Section 3.2.2).

The extended updating rule is then obtained by multiplying each $x_{ij}^l$ term from (3.2) by its associated weight (3.4):

$$p_{ij}^k = \frac{c_i^0}{c_i^0 + k} p_{ij}^0 + \frac{k}{c_i^0 + k} \frac{\sum_{l=1}^k w_i^l x_{ij}^l}{\sum_{l=1}^k w_i^l}, \tag{3.5}$$

for $1 \leq k \leq N_i$. Note that the denominator for the last part of this updating rule was adjusted to $\sum_{l=1}^k w_l$ in order to satisfy the invariant $\sum_{j=1}^n p_{ij}^k = 1$:

[2] Recording this additional information typically requires only a small extension to the monitoring part of a system.

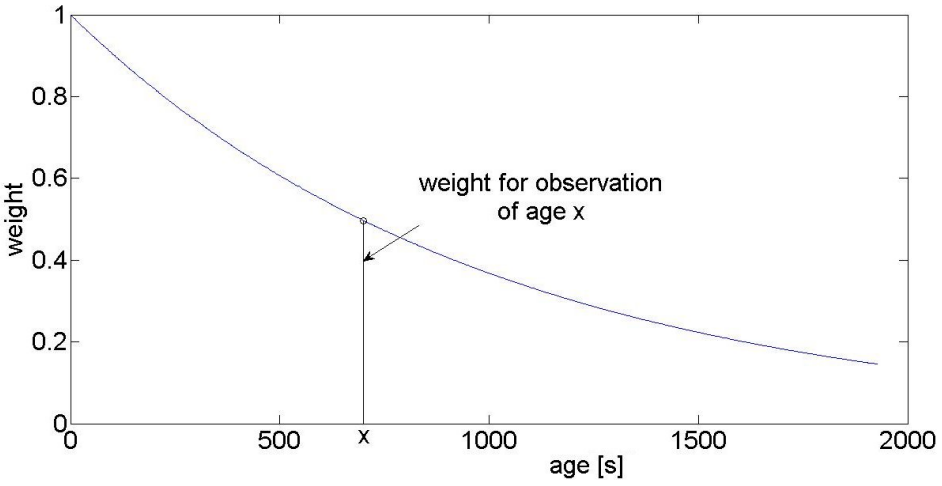


Figure 3.1: The observation ageing function $\alpha^{-age}$, shown for $\alpha = 1.001$

$$\begin{aligned}\sum_{j=1}^{n} p_{ij}^{k} =& \frac{c_i^0}{c_i^0+k}\sum_{j=1}^{n} p_{ij}^{0} + \frac{k}{c_i^0+k}\sum_{j=1}^{n}\frac{\sum_{l=1}^{k} w_l x_{ij}^{l}}{\sum_{l=1}^{k} w_l} = \\ =& \frac{c_i^0}{c_i^0+k}\times 1 + \frac{k}{c_i^0+k}\frac{\sum_{l=1}^{k}\left(w_l \sum_{j=1}^{n} x_{ij}^{l}\right)}{\sum_{l=1}^{k} w_l} = \\ =& \frac{c_i^0}{c_i^0+k} + \frac{k}{c_i^0+k}\frac{\sum_{l=1}^{k}(w_l\times 1)}{\sum_{l=1}^{k} w_l} = \frac{c_i^0}{c_i^0+k} + \frac{k}{c_i^0+k} = 1.\end{aligned}$$

Finally, note that selecting an ageing coefficient $\alpha = 1$ makes all weights $w_l = 1$, thus reducing the extended updating rule (3.5) to the base updating rule in (3.2). This property represents another advantage of using the ageing function in Figure 3.1.

### 3.2.2 Analysis of the learning algorithm

The pseudocode for the algorithm is shown in Algorithm 1. In devising this pseudocode, we used the notation $f_{ij}^{k} = \sum_{l=1}^{k} w_l x_{ij}^{l}$ and $g_{ij}^{k} = \sum_{l=1}^{k} w_l$, and the observation that:

$$\begin{aligned}f_{ij}^{k} =& \sum_{l=1}^{k} w_l x_{ij}^{l} = \sum_{l=1}^{k}\frac{x_{ij}^{l}}{\alpha^{t_k-t_l}} = \frac{x_{ij}^{k}}{\alpha^{t_k-t_k}} + \sum_{l=1}^{k-1}\frac{x_{ij}^{l}}{\alpha^{t_k-t_l}} = \\ =& x_{ij}^{k} + \sum_{l=1}^{k-1}\frac{x_{ij}^{l}}{\alpha^{t_k-t_{k-1}}\alpha^{t_{k-1}-t_l}} = x_{ij}^{k} + \frac{f_{ij}^{k-1}}{\alpha^{t_k-t_{k-1}}}\end{aligned}$$

and, following a similar proof,

$$g_{ij}^{k} = 1 + \frac{g_{ij}^{k-1}}{\alpha^{t_k-t_{k-1}}}.$$

As mentioned earlier, the choice of an exponential negative ageing function simplifies the application of the updating rule (3.5). Thus, the first term (i.e., $\frac{c_i^0}{c_i^0+k}p_{ij}^0$) and the multiplicative factor $\frac{k}{c_i^0+k}$ can both be calculated in constant, $O(1)$ time. As a result, the part $\frac{f_{ij}^k}{g_{ij}^k} = \frac{\sum_{l=1}^{k} w_l x_{ij}^{l}}{\sum_{l=1}^{k} w_l}$ from rule (3.5) can also be calculated in $O(1)$ time, by using the recursive definitions above for $k \geq 2$, and $f_{ij}^{1} = x_{ij}^{1}$ and $g_{ij}^{1} = 1$ for $k = 1$. Furthermore,

**Algorithm 1** Bayesian learning algorithm with observation ageing

1: $k \leftarrow 0$

2: **function** UPDATE($x_{ij}^k$, $t_{ij}^k$)
3: $k \leftarrow k + 1$
4: **if** $k = 1$ **then**
5: $f_{ij}^k \leftarrow x_{ij}^k$
6: $g_{ij}^k \leftarrow 1$
7: **else**
8: $f_{ij}^k \leftarrow f_{ij}^k + x_{ij}^k$
9: $g_{ij}^k \leftarrow g_{ij}^k + 1$
10: **end if**
11: $p_{ij}^k$ = calculate new estimate using Equation (3.5)
12: **return** $p_{ij}^k$
13: **end function**

calculating the right-hand side of (3.5) at step $k$ does not require the technique to maintain a record of all $k$ observations (i.e., of $x_{ij}^l$ and $t_l$ for all $1 \leq l \leq k$). The only observation-related values required to carry out the calculation at step $k$ are $t_{k-1}$, $f_{ij}^{k-1}$, $g_{ij}^{k-1}$, $x_{ij}^k$ and $t_k$. Thus, the memory complexity of the learning algorithm is also $O(1)$.

### 3.2.3 Case study

#### 3.2.3.1 Bioinformatic Workflow

To illustrate the process of updating the QoS parameters of a MC model, we carried out a case study in which the approach presented in Section 3.3.1 was applied to a real-world bioinformatic workflow taken from the Taverna repository of scientific workflows myExperiment [3] (Figure 3.2). Taverna [98] is a workflow engine widely used by research communities from bioinformatics, astronomy and social sciences. The workflow in Figure 3.2 has been used in studies of an autoimmune disease that represents the most common cause of hyperthyroidism in young people and children (i.e., the *Graves disease*). This workflow was chosen for our case study because it invokes 18 different web services running at four research centres in the UK and Japan; it is also one of the most complex and most used workflows from the repository[4].

#### 3.2.3.2 DTMC Model

A fragment of the PRISM model for this workflow is shown in Figure 3.3, where $p1$ to $p18$ represent *a priori* estimates of the probabilities that the 18 web service invocations complete successfully, and a PRISM module is used to model each web service. An

[3] http://www.myexperiment.org

[4] It had a download count of 2484 when last checked in May 2015.

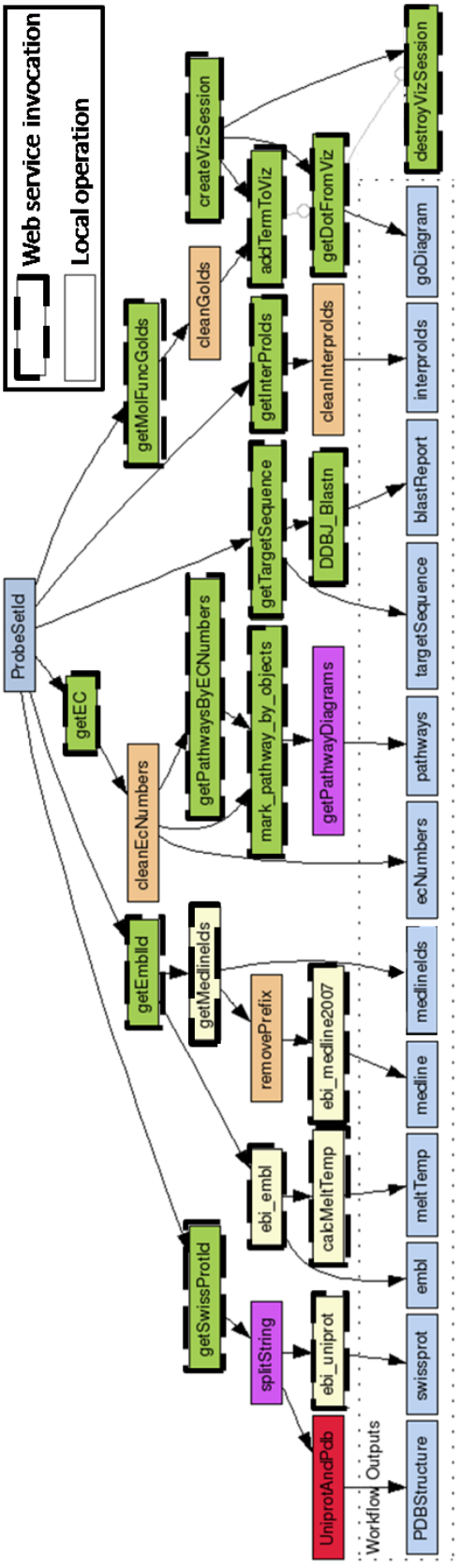


Figure 3.2: Bioinformatics workflow taken from `http://www.myexperiment.org/workflows/28.html`

additional module (Workflow) is used to model the workflow as a whole, thus enabling the specification of a reliability QoS requirement such as "the workflow must complete

```
dtmc
//This is a PRISM model of a Taverna workflow used in studies for Graves diesease.
//The workflow can be found at: http://www.myexpriment.org/workflows/28.html
//constants p1 to p18 represent (a priori estimates of) the probabilities that the 18
//web service invocations complete successfully, and a PRISM module is used to model each
//web service. An additional module "Workflow" is used to model the workflow as a whole.

const double p1=0.999;
...
const double p18=0.99; //calcMeltTemp
const int SUCC=1;
const int FAIL=2;
-------------------------------------------------------------------------------------------
module SgetEC //Service 1
     getEC : [0..2] init 0;
  [] getEC=0 -> p1:(getEC'=SUCC) + (1-p1):(getEC'=FAIL);
endmodule
-------------------------------------------------------------------------------------------
...
-------------------------------------------------------------------------------------------
module ScalcMeltTemp //Service 18
     calcMeltTemp : [0..2] init 0;
  [] calcMeltTemp=0 -> p18:(calcMeltTemp'=SUCC)+(1-p18):(calcMeltTemp'=FAIL);
  [] (calcMeltTemp=0)&(ebi_embl=FAIL) -> 1:(calcMeltTemp'=FAIL);
endmodule
-------------------------------------------------------------------------------------------
module WorkFlow
     wf : [0..2] init 0; // 0 - init; 1 - success; 2 - fail
  [] (wf=0)&((ebi_uniprot=FAIL)| (calcMeltTemp=FAIL)| (ebi_medline2007=FAIL)
        | (markPathwayByObjects=FAIL)| (DDBJBlastn=FAIL)| (getInterProIds=FAIL)|
        (getDotFromViz=FAIL)) -> 1 : (wf'=FAIL);
  [] (wf=0)&((ebi_uniprot=SUCC)&(calcMeltTemp=SUCC)&(ebi_medline2007=SUCC)
        &(markPathwayByObjects=SUCC)&(DDBJBlastn=SUCC)&(getInterProIds=SUCC)
        &(getDotFromViz=SUCC)) -> 1:(wf'=SUCC);
  [] wf=SUCC -> 1:(wf'=SUCC);
  [] wf=FAIL -> 1:(wf'=FAIL);
endmodule
-------------------------------------------------------------------------------------------
```

Figure 3.3: A fragment of the PRISM model for the workflow in Figure 3.2.

successfully with a probability of at least 0.95" as the PCTL reachability property:

$$\text{“}init\text{”} \Rightarrow P_{\geq 0.95}[\text{F } wf = \text{ SUCC}] \tag{3.6}$$

The complete PRISM model of the workflow is available in Appendix A.

#### 3.2.3.3 Experiments and results

The MC parameter learning algorithm in Section 3.2.1 was validated through the simulation of a wide range of scenarios for the bioinformatics workflow from Figure 3.2.

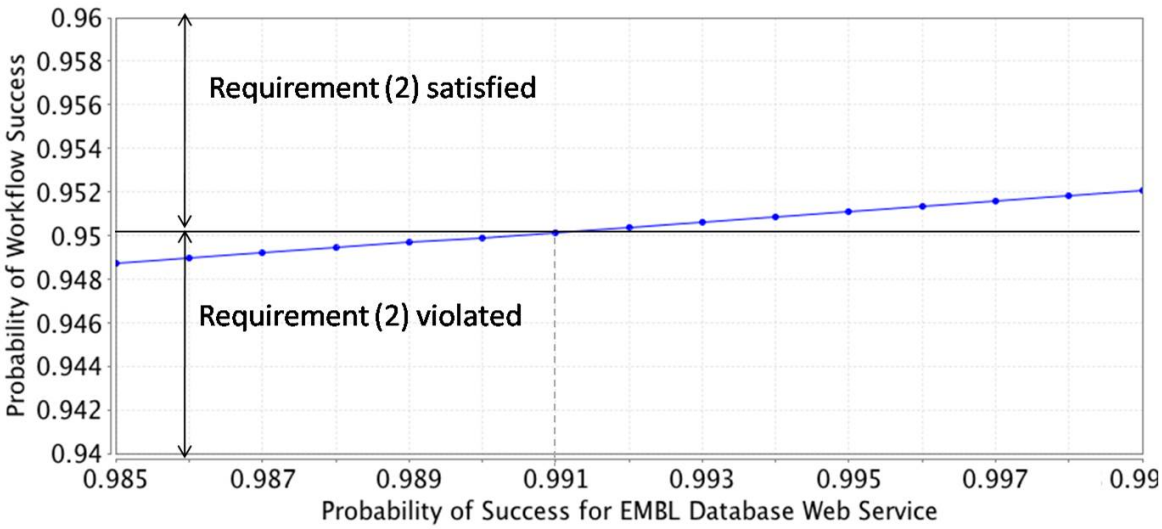


Figure 3.4: PRISM analysis of the workflow compliance with requirement (3.6)

This section presents a subset of these scenarios that involve learning the probability of a successful invocation of the web service **ebi_embl** from the workflow, based on an initial estimate and on observations obtained through monitoring the service. This service was selected because its execution involves complex bioinformatics database operations that expose its invocation to variations in performance and reliability.

Assuming that the probabilities of all other service invocations are fixed, the probabilistic model checker PRISM was used to analyse the impact of changes in the success probability of service **ebi_embl** on the compliance of the workflow with requirement (3.6). Figure 3.4 depicts the result of this analysis, showing that the workflow satisfies the requirement if and only if service **ebi_embl** has a probability of success of at least $pRequired = 0.991$.

Figure 3.5 presents the experimental results for three scenarios involving dynamic changes to the actual probability of success $pActual$ for the simulated web service over 30,000s of simulated time. Each scenario corresponds to a different $pActual$ change pattern, and the experimental results were obtained through averaging the results of 100 Matlab simulations of each of the three patterns. The observation timestamps $t_k, 1 \leq k \leq N_i$, were selected from a Poisson distribution with a mean inter-arrival time of $1/\lambda = 1s$, and the observations $x_{ij}^k, 1 \leq k \leq N_i$, were taken from a Bernoulli distribution with parameter $pActual$. The smoothing parameter was fixed at $c_i^0 = 50$, and the *a priori* probability was set to 0.991, i.e., the minimum probability for which requirement (3.6) is satisfied. Simulations were carried out for a large number of values of the ageing coefficient $\alpha$, though, in the interest of readability, the diagrams in Figure 3.5 show the results for only two of these values, i.e., $\alpha = 1$ (which corresponds to the base learning algorithm) and $\alpha = 1.001$.

In each scenario from Figure 3.5, the probability of success for the simulated service starts and ends at a value $pActual = 0.991$ that ensure compliance with requirement (3.6). However, this probability drops to a value $pActual = 0.99$ that violates the requirement during a time interval whose bounds $a, a^{'}, b$ and $b^{'}$ marked on each scenario denote time intervals when the estimated probability of successful service invocation does not sufficiently reflect the actual probability $pActual$. This leads to an erroneous verification result of requirement (3.6). Thus, $a$ and $a^{'}$ are the time intervals during which the requirement violation is undetected for $\alpha = 1.001$ and $\alpha = 1$, respectively (false negative); and $b$ and $b^{'}$ represent time intervals during which the requirement is satisfied but wrongly classified as violated for $\alpha = 1.001$ and $\alpha = 1$, respectively (false positive).

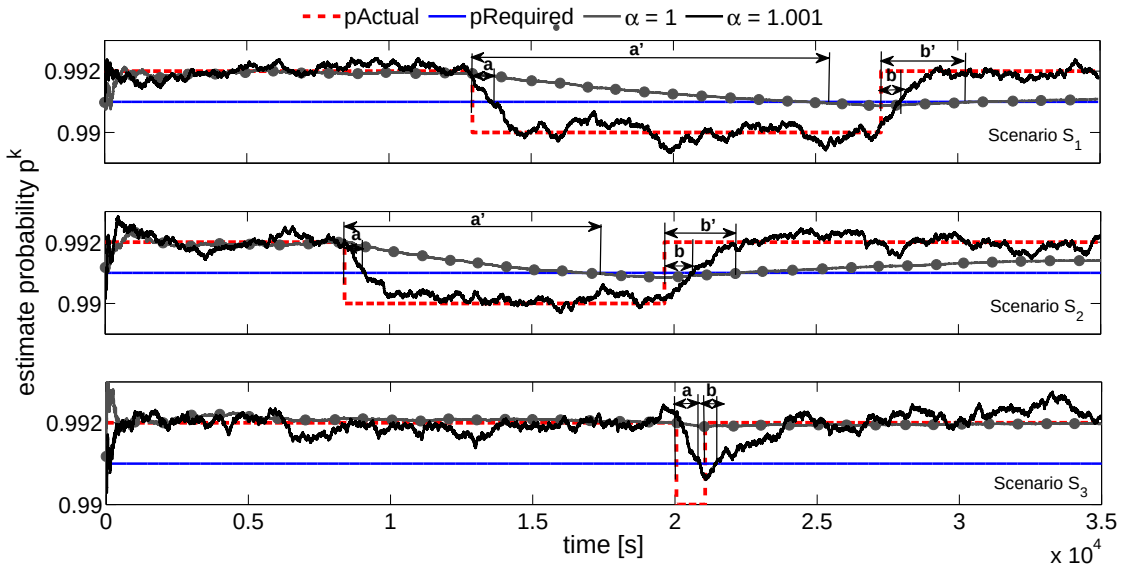


Figure 3.5: Experimental results contrasting the effectiveness of the observation-ageing and base learning techniques in scenarios when the actual probability of success *pActual* changes dynamically

Table 3.2: Quantitative analysis of the experiments in Scenarios 1–3

| Scenario | $\alpha = 1.001$ | | $\alpha = 1$ | |
|---|---|---|---|---|
| $(S_X)$ | $a$ [s] | $b$ [s] | $a'$ [s] | $b'$ [s] |
| $S_1$ | 680 | 700 | 12990 | 3130 |
| $S_2$ | 699 | 1050 | 8694 | 2890 |
| $S_3$ | 840 | 430 | – | – |

A comparison of the lengths of time intervals $a$ and $a^{'}$, and of $b$ and $b^{'}$ summarised in Table 3.2 shows that our learning method (which corresponds to $\alpha = 1.001$) provides a more precise estimation of *pActual* than the base Bayesian learning algorithm. In particular, Scenario 3 shows that our method provides a good estimate even for a short degradation in the probability of success for the simulated service; in contrast, the base method is unable to detect this degradation.

Finally, the results from Scenarios 1 and 2 show that the effectiveness of our method does not depend on the timing of the changes in the value of *pActual*. Thus, the time intervals $a$ for the two scenarios are of similar length, and so are the time intervals $b$. In contrast, when the base method is used, the length of the time interval $a^{'}$ is much longer for scenario 1, in which *pActual* had the same value (i.e., 0.992) for a longer period of time and the length of $b^{'}$ is much longer in scenario 2, in which *pActual* maintained a value of 0.99 for an extended period of time.

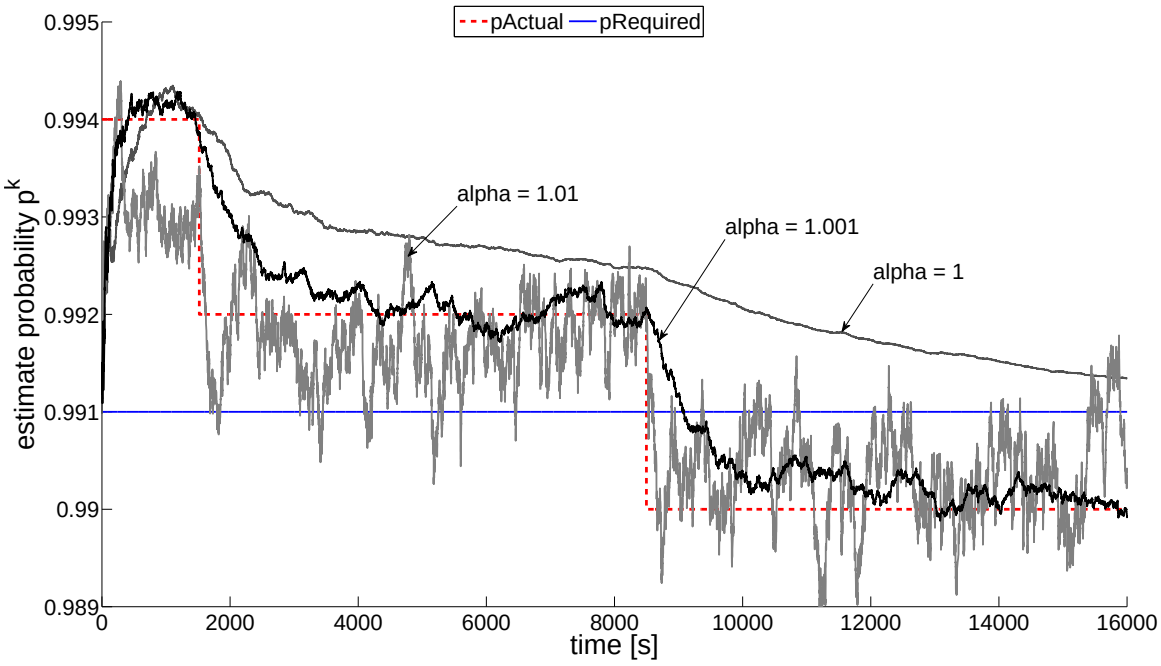


Figure 3.6: Effect of different choices for the ageing coefficient $\alpha$

### 3.2.4 A rule of thumb for choosing the ageing coefficient

The value of the ageing coefficient $\alpha$ determines the age range of the observations that contribute effectively to the most significant $d < 0$ digits of the probability estimate $p_{ij}^k$ in (3.5). If too few of the observations $x_{ij}^l, 1 \leq l \leq k$, are within this age range, then the probability estimate $p_{ij}^k$ will depend too heavily on the very recent past and will oscillate as shown in Figure 3.6 for $\alpha = 1.01$.

To avoid this undesirable effect, more of the observations $x_{ij}^l$ must contribute to the $d$ most significant digits of $p_{ij}^k$, i.e., have a coefficient larger than $10^{-d}$ in (3.5):

$$\frac{k}{c_i^0 + k} \frac{w_l}{\sum_{l=1}^k w_l} \geq 10^{-d} \tag{3.7}$$

Assuming that the mean distance between successive observations is $\mu > 0$, then, for large values of $k$, $\frac{k}{c_i^0+k} \approx 1$ and $\sum_{l=1}^k w_l = \sum_{l=1}^k \frac{1}{\alpha^{t_k - t_l}} \approx \sum_{l=1}^k \frac{1}{\alpha^{(k-l)\mu}} = 1 + \frac{1}{\alpha^\mu} + \frac{1}{\alpha^{2\mu}} + \cdots + \frac{1}{\alpha^{(k-1)\mu}} = (1 - \frac{1}{\alpha^{k\mu}})/(1 - \frac{1}{\alpha^\mu}) \approx 1/(1 - \frac{1}{\alpha^\mu}) = \frac{\alpha^\mu}{\alpha^\mu - 1}$, so (3.7) becomes $w_l / \left(\frac{\alpha^\mu}{\alpha^\mu - 1}\right) \approx \frac{\alpha^\mu - 1}{\alpha^{(k-l)\mu}\alpha^\mu} \geq 10^{-d}$. The observations that satisfy this inequality contribution to the $d$ most significant digits of the probability estimate $p_{ij}^k$. To ensure that at least $m > 0$ observations are taken into account, the inequality must be satisfied for $k - l = m$ (Figure 3.7). The following section describes a more rigorous approach that uses Chebyshevs Inequality to select a suitable value for $\alpha$ dynamically, depending on the frequency of the observations.

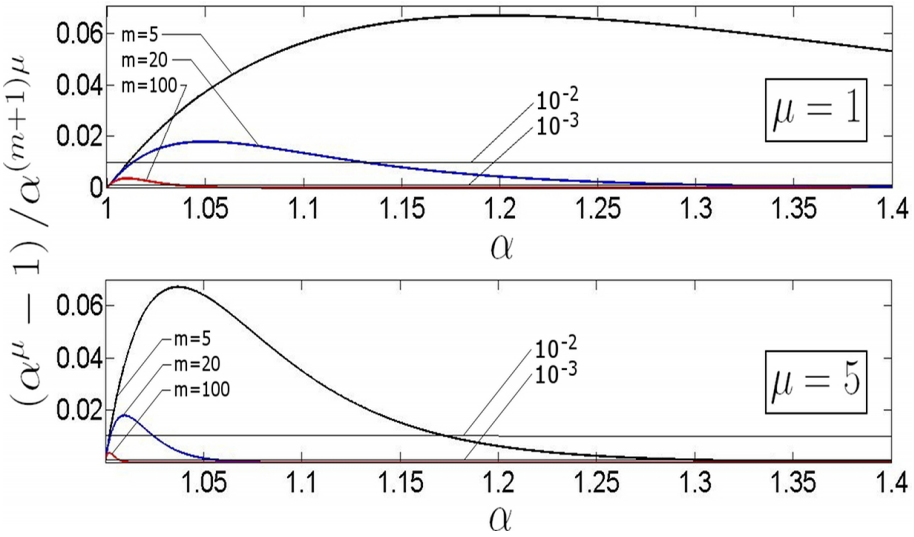


Figure 3.7: Choosing the ageing coefficient: $\alpha$ values for which $(\alpha^{\mu}-1)/\alpha^{(m+1)\mu} \geq 10^{-d}$ ensure that the $m$ most recent observations contribute to the $d$ most significant digits of the probability estimate.

## 3.3 Adaptive Markov chain transition-probability learning

### 3.3.1 Description

The transition-probability learning algorithm (3.4) and (3.5) depends on the choice of the smoothing parameter $c_i^0$ and the ageing parameter $\alpha_i$. However, no combination of values for these parameters is suitable for all scenarios. This section describes the technique developed by the project to address this limitation, through extending the learning algorithm in Section 3.2.1 with the ability to select suitable parameters $c_i^0$ and $\alpha_i$ at runtime. The dynamic selection of these parameters adapts the learning algorithm to the frequency of the observations, and is based on the following theoretical results.

**PROPOSITION 3.1.** Let $x_1, x_2, \cdots, x_k$ be an independent trials process with expected value $E(x_l)-\mu$ and variance $V(x_l) = \sigma^2$, for $l = 1, 2, \cdots, k$. Let $w_1, w_2, \cdots, w_k > 0$ be a set of weights, and $A_k = \frac{\sum_{l=1}^{k} w_l x_l}{\sum_{l=1}^{k} w_l}$ be the weighted average of $x_1, x_2, \cdots, x_k$. Then

$$E(A_k) = \mu \text{ and } V(A_k) = \frac{\sum_{l=1}^{k} (w_l)^2}{\left(\sum_{l=1}^{k} w_l\right)^2}\sigma^2$$

*Proof.* The expected value $E(A_k)$ can be calculated as

$$E\left(\frac{\sum_{l=1}^{k} w_l x_l}{\sum_{l=1}^{k} w_l}\right) = \frac{E\left(\sum_{l=1}^{k} w_l x_l\right)}{\sum_{l=1}^{k} w_l} =$$

$$\text{(since } \sum_{l=1}^{k} w_l \text{ is a constant, cf. Theorem 6.2 [89])}$$

$$= \frac{\sum_{l=1}^{k} E(w_l x_l)}{\sum_{l=1}^{k} wl} =$$

(since $w_1x_1, w_2x_1, \cdots, w_kx_k$ are random variables with finite expected values, cf. Theorem 6.2 [89])

$$= \frac{\sum_{l=1}^{k} w_l E(x_l)}{\sum_{l=1}^{k} w_l} = \frac{\sum_{l=1}^{k} w_l \mu}{\sum_{l=1}^{k} w_l} = \mu$$

(since $w_l$ is a constant, cf. Theorem 6.2 [89])

In a similar way, the variance $V(A_k)$ is given by

$$V(\frac{\sum_{l=1}^{k} w_l x_l}{\sum_{l=1}^{k}} w_l) = \frac{v\left(\sum_{l=1}^{k} w_l x_l\right)}{\left(\sum_{l=1}^{k} w_l\right)^2}$$

(since $\sum_{l=1}^{k} w_l$ is a constant, cf. Theorem 6.7 [89])

$$= \frac{\sum_{l=1}^{k} V(w_l x_l)}{\left(\sum_{l=1}^{k} w_l\right)^2} =$$

(since $w_1x_1, w_2x_1, \cdots, w_kx_k$ are independent random variables with finite expected values, cf. Theorem 6.8 [89]) □

**COROLLARY 3.2.** Consider again the independent trials process $x_1, x_2, \cdots, x_k$ from Proposition 3.1, and let $\epsilon > 0$. Then

$$P\left(\left|\frac{\sum_{l=1}^{k} w_l x_l}{\sum_{l=1}^{k} w_l} - \mu\right| \geq \epsilon\right) \leq \frac{\sum_{l=1}^{k} (w_l)^2}{\left(\sum_{l=1}^{k} w_l\right)^2 \epsilon^2} \sigma^2 \tag{3.8}$$

*Proof.* The result is a direct application of Chebyshev's Inequality (e.g., Theorem 8.1 [89]) to the discrete random variable $A_k$ with the expected value and variance from (**??**). □

**PROPOSITION 3.3.** Consider the transition-probability learning algorithm (3.4)–(3.5), and let $\epsilon > 0$. Then

$$P\left(\left|\frac{\sum_{l=1}^{k} w_i^l x_{ij}^l}{\sum_{l=1}^{k} w_i^l} - p_{ij}\right| \geq \epsilon\right) \leq \frac{\sum_{l=1}^{k} (w_l)^2}{4\left(\sum_{l=1}^{k} w_l\right)^2 \epsilon^2}, \tag{3.9}$$

where $p_{ij}$ represents the actual transition probability between states $s_i$ and $s_j$ of the model $\mathcal{M}$.

*Proof.* Since the actual transition probability between states $s_i$ and $s_j$ is $p_{ij}$, $x_{ij}^l \in \{0,1\}$, $1 \leq l \leq k$, are discrete random variables with (a) distribution function $P(1) = p_{ij}$ and $P(0) = 1 - p_{ij}$; (b) expected value $\mu = E(x_{ij}^l = 1xp_{ij} + x(1-p_{ij}) = p_{ij}$; and (c) variance $\sigma^2 = V(x_{ij}^l = E\left((x_{ij}^l)^2\right) - \left(E(x_{ij}^l)\right)^2 = (1^2xp_{ij} + 0^2x(1-p_{ij})) - (p_{ij})^2 = p_{ij} - (p_{ij})^2$. The inequality (3.9) is now easy to obtain by replacing these $\mu$ and $\sigma^2$ values in (3.8), and noting that $\sigma^2 = p_{ij} - (p_{ij})^2 \leq \frac{1}{4}$ for all possible values of $p_{ij}$. □

### 3.3.2 Dynamic selection of learning algorithm parameters

To take advantage of the results from Proposition 3.3, we consider a time interval during which the mean distance between successive observations is $\bar{t} > 0$. Accordingly, $w_i^l = \alpha_i^{-(t_k - t_l)} \approx \alpha_i^{-(k-l)^{\bar{t}}}$ and, after straightforward algebraic manipulations,

$$\frac{\sum_{l=1}^k (w_l)^2}{\left(\sum_{l=1}^k w_l\right)^2} \approx \frac{\sum_{l=1}^k \alpha_i^{-2(k-l)^{\bar{t}}}}{\left(\sum_{l=1}^k \alpha_i^{-(k-l)\bar{t}}\right)^2} = \frac{(\alpha_i^{k\bar{t}} + 1)(\alpha_i^{\bar{t}} - 1)}{(\alpha_i^{k\bar{t}} - 1)(\alpha_i^{\bar{t}} + 1)} \approx \frac{\alpha_i^{\bar{t}} - 1}{\alpha_i^{\bar{t}} + 1},$$

if $\alpha^{k\bar{t}} \gg 1$. Replacing this result in (3.3) we obtain:

$$P\left(\left|\frac{\sum_{l=1}^k w_i^l x_{ij}^l}{\sum_{l=1}^k w_i^l} - p_{ij}\right| \geq \epsilon\right) \leq \frac{1}{4\epsilon^2}\frac{\alpha_i^{\bar{t}} - 1}{\alpha_i^{\bar{t}} + 1}, \text{if } \alpha_i^{k\bar{t}} \gg 1. \tag{3.10}$$

Our adaptive transition-probability learning algorithm uses the result in (3.10) to adjust the smoothing parameter $c_i^0$ and the ageing parameter $\alpha_i$ from (3.4) and (3.5) dynamically, based on the mean distance between recent observations $\bar{t}$ as follows:

1. Given a small $\epsilon$, we select $\alpha_i$ such that the probability from (3.10) is below a small value $p_{max}$, i.e.,

$$\frac{1}{4\epsilon^2}\frac{\alpha_i^{\bar{t}} - 1}{\alpha_i^{\bar{t}} + 1} \leq p_{max} \implies \alpha_i \leq \left(\frac{1 + 4\epsilon^2 p_{max}}{1 - 4\epsilon^2 p_{max}}\right). \tag{3.11}$$

2. Having selected the $\alpha_i, c_i^0$ is chosen such that $\alpha_i^{c_i^{0\bar{t}}} \gg 1$. Since the first term of (3.5) dominates the calculation of $p_{ij}^k$ until the number of observations $k$ is larger than $c_i^0$, this ensures that the $k$ observations play a major role in the $p_{ij}^k$ estimate only once $\alpha_i^{k\bar{t}} \gg 1$ as well. in practice, we use $\alpha_i^{c_i^{0\bar{t}}} = 10$, or

$$c_i^0 = \frac{1}{\bar{t} log_{10} \alpha} \tag{3.12}$$

### 3.3.3 Complexity analysis

Our adaptive learning method requires the calculation of the ageing parameter $\alpha_i$ from (3.11), smoothing parameter $c_i^0$ from (3.12), weights $w_i^l$ from (3.4) and probability estimates $p_{ij}^k$ from 3.5) after each observation. As we show in Section 3.2.1, algebraic manipulation can be used to rearrange (3.4)-(3.5) so that the last two calculations can be performed in O(1) time and using constant, O(1) space. Calculating the mean distance $\bar{t}$ between recent observations - used to compute $\alpha_i$ in (3.11) - requires the algorithm to

Table 3.3: Learning methods compared in the evaluation experiments

| Method | Description |
|---|---|
| Method 1 | basic Bayesian learning from Section 3.1, obtained by setting $w_i^l = 1$ in (Section 3.2.1) for all $1 \leq l \leq k$, and using the smoothing parameter $c_i^0 = 500$. |
| Method 2 | fixed-parameter, ageing-enabled learning algorithm from Section 3.2.1, obtained by setting $c_i^0 = 500$ and $\alpha = 1.001$ in (3.5)–(3.4). |
| Method 3 | fixed-parameter, ageing-enabled learning algorithm from Section 3.2.1, obtained by setting $c_i^0 = 500$ and $\alpha = 1.01$ in (3.5)–(3.4). |
| Method 4 | our new adaptive learning algorithm with smoothing parameter $c_0$ and ageing parameter $\alpha$ given by (3.11)–(3.12) for $p_{max} = \epsilon = 0.05$. |

store the timestamps of all observation. The number of such timestamps is proportional to the frequency $f$ of observations, so the space complexity of this calculation is $O(f)$. The actual calculation of $\bar{t}$, however, can be carried out in $O(1)$ time using a running sum, and computing $\alpha_i$ and $c_i^0$ also takes constant time. Accordingly, the overall space complexity of the adaptive learning is $O(f)$, and its time complexity is $O(1)$.

### 3.3.4 Evaluation

To evaluate the effectiveness of the adaptive learning method, we carried out a broad range of experiments in which we compared its results with those produced by existing learning methods. The existing methods selected for this comparison were the Bayesian learning method from Section 3.1, and the fixed-parameter, ageing-enabled learning method from Section 3.2.1. The concrete methods compared in these experiments and their parameters are summarised in Table 3.3.

Figures 3.8–3.11 depict the experimental results of four scenarios in which we assessed the effectiveness of the adaptive learning method. The four scenarios involved learning the probability $p$ of tossing heads with a biased coin from observations of coin tosses, when $p$ changes over time between a "normal" value of $p = 0.96$ and a lower value. The aim of these scenarios was to simulate a degradation in the reliability with which a system component completed a given task within a predefined amount of time, and to test the ability of the four learning methods to identify this degradation. The four scenarios considered different types of reliability degradation—a longer (i.e., 8000-second) and more significant (i.e., down to $p = 0.87$) one in Scenario 1, a shorter (1200-second) and less significant (down to $p = 0.9$) one in Scenario 2. In Scenario 3 we considered shorter (1200-second) reliability degradation (down to $p = 0.9$) in the first half of the experiment and a longer (5400-second) in the latter half of the experiment, and vice-versa in Scenario 4. Finally, learning each type of reliability degradation was attempted for two different observation frequencies. Thus, observation "inter-arrival" time was exponentially distributed, with a mean of 100ms (or a mean frequency of $10\text{s}^{-1}$) during the first half of the experiments, and a mean of 500ms (i.e., a mean frequency of $2\text{s}^{-1}$) during the second half of the experiments. A qualitative analysis of the experimental results in Figures 3.8–3.11 shows that the adaptive learning algorithm (Method 4) outperforms the existing learning algorithms (Methods 1–3) as follows:

- At the beginning of the experiment, the $p^k$ estimate probability for the adaptive

algorithm approaches $p$ faster than for the basic algorithm in Method 1 and the two combinations of fixed-parameter ageing-enabled algorithms in Methods 2–3.

- During the "high frequency" half of the experiments, the adaptive algorithm is as good at detecting the decrease in the value of $p$ as the "high $\alpha$" algorithm in Method 3 (but with a $p^k$ estimate that oscillates less around the actual $p$), and far better than the "low $\alpha$" algorithm in Method 2 and the basic algorithm in Method 1;

- During the "low frequency" half of the experiments, the adaptive algorithm produces estimates that are as accurate and as smooth as the "low $\alpha$" algorithm (Method 2), and much smoother than the "high $\alpha$" algorithm (Method 3).

Although some of the estimates produced by the "high alpha" algorithm in Method 3 during the decrease in the value of $p$ in the second half of the experiment are closer to $p$ than the estimates produced by the adaptive algorithm, this is achieved at the expense of significant oscillation. Such oscillation is likely to trigger false alarms in a real-world scenario. If this is not a problem, then the adaptive algorithm can be configured to provide similar estimates by adjusting its confidence interval through increasing $\epsilon$ and/or $p_{max}$.

For a quantitative evaluation of the effectiveness of our adaptive learning method, consider a situation in which an alarm is triggered if the estimate probability $p^k$ (representing the reliability of a system component, as explained above) drops below a threshold value $p^{required} = 0.95$. This threshold value is shown as a dotted line in all graphs in Figures 3.8–3.9. Assuming that the learning methods are used to detect such violations of a reliability threshold, we measured the following three non-functional properties of the learnt $p^k$ values from Scenarios 1 and 4:

- The time $t_{down}$ elapsed between the drop in the value of $p$ and the moment when the estimate $p^k$ becomes smaller than $p^{required}$.

- The time $t_{up}$ elapsed between the moment when $p$ regains its "normal" value (i.e., $p = 0.96$) after a period of degraded reliability, and the moment when the estimate $p^k$ becomes at least $p^{required}$.

- The number of false positives $n_+$, i.e., instances when $p^k$ drops below $p^{required}$ although $p$ has its normal value.

Table 3.4 shows the value of these properties, separately for the periods of high-frequency and low-frequency observations from the experiments. These results indicate that Method 1 is mostly suited for identifying only the first change in the probability $p$, whereas the other methods yield $p^k$ probability estimates that follow the changes in $p$ with more or less accuracy. The adaptive learning algorithm (Method 4) detects the changes in the value of $p$ faster than Method 2 in the high-frequency observation area, and, like this method, produces no false positives. In the low-frequency observation area, the two methods are comparable, while Method 3 achieves slightly lower $t_{down}$ and $t_{up}$ but has the significant disadvantage of generating tens of false positives. As mentioned before, if these false positives are deemed acceptable, then Method 4 can achieve similar results by choosing larger $p_{max}$ and $\epsilon$ values than those in Table 3.3.

The last set of experiments are carried out for the scenario illustrated in Figure 3.12. In this scenario, we assume that $p$ represents the probability that a system will perform an operation or task over another (or over remaining idle), and we suppose that $p$ varies over a 10-hour time period (e.g., between 8am and 6pm during a working day) as shown by the thick dashed line. Our experiments assessed to what extend the estimate probability $p^k$ provided by each of the four learning remained with the interval

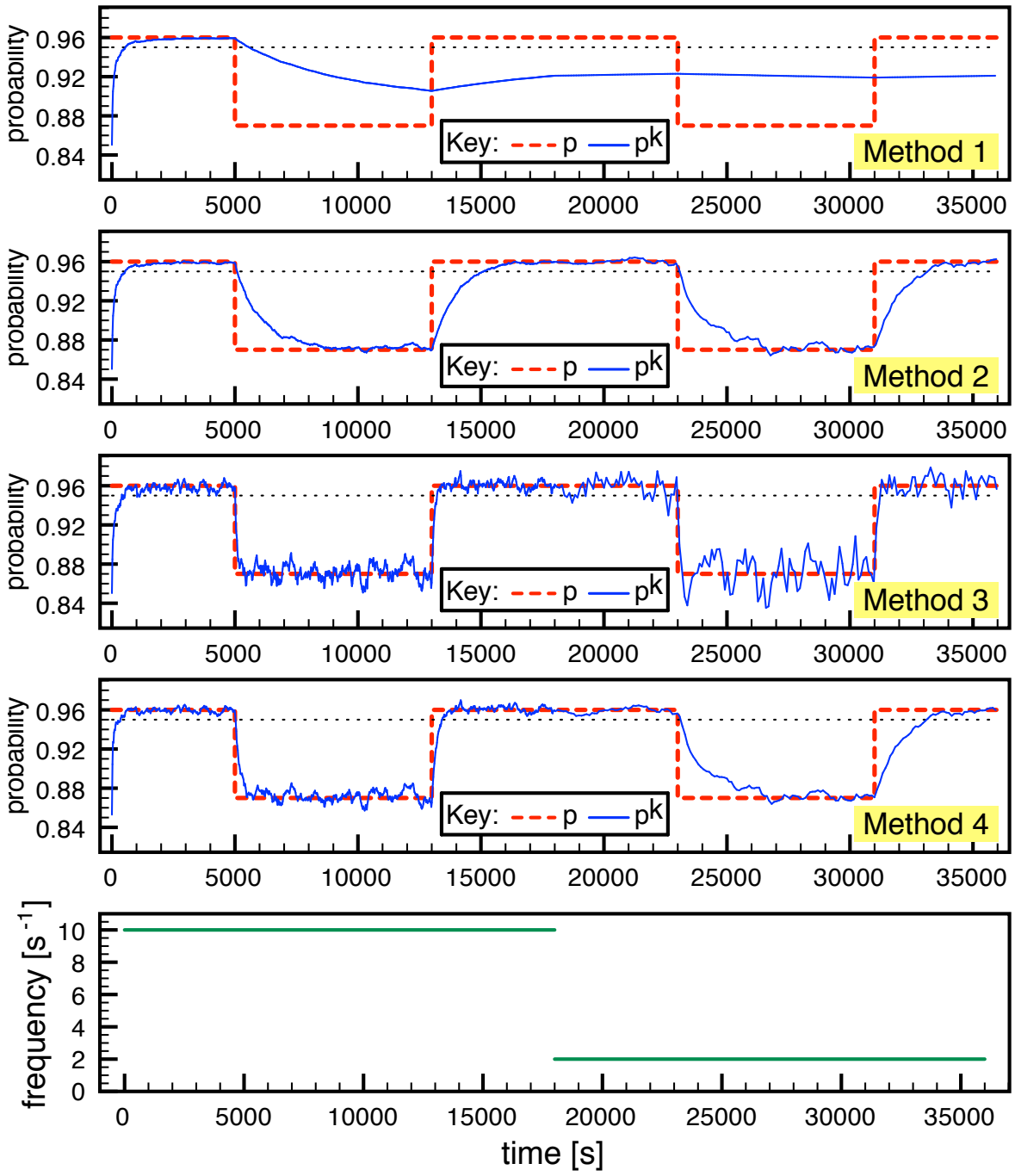


Figure 3.8: Experimental results—scenario 1

$[p-\epsilon, p+\epsilon]$ while the observation frequency was decreased linearly from $10\text{s}^{-1}$ to $2\text{s}^{-1}$. The value $\epsilon = 0.05$ was chosen, in order to match the value of $\epsilon$ used by the adaptive learning algorithm (cf. Table 3.3). The typical experimental results in Figure 3.12 show that Method 1 cannot handle this degree of variability, while Method 2 yields $p^k$ estimates within the desired interval around $p$ most of the time. In contrast, Methods 3–4 produce estimates that remain within this interval throughout the 10-hour simulated time period. The main difference between these two methods is that Method 4 (the adaptive learning algorithm) achieves this objective with much less oscillation around the actual value $p$.

In order to measure the accuracy of the estimates quantitatively, we performed 100 experiments simular to those from Figure 3.10, for each of the four learning methods. For each experiment, we measured the cumulated time $t_{outside}$ during which the estimate probability $p^k$ resided outside the interval $[p-\epsilon, p+\epsilon]$. Table 3.5 reports these times,

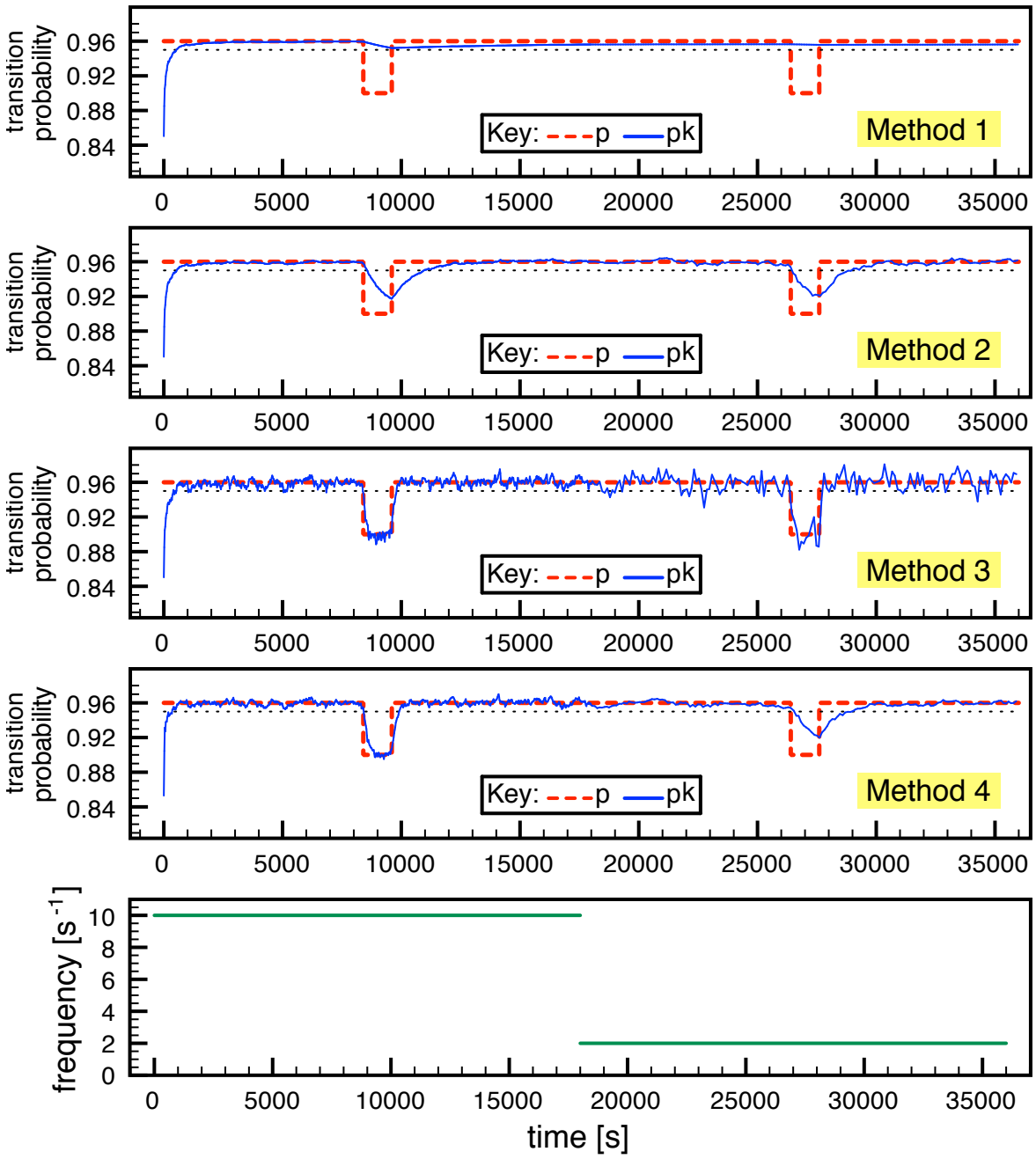


Figure 3.9: Experimental results—scenario 2

averaged over the 100 experiments carried out for each learning method, confirming that the adaptive learning methods outperforms the other three methods according to this criterion.

## 3.4 Learning the costs/reward structures of Markov chain models

### 3.4.1 Learning technique

This section introduces a technique for learning the costs/reward structures of MC models associated with component-based systems such as workflows and embedded

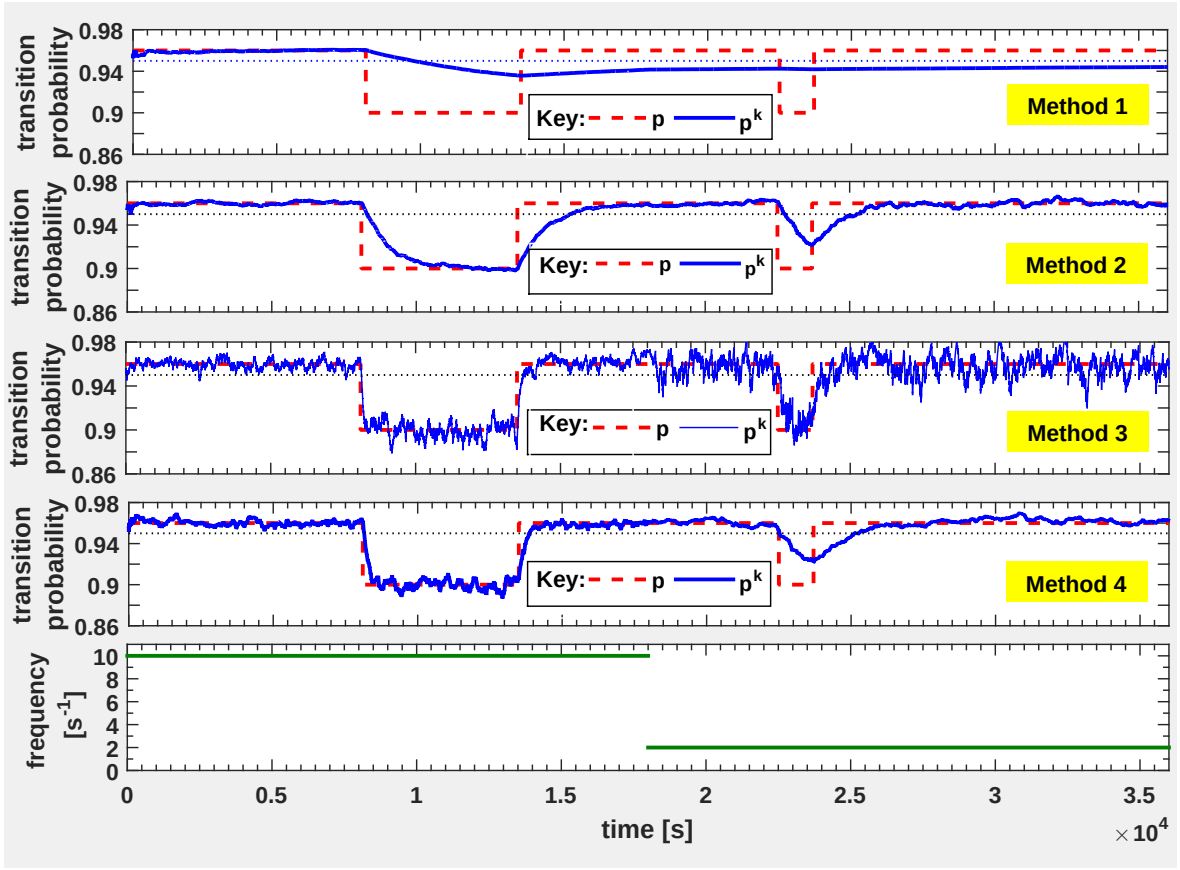


Figure 3.10: Experimental results—scenario 3

systems. The technique is applicable to systems that execute multiple sequences of operations such that the cumulative costs/rewards associated with each sequence can be measured, but it is not possible to measure the cost/reward for individual operations and/or transitions. This is a scenario that is frequently encountered in systems that use third-party components and in embedded systems. Using our technique to establish the unknown costs/rewards of MC models of these systems enables:

1. the identification of under-performing components;
2. establishing QoS requirement compliance/violation;
3. predicting the system performance/cost for infrequent execution paths;
4. the dynamic reconfiguration of the system to overcome the impact of under-performing overly costly components.

Formally, a system targeted by our technique comprises components that can perform $n > 1$ operations and is modelled by a MC $\mathcal{M} = (S, s_0, \mathcal{P}, L)$, where:

- the $i$-th system operation, $1 \leq i \leq n$, is associated with a state $s_i \in S$;
- the state $s_i$ associated with operation $i$ has a single outgoing transition, and an unknown state cost/reward $x_i$ corresponding to a QoS parameter of operation $i$, such as cost, energy usage or profit;
- the system has a finite set of frequently executed operation sequences that correspond to $m \geq 1$ known paths $\pi_1, \pi_2, \ldots, \pi_m$ of $\mathcal{M}$.

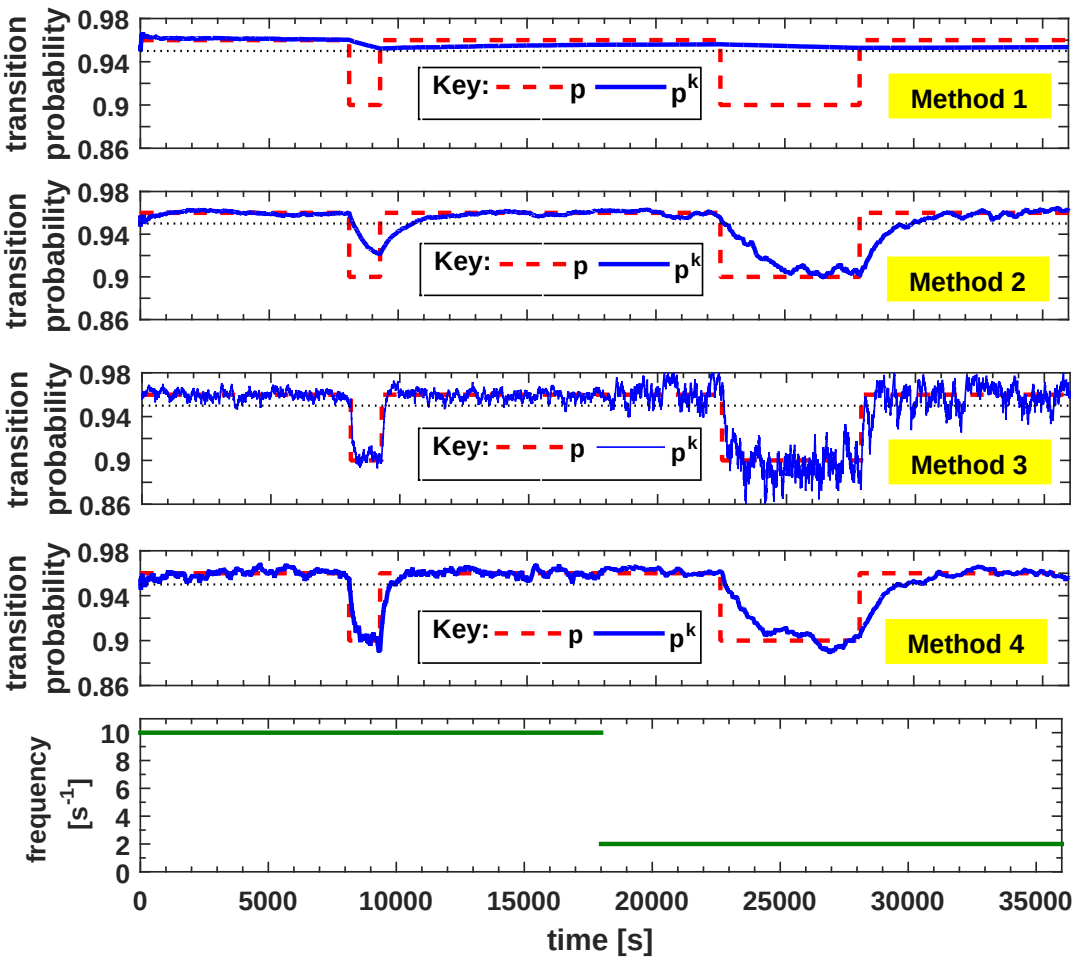


Figure 3.11: Experimental results—scenario 4

The values $x_1, x_2, \ldots x_n$ define a cost/reward structure $\iota : S \to R_+$ over $\mathcal{M} = (S, s_0, \mathcal{P}, L)$, where $\iota(s_i) = x_i$ for $1 \leq i \leq n$ and $\iota(s) = 0$ for $s \in S \setminus \{s_1, s_2, \ldots, s_n\}$. In addition to the states associated with system operations, the state set $S$ comprises:

(i) A "start" state $s_0$ and an "end" state $s_{\mathsf{end}}$ such that $\pi_j = s_0 \ldots s_{\mathsf{end}}$ for all $1 \leq j \leq m$;

(ii) States corresponding to decision points in the workflow implemented by the system. The states have two or more outgoing transitions to other states.

These elements of the class of MC models considered in this section are depicted in Figure 3.13. In addition, we require that the decision state outgoing transitions ensure that the MC graph is "structured" in the sense from structured programming [56]. In other words, the MC graph is a sequence, a (probabilistic) selection or a (probabilistic) loop of states or subgraphs with the same property. As demonstrated by structural induction in Figure 3.14, this ensures that the cost/reward for any of the $m$ operation sequences is a linear combination of the $n$ operation cost/reward values $x_1$, $x_2$, …, $x_n$. Note that this assumption does not constrain the types of component-based systems that can be handled using the learning technique to be introduced in this section, since according to the Böhm-Jacopini structured program theorem [22] any such system can be represented using a structured model.

We assume that a system with the characteristics described above is continuously monitored, and the observed values of the cumulative costs/rewards $z_1$, $z_2$, …, $z_m$ for the $m$ operation sequences are being recorded. Our technique uses this information and

Table 3.4: Quantitative analysis of the experiments in Scenarios 1–2

| Scenario & Method (S$x$_M$y$) | high-frequency observations | | | low-frequency observations | | |
|---|---|---|---|---|---|---|
| | $t_{down}$ [s] | $t_{up}$ [s] | $n_+$ | $t_{down}$ [s] | $t_{up}$ [s] | $n_+$ |
| S1_M1 | 570 | — | 0 | — | — | — |
| S1_M2 | 95 | 2110 | 0 | 76 | 2149 | 0 |
| S1_M3 | 6 | 217 | 54 | 33 | 239 | 38 |
| S1_M4 | 26 | 405 | 0 | 128 | 2100 | 0 |
| S2_M1 | 150 | — | 0 | — | — | — |
| S2_M2 | 162 | 1350 | 0 | 46 | 1156 | 0 |
| S2_M3 | 2 | 195 | 65 | 0 | 113 | 134 |
| S2_M4 | 13 | 303 | 0 | 131 | 1120 | 0 |
| S3_M1 | 27 | — | 0 | — | — | — |
| S3_M2 | 195 | 1851 | 0 | 268 | 1625 | 0 |
| S3_M3 | 29 | 210 | 48 | 14 | 271 | 274 |
| S3_M4 | 49 | 377 | 0 | 30 | 1410 | 0 |
| S4_M1 | — | — | — | — | — | — |
| S4_M2 | 169 | 1290 | 0 | 180 | 2050 | 0 |
| S4_M3 | 11 | 151 | 61 | 80 | 120 | 247 |
| S4_M4 | 15 | 396 | 0 | 180 | 1470 | 0 |

Table 3.5: Cumulative times when the estimate probability $p^k$ is outside the interval $[p-\epsilon, p+\epsilon]$, averaged over 100 36,000-second experiments

| Method | $t_{outside}$ [s] |
|---|---|
| Method 1 | 31390.64 |
| Method 2 | 4791.11 |
| Method 3 | 60.91 |
| Method 4 | 15.17 |

one of the optimal filters described in the background Sections 2.4–2.5 to estimate the unknown costs/rewards function $\iota$. For this purpose, it starts by using the MC model and the results in Figure 3.14 to derive an algebraic expression of the expected cumulative cost/reward $E(z_j)$ for each path $\pi_j$, $1 \leq j \leq m$. Given the assumptions above, the algebraic expressions for the $m$ paths are linear combinations of the mean operation cost/reward $x_1, x_2, \ldots, x_n$. Using the notation $E(\mathbf{z}) = [E(z_1)\ E(z_2)\ \cdots\ E(z_m)]^{\mathrm{T}}$ and $x = [x_1\ x_2\ \cdots\ x_n]^{\mathrm{T}}$, we have

$$E(\mathbf{z}) = \mathbf{H}\mathbf{x}, \tag{3.13}$$

where $\mathbf{H}$ is a $m \times n$ matrix with non-negative elements.

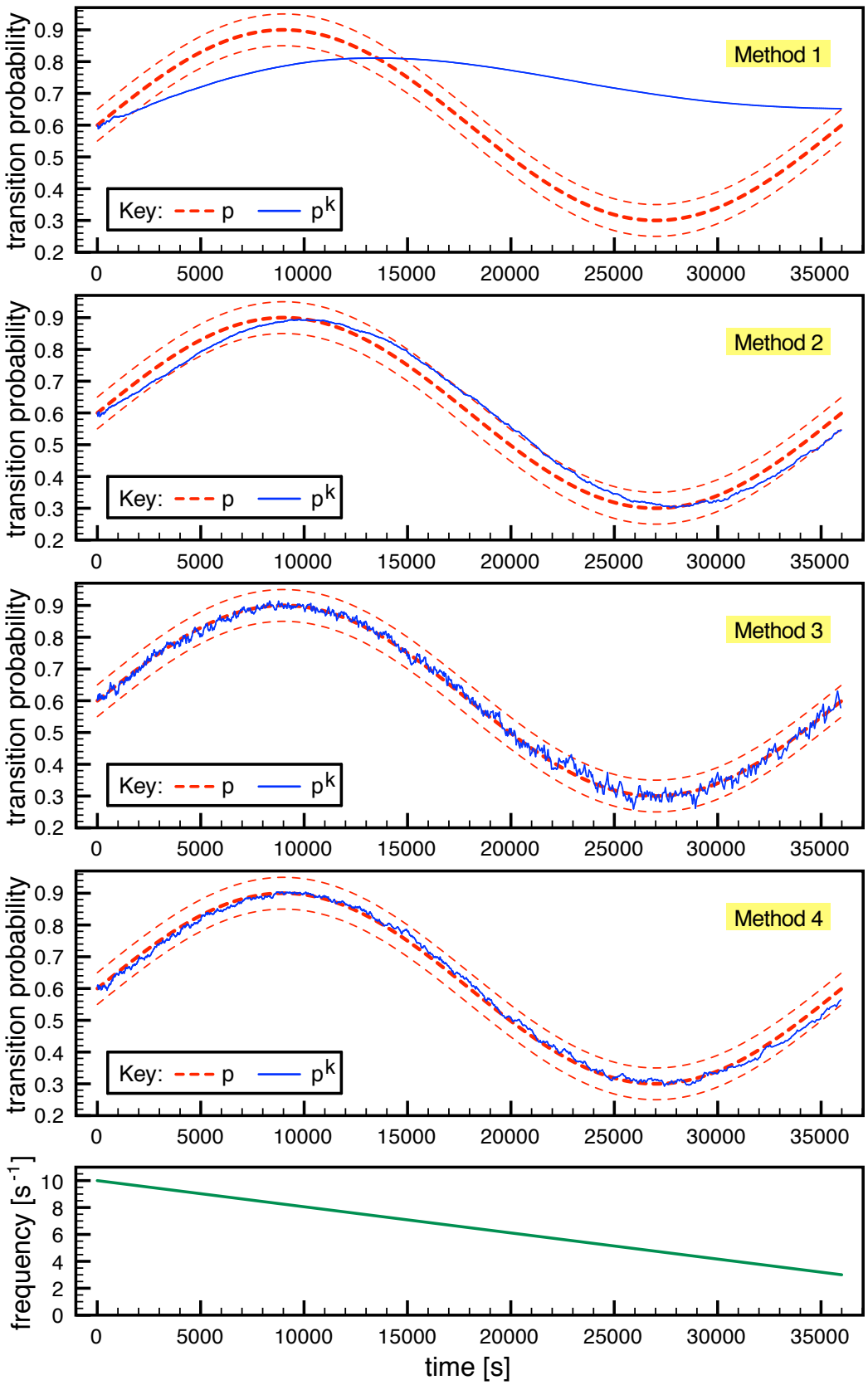


Figure 3.12: Experimental results—scenario 5

In order to use the above result for estimating the values $x_1$, $x_2$, ..., $x_n$, the observed cumulative cost/reward for the $m$ operation sequences are averaged within time windows of fixed duration $T > 0$. The value of $T$ is chosen such that a large

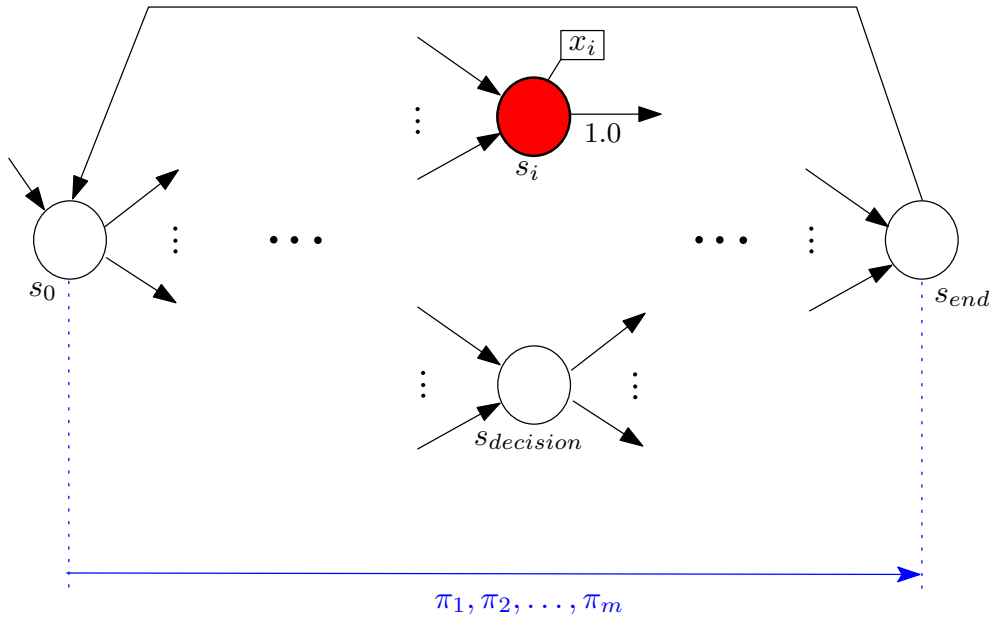


Figure 3.13: Elements of the Markov chain model.

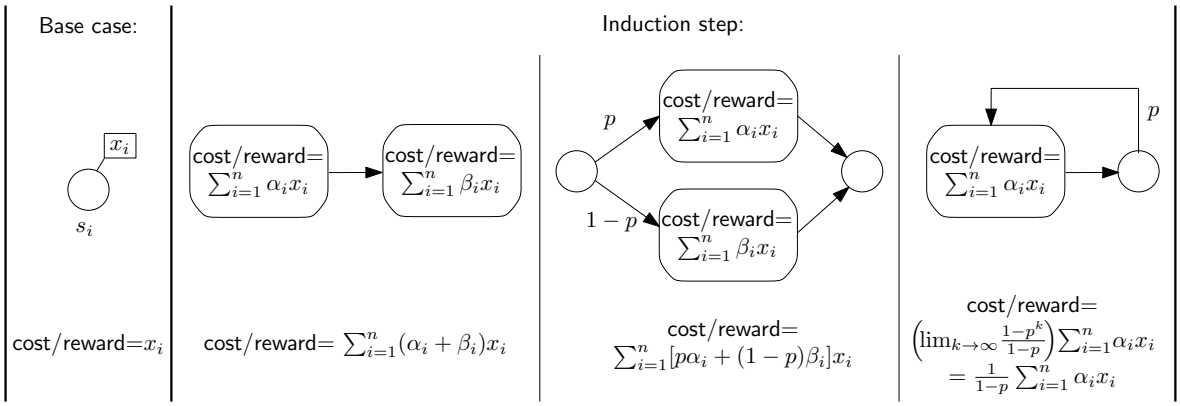


Figure 3.14: Structured Markov chain graph with cost/reward calculation.

number of observations of each operation sequence are made in each time window. Let $z_{1,t}$, $z_{2,t}$, ..., $z_{m,t}$ be the average cost/reward for the $m$ operation sequences within time window $t$, and $\mathbf{z}_t = [z_{1,t}\ z_{2,t}\ \cdots\ z_{m,t}]^{\mathrm{T}}$. Each $z_{j,t}$, $1 \leq j \leq m$, is the average of a large number of observations, so according to the Central Limit Theorem [89] $z_{j,t}$ is normally distributed with mean $E(z_j)$, and we have

$$\mathbf{z}_t = \mathbf{H}\mathbf{x}_t + \mathbf{v}_t, \tag{3.14}$$

where $\mathbf{v}_t$ is a normally distributed measurement noise with zero mean, and $\mathbf{x}_t$ is an $n \times 1$ vector whose elements are the unknown cost/reward values for the $n$ system operations in time window $t$.

Assuming that rank $\mathbf{H} = n$, the system

$$\begin{aligned} \mathbf{x_{t+1}} &= \mathbf{I_n}\mathbf{x_t} + \mathbf{w_t} \\ \mathbf{z_t} &= \mathbf{H}\mathbf{x_t} + \mathbf{v_t} \end{aligned} \tag{3.15}$$

where $\mathbf{I_n}$ is the $n \times n$ identity matrix is observable, since the observability matrix $\mathbf{M} = [\mathbf{H^T}\ \ (\mathbf{A^T})\mathbf{H^T}\ \ \cdots(\mathbf{A^T})^{\mathbf{n-1}}\mathbf{H^T}]$ from (2.9) reduces to $\mathbf{M} = [\mathbf{H^T}\ \ \mathbf{H^T}\ \ \cdots\mathbf{H^T}]$ when $\mathbf{A} = \mathbf{I_n}$. Under this assumption, the above formulation of the relationship between

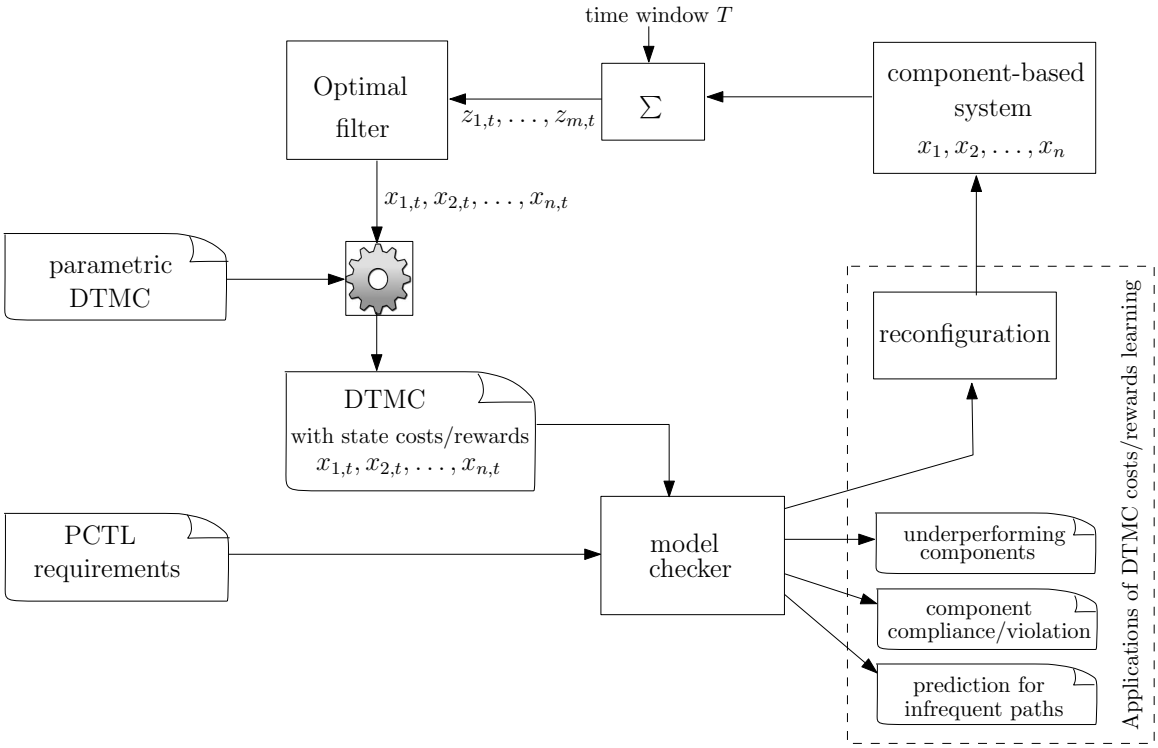


Figure 3.15: Online learning of Markov chain costs/rewards.

the unknown operation cost/reward and the observed cumulative cost/reward for the $m$ operation sequences allows the application of the optimal filters in Sections 2.4–2.5.

Figure 3.15 depicts the application of the technique described in this section to a black-box component-based system with $n > 1$ operations for which the cumulative value of a QoS attribute can be observed for $m$ sequences of operations. For a given time window $T$ the average cumulative cost/reward for the $m$ operation sequences are calculated, and an optimal filter is used to estimate the unknown cost/reward values $x_1, x_2, \ldots, x_n$ of the $n$ operations. A MC model parameterised by these unknown values is continually updated, and a model checker is used to verify that the system continues to meet PCTL-expressed requirements associated with the considered QoS attribute. The applications of the technique are multifold. First, it can be used to identify under-performing components and components that violate the system requirements. Second, it supports predicting the system behaviour for infrequent execution paths. Finally, it can be used to trigger dynamic reconfiguration within self-adaptive systems, to overcome the impact of under-performing components.

In the next section, we present the application of this technique for a case study, and we examine the effectiveness of the Kalman filter and the recursive weighted least square filter presented in Sections 2.4–2.5 respectively.

## 3.4.2 Case study

### 3.4.2.1 Description

We use a system from the telematics domain to illustrate the application of the learning technique in this section, and to assess its efficiency (i.e., time to detect underperforming components) and accuracy (i.e., ability to detect all/most violations in the QoS requirements). Telematics Systems (TSs) are emerging automotive technologies that use advanced communication and vehicle technologies to improve vehicle control and

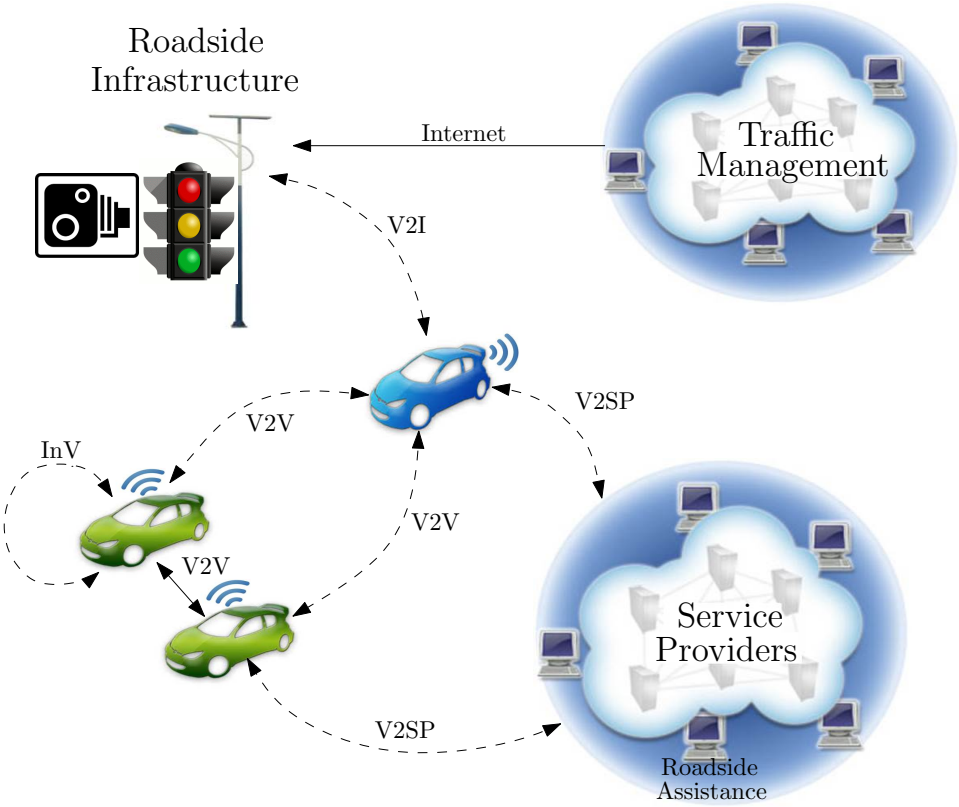


Figure 3.16: The three main V2X classification for automotive telematics systems

safety [40, 61, 62], to make vehicles more environment-friendly [5], and to enhance driver experience [41, 91, 99, 152]. TSs can be classified into three groups based on their communication range (Figure 3.16). In-vehicle (InV) TSs support the interaction between user portable devices (e.g., phones, tablets, etc.) and vehicle built-in infotainment systems. Vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) systems allow vehicles to communicate with each other and to roadside infrastructure, so that they can share safety-critical information, e.g., to avoid hazards ahead. Finally, vehicle-to-service provider (V2SP) systems allows the driver to access value-added services provided by third party providers, such as location based services.

Figure 3.17 depicts the UML activity diagram for the workflow of a telematics system adapted from [152], which we use in our case study. This system comprises a combination of InV, V2SP and V2I components that provide information and/or facilitate context-aware interactions between other components. The decision node branches in Figure 3.17 are annotated with the probabilities that those branches are being taken. Each execution of the workflow starts with one of the following four operations:

- The *weatherForecast* operation retrieves the latest weather information by invoking a remote service (V2SP).
- The *trafficCondition* operation retrieves the latest traffic news by invoking a remote service (V2SP).
- The *auxDiagnostic* operation performs a test on the built-in infotainment system (InV).
- The *primaryDiagnostic* operation carries out a built-in self-test to assess the state of mechanical parts, i.e. test the anti-lock brakes (InV).

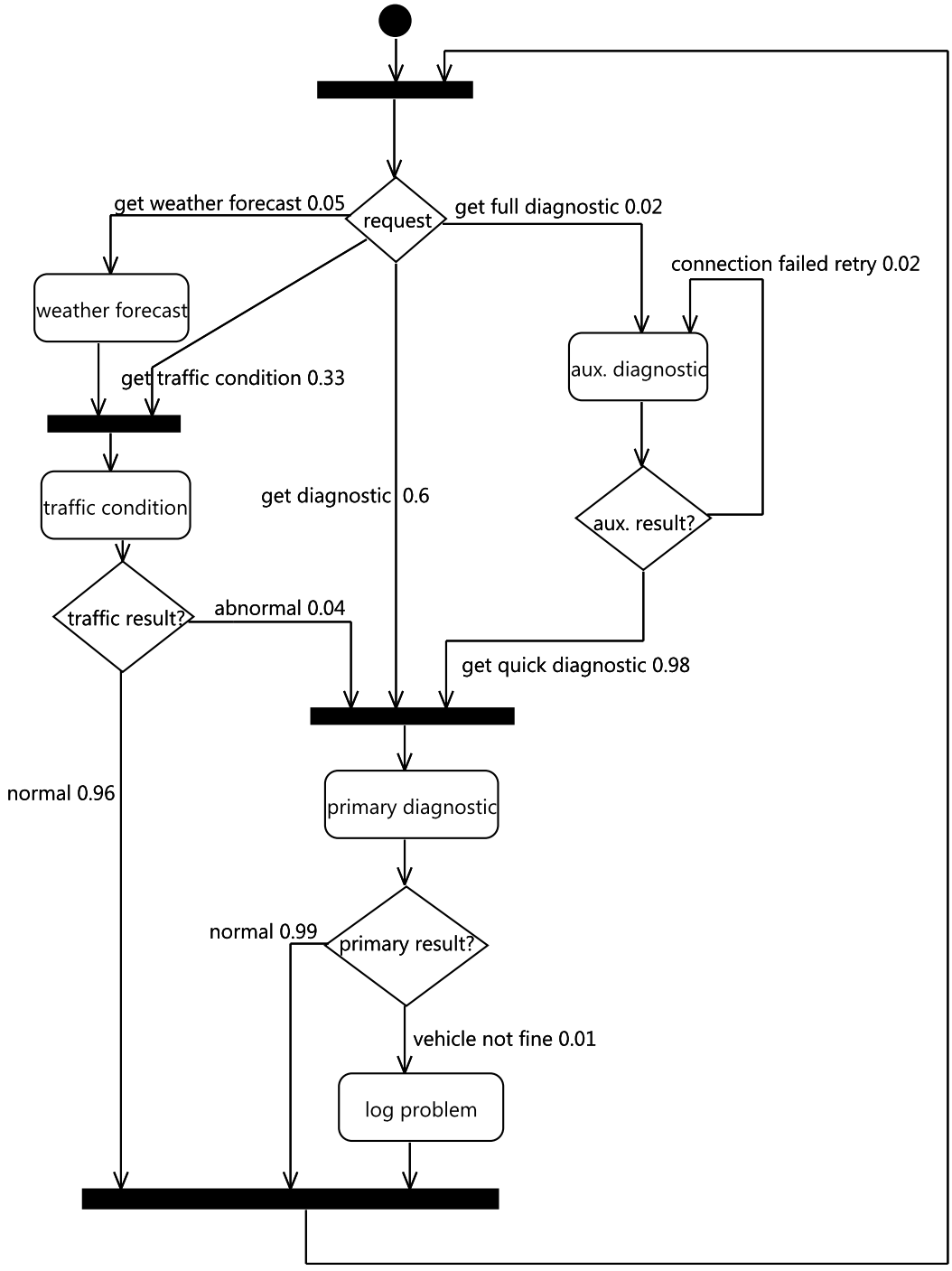


Figure 3.17: UML activity diagram of the telematics system

The execution of one of these four operations is followed by running a sequence of additional operations, as detailed in Figure 3.17. For example, the TS may first invoke the *weatherForecast* operation, followed by the *trafficCondition*. If the normal flow of traffic is disrupted and there are severe weather warnings, then the TS may invoke the *primaryDiagnostic* operation to carry out checks on the mechanical components of the vehicle. If the results from the diagnosis reveal that there is a fault in the component(s), then the *logProblem* operation is invoked to register the problem, so that it could be attended to when the vehicle is serviced next time. In addition to the primary diagnostic

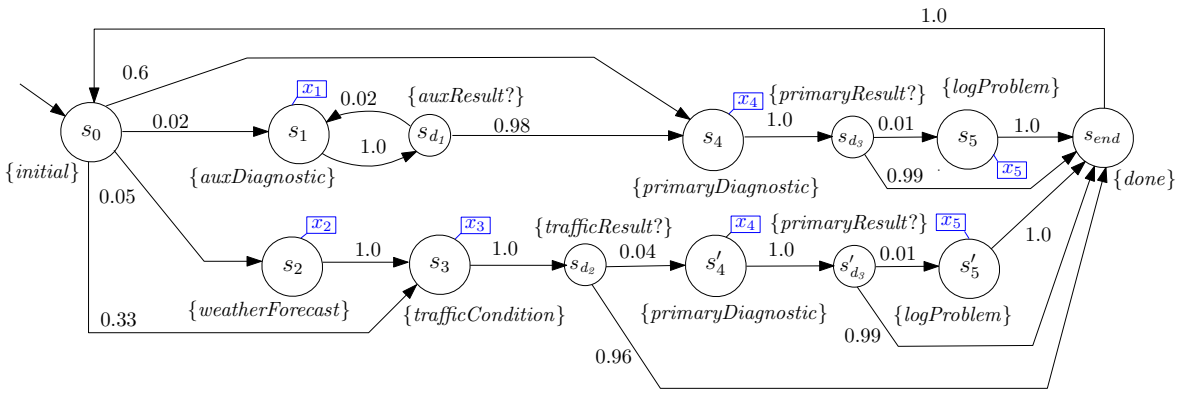


Figure 3.18: Markov chain model for the SBS workflow from Figure 3.17, with states $s_4$, $s_{d_3}$ and $s_5$ duplicated to ensure that the model is "structured".

operation, the TS may invoke the auxiliary (*auxDiagnostic*) diagnostic operation to carry out checks on non-critical components, i.e., CD drivers, radio, etc.

We assume that the TS must satisfy the following QoS requirements:

$R_1$**:** The expected workflow execution time is at most $320ms$.

$R_2$**:** The expected time of workflow executions that start with the a *primaryDiagnostic* operation does not exceed $200ms$.

$R_3$**:** The expected time of workflow executions that start with a *weatherForecast* operation does not exceed $310ms$.

$R_4$**:** The expected execution time for workflow instances that start with the *weatherForecast* operation and include the logging operation do not exceed $480ms$.

#### 3.4.2.2 Formal model and requirements

The telematics system described above fits the assumptions from Section 3.4.1, so we will use it to evaluate the learning technique presented in this section. Figure 3.18 shows the MC model obtained for the TS workflow using the template from Figure 3.13, with states $s_1$ to $s_5$ corresponding to the five operations of the system, and the cost/rewards $x_1$ to $x_5$ their unknown execution times. The transition probabilities are taken from the activity diagram.

We can now formalise the system requirements $R_1 - R_4$ from Section 3.4.2.1 as PCTL formulae:

$$R_1 : \quad R_{\leq 320}[F\ done],$$

$$R_2 : \quad R_{<=200}[F\ done\ \{primaryDiagnostic\}],$$

$$R_3 : \quad R_{<=310}[F\ done\ \{weatherForecast\}],$$

$$R_4 : \quad R_{=?}[F\ trafficResult?\ \{weatherForecast\}]+ \\ R_{=?}[F\ primaryResult?\ \{primaryDiagnostic\}]+ \\ R_{=?}[F\ done\ \{logProblem\}] \leq 480.$$

where $R_2$–$R_4$ use a PRISM extension of PCTL in which the state selector $\{\phi\}$ enforces the evaluation of the property for a start state that satisfies $\phi$.

We assume that we can measure the overall execution time of the $m = 5$ execution paths (i.e., operation sequences) described below.

**Path** $\pi_1$ - The TS invokes the *primaryDiagnostic* service to carry out routine checks on the essential components of the vehicle, to ensure that it is safe to drive. The results from the *primaryDiagnostic* service reveal that one of the components has a mechanical fault and in turn the *logProblem* service is invoked to register the problem, so that it is attended to when the vehicle is serviced next time. The expected execution time for this path is $E(z_1) = x_4 + x_5$.

**Path** $\pi_2$ - The *auxDiagnostic* service is invoked to carry out a routine check on all the auxiliary components, which is followed by the invocation of the *primaryDiagnostic* service. The results from the complete diagnosis service are normal and the total expected response time to execute these operations in the TS workflow are according to the MC in Figure 3.18. Accordingly, $E(z_2) = (1 + 0.02 + 0.02^2 + \ldots)x_1 + x_4 = \left(\lim_{k\to\infty} \frac{1-0.02^k}{1-0.02}\right) x_1 + x_4 = \frac{1}{0.98}x_1 + x_4$.

**Path** $\pi_3$ - The *weatherForecast* service followed by the *trafficCondition* service are invoked to retrieve the latest weather and road traffic news. The *primaryDiagnostic* service is also invoked and the results are normal, i.e., $E(z_3) = x_2 + x_3 + x_4$.

**Path** $\pi_4$ - The *trafficCondition* service is invoked to retrieve the latest road traffic news, which is then followed by the *primaryDiagnostic* service. The results from the *primaryDiagnostic* service are normal. Accordingly, the total expected time to execute this sequence of operations is $E(z_4) = x_3 + x_4$.

**Path** $\pi_5$ - The *weatherForecast* service followed by the *trafficCondition* service are invoked to retrieve the latest weather and road traffic news. The total expected time for this path is $E(z_5) = x_2 + x_3$.

An execution of the workflow corresponds to one of these five sequences of operations with probability 0.006, 0.0202, 0.00198, 0.013068 and 0.048, respectively, as calculated by multiplying the relevant transition probabilities along the five paths. These are relatively high probabilities, meaning that each operation sequence will be observed relatively often.

In addition to the five sequences of operations, we are interested in analysing the following infrequently executed path that corresponds to requirement $R_4$, and whose probability of execution is only 0.00002.

**Path** $\pi_6$ - The *weatherForecast* service is invoked, followed by an invocation of the *trafficCondition* service. The *trafficCondition* service returns information pertaining to hazardous traffic conditions that are causing disruptions ahead, so the TS invokes the *primaryDiagnostic* service. The *primaryDiagnostic* service carries out checks on all of the essential components, to ensure that the vehicle is safe to drive under such conditions. The results from the *primaryDiagnostic* service reveals that one of the components has a mechanical fault and in turn the *logProblem* service is invoked to register the problem, to be attended to when the vehicle is serviced next time. The expected execution time for this path is $x_2 + x_3 + x_4 + x_5$.

Our technique uses the estimated execution times of the individual operations observed from the overall execution time of the frequently executed paths to establish the system behaviour along this infrequent path. To this end, the observed execution times of the $m = 5$ frequent operation sequences are averaged over time windows of fixed duration $T > 0$. According to (3.14), the average observation vector for time

window $t$, i.e., $\mathbf{z_t} = [z_{1,t}\ z_{2,t}\ \ldots\ z_{5,t}]^T$ will satisfy

$$\mathbf{z_t} = \begin{bmatrix} z_{1,t} \\ z_{2,t} \\ z_{3,t} \\ z_{4,t} \\ z_{5,t} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ \frac{1}{0.98} & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} = \mathbf{H}\mathbf{x_t} + \mathbf{v_t}, \tag{3.16}$$

where $\mathbf{x_t}$ is the vector whose elements are the execution times of the $n$ system operations and $\mathbf{v_t}$ represents the normally distributed measurement noise. In addition, since rank$\mathbf{H} = 5$, the system

$$\begin{aligned} \mathbf{x_{t+1}} &= \mathbf{I_5}\mathbf{x_t} + \mathbf{w_t} \\ \mathbf{z_t} &= \mathbf{H}\mathbf{x_t} + \mathbf{v_t} \end{aligned} \tag{3.17}$$

is observable, and we can use the optimal filters in Sections 2.4–2.5 to estimate the unknown transition costs/rewards $x_1$ to $x_5$ of the MC from Figure 3.18. These estimates can then be used to validate if the system meets its requirements $R_1$ to $R_4$.

### 3.4.3 Experiment setup and results

To evaluate the effectiveness of our learning technique, and to compare the optimal filters from Sections 2.4–2.5, we implemented a Java simulator of the MC model from Figure 3.18, and used it to run a wide range of experiments. In these experiments, the operation execution times $x_1$ to $x_5$ were assumed normally distributed, with a mean that changed according to different patterns of component failure and variation around a "nominal" value for which requirements $R_1$–$R_4$ from Sections 3.4.2.1 and 3.4.2.2 were satisfied. For each experiment, the standard deviation for the execution time of the $i$-th operation was selected as one of $sd_i = 0.1x_i$, $sd_i = 0.2x_i$, ..., $sd_i = 0.5x_i$.

All experiments used a normally distributed delay with a mean of 5s and a standard deviation of 1s between successive executions of the TS workflow. Observations of the execution times of the $m = 5$ operation sequences from Section 3.4.2.2 were averaged across $T = 600s$ and $T = 1800s$ time windows, and the optimal filters were used to estimate the execution times $x_1$ to $x_5$. For each experiment, we measured the following attributes of the costs/rewards learning:

1. *Learning time* (LT), i.e., the time elapsed between the moment when there was a change in the execution time $x_i$ of an operation, and the moment when the learning produced an estimate within 10% of the new $x_i$ value;

2. *False positives* (FP) and *false negatives* (FN), i.e., the percentages of time for which using the estimated $x_1$ to $x_5$ values to establish if requirements $R_1$–$R_4$ were met produced false positives and false negatives.

The rest of the section presents and discusses these experimental results.

**Optimal filter comparison** We start with a typical experiment that illustrates the effectiveness of the Kalman filter (KF) and the recursive weighted least square (RWLS) filter for a scenario in which $sd_i = 0.3x_i$, $1 \leq i \leq 5$. The results of this experiment are shown in Figure 3.19, where the time intervals labelled $a$, $b$, etc. represent LTs for the KF estimation, and the time intervals $a'$, $b'$, etc. are LTs for the RWLS filter estimation. Note that for several changes in the execution time $x_i$ of the $i$-th operation, estimates within 10% of the actual $x_i$ value (which we deem "accurate estimates") were

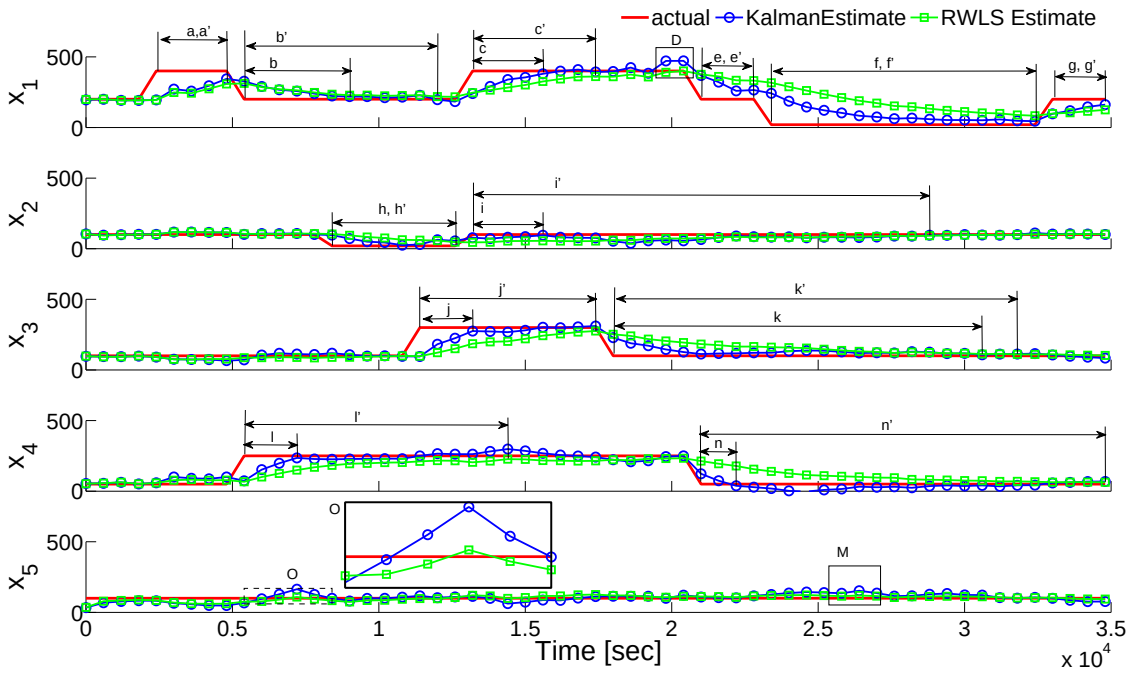


Figure 3.19: Kalman-filter versus RWLS-filter costs/rewards learning

not produced before the next change. This is the case for the time intervals marked $a, a'$; $e, e'$; $g, g'$; and $h, h'$. For all the remaining changes, the Kalman filter was consistently faster at producing accurate estimates than the RWLS filter. This was true both in detecting degradations in the response time of an operation (e.g., for LTs $c, c'$, $j, j'$ and $l, l'$) and reductions in response times (e.g., for LTs $b, b'$ and $n, n'$).

However, the Kalman filter estimates are characterised by a far more significant oscillation around the actual $x_i$ values than the estimates produced by the RWLS filter, as illustrated by the regions D, O and M from Figure 3.19. The oscillations in the KF estimates are caused by the high values of the elements of the covariance matrix $Q$ from the error covariance equation (2.14) in Section 2.4.2. These values increase the Kalman gain, to ensure the filter places sufficient confidence in the new observations. Otherwise, i.e., if the $Q$ matrix elements are smaller, the filter becomes numerically unstable and the learning becomes slow. In contrast, the RWLS filter uses a forgetting factor (or weighting factor) $\lambda$ (cf. the error covariance equation (2.4.2) in Section 2.5) that reduces the influence of the old observations, therefore reduces the error in the state estimate and the learning is always smoother.

**Analysis of the effect of variation in component execution times** The next two experiments illustrate the effectiveness of the KF and the RWLS filter for scenarios in which the optimal filters were used to estimate the mean execution times $x_i$ $1 \leq i \leq 5$ for multiple standard deviation values, $sd_i = \{0.1x_i, 0.2x_i, 0.5x_i\}$. The results of these experiments are shown in Figures 3.20 and 3.21, where the time intervals labelled $1a, 1b$ etc. represent LTs for the estimations with standard deviation $sd_i = 0.1x_i$, the time intervals labelled $2a, 2b$ etc. represent LTs for the estimations with standard deviation $sd_i = 0.2x_i$, and the time intervals labelled $3a, 3b$ etc. represent LTs for the estimations with standard deviation $sd_i = 0.5x_i$. Figure 3.20 depicts the results for the KF estimation. The time intervals marked $1b, 2b, 3b$ and $1f, 2f, 3f$ show instances when KF estimates within 10% of the actual $x_i$ mean value were not produced before the next change. For all the remaining changes, the KF was mostly faster at detecting

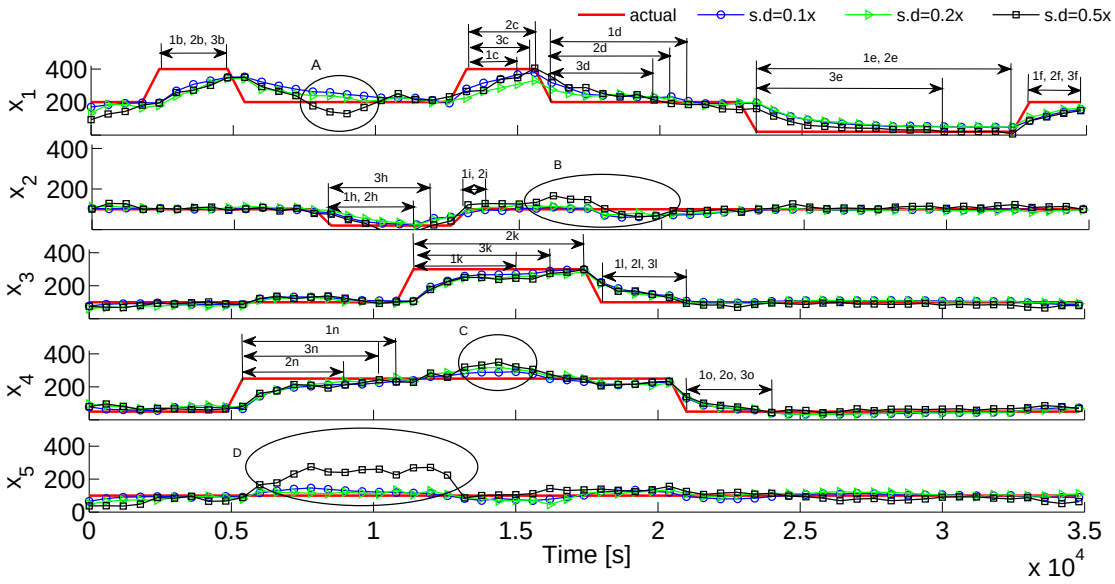


Figure 3.20: Experimental results comparing the effectiveness of the Kalman filter for multiple standard deviation values

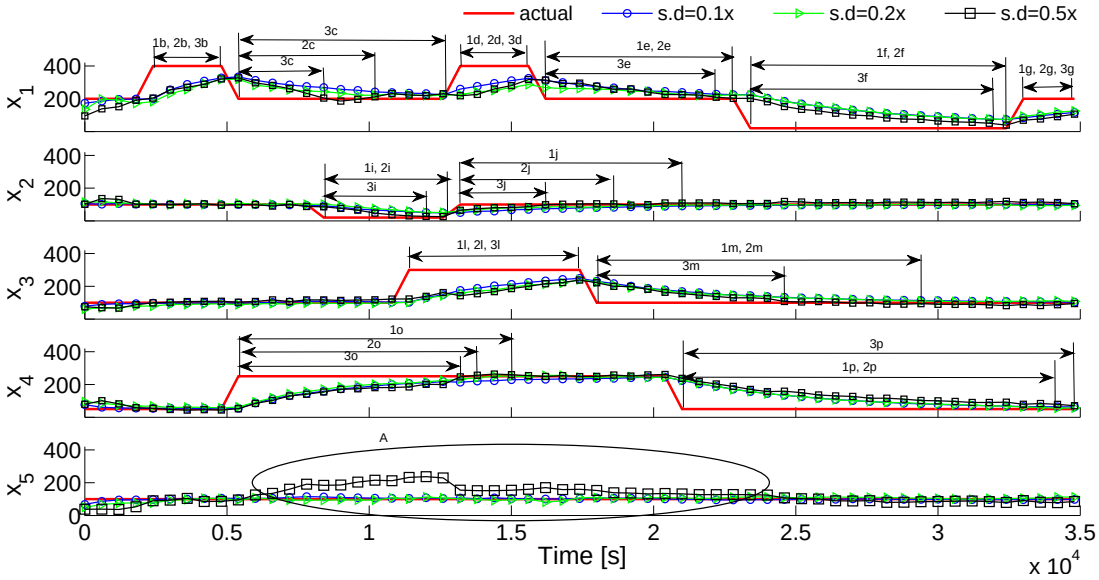


Figure 3.21: Experimental results comparing the effectiveness of the recursive weighted least-square filter for multiple standard deviation values

degradations in the response time of an operation with standard deviation $sd_i = 0.1x_i$ (e.g., for LTs $1c, 1i, 1k$); and was in general faster at detecting reductions in the response time of an operation with standard deviation $sd_i = 0.5x_i$ (e.g., for LTs $3d, 3e$).

Nevertheless, the areas labelled $A, B, C$, and $D$ show that the KF estimate with standard deviation $sd = 0.5x_i$ are characterised by significant oscillation around the actual $x_i$ values than the estimates produced with standard deviation $sd_i = 0.1x_i$ and $sd_i = 0.2x_i$.

Figure 3.21 depicts the experimental results comparing the effectiveness of the RWLS filter for multiple standard deviation values. Note that for several changes in the mean execution time $x_i$ of the $i-$th operation, estimates within 10% of the actual $x_i$ value were not produced before the next change. This is the case for the

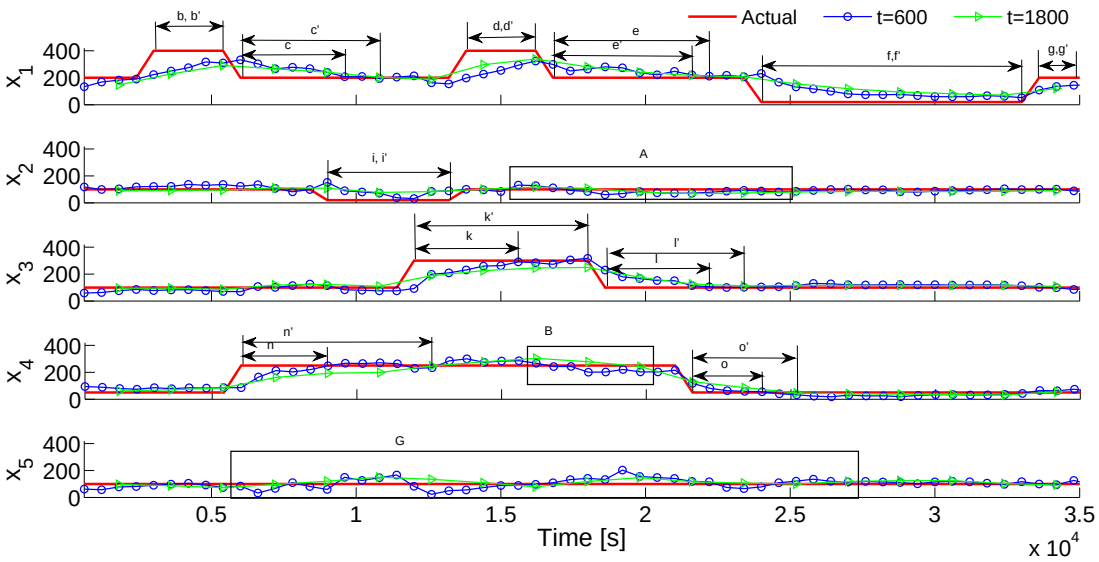


Figure 3.22: Experimental results examining the effect of different time windows on the effectiveness of Kalman filter

time intervals marked $1b, 2b, 3b; 1d, 2d, 3d; 1g, 2g, 3g$ and $1l, 2l, 3l$. For all the remaining changes, mostly the RWLS filter was faster at producing accurate estimates with standard deviation $sd = 0.5x_i$. This was true both in detecting degradations in the response time of an operation (e.g., for LTs $3j$ and $3o$) and reduction in response time (e.g., for LTs $3c; 3e; 3f; 3i, 2i$ and $3m$).

Like in the case of the KF estimates, the RWLS filter estimates with standard deviation $sd = 0.5x_i$ are characterised by significant oscillation around the actual $x_i$ value when the actual $x_i$ value is constant than the estimates produced with standard deviation $sd = 0.1x_i$ and $sd = 0.2x_i$, as illustrated by the region marked $A$ from Figure 3.21.

**Analysis of the effect of the time window length** Figures 3.22 and 3.23 depict the experimental results for the KF and RWLS filter respectively, for scenarios with time windows $T_1 = 600s$ and $T_2 = 1800s$, with a fixed standard deviation $sd_i = 0.4x_i$, $1 \leq i \leq 5$. The time intervals labelled $a, b$ etc. represent LTs for the estimations from time window $T_1$, and the time intervals labelled $a', b'$ etc. represent LTs for the estimations from time window $T_2$.

Note that for several changes in the mean execution time $x_i$ in Figure 3.22, estimates within 10% of the actual $x_i$ value were not produced before the next change. This is the case for the time intervals marked $b, b'; d, d'; f, f'; g, g'; i, i'$ and $m, m'$. For the remaining changes, in general the KF estimates in time window $T_1$ were faster at producing accurate estimates than the estimates from time window $T_2$. This is true both in detecting degradations in the response time of an operation (e.g., for LTs $j$ and $m$) and reductions in response times (e.g., for LTs $c, k$ and $n$).

In general, the Kalman filter estimates in time window $T_1$ are characterised by a far more significant oscillation around the actual $x_i$ values than the estimates produced by the KF filter in time window $T_2$, as illustrated by the regions A, B and G from Figure 3.22.

Figure 3.23 depicts the experimental results comparing the effectiveness of the RWLS filter with the longer time windows. Note that for several changes in the execution time $x_i$ of the $i-$th operation, estimates within 10% of the actual $x_i$ value were not

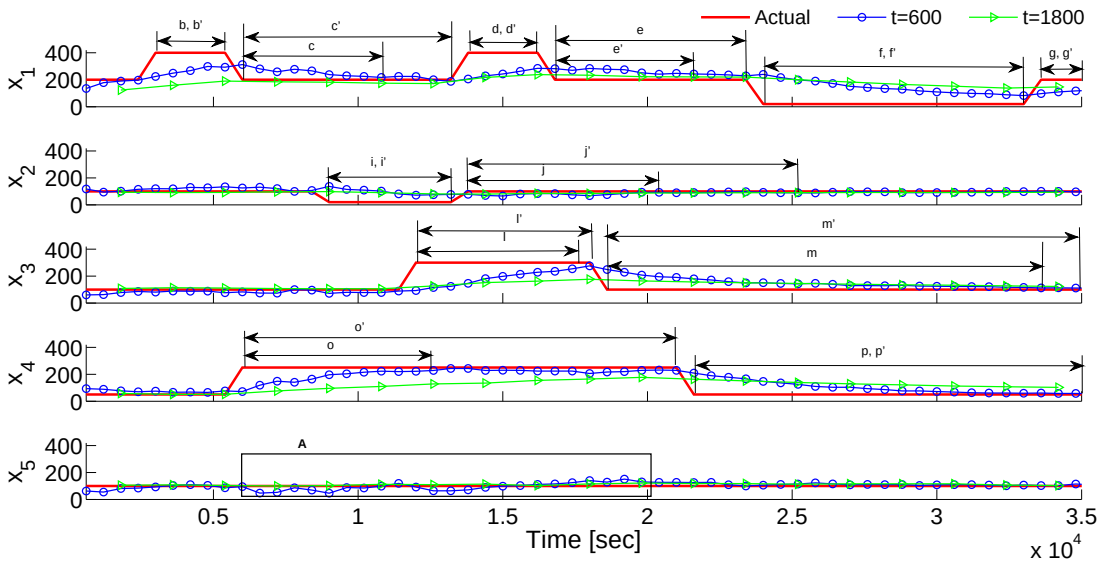


Figure 3.23: Experimental results examining the effect of different time windows on the recursive weighted least square filter

Table 3.6: False positives and false negatives in the analysis of requirement R1

| Requirement R1 | Negatives, i.e., requirement met | | Positives, i.e., requirement violated | |
|---|---|---|---|---|
| | TN | FP | TP | FN |
| $sd_i = 0.1x_i$ | K: 100%<br>RWLS: 83% | K: 0%<br>RWLS: 17% | K: 95%<br>RWLS: 70% | K: 5%<br>RWLS: 30% |
| $sd_i = 0.2x_i$ | K: 100%<br>RWLS: 82% | K: 0%<br>RWLS: 18% | K: 95%<br>RWLS: 70% | K: 5%<br>RWLS: 30% |
| $sd_i = 0.5x_i$ | K: 99%<br>RWLS: 83% | K: 1%<br>RWLS: 17% | K: 92%<br>RWLS: 74% | K: 8%<br>RWLS: 26% |

produced before the next change. This is the case for the time intervals marked $b, b'; d, d'; f, f'; g, g'; i, i'$ and $p, p'$. For all the remaining changes, mostly the RWLS filter estimates from time window $T_1$ were faster in producing accurate estimates than the estimates from the time window $T_2$. This was true both in detecting degradations in the response time of an operation (e.g., for LTs $j, l$, and $o$) and reduction in response times (e.g., for LTs $c, m$ and $p$).

However the RWLS filter estimates from time window $T_1$ are characterised by more sporadic oscillations around the actual $x_i$ values than the estimates produced by the RWLS filter from time window $T_2$, as illustrated by the region marked A from Figure 3.23.

**False negatives and false positives** Tables 3.6–3.9 show the experimental results for the analysis of the *sensitivity* of our learning technique (i.e., the percentage of false positives detected) and its *specificity* (i.e., the percentage of false negatives detected). To this end, we examined the ability of the learning technique to establish whether

Table 3.7: False positives and false negatives in the analysis of requirement R2

| Requirement R2 | Negatives, i.e., requirement met | | Positives, i.e., requirement violated | |
|---|---|---|---|---|
| | TN | FP | TP | FN |
| $sd_i = 0.1x_i$ | K: 100%<br>RWLS: 94% | K: 0%<br>RWLS: 6% | K: 87%<br>RWLS: 66% | K: 13%<br>RWLS: 34% |
| $sd_i = 0.2x_i$ | K: 100%<br>RWLS: 95% | K: 0%<br>RWLS: 5% | K: 86%<br>RWLS: 54% | K: 14%<br>RWLS: 46% |
| $sd_i = 0.5x_i$ | K: 100%<br>RWLS: 97% | K: 0%<br>RWLS: 3% | K: 70%<br>RWLS: 45% | K: 30%<br>RWLS: 55% |

Table 3.8: False positives and false negatives in the analysis of requirement R3

| Requirement R3 | Negatives, i.e., requirement met | | Positives, i.e., requirement violated | |
|---|---|---|---|---|
| | TN | FP | TP | FN |
| $sd_i = 0.1x_i$ | K: 100%<br>RWLS: 99% | K: 0%<br>RWLS: 1% | K: 73%<br>RWLS: 20% | K: 27%<br>RWLS: 80% |
| $sd_i = 0.2x_i$ | K: 100%<br>RWLS: 99% | K: 0%<br>RWLS: 1% | K: 74%<br>RWLS: 12% | K: 26%<br>RWLS: 88% |
| $sd_i = 0.5x_i$ | K: 99%<br>RWLS: 98% | K: 1%<br>RWLS: 2% | K: 74%<br>RWLS: 21% | K: 26%<br>RWLS: 79% |

the four requirements $R1 - R4$ from Sections 3.4.2.1 and 3.4.2.2 were satisfied or violated. The analysis for the false positives and the false negatives was averaged over ten experiments for both filters and repeated for multiple values of the standard deviation.

For all experiments and irrespective of the value of the standard deviation, the KF learning led to lower FP and FN percentages than the RWLS learning, e.g., for requirement R1 (shown in Table 3.6) only 5% FN for $sd_1 = 0.1x_1$ was obtained for the KF, compared to 30% for the RWLS filter for the same standard deviation. In general, the FP and FN percentages increased with the standard deviation, e.g., for requirement R4 (Table 3.9) the FN increased from 5% for $sd_4 = 0.1x_2$ to 14% for $sd_2 = 0.5x_2$ for the KF, and from 29% to 40% for the RWLS filter. This increase was more pronounced for the RWLS filter across all requirements, e.g., with FN growing from 34% for $sd_2 = 0.1x_2$ to 55% for $sd_2 = 0.5x_2$ for requirement R2 (see Table 3.7). In contrast, the FP percentages increased very slightly with the standard deviation, e.g., in Table 3.8 the FP for requirement R3 increased from 0% for $sd_3 = 0.1x_3$ to just 1% for $sd_3 = 0.5x_3$ for KF, and from 1% to only 2% for the RWLS filter.

All the experiments described so far suggest that the Kalman filter is better than the RWLS filter in terms of both false positives and false negatives. However, there are scenarios in which the slower but smoother learning provided by the RWLS filter is

Table 3.9: False positives and false negatives in the analysis of requirement R4

| Requirement R4 | Negatives, i.e., requirement met | | Positives, i.e., requirement violated | |
|---|---|---|---|---|
| | TN | FP | TP | FN |
| $sd_i = 0.1x_i$ | K: 100%<br>RWLS: 81% | K: 0%<br>RWLS: 19% | K: 95%<br>RWLS: 71% | K: 5%<br>RWLS: 29% |
| $sd_i = 0.2x_i$ | K: 100%<br>RWLS: 81% | K: 0%<br>RWLS: 19% | K: 92%<br>RWLS: 57% | K: 8%<br>RWLS: 43% |
| $sd_i = 0.5x_i$ | K: 100%<br>RWLS: 80% | K: 0%<br>RWLS: 20% | K: 86%<br>RWLS: 60% | K: 14%<br>RWLS: 40% |

Table 3.10: False positives and false negatives for the experiment in Figure 3.24

| Standard deviation | True negatives | False positives |
|---|---|---|
| $sd_i = 0.1x_i$ | K: 78%<br>RWLS: 100% | K: 22%<br>RWLS: 0% |
| $sd_i = 0.2x_i$ | K: 75%<br>RWLS: 100% | K: 25%<br>RWLS: 0% |
| $sd_i = 0.5x_i$ | K: 55%<br>RWLS: 100% | K: 45%<br>RWLS: 0% |

preferable to the faster Kalman-filter learning. This is the case, for instance, when a system operates close to the bounds specified in its requirements—a situation that is likely to be common in practice, since using components that provide a lot of slack is uneconomical. One such scenario is illustrated for our TS system in Figure 3.24, which shows experimental results obtained for the analysis of the requirement

> "The expected time of workflow executions that consist of an invocation of the *primaryDiagnostic* service followed by an invocation of the *logProblem* service must not exceed $160ms$."

when the execution times of the two services are close to bounds beyond which the requirement is violated. Note that these workflow executions correspond to path $\pi_1$ from Section 3.4.2.2, and that the expected execution time for this path is $time(\pi_1) = x_4 + x_5 \leq 160ms$. Under these circumstances, the oscillations in the $x_4$ and $x_5$ estimates produced by the Kalman filter introduce numerous false positives, whereas the RWLS filter does not (Table 3.10). This limitation of using the Kalman filter with our learning technique is made worse by increases in the standard deviation for the actual execution times of the two services.

As shown in Figure 3.24, short-lived downgrades in service performance leading to temporary requirement violations are not identified as such when the RWLS filter is used. This is a limitation of RWLS that needs to be taken into account when choosing between the two filters. If such short-lived downgrades in service performance are unlikely (e.g., because services that experience problems become unavailable or are

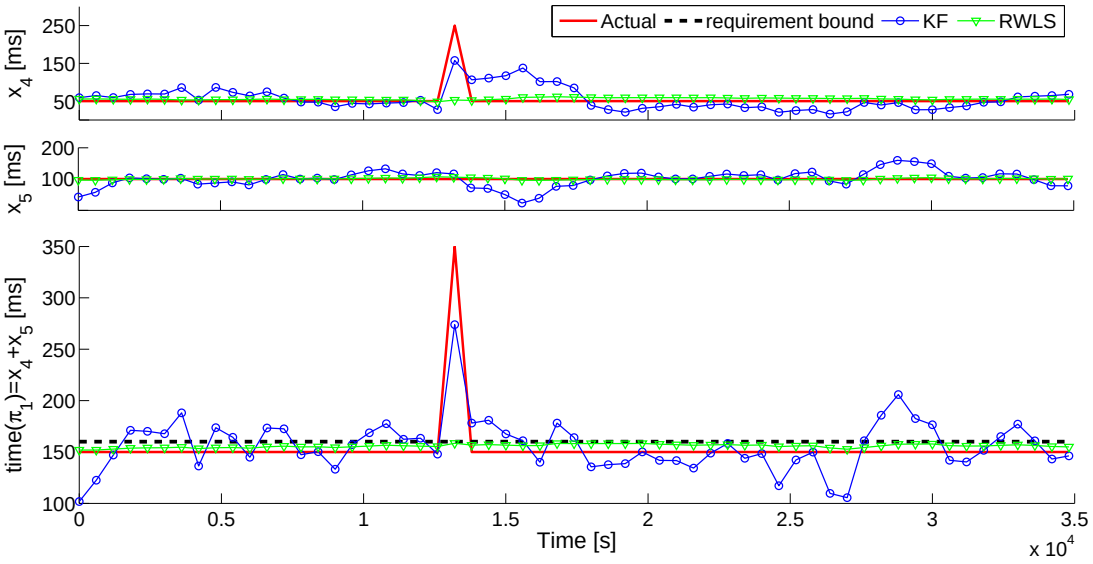


Figure 3.24: Scenario in which the RWLS filter is a better option

performing poorly for long periods of time before recovering) or are not of interest (e.g., because short periods of degraded operation are acceptable) then the RWLS filter is the best option.

## 3.5 Summary

This chapter has presented several approaches to the online learning of Markovian model parameters. These techniques are useful in scenarios where state transition probabilities and response times of the modelled system and its components are prone to changing dynamically.

First, we introduced a new algorithm for learning the state transition probabilities from runtime observations of the behaviour of the modelled system. Our Bayesian derived technique weighs observations based on their age, to account for the fact that older observations are less relevant than more recent ones.

Second, the learning technique summarised above was enhanced by adjusting its parameters dynamically depending on the frequency of the observations. This enhanced adaptive learning technique leads to a faster and more accurate inference of the transition probabilities than that provided by existing methods. Rigorous theoretical results link the parameters chosen dynamically by our learning methods to the expected error in the accuracy of the learnt state transition probabilities. This allows the configuration of the adaptive learning method so that it yields results within an acceptable expected error range.

Finally, a novel approach that uses optimal filters to establish the performance related QoS properties for black-box component based systems was introduced. This approach establishes the expected execution time of individual components and uses these values to determine the expected durations of rare/exceptional tasks (too infrequent to be measured, but highly critical), through the verification of costs/reward structures of Markov chain models. However, this technique is only applicable in scenarios where very large numbers of observations are made available and large number of paths in the model can be observed.(What constitutes a sufficiently large number of observations is quantified in Section 3.3.1 and 3.3.2)

# Chapter 4

# Online learning of Markov chain structure for service-based systems

## 4.1 Introduction

As complex embedded applications are increasingly used to build modern software, so too is the need to build accurate models that can be formally verified to ensure QoS compliance during operation. In the previous chapter we presented techniques that update the parameters of Markovian models at runtime to synchronise the model with the actual behavior of the implemented system. However, using these techniques assumes that the structure of the model is known. When this assumption does not hold, such as in the case of "black-box" software systems, the approaches presented in the previous chapter cannot be applied. To address this limitation, we introduce a new technique for learning a (discrete-time) Markov chain (MC) model of service-based system (SBS) by making runtime observations.

## 4.2 Related work

Model learning techniques are being used in many applications to accurately construct and later update model parameters for verification purposes. For example, in [44, 123, 137] automata learning techniques are used to extract behavioral models of software systems. In [137] Raffelt et al. introduces a modular framework for automata learning based on several variants of Angluin's algorithm. This framework, called LearnLib, was originally designed to systematically build finite state machine models of real-world systems. More recently, LearnLib has become a platform for experimenting with different learning algorithms, supporting the statistical analysis of their characteristics in terms of the learning effort, run time and memory consumption.

Sen et al. [146] base their learning of system models on the *Alergia* algorithm that is used in learning finite deterministic stochastic automata, a technique that is further developed by Mao et al. [123]. Both [123] and [146] use independent finite samples of the execution runs of the system to learn the system model, each starting at the same initial state. While is possible only under laboratory conditions, it is unfeasible when a system is deployed and operates in real world environment. Finally, Chen et al. [44] investigates the learning of system models by passively observing a single, ongoing execution of the system, i.e., from data that consists of a single, long observation

sequence, which may start at any point in the operation of the system.

On the other hand, Roshandel et al. [49, 140, 141] propose a framework for component reliability estimation at the software architectural level. They use the states in a behaviour model to correspond to states in the Hidden Markov model (HMM), such that the dynamic behaviour model becomes observations of the HMM. A HMM solver is used to leverage a set of synthesized or simulated training data and the Baum-Welch algorithm is used to learn the unknown parameters of the HMM.

The inferred models produced by the approaches in [44, 49, 123, 137, 140, 141, 146] represent formal representations of a software system and are used as input for probabilistic model checking tools that verify the system specifications at design time. However, unlike the models produced by the approach introduced in this chapter, these models (and thus the results of their verification) may easily become inaccurate when the application is running in a dynamic environment.

In a related study, Ghezzi et al. in [85] extract a MC model based on the users' interaction history of navigational behaviors given in the form of a log file for a Web application. The Markov model is built incrementally by examining each entry of the log file once. Our approach resembles the one in [85], as it constructs the underlying system model based on observations taken from the running system, to drive progressive maintenance and adaptation of the modelled system as it evolves. However, our approach is set in the context of service-based systems (SBSs), and it automates the insertion of success and fail states for each of the SBS operations.

In [44] Chen et al. delay the learning of the system model until a possible initial "burn-in" phase has passed, such that the inference process starts with observation sampled from a stationary distribution. In contrast, our approach starts learning the model as soon as the system starts running.

Sun et al. in [150] record all user operations, and use an inference engine to infer a user's intention in a model transformation task. A transformation pattern is generated from the inference, specifying the precondition of the transformation and the sequence of operations needed to realize the transformation. In contrast, our learning technique described in Section 4.3 uses a predefined MC model pattern and runtime observations of systems events to generate new states in the model. When a SBS operation is invoked for the first time, three new states are added to the MC and their associated labels. The three states correspond to the invocation of the operation, to its successful completion, and to its failure.

Last but not least, a unique characteristic of our approach is its use of a distance function to compare the model versions learnt after every $N$ observations of SBS operations, where $N \gg 1$ is a parameter of the approach. When the distance between two successive model versions compared in this way drops below an *acceptable distance* $\epsilon > 0$, the latest model is deemed "stable" enough for use in verification, and ultimately to support adaptation decisions. By varying the acceptable distance parameter $\epsilon$ and the number of observations $N$ between successive *checkpoints*, we are able to provide insight into the impact that they have on the ability to accurately analyse the compliance of system with a range of requirements.

## 4.3 Learning technique

We now introduce the online algorithm for learning a MC model of an SBS workflow comprising $m \geq 1$ operations executed by external services. We assume that $op_1, op_2, \cdots, op_m$ represent the (String-valued) names of the $m$ SBS operations. The

```
op_i_proxy_wrapper(...) {
    UPDATE(op_i, "invoke", N, ε)
    try {
        op_i_proxy(...)
        UPDATE(op_i, "succ", N, ε)
    } catch {
        UPDATE(op_i, "fail",N, ε)
    }
}
```

Figure 4.1: Using a service proxy wrapper to instrument workflows with model learning capabilities

output of the algorithm is a MC $\mathcal{M} = (S, s_0, P, L)$ over the atomic proposition set

$$\begin{aligned} AP = \{ & \text{“invoke”}^\frown op_1, \text{“succ”}^\frown op_1, \text{“fail”}^\frown op_1, \\ & \text{“invoke”}^\frown op_2, \text{“succ”}^\frown op_2, \text{“fail”}^\frown op_2, \\ & \ldots, \\ & \text{“invoke”}^\frown op_m, \text{“succ”}^\frown op_m, \text{“fail”}^\frown op_m, \\ & \text{START}, \text{STOP}\}. \end{aligned}$$

Algorithm 2 shows the pseudocode for our learning technique. First, lines 1 to 5 initialise the variables used by the algorithm as follows:

- the state set $S$ initially contains a start state $s_{START}$ and a stop state $s_{STOP}$;
- the labelling function $L$ associates appropriate labels to $s_{START}$ and $s_{STOP}$;
- the partial function[1] $transitions : S \times S \nrightarrow \mathbb{N}$ that counts the transitions between states ( i.e., for any $s, s' \in$ dom $transitions$, $transitions(s, s')$ represents the number of observed transitions from state $s$ to state $s'$) initially contains no mapping;
- the current state *crtState* is initially *null*;
- the observation *counter* is initially zero;
- the learnt model is initially *null*.

The function UPDATE in lines 7–24 is called before and after each invocation of one of the $m$ operations of the workflow, as well as at the start and end of the workflow execution. Note that calling UPDATE before and after operation executions can be easily done by using wrappers around standard services proxies (Figure 4.1).

The arguments for this function are:

- the name *op* of the invoked operation, i.e., one of $op_1, op_2, \cdots, op_m$, “START” or “STOP”;

[1] A partial function is denoted by $\nrightarrow$

**Algorithm 2** MC learning algorithm

1: $S \leftarrow \{s_{START}, s_{STOP}\}$

2: $L \leftarrow \{s_{START} \mapsto \{\text{“}START\text{”}\}, s_{STOP} \mapsto \{\text{“}STOP\text{”}\}\}$

3: $transitions \leftarrow \{\}$

4: $crtState \leftarrow null$

5: $counter \leftarrow 0$

6: $\mathcal{M} \leftarrow null$

7: **function** UPDATE($op$, $type$, $N$, $\epsilon$)

8: **if** $type =$ “invoke” $\wedge\ s_{\text{“invoke”}^\frown op} \notin S$ **then**

9: $S \leftarrow S \cup \{s_{\text{“invoke”}^\frown op}, s_{\text{“succ”}^\frown op}, s_{\text{“fail}^\frown op}\}$

10: $L \leftarrow L \cup \{s_{\text{“invoke”}^\frown op} \mapsto \{\text{“invoke”}^\frown op\}, s_{\text{“succ”}^\frown op} \mapsto \{\text{“succ”}^\frown op\},\ s_{\text{“fail}^\frown op} \mapsto \{\text{“fail”}^\frown op\}\}$

11: **end if**

12: **if** $crtState \neq null$ **then**

13: **if** $(crtState, s_{type^\frown op}) \in$ dom $transitions$ **then**

14: $transitions(crtState, s_{type^\frown op})$++

15: **else**

16: $transitions \leftarrow transitions \cup \{(crtState, s_{type^\frown op}) \mapsto 1\}$

17: **end if**

18: **end if**

19: $crtState \leftarrow s_{type^\frown op}$

20: **if** $++counter \bmod N = 0$ **then**

21: $\mathcal{M} \leftarrow$ CHECKMODEL($S, transitions, L, N, \epsilon$)

22: **end if**

23: **return** $\mathcal{M}$

24: **end function**

- the *type* of the UPDATE invocation, i.e., one of “invoke” (used when UPDATE is called before the execution of the operation), “succ” or “fail” (used for UPDATE calls after the successful or failed execution of the operation, respectively), and “” (used when UPDATE is invoked for “START” or “STOP”);
- $N \gg 1$ and $\epsilon$, two parameters of the learning algorithm whose roles are explained below.

The function UPDATE works as follows. First, the if statement in lines 8–11 checks if UPDATE was called for the first time for one of the $m$ SBS operations, in which case three new states are added to the state set $S$ in line 9, and sets of labels are associated with them in line 10. The three states correspond to the invocation of the operation, to its successful completion, and to its failure.

Second, the if statement in lines 12–18 updates the *transitions* counters, except for the first invocation of UPDATE, when *crtState* is *null*. The update involves incrementing

the counter of transitions between *crtState* and the state corresponding to the new invocation of UPDATE (i.e., $s_{type^\frown op}$) if such a counter exists, or the creation of this counter with an initial value of 1, otherwise. The value of *crtState* is then updated in line 19.

Finally, the if statement in lines 20–22 invokes the CHECKMODEL function in Algorithm 3 after every $N$ calls to UPDATE, in order to check whether the learnt model is sufficiently stable to be used (in which case CHECKMODEL returns the learnt model) or not (in which case CHECKMODEL returns *null*). The model (or the value '*null*') obtained from CHECKMODEL is returned by UPDATE in line 23.

The CHECKMODEL function comprises two parts. The first part (lines 4–18) calculates the relative frequency (i.e., the estimate probabilities) of the transitions from each state $s_1 \in S$ to every state $s_2 \in S$. To this end, the for loop in lines 6–10 counts the number of transitions from $s_1$ to other states, and the for loop in lines 11–17 calculates the relative transition frequencies from $s_1$ to other states, when such transitions exist.

The second part of the function starts by calculating the "distance" between the current version of the learnt model and the version from the previous invocation of CHECKMODEL (lines 19-25). This is done only if the model is not already deemed *learnt* and the state sets corresponding to the current and previous model assembled by the function are identical.[2] The distance is calculated as the sum of the differences between the current and the previous transition probability estimate associated with each pair of states. If this distance, normalised through division by the number of transitions observed, drops below $\epsilon$, the model is deemed *learnt* (line 24), and is returned in line 28. Otherwise, the function returns *null* (line 30). Finally, the current state set and estimate transition probabilities are retained (in line 26) for use during the next invocation of CHECKMODEL.

## 4.4 Complexity analysis

To analyse the memory complexity of Algorithms 2—3, note that $\#S \leq 3m + 2$. Accordingly, the domain of *transition* in UPDATE and the relative frequency arrays $p$ and *oldP* in CHECKMODEL have at most $(3m+2)^2$ elements, so the memory complexity of the technique is $O(m^2)$.

Checking set memberships in lines 8 and 13 of UPDATE is at most linear in the size of the two sets (i.e., $S$ and dom *transitions*, so at most $O(m)$. The for loops in lines 4–18 and 21–23 from CHECKMODEL operate with each pair of states from $S$, i.e, with at most $(3m + 2)^2$ pairs of states. Accordingly, the combined time complexity of one invocation of the two algorithms is also $O(m^2)$.

## 4.5 Evaluation

### 4.5.1 Case study

We use a telehealth service-based system (SBS) originally introduced in [16] and also used [28, 29, 66] to evaluate the efficiency (i.e., time to detect all the states and the transitions between the states of the model of the system) and accuracy (i.e., ability to

[2]Since UPDATE can add new states to $S$ but cannot remove or modify existing states, checking that the two sets have the same number of elements is sufficient.

**Algorithm 3** Transition probability calculator and model comparison algorithm

1: $oldS \leftarrow \{\}$, $oldP \leftarrow null$
2: $learnt \leftarrow false$
3: **function** CHECKMODEL($S$,$transitions$,$L$,$N$,$\epsilon$)
4: **for** $s_1 \in S$ **do**
5: $nTrans \leftarrow 0$
6: **for** $s_2 \in S$ **do**
7: **if** $(s_1, s_2) \in$ dom $transitions$ **then**
8: $nTrans \leftarrow nTrans + transitions(s_1, s_2)$
9: **end if**
10: **end for**
11: **for** $s_2 \in S$ **do**
12: **if** $nTrans \neq 0 \wedge (s_1, s_2) \in$ dom $transitions$ **then**
13: $p(s_1, s_2) \leftarrow transitions(s_1, s_2)/nTrans$
14: **else**
15: $p(s_1, s_2) \leftarrow 0$
16: **end if**
17: **end for**
18: **end for**
19: **if** $\neg learnt \wedge \#S = \#oldS$ **then**
20: $dist \leftarrow 0$
21: **for** $(s_1, s_2) \in S \times S$ **do**
22: $dist \leftarrow dist + \mid p(s_1, s_2) - oldP(s_1, s_2) \mid$
23: **end for**
24: $learnt \leftarrow dist/\#(\text{dom } transactions < \epsilon$
25: **end if**
26: $oldS \leftarrow S$, $oldP \leftarrow P$
27: **if** $learnt$ **then**
28: **return** $(S, s_{\text{START}}, p, L)$
29: **else**
30: **return** null
31: **end if**
32: **end function**

detect all/most states and the transition probabilities between the states of the model) of the learning algorithm from the previous section. In this SBS, the vital parameters of a patient are periodically measured by a wearable device and analysed by third-party medical services. The result of the analysis may trigger the invocation of an alarm service (that determines, for instance, the dispatch of an ambulance), may lead to the

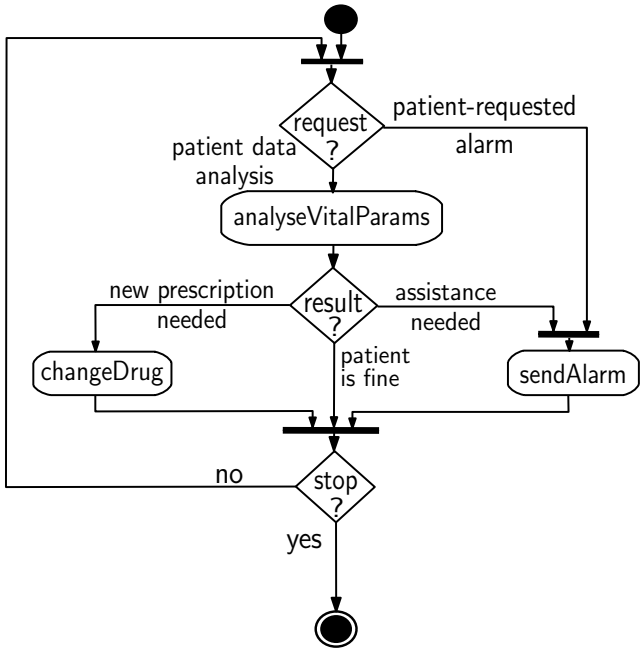


Figure 4.2: UML activity diagram of the telehealth service-based system

invocation of a pharmacy service to deliver new medication to the patient, or may confirm that the patient is fine. In addition, the patient can initiate an alarm by using a panic button on the wearable device. The workflow of the telehealth SBS is shown in Fig 4.2, and we will consider that it must comply with the following five reliability requirements:

**R1** The probability that one execution of the workflow ends in a alarm failure is at most $p_{R1} = 0.00001$.

**R2** The probability that one execution of the workflow ends in a service failure is at most $p_{R2} = 0.008$.

**R3** The cost of one execution of the workflow is at most $R_{R3} = 2.5$.

**R4** The probability that the analysis service fails within 100 seconds is at most $p_{R_4} = 0.002$.

**R5** The probability that an invocation of the analysis service is followed by an alarm failure is at most $p_{R_5} = 0.0002$.

Note that although we provided a full description of the system in the remainder of this section we will assume that it consists of only three SBS operations.

### 4.5.2 Experiment setup

A Java implementation of the telehealth SBS workflow was integrated using a purpose-built proxies that we implemented to ensure that the UPDATE function in Algorithm 2 was called before and after each invocation of a service. The MC models were learnt for multiple values of the acceptable distance parameter $\epsilon = \{0.05, 0.02, 0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.00$
Each experiment was carried out until the model was deemed stable, and we report the number of workflow iterations required for this. All experiments were run on a 2.66 GHz Intel Core 2 Duo Macbook Pro computer.

The learnt models were analysed to verify the QoS requirements $R1 - R5$ described in Section 4.5.1. We used the probabilistic model checker PRISM for this purpose. The probabilistic temporal logic formulae for requirements $R1 - R5$ are:

$R1 \quad P =?[!(\text{“}STOP\text{”})U(\text{“}failAlarm\text{”})]$,

$R2 \quad P =?[!(\text{“}STOP\text{”})U(\text{“}failAlarm\text{”}|\text{“}failDrug\text{”}|\text{“}failAnalysis\text{”})]$,

$R3 \quad R =?[F\text{“}STOP\text{”}]$,

$R4 \quad P =?[trueU <= 100\text{“}failAnalysis\text{”}]$,

$R5 \quad filter(max, P =?[!\text{“}STOP\text{”}U\text{”}failAlarm\text{”}], \text{“}invokeAnalysis\text{”})$.

Figure 4.3 depicts the parts of the Markov model learnt at several stages of the learning process. First, Figure 4.3a illustrates the initial state of the model (i.e., before the first invocation of UPDATE). Next, Figure 4.3b shows the model components inferred after one workflow iteration. This workflow iteration happened to be a successful invocation of the analysis service that found the patient to be fine, so the states *failedAVP* and *successAVP* were also created. There are no transitions associated with the *failedAVP* state because in this single iteration the analysis service was invoked successfully. Figure 4.3c illustrates the model inferred after five workflow iterations, where 96% of the the *analyseVitalParams* service invocations were successful, 60% of the analysis results confirmed that the patient was fine, and 40% required new medication. Finally, Figure 4.3d illustrates the inferred model after ten workflow iterations. In this scenario all the states of the Markov chain have been discovered, and a structurally complete model has been inferred, although the state transition probabilities are still very coarse estimates of the actual transition probabilities for the system. For comparison, Figure 4.4 depicts the manually derived Markov model, which of course is unknown in the scenarios where the learning technique is applied.

### 4.5.3 Results

This section presents the experimental results from the evaluation of our learning technique and of the role of its two parameters, $\epsilon$ and $N$. The probabilistic model checker PRISM is used to quantitatively establish the values of properties $R1$–$R5$ from the previous section given “stable” models learnt using different range of $\epsilon$ and $N$ values. For comparison, we also analysed the manually derived model from Figure 4.4 to determine the actual values of $R1$–$R5$. This provided insight into how the two parameters should be configured in order to obtain accurate estimates for the QoS properties that requirements $R1$–$R5$ are based on.

**Evaluation of the effect of $\epsilon$ on the accuracy of the learnt model** The boxplots in Figure 4.5 represent the range of values for $R_n$, $1 \leq n \leq 5$, obtained through analysing 20 Markov models learnt for each value of $\epsilon$ in the set $\{0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05\}$, and for a fixed $N = 50$. These results show that the estimate $R1$–$R5$ property values are distributed in the vicinity of their actual values for all examined values of $\epsilon$. However, the interquartile range for the estimate $R1$–$R5$ property values grows with larger $\epsilon$ values. Thus, for a small value of $\epsilon$ (approximately for $\epsilon$ not exceeding 0.001), the learnt model is “closer” to the actual model, and can be used to obtain accurate estimates of the properties of interest.

Obtaining more accurate models by using low values of $\epsilon$ comes at a cost. Table 4.1 shows, for each $\epsilon$ value used in our experiments, the average *counter* value from Algorithm 2 when the model was deemed “stable”, averaged over the 20 experiments

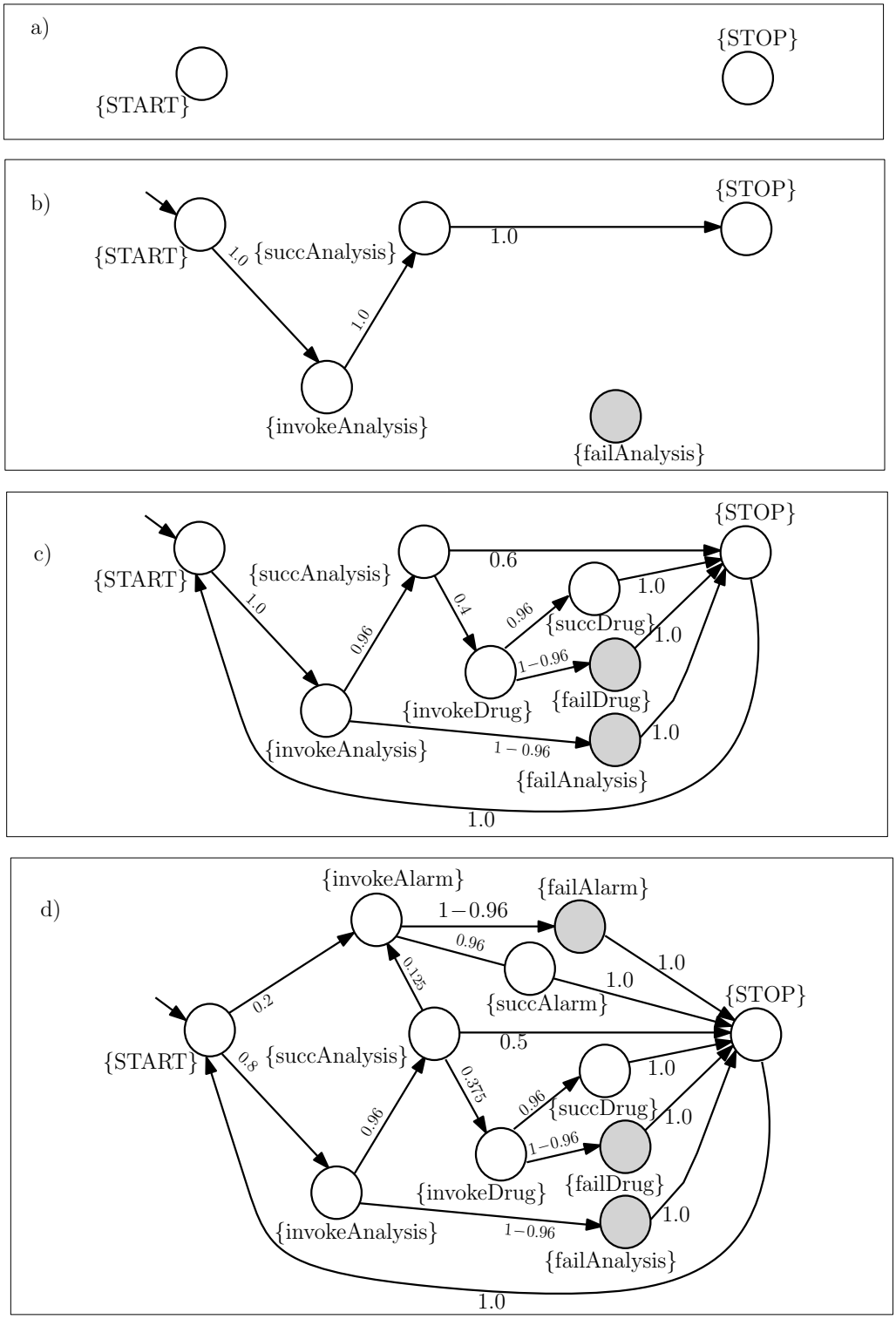


Figure 4.3: Markov model learnt after: a) no workflow iterations; b) one workflow iteration; c) five workflow iterations; and d) 10 workflow iterations.

carried out for each $\epsilon$ value. As expected intuitively, *counter* increases with decreasing $\epsilon$. This means that whilst the learnt model is closer to the actual value for a small $\epsilon$ than for a larger $\epsilon$, the time taken to reach a stable model is much longer, e.g., for $\epsilon = 0.0005$ the average *counter* value is 3280.8, compared to an average *counter* value of 236.0 for the less accurate learnt model for $\epsilon = 0.02$.

**Evaluation of the effect of $N$ on the accuracy of the learnt model** The boxplots in Figure 4.6 represent the range of values for $R_n$, $1 \leq n \leq 5$, obtained through

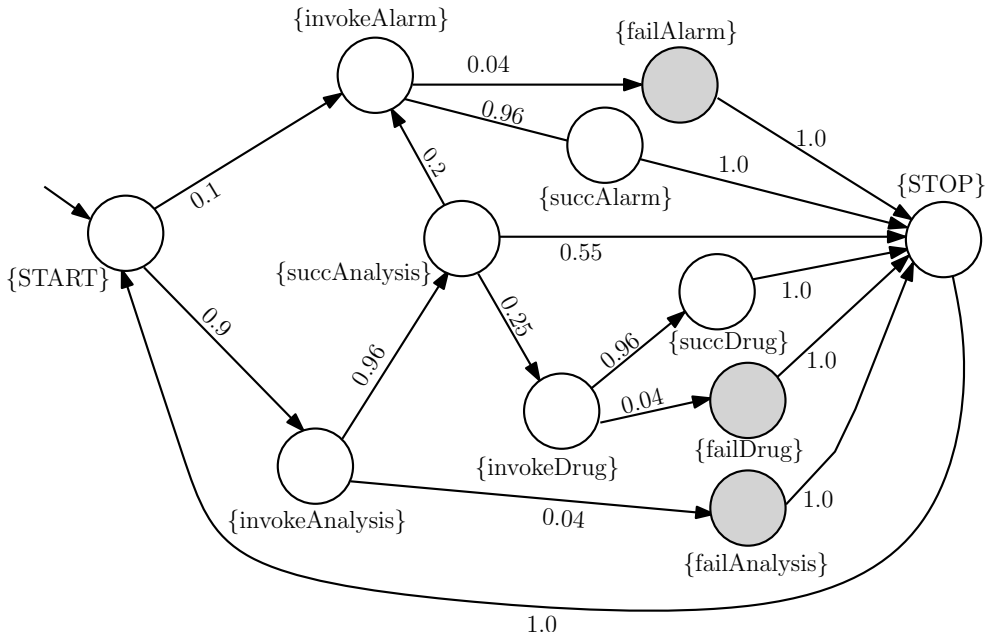


Figure 4.4: Manually derived Markov model for the telehealth workflow

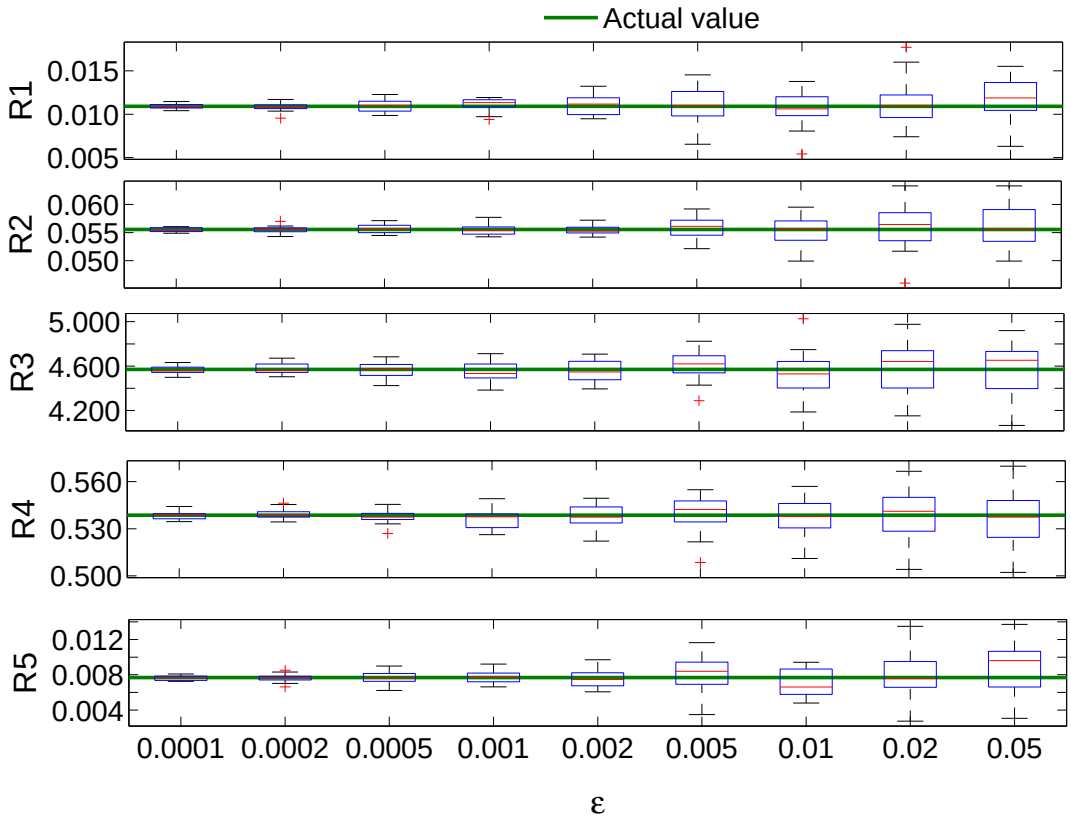


Figure 4.5: Analysis of $R1 - R5$ with varying $\epsilon$ and $N = 50$

analysing 20 learnt models for each value of $N$ in the set $\{15, 50, 100, 500\}$, with a fixed $\epsilon = 0.0005$. A comparison of the interquartile range of the box plots shows that the interquartile range decreases as $N$ grows, suggesting that for large values of $N$ the learnt model is closer to the actual model. However, this trend becomes less pronounced beyond $N = 100$, as the model obtained for this value of $N$ is already yielding very

Table 4.1: Average *counter* value when the learnt model becomes "stable" for different $\epsilon$ values

| $\epsilon$ | Average counter value |
|---|---|
| 0.0001 | 11793.0 |
| 0.0002 | 6976.5 |
| 0.0005 | 3280.8 |
| 0.001 | 1875.9 |
| 0.002 | 1127.5 |
| 0.005 | 591.2 |
| 0.01 | 361.9 |
| 0.02 | 236.0 |
| 0.05 | 149.3 |

Table 4.2: Average *counter* when the learnt model becomes "stable" for different $N$ values

| $N$ | Average counter value |
|---|---|
| 15 | 1752.9 |
| 50 | 3280.8 |
| 100 | 4580.0 |
| 500 | 14665.0 |

accurate results for all properties $R1$–$R5$.

As before, obtaining a more accurate model (this time through using a larger $N$) requires more observations to be made. The number of observations until the learnt model is deemed "stable" increases rapidly with $N$, as shown in Table 4.2, which gives the average *counter* values from Algorithm 2 for when the model is deemed stable. The results summarised in Table 4.2 also show that by reducing the value of $N$ from 50 to 15, the average *counter* value is reduced by over 45%. Furthermore, this reduction is achieved without too significant an impact on the ability of the learnt model to produce relatively accurate estimates of the analysed properties. This means that when selecting a small $\epsilon$ value together with a small $N$, the waiting time to declare a reasonably accurate model "stable" could potentially be almost halved. However, note that for very small values of $N$ (i.e., $N \leq 10$ for this system) the number of observations between adjacent checkpoints becomes very small and runs the risk of having the learnt model declared "stable" too early—often before all states and state transitions are identified.

### 4.5.4 Further applications

We carried out additional experiments to ensure the approach is not specific to the example SBS from the case study, and to increase our confidence that it is also useful for other SBSs. We selected the following workflows, used by a number of projects in

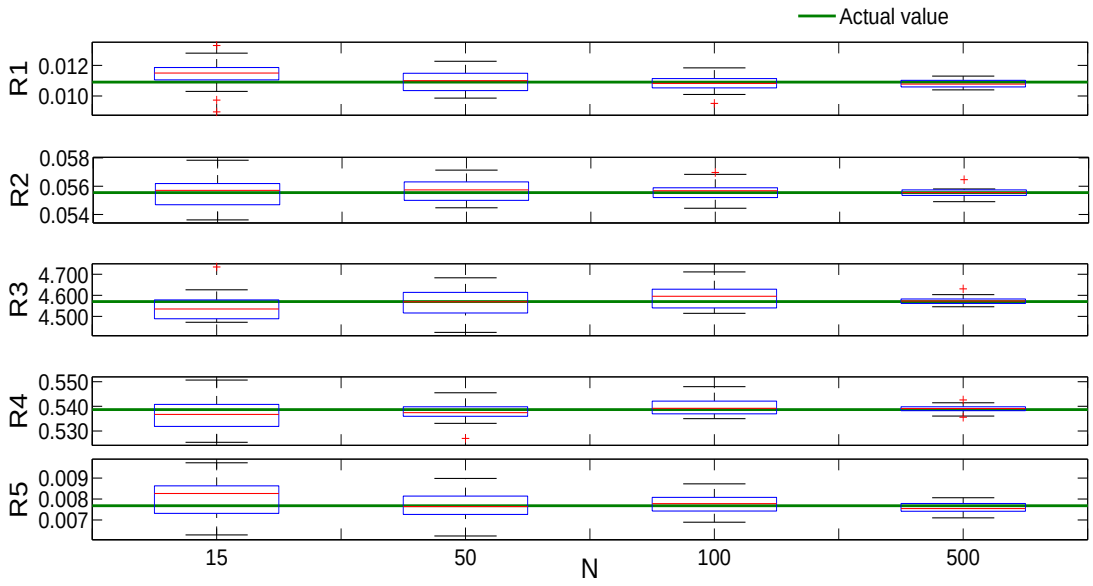


Figure 4.6: Analysis of properties $R1$–$R5$ with varying $N$ and $\epsilon = 0.0005$

this area:

1. TeleHealth—the healthcare case study described in Section 4.5.1, and previously used in [16, 28, 29, 66];

2. Ecommerce—the e-commerce workflow obtained from [74];

3. TravelPlanner—the travel assistant workflow derived from the state-chart representation presented in [164].

We repeated the experiments described in Section 4.5.3 for the additional two service-based systems mentioned above. The numbers of requests (i.e., SBS workflow executions) that need to be observed in order to learn a stable Markov model for different values of $\epsilon$ (when the checking interval parameter is fixed, $N = 45$) is shown Figure 4.7. Somewhat unexpectedly, this number did not increase with the number of states for this model. At a closer examination, it turned out that the factor that influences the required number of observations is the fraction of states with multiple outgoing transitions. This is explained by the fact that states with a single transition will always have an exact transition probability (i.e., 1.0) associated with this transition, so they will bring a zero contribution to the distance *dist* calculated in lines 20–23 of Algorithm 3. The percentages of states with multiple outgoing transitions for our three SBS workflows are 15% for TeleHealth, 13% for Ecommerce and just 5% for TravelPlanner, and the dependency between these percentages and the number of observations required to learn the Markov model is shown in Figure 4.7 and in Table 4.3.

## 4.6 Summary

This chapter has introduced a new technique for learning the Markov model of a service-based system from runtime observations of its service invocations and of the outcome of these invocations. Our learning technique uses a distance function to compare the model versions learnt after every $N$ observations of SBS operations, and when the distance between two successive model versions is below a predefined value $\epsilon$, the latest model is

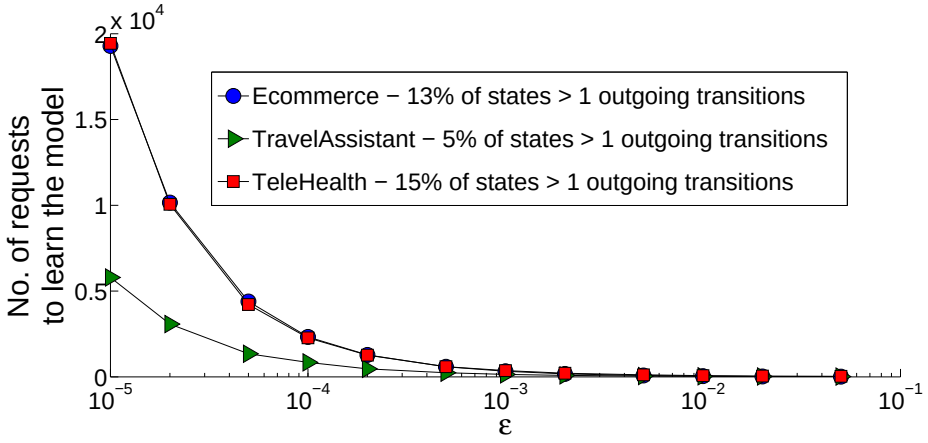


Figure 4.7: Scalability results for three SBS-systems, Telehealth - 11 states, Ecommerce-38 states and TravelPlanner - 20 states.

Table 4.3: Counter value associated with $\epsilon$ for each workflow, when model becomes stable with $N = 45$

| $\epsilon$ | Number of requests observed to learn model | | |
|---|---|---|---|
| | TeleHealth | Ecommerce | TravelPlanner |
| 0.5 | 586 | 206 | 338 |
| 0.2 | 586 | 206 | 338 |
| 0.1 | 586 | 206 | 338 |
| 0.05 | 587 | 206 | 338 |
| 0.02 | 587 | 206 | 338 |
| 0.01 | 601 | 241 | 340 |
| 0.005 | 803 | 351 | 372 |
| 0.002 | 1330 | 657 | 537 |
| 0.001 | 2111 | 1150 | 757 |
| 5.0E-4 | 3394 | 1960 | 1089 |
| 2.0E-4 | 6418 | 3855 | 1868 |
| 1.0E-4 | 10922 | 6747 | 3019 |
| 5.0E-5 | 16443 | 12094 | 5011 |
| 2.0E-5 | 33445 | 25485 | 8360 |
| 1.0E-5 | 58119 | 43640 | 14347 |

deemed "stable" enough for use in verification, and ultimately to support adaptation decisions. By varying the two parameters, $N$ and $\epsilon$, in experiments that learnt the Markov model of a telehealth service-based system, we provided insight into the role of

these parameters in the learning of models that support the accurate analysis of the compliance of a system with a range of reliability and cost-related QoS requirements.

# Chapter 5

# Model-driven QoS management for service-based systems

An ever growing number of software applications are developed through the integration of third-party services deployed on remote cloud data-centres and accessed over the Internet. The application domains that adopted *service-based systems* (SBSs) as a *de facto* standard range from e-commerce to online banking and e-government. The approach reduces the time, cost and expertise required to develop software systems, lowering the entry barrier for the providers of new applications.

There is, however, a limitation to these advantages: remote third-party services tend to vary in reliability and performance over time. Even selecting the services that implement critical business workflows from the most reputable providers is not a guarantee that the resulting SBS will comply with its requirements at all times. Addressing this limitation through replacing underperforming services with functionally equivalent ones "on the fly" has preoccupied the research community for over a decade. The numerous solutions proposed by the ensuing research range from approaches that use intelligent control loops (e.g, [10, 29, 37, 127]) to approaches which emulate the cooperative behaviour of biological systems (e.g., [77, 143]).

Although these approaches to developing *self-adaptive* SBSs were shown to be effective in lab-based scenarios, none has yet been adopted in SBS engineering practice. The COntinual VErification (COVE) SBS development framework introduced in this chapter aims to reduce this gap between state-of-the-art research and the current state of practice by integrating and exploiting key benefits of several software engineering paradigms. Similar to other self-adaptive SBS frameworks (e.g., [29]), COVE selects the service used to execute each operation of an SBS workflow at runtime, from a set of services that provide the same functionality with different levels of reliability and at different cost. However, COVE has a number of unique advantages over the existing approaches to developing and operating self-adaptive SBSs:

1. COVE self-adaptive SBSs are *self-verifying*, i.e., they employ continual formal verification to select the service combination that guarantees the realisation of the SBS reliability requirements with minimal cost. This verification uses an embedded version of the quantitative model checker PRISM to analyse a (discrete-time) Markov chain model of the SBS workflow.

2. The Markov chain model used by COVE is updated online to reflect changes in the service reliability and in the frequency with which the SBS operations are invoked.

3. The continual verification, model updating and service selection capabilities of COVE are fully automated, and are provided by a combination of reusable and automatically generated software components.

4. The tool-supported COVE development process resembles the traditional SBS development process, except that: (a) the COVE proxy is synthesised from a set of web service WSDL definitions instead of a single one; and (b)needs to specify the functionally equivalent methods of the web services in a preliminary, GUI-supported step of the generation process. These COVE features are intended to ensure that practitioners can use the framework with little learning effort, although additional studies are needed to confirm that this is the case.

The rest of this chapter is organised as follows. Related work is discussed in Section 5.1. Section 5.2 introduces a telehealth service-based system that is used to present the architecture of a COVE self-verifying SBS in Section 5.3, and to illustrate the application of the COVE development approach in Section 5.4. Next, Section 5.5 presents the continual verification process, and discusses options for dynamically switching between equivalent services in service-based systems. Finally, the effectiveness of the framework is evaluated in Secton 5.6.

## 5.1 Related work

The management and optimisation of SBS properties through dynamic service selection has been the focus of significant research over the past decade. The solutions proposed by this research include approaches that use intelligent control loops (e.g., [10, 37, 55, 127]) and approaches that emulate the cooperative behaviour of biological systems (e.g., [77, 143]). COVE belongs to the first category of approaches, so this section focuses on comparing our work with results from this area, and in particular with solutions that employ formal models that can represent SBSs accurately and in a realistic way.

The approaches proposed in [80, 124, 127, 145] use UML activity diagrams or directed acyclic graphs to synthesise simple performance models based on queuing networks [124, 127] or, like COVE, Markovian reliability models [80, 145]. These models are then used to establish the quality-of-service (QoS) properties of the analysed SBS systems. However, unlike these approaches, COVE also uses an adaptive learning technique to update the initial model based on observations of the system behaviour. The QoS-driven selection of services in self-adaptive service-based systems is addressed in [10, 37, 164]. For example, in [55] Cortellessa *et al.* present a service selection method based on the definition of a set of optimization models that are solved using Couenne solver.[1] The optimization models satisfy both costs and reliability constraints under the hypothesis that repair and mitigation actions can be undertaken to maintain service's reliability over a given threshold.

However, all of the above approaches lack the adaptive learning capabilities of COVE, and propose theoretical solutions that are hard to replicate in practical SBSs. In particular, approaches such as [10, 37, 116, 164] involve the optimisation of the service selection on a per request basis. These approaches require perfect knowledge of the QoS capabilities of the available services, which renders them ineffective in the scenarios targeted by COVE, where the characteristics of services need to be learnt from observations of their behaviour.

[1] Couenne (Convex Over and Under ENvelopes for Nonlinear Estimation) is a branch&bound algorithm that solves Mixed-Integer Nonlinear Programming (MINLP) problems (https://projects.coin-or.org/Couenne)

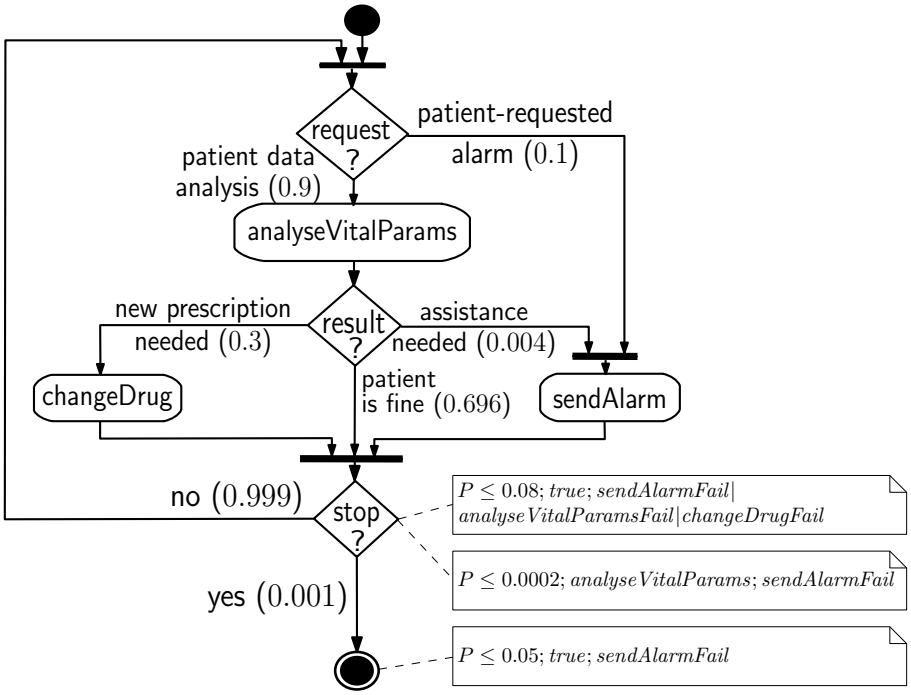


Figure 5.1: UML activity diagram of the telehealth SBS. Estimate *a priori* probabilities are associated with the outgoing edges of decision nodes, and comments defining the SBS requirements are associated with relevant nodes.

## 5.2 Telehealth service-based system

We will use the running example from Section 4.5.1 of the telehealth SBS taken from [28, 29, 66]. The workflow of the telehealth SBS is shown in Figure 5.1, and we will consider that it must comply with three reliability requirements:

$R_1$**:** The probability that one execution of the workflow ends in a service failure is at most $p_{R_1} = 0.8$.

$R_2$**:** The probability that an alarm failure occurs within $N = 10$ execution of the workflow is at most $p_{R_2} = 0.05$.

$R_3$**:** The probability that an invocation of the analysis service is followed by an alarm is at most $p_{R_3} = 0.0002$.

## 5.3 Architecture of a self-verifying service-based system

The architecture of a COVE self-adaptive SBS comprising $n \geq 1$ operations performed by remote third-party services resembles the implicit invocation style as defined by Garlan and Shaw in [82], and is depicted in Figure 5.2. The $n > 0$ COVE service proxies in this architecture interface the SBS workflow with sets of remote services such that the $i$-th SBS operation can be carried out by $m_i \geq 1$ functionally equivalent services. The runtime selection of the service for each SBS operation and, as in the case of traditional web service proxies, the interactions with the selected services are handled automatically

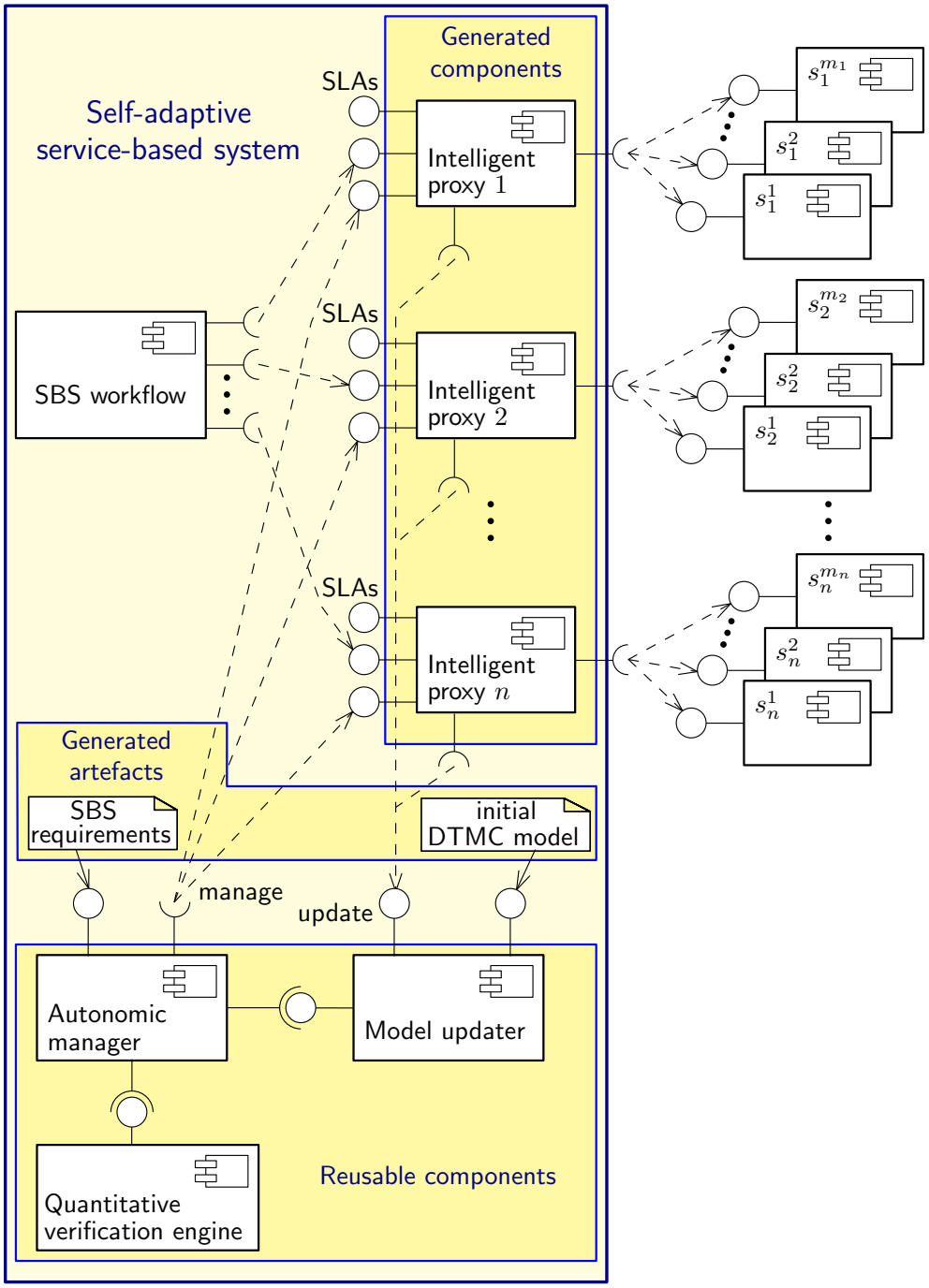


Figure 5.2: Architecture of a COVE self-verifying SBS

by the COVE proxies. When an instance of the $i$-th proxy is created, it is initialised with a sequence of (promised) *service level agreements* (SLAs) $sla_{ij} = (p_{ij}^0, c_{ij}), 1 \leq j \leq m_i$, where $p_{ij}^0 \in [0, 1]$ and $c_{ij} > 0$ represent the provider-supplied probability of success and the cost for an invocation of service $s_{ij}$, respectively.

The $n$ proxies are also responsible for notifying a *model updater* about each service invocation and its outcome. The COVE model updater starts from an initial Markov chain (MC) *model* of the SBS workflow, and uses our adaptive online learning technique presented in Section 3.3 to adjust the model parameters in line with these proxy notifications. The updated SBS model is then used by an *autonomic manager* that controls the services selected by the $n$ proxies, to ensure that the service combination which satisfies the SBS *requirements* with minimal cost is selected at all times. Accordingly, the COVE proxies, model updater and autonomic manager implements a monitor-analyse-plan-execute (MAPE) autonomic computing loop [105].

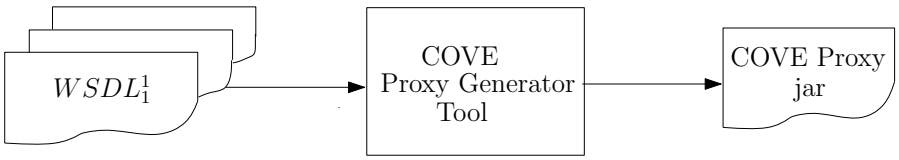


Figure 5.3: Diagram of COVE proxy synthesis process

A key advantage of our COVE framework is that its model updater and autonomic manager components are SBS-independent and therefore reusable across applications, while the SBS-specific proxies are generated automatically using a COVE software engineering tool described in Section 5.4.

## 5.4 Tool-supported framework for the engineering of service-based systems

The tool-supported COVE development process comprises three stages, each of which is described in detail in this section.

### 5.4.1 Stage 1: Proxy generation

In this stage, COVE service proxies are generated for the $n$ SBS operations. For the $i$-th operation, $1 \leq i \leq n$, the developer first selects $m_i \geq 1$ functionally equivalent services that implement the operation, but which may be associated with different levels of reliability and different costs (COVE does not support services discovered dynamically). Once the $m_i$ candidate services have been selected, the developer uses the COVE *proxy generator tool* to produce a (Java package) proxy for the $i$-th SBS operation. The functionality of the COVE proxy generator resembles that of standard web service proxy generators such as `WSDL2Java` and `wsdle2php`, except that: (a) the COVE proxy is synthesised from $m_i$ web service WSDL definitions instead of a single one; and (b) the developer needs to specify the functionally equivalent methods of the $m_i$ web services in a preliminary, GUI-supported step of the generation process. Figure 5.3 illustrates the COVE proxy synthesis process, and Figure 5.4 depicts the class diagram of the proxy generator and its associated auxiliary classes. The COVE proxy is extensible and additional methods can be implemented, e.g., to apply the optimal filter analysis from Section 3.4.

**Example 8.** Figure 5.5 depicts the generation of a COVE proxy for the `sendAlarm` operation of the telehealth SBS from Figure 5.1. Two "concrete" services deployed on Amazon EC2 virtual machines were selected as candidates for executing the "abstract" SBS operation, which is mapped to the `sendAlarm` method of the first service and to the `sendAlarm2` method of the second service. In the general case, a single COVE proxy could map each of several SBS operations to different methods belonging to a subset of the concrete services it relies upon.

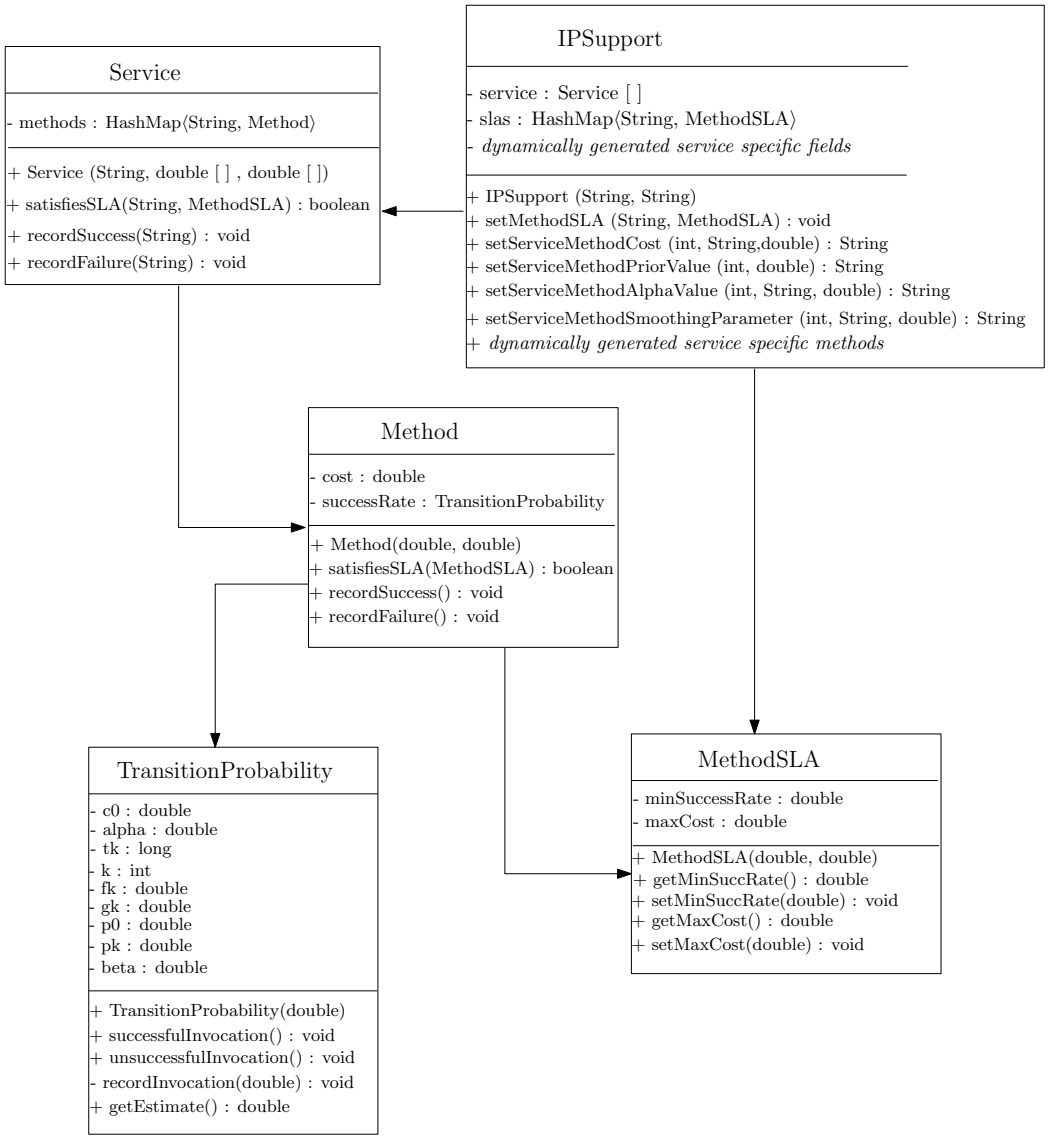


Figure 5.4: Class diagram of COVE proxy generator

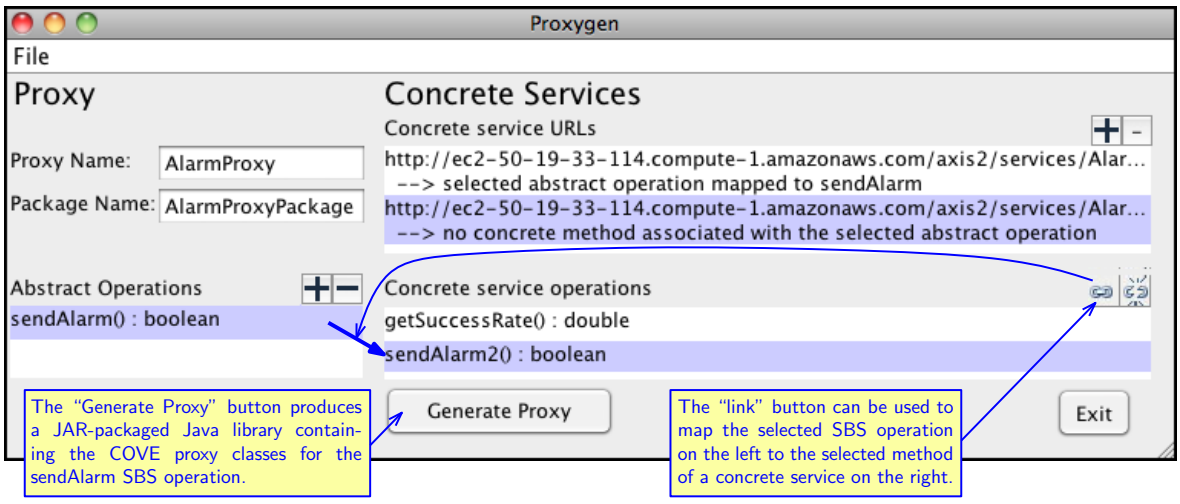


Figure 5.5: Proxy generation for the `sendAlarm` operation from Figure 5.1

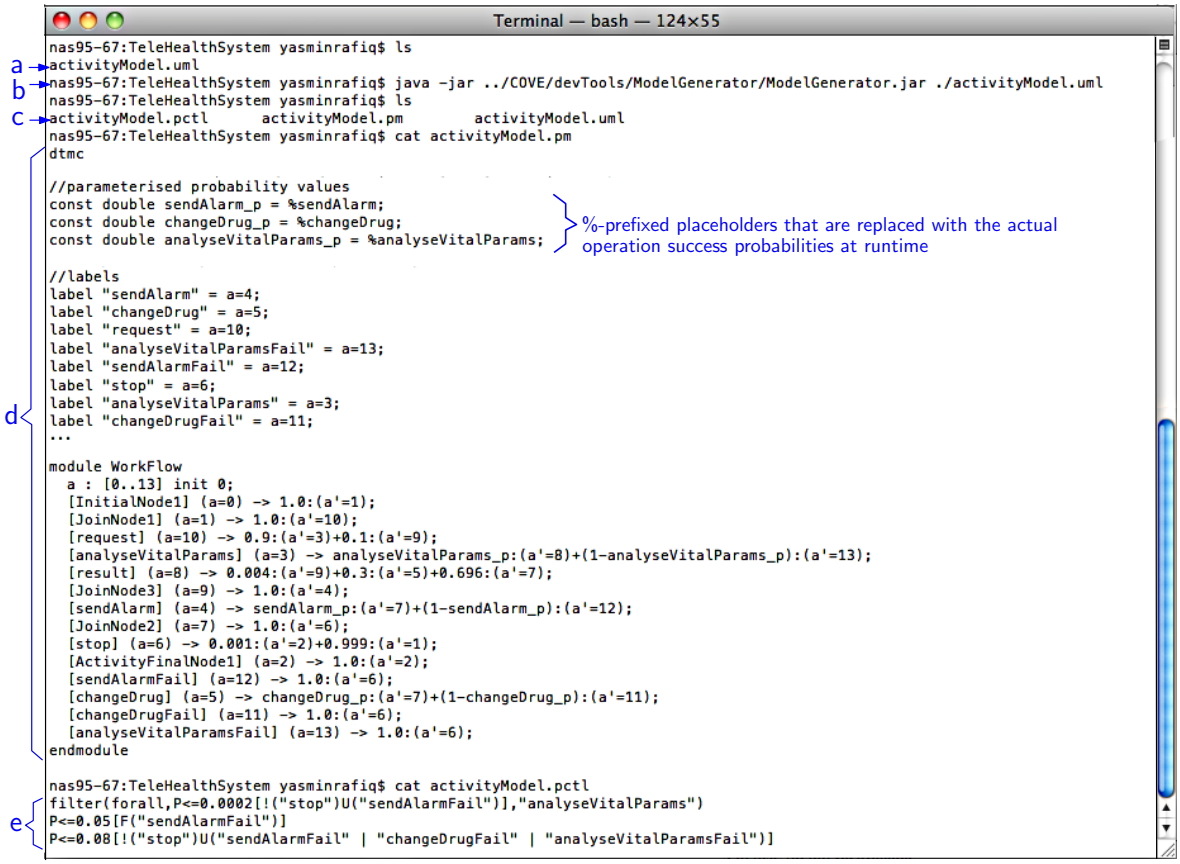


Figure 5.6: Using the Markovian model and PCTL requirements generator. The activity diagram **a** is used by the COVE tool in line **b**, to generate the **.pm** MC model file and the **.pctl** PCTL property files in line **c**, the two artefacts are shown in areas **d** and **e**, respectively.

### 5.4.2 Stage 2: Initial model construction and requirement formalisation

The second stage of the development process involves the construction of the initial Markov model used to configure the COVE model updater, and the formalisation of the SBS requirements used to configure the COVE autonomic manager. This development stage is supported by a COVE *model and requirements generator tool* that takes as input the XMI-encoded UML activity diagram of an SBS workflow, as shown in Figure 5.6 and produces:

- a cost-annotated Markov model $\mathcal{M} = (S, s_0, \mathcal{P}, L)$, $\rho : S \rightarrow R_{\geq 0}$ of the SBS, expressed in the PRISM high-level modelling language and in the format required by the COVE model updater; and
- the set of PCTL-encoded SBS requirements, in the format required by the COVE autonomic manager.

The four components of the Markov model $\mathcal{M}$ and the cost function $\rho$ are synthesised by the tool as described below:

**$S$:** A distinct state is included in $S$ (a) for each node in the activity diagram; and (b) for the failure of each of the $n$ SBS operations. The initial state $s_0 \in S$ corresponds to the start node in the diagram.

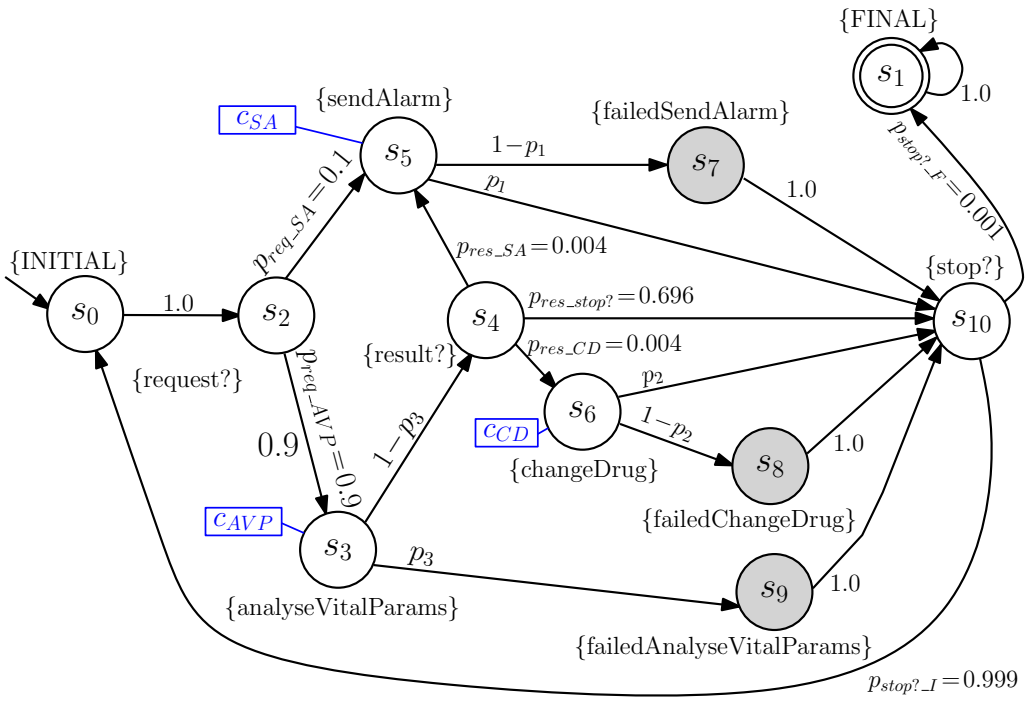


Figure 5.7: Initial Markov model for the SBS workflow from Figure 5.1

$P$**:** Given two states $s_x, s_y \in S$, the state transition probability $p_{xy}$ is set as follows:

- **(a)** if $s_x$ corresponds to the $i$-th SBS operation and $s_y$ corresponds to the failure state for the operation, then $p_{xy} = 1 - p_i$ (where $p_i \in [0, 1]$ is the probability of success for operation $i$, and represents a model parameter);
- **(b)** otherwise if $s_x$ corresponds to the $i$-th SBS operation and $s_y$ corresponds to the node that follows operation $i$ in the activity diagram, then $p_{xy} = p_i$;
- **(c)** otherwise, if $s_x, s_y$ corresponds directly to connected nodes X and Y from the UML activity diagram, then $p_{xy} = 1.0$ if the node associated with $s_x$ is not a decision node; if $s_x$ is a decision node, $p_{xy} = p\text{X\_Y}$ (where $p\text{X\_Y} \in [0, 1]$ is a model parameter initialised with the corresponding *a priori* probability estimate from the activity diagram);
- **(d)** otherwise, $p_{xy} = 0$.

$L$**:** Each state that corresponds to a node $X$ in the activity diagram is labelled $\{X\}$, and each state that corresponds to a failed operation $X$ is labelled $\{\text{Failed}X\}$.

$\rho$**:** A cost $c_X$ is associated with each state that corresponds to an operation $X$ from the activity diagram (where $c_X > 0$ is a model parameter), and all other states are assigned zero cost.

**Example 9.** Figure 5.7 depicts the initial Markov model obtained by applying the method described above to the telehealth SBS from our running example. The model comprises a state for each node from the UML activity diagram in Figure 5.1, and a "failure" state for each of the three SBS operations—the "failure" states are shaded in Figure 5.7. To improve readability, the names of the three SBS operations and of the other activity diagram nodes were abbreviated in the name of the state transition probabilities that represent model parameters (i.e., SA-sendAlarm, AVP=analyseVitalParams,

CD=changeDrug, I=INITIAL, F=FINAL, req=request? and res=result?). Only the non-zero costs $c_{SA}, c_{AVP}$ and $c_{CD}$ are shown in the diagram, next to the states corresponding to the three SBS operations.

The PCTL encoding of the SBS requirements is automated by the tool for requirements that can be represented using instances of the pattern

$$P \bowtie p; condition, outcome$$

that are associated with activity-diagram nodes as illustrated in Figure 5.1. The three elements of this pattern represent the probability bound for the requirement (where $\bowtie \in \{<, \leq, =, \geq, >\}$), a boolean *condition* that holds in the scenarios in which the requirement must be satisfied, and the *outcome* that the probability bound is associated to, respectively. Note that this pattern can be used to describe only a subset of the requirements that can be expressed in PCTL, i.e., the subset of requirements that can be encoded as unbounded until PCTL formula from the Background Chapter 2. Nevertheless, other tools exist that can automate the PCTL generation process for other type of SBS requirements. The ProProST tool [90] that was used for the same purpose in the related work from [29] is particularly suitable for this role. ProProST takes as input a plain-English description of the requirements of a system that is expressed using a constrained vocabulary, and generates PRISM PCTL formulae.

**Example 10.** The probabilistic temporal logic formula generated by the COVE tool for requirements $R_1 - R_3$ from Section 5.2 are:

$R_1$**:** $P \leq_{0.08} [!stop?\ \mathrm{U}\ failedSendAlarm \mid failedChangeDrug \mid failedAnalyseVitalParams]$

$R_2$**:** $P \leq_{0.05} [true\ \mathrm{U}\ failedSendAlarm]$

$R_3$**:** $analyseVitalParams \Rightarrow P \leq_{0.0002} [!stop\ ?U\ failedSendAlarm]$

where the atomic proposition from the PCTL formula are taken from the Markov model depicted in Figure 5.7.

### 5.4.3 Stage 3: Service-based system construction

In this stage, the $n$ COVE proxies are integrated with the code that implements the SBS workflow, in a similar manner to standard web service proxies. Additionally, an instance of the COVE model updater and an instance of the COVE autonomic manager are created and initialised (through their constructor parameters) with the initial Markov model and the array of PCTL requirements from the previous stage, respectively. Standard development tools/integrated development environments can be employed in this stage.

**Example 11.** The code in Figure 5.8 creates and initialises instances of these components as follows:

```
public class TeleHealthSystem {
     ...
     public static void main(String[] args) throws Exception {
          // 1. Create COVE model updater, and initialise with the Markovian model generated by the COVE tool
          ipSupportTool.Model model =
               new ipSupportTool.Model("/Users/yasminrafiq/TeleHealthSystem/activityModel.pm");

          // 2. Create COVE proxy for the sendAlarm operation
          String methodNameAlarm = "sendAlarm";
          AlarmProxyPackage.AlarmProxy sendAlarmProxy = new AlarmProxyPackage.AlarmProxy(methodNameAlarm,
                 "[AS:Alarm]", model);

          // 2.1. Set parameter values for the concrete service #1 used for the "sendAlarm" operation
          sendAlarmProxy.setServiceMethodCost(0, methodNameAlarm, 0.02);
          sendAlarmProxy.setServiceMethodPriorValue(0, methodNameAlarm, 0.968);
          sendAlarmProxy.setServiceMethodAlphaValue(0, methodNameAlarm, 1.004);
          sendAlarmProxy.setServiceMethodSmoothingParameter(0, methodNameAlarm, 50);

          // 2.2. Set parameter values for the concrete service #2 used for the "sendAlarm" operation
          ...

          // 3. Create COVE proxy for the changeDrug operation
          String methodNameDrug = "changeDrug";
          ...

          // 4. Create COVE proxy for the analyseVitalParams operation
          String methodNameAnalysis = "analyzeVitalParams";
          ...

          // 5. Create COVE autonomic manager object
          AutonomicManager am = new AutonomicManager(model, 3,
               new IPSupport[] {sendAlarmProxy, analyseVitalParamsProxy, changeDrugProxy},
               new String[] {methodNameAlarm, methodNameAnalysis, methodNameDrug});

          // 6. PCTL system requirements
          String[] pctls = {
               "filter(forall,P<=0.0002[!(\"stop\")U(\"sendAlarmFail\")],\"analyseVitalParams\")",            // R1
               "P<=0.05[F(\"sendAlarmFail\")]",                                                               // R2
               "P<=0.08[!(\"stop\")U(\"sendAlarmFail\" | \"changeDrugFail\" | \"analyseVitalParamsFail\")]"}; // R3

          ...
```

Figure 5.8: The initialisation of the proxies, model updater and autonomic manager components of the telehealth self-verifying service-based system

- The COVE model updater class is instantiated in step 1, and initialised using the (discrete-time) Markov chain model generated by the COVE tool in stage 2 of the development process;

- The COVE proxy for the sendAlarm operation (generated automatically in stage 1 of the development process, see Figure 5.5) is then instantiated in step 2 from Figure 5.8, and its parameters associated with the two concrete web services that implement this SBS operation are initialised in substeps 2.1 and 2.2, respectively. Steps 3 and 4 carry out similar initialisations for the other two SBS operations.

- The code in step 5 instantiates the COVE autonomic manager.

- Finally, the PCTL-encoded SBS requirements generated in the second stage of the development process are organised into an array in step 6; in the current version of COVE, this involves taking a copy of these properties from the `.pctl` file in Figure 5.6.

Figure 5.9 illustrates the actual workflow code (in substep 7.1), and the periodical invocation of the COVE autonomic manager (in substep 7.2).

```
        ...

        // 7. The telehealth workflow
        while(true) {
            ...
            // 7.1. The actual telehealth workflow
            if (nextRequest() == RequestType.PatientRequestedAlarm) {
                sendAlarmProxy.sendAlarm();
            }
            else // NB: the next request is of type RequestType.PatientDataAnalysis
                switch (getAnalysisResultType(analyseVitalParamsProxy.analyzeData(...))) {
                case NewPrescription:
                    changeDrugProxy.changeDrug(...);
                    break;
                case AssistanceNeeded:
                    sendAlarmProxy.sendAlarm();
                    break;
                case PatientFine:
                default:
                    break;
                }
            ...

            // 7.2. Periodical invocation of the COVE autonomic manager
            if (System.currentTimeMillis() >= nextAMrun){
                am.selectConcreteServices(pctls);
                nextAMrun = System.currentTimeMillis() + durationBetweenAMInvocations;
            }
        }
    }
}
```

Figure 5.9: The implementation of the self-verifying telehealth SBS workflow

## 5.5 Dynamic service selection

### 5.5.1 Continual verification

The continual verification function of a COVE self-adaptive SBS is performed by its autonomic manager component (Figure 5.2). The autonomic manager is invoked periodically, and operates by verifying the latest version of the system model as follows. First, the updated Markov model of the SBS workflow is obtained from the COVE model updater. This model is parameterised by the success probabilities $p_i$ and costs $c_i, 1 \leq i \leq n$, of the $n$ SBS operations. Accordingly, the autonomic manager obtains the estimate success probabilities and costs of the services available for each operation from the appropriate COVE proxy. The estimate success probabilities and costs associated with every $n$-service combination that can execute the SBS operations are then used to generate a fully specified instance of the SBS model. Next, these model instances are verified using an embedded version of the PRISM model checker, to identify the service combinations that satisfy all SBS requirements. (The scalability of the approach is evaluated and discussed in Section 5.6.3.) Finally, a service combination that satisfies the SBS requirements and has minimal cost is selected by the autonomic manager, and the COVE proxies are configured to start using this combination. The decision taken when no suitable service combination is available depends on the configuration of the autonomic manager, the options being to select the lowest-cost service combination, or to choose cease executing the workflow until the services recover or SBS requirements are relaxed (e.g., by the system administrator).

**Example 12.** For illustration purposes, we used the standalone version of the probabilistic model checker PRISM to reproduce the verification task carried out by the COVE autonomic manager for the telehealth SBS from our running example. We assume that the state transition probabilities for the SBS Markov model are those

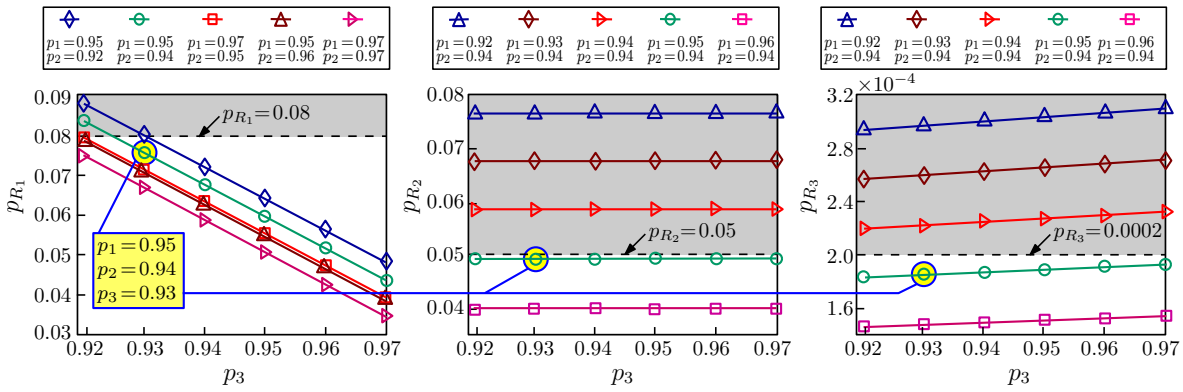


Figure 5.10: Experimental results for Example 12

from Figure 5.7, with the exception of the operation success probabilities $p_1, p_2$ and $p_3$, which represent model parameters. We then analysed the three PCTL requirements from Example 9 for model instances associated with a range of $(p_1, p_2, p_3)$ combinations. The results from the verification of the three requirements are shown in the three diagrams from Figure 5.10. The shaded areas in each graph corresponds to $(p_1, p_2, p_3)$ combinations that violate the SBS requirement associated with that graph. In contrast, $(p_1, p_2, p_3)$ combinations that appear outside the shaded area of each diagram correspond to acceptable service combinations[2]. The acceptable service combination with the lowest cost (or one of them, if several acceptable combinations have the lowest cost) is eventually selected, e.g., $(p_1, p_2, p_3) = (0.95, 0.94, 0.93)$ in the scenario from Figure 5.10.

### 5.5.2 Heuristics for switching between concrete services

In many applications that require the use of online learning techniques such as those described in Sections 3.2.1 and 3.3, these techniques are employed not only to detect violations of QoS requirements, but also to recover from such violations. This recovery from QoS violations typically involves reconfigurations of the adaptive system, e.g., through the dynamic replacement of underperforming or failed components with alternative components. These alternative components may be functionally equivalent but more expensive to use, or may provide a limited version of the same functionality to allow for a graceful degradation of the service provided by the system. In these circumstances, the sudden configuration switching means that the observations for an SBS operation are associated with a concrete service that is no longer in use. Accordingly, the calculation of the transition probability from (3.5) needs to be restarted using a new prior value $p_{ij}^0$ that corresponds to the newly selected service. Furthermore, the probability estimate $p_{ij}^k$ learnt prior to the change and associated with a system component or configuration that may be of interest again in the future needs to be updated using another mechanism because observations of its behaviour are no longer available.

In this section, we present several heuristics for handling this scenario. These

[2]Note that the results from this exhaustive analysis can be used to derive the a Pareto optimal set of $(p_1, p_2, p_3)$ combinations and the Pareto front for this system [172, 173].

heuristics assume that the learning algorithm from Section 3.2.1 can be applied to online learning of the reliability QoS property for a self-adaptive SBSs, where the underlying services are dynamically selected at runtime (this framework is described in more detail in Section 5.4). However at any point in time, the online estimation technique from (3.5) is only applied to the selected service from a set of functionally equivalent services, namely to the service that was used for the last execution of the SBS operation. This section describes an extension to the technique to include the inactive concrete services in the set as follows:

- A service never used by the SBS workflow is assumed to operate with its "advertised" success probability $p^0$.
- A service whose use was discontinued because a cheaper service that satisfies the SBS requirements became available is assumed to return to its "advertised" success probability $p^0$.
- A service "discontinued" after $k > 0$ invocations because it ceased to satisfy the lower bound $p^{required}$ success probability is dealt with using one of the techniques summarised in Table 5.1 and described below.

First, the "cooling off" technique for handling underperforming services is suitable for scenarios in which a service becomes unavailable suddenly. Our COVE proxy described in Section 5.4.1, is configured to use this technique, so when a selected service's success probability $p^k$ from (3.5) drops below $p^{required}$, the proxy deselects the underperforming service and replaces it with another service from the set of functionally equivalent services. The service is reconsidered as (potentially) suitable after a fixed *cooling-off time* $t_{co} > 0$, which is a parameter of the technique.

Second, the "simple" technique is useful in scenarios in which a service is temporarily overloaded, so its probability of success decreases below the threshold specified in the operation SLA, but remains non-zero. When this technique is used for a service deemed unsuitable after the $k-$th invocation because $p^k < p^{required}$, the probability of successful service invocation at time $t_{k'} > t_k$ is estimated as

$$p^{k'} = \begin{cases} p^k, & \text{if } t_{k'} < t_k + t_{co} \\ \frac{\beta^{t_{k'}-(t_k+t_{co})}-1}{\beta^{t_{k'}-(t_k+t_{co})}} p^0 + \frac{1}{\beta^{t_{k'}-(t_k+t_{co})}} p^k, & \text{otherwise} \end{cases} \tag{5.1}$$

where $\beta > 1$ is a *recovery* parameter, and $t_{co} > 0$ is a cooling-off time as before. As a result, the time until the service is reconsidered depends on how much the service reliability $p^k$ dropped below the required value $p_i$. This advantage is shared by the "hysteresis" technique, which uses the same approach to estimating $p$, but additionally avoids frequent service changes by using a *hysteresis parameter* $\gamma$.

## 5.6 Implementation and evaluation

### 5.6.1 Implementation

We developed a prototype implementation of the COVE framework as an open-source Java toolset. To ensure that the toolset remains accessible after the end of this PhD project, we made it available from the project supervisor's public webpage at `http://www-users.cs.york.ac.uk/~raduc/COVE/`. The core components of the COVE toolset are:

Table 5.1: Handling services that cease to be part of service combinations satisfying system requirements.

| Technique | Criteria for deciding that:<br>(a) service is no longer suitable<br>(b) unsuitable service is again suitable | Estimate for discontinued service |
|---|---|---|
| cooling off | (a) $p^k < p_i$<br>(b) $t_{co}$ time units elapsed since $t_k$ | not required |
| simple | (a) $p^k < p_i$<br>(b) $p^{k'} \geq p_i$ | see eq. (5.1) |
| hysteresis | (a) $p^k < (1-\gamma)p_i$<br>(b) $p^{k'} > (1+\gamma)p_i$ where<br>$0 < (1-\gamma)p_i < p_i < (1+\gamma)p_i < 1$ | see eq. (5.1) |

- The SBS-independent autonomic manager and model updater, which are provided as a JAR library for inclusion in COVE-based self-adaptive SBSs. The autonomic manager uses a PRISM quantitative verification library [111], and implements the continual verification technique described in Section 5.5.1. The model updater implements the adaptive learning algorithm presented in Section 3.3 in order to maintain the state transition probabilities of the SBS model verified by the autonomic manager in step with changes in the SBS workflow.

- The COVE proxy-generator tool, which is implemented as a Java Swing application that employs open-source Apache Axis2 technology (`http://axis.apache.org/axis2/java/core`) to generate the COVE proxies as described in Section 5.4.1.

- The COVE model generator and requirement formalisation tool, which is implemented as a Java command-line application that takes as input SBS activity diagram in the XMI format generated by the Eclipse-based Papyrus graphical editing tool for UML 2 (`http://www.eclipse.org/papyrus/`).

The auxilliary COVE class supporting the runtime manipulation of discrete-time Markov chain models, and the creation of an audit trail of autonomic manager verification steps and decisions with log4j (`http://logging.apache.org/log4j/`) are bundled together with the autonomic manager and model updater in the reusable COVE JAR.

### 5.6.2 Case study

We used the COVE toolset and the development process described in Section 5.4 to implement a self-adaptive version of our telehealth SBS that used $m_1 = 2$ `sendAlarm` services, $m_2 = 2$ `changeDrug` services, and $m_3 = 3$ `analyseVitalParams` services. These seven services were simulated using real Java web services deployed on Amazon EC2 (`http://aws.amazon.com/ec2/`) "small instance" virtual machines. Individual configuration files were used to specify the variation of the actual probability of successful invocation for each web service, $p_{ij}, 1 \leq i \leq 3, 1 \leq j \leq m_i$, over the duration of each

Table 5.2: Service prior success probabilities and costs

| service | prior success probability ($p^0_{i,j}$) | cost ($c_{i,j}$) |
|---|---|---|
| $\text{sendAlarm}_1$ | 0.968 | 0.02 |
| $\text{sendAlarm}_2$ | 0.968 | 0.01 |
| $\text{changeDrug}_1$ | 0.96 | 0.3 |
| $\text{changeDrug}_2$ | 0.95 | 0.1 |
| $\text{analyseVitalParams}_1$ | 0.965 | 5.0 |
| $\text{analyseVitalParams}_2$ | 0.95 | 4.0 |
| $\text{analyseVitalParams}_3$ | 0.96 | 3.0 |

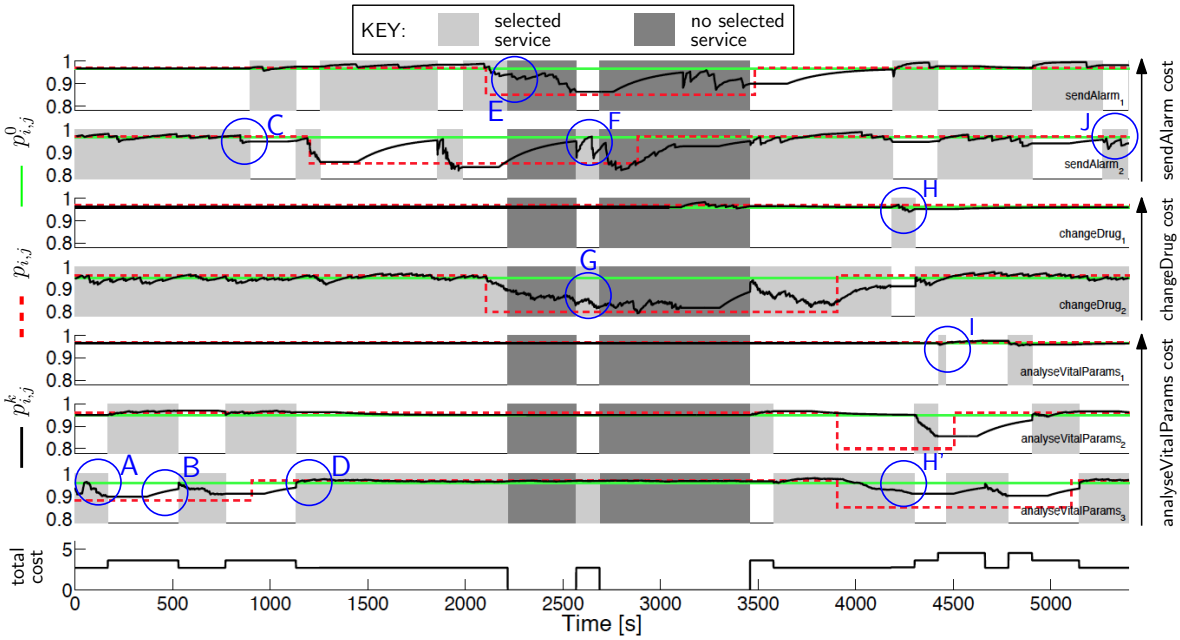


Figure 5.11: Automated service selection for the telehealth service-based system; the circular areas labelled 'A', 'B', etc. are analysed in Section 5.6.2

experiment. The *a priori* success probability $p^0_{i,j}$ and the costs $c_{i,j}$ for an invocation of each of these services are shown in Table 5.2. A Java implementation of the telehealth SBS workflow from Figure 5.1 was integrated with COVE proxies for its three operations, and was run on a standard 2.66 GHz Intel Core 2 Duo Macbook Pro computer.

Figure 5.6.2 depicts a typical experiment in which the COVE autonomic manager selects the service combinations for the telehealth SBS dynamically, over a 1.5-hour wall clock time period. Low-cost combinations of services are preferred when their combined probabilities of successful completion satisfies all SBS reliability requirements, and are discarded in favour of high-cost service combinations when their joint reliability violates one or more of these SBS requirements. These decisions are taken based on the estimate probabilities of success $p^k_{ij}$ calculated by the COVE adaptive learning algorithm (initialised with $\epsilon = p_{max} = 0.05$), and on the continual verification of the updated SBS model:

- At the beginning of the experiment, the lowest-cost service combination is selected by the autonomic manager, as the high *a priori* success probabilities $p^0_{i,j}$ of

all services make all service combinations seem suitable. This is the expected behaviour, since a service whose provider-specified SLA does not satisfy the SBS requirements should not be included in the system.

- When the autonomic manager learns that $\text{analyseVitalParams}_3$ is underperforming in the area labelled 'A' in the diagram, it switches to using the higher-cost $\text{analyseVitalParams}_2$ service.

- While a higher-cost service is used for an SBS operation, the adaptive learning algorithm "rebuilds trust" in the temporarily discarded lower cost service (area labelled 'B' in the diagram). This is due to the fact that the observations of frequent failures from area 'A' are associated weights that decrease over time, so the estimate $p^k_{3,3}$ slowly approaches the prior value $p^0_{3,3}$. The learning algorithm was configured to assume that a service returned to its prior success probability when the autonomic manager resumes using it, which explains why $p^k_{3,3}$ grows suddenly to $p^0_{3,3}$ when the $\text{analyseVitalParams}_3$ is selected again in area B.

- In area C, a slight variation in the estimate success probability of the $\text{sendAlarm}_2$ service triggers a potentially unnecessary transition to the more expensive service $\text{sendAlarm}_1$. Choosing strict intervals of confidence (i.e., smaller $\epsilon$ and/or $p_{max}$ parameters) for the COVE adaptive learning could reduce such "false positives", although eliminating them altogether is not possible (cf Proposition 3.3).

- In area D, the autonomic manager resumes using $\text{analyseVitalParams}_3$, which has now recovered.

- In area E, the autonomic manager learns that even the high-cost alarm service (i.e., $\text{sendAlarm}_2$) is unreliable, to the extent that the SBS requirements are no longer satisfied. Under the autonomic manager configuration used in the experiment, no service was selected in this scenario, and an error message was generated instead to alert the system administrator.

- In area F, the autonomic manager retries to use the alarm service that experienced a low success rate first, learns that this service has not yet recovered.

- Area G shows that some services have little impact on the overall SBS compliance with its requirements—as only requirement $R_1$ depends on a successful completion of the changeDrug SBS operation (and only marginally), a decrease in the reliability of $\text{changeDrug}_2$ does not determine the autonomic manager to abandon this service.

- Nevertheless, the autonomic manager does switch to the more expensive $\text{changeDrug}_1$ service in area H-H', at a moment when $\text{changeDrug}_2$ is actually more reliable than it was in area G. The decision is motivated by the decrease in the reliability of $\text{dataAnalysis}_3$, which the autonomic manager compensates for by choosing a slightly more expensive drug service (the cost difference between $\text{changeDrug}_2$ and $\text{changeDrug}_1$ is only 0.2) instead of switching to a significantly more expensive analysis service (moving away from $\text{analyseVitalParams}_3$ would have amounted to a cost increase of at least 1.0 for this operation.

- The strategy adopted in area H-H' is unsuccessful, so the most expensive analysis service is eventually selected in area I.

- Finally, in area J all service have recovered and operated close to their advertised SLAs, so the autonomic manager returns to using the lowest-cost service combination for the telehealth service-based system.

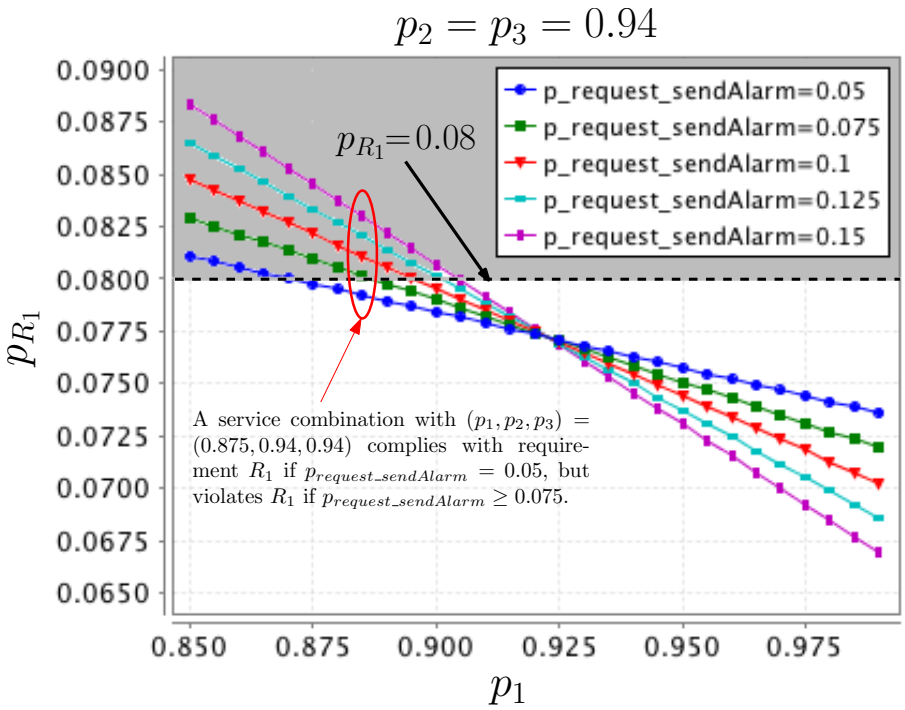


Figure 5.12: The effect of changes in the probability of alarm requests

A key capability of COVE is its ability to learn not only changes in the reliability of individual services, but also changes in the rates with which the SBS operations are performed. To evaluate this functionality, we considered the effect of changes in the probability $p_{request\_sendAlarm}$ (abbreviated $p_{req\_SA}$ in Figure 5.7) that a request handled by the telehealth SBS is a patient-initiated alarm. A temporary increase in this probability may be caused, for instance, by a flu outbreak. Figure 5.12 depicts the analysis of requirement $R_1$ from our case study, for a range of service combinations and for $p_{request\_sendAlarm}$ values between 0.05 and 0.15. This analysis shows that even a small change in the probability of alarm requests is sufficient to render unacceptable a service combination that was previously compliant with requirement $R_1$. This confirms the importance of updating the SBS model in line with any fluctuations in the probabilities with which the SBS operations are executed.

### 5.6.3 Applicability to larger systems

We evaluated the ability of our approach to handle a range of system sizes, by carrying out a number of experiments that assessed the applicability and overheads of executing the runtime analysis within the autonomic manager in multiple scenarios. We selected the following workflows, used by a number of projects in this area:

1. the healthcare case study described in this report, and previously used in [28, 29, 66];

2. the e-commerce workflow obtained from [74];

3. the travel assistant workflow derived from the state-chart representation presented in [164].

These workflows comprise invocations of three, four and five abstract operations, respectively.

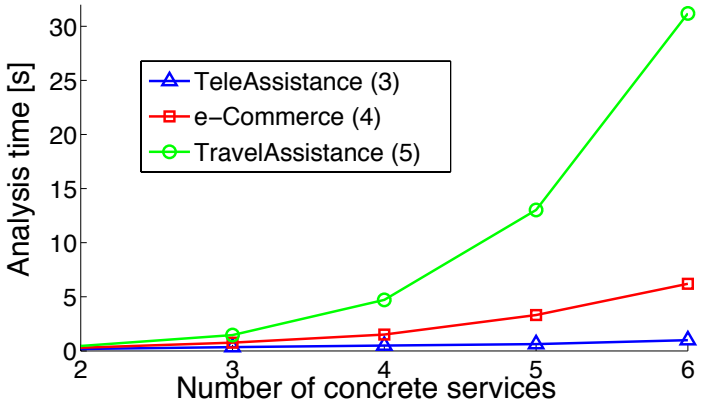


Figure 5.13: Scalability results of the autonomic manager

For each of the workflows we devised a parameterised Markov chain, and we defined four PCTL requirements, including one PCTL property to determine the expected cost of a single invocation of the workflow. The size of the models ranged between 11 and 17 states. As we envisage that practical self-adaptive service-based systems will rarely use more than two or three concrete services for each abstract operation, we then ran experiments that considered between to and five concrete services for each of the abstract operations. Due to space constraints, we do not include the Markov models and properties for the e-commerce and travel assistant workflows in the main body of this report. However, these Markov models and properties, and detailed descriptions of each of the experiments are available from the project website at `http://www-users.cs.york.ac.uk/~raduc/COVE/` and in the Appendix F and G.

Each experiment measured the time taken to initialise the autonomic manager and select the optimal concrete service configuration in the worst-case scenario whereby all combinations of concrete services satisfied verifying the suitability of a service combination as soon as it learns that the combination violates one of the requirements. Figure 5.13 summarises the results of these experiments, averaged over multiple runs. According to these results, up to three services per SBS operation can be analysed within two seconds for each of the considered workflows, which confirms the feasibility of the approach for typical SBSs of practical importance from the domains explored in our experiments. Increasing this to four services for each SBS operation leads to verification times of up to $5s$, which is likely to be acceptable for many practical applications. This is particularly true when large numbers of false positives and false negatives in the associated learning process need to be avoided, so longer time is already needed to identify the changes on which the autonomic manager must act.

The exponential growth in analysis time shown in Figure 5.13 limits the applicability of the current COVE version to systems comprising small to medium numbers of operations, and using small numbers of services (i.e., between 2 and 4 services) per SBS operation. While the second constraint is, in our opinion, not significant, the former implies that SBSs comprising large numbers of operations cannot yet benefit from this approach. However, recent work by several research groups and ourselves has led to significant advances in the use of incremental and compositional techniques to reduce quantitative verification times, often by multiple orders of magnitude [33, 73, 102, 112]. We envisage that integrating these techniques into COVE will significantly enhance its ability to support the development and operation of much larger service-based systems.

As always, the addition of monitors introduces overheads and may cause additional problems due, for instance, to computational resources including CPU, memory, I/O, and network bandwidth [125]. However, as described in Section 3.3.3, in our case

the overheads are minimal since the overall space complexity of the adaptive learning algorithm is $O(f)$ (i.e., timestamps is proportional to the frequency $f$ of observations), and the time complexity of the learning algorithm is $O(1)$. Then again, the other effects cannot be ruled out. In particular, robustness, extensibility and manageability need to be assessed for any use of our approach in a real-world system.

## 5.7 Summary

This chapter introduced our COntinual VErification (COVE) framework for the tool-supported engineering of self-adaptive service-based systems. The architecture of COVE self-adaptive service-based systems was presented using a case study from the telehealth domain. Among the key components of this architecture are the COVE service proxies that interface the service-based system (SBS) workflow with sets of functionally equivalent remote services. The runtime selection of the service for each SBS operation and, as in the case of traditional web service proxies, the interactions with the selected services are handled automatically by the COVE proxies.

The COVE proxies are also responsible for observing all service invocations and their outcome. These observations are used to keep up to data a Markov model of the SBS workflow, starting from a developer-supplied initial version of this model. The model updates are carried out using the adaptive online learning technique presented in Chapter 3. Finally, the resulting up-to-date model is used by an *autonomic manager* that controls the services selected by the COVE proxies, to ensure that the service combination which satisfies the SBS requirements with minimal cost is selected at all times.

The COVE framework, which we implemented as an open-source Java toolset, comprises SBS-independent components (i.e., the autonomic manager and model updater), a COVE proxy generator tool and an initial model generator. Accordingly, the COVE model updater and autonomic manager are reusable across applications, while the SBS-specific COVE proxies and initial Markov model are generated automatically by the proxy and model generators, respectively.

# Chapter 6

# Conclusion and future work

This thesis introduced a suite of online techniques for learning and continually updating the parameters and structure of Markov chains, a class of models used to establish key reliability, performance and other QoS properties of real-world systems. The analysis of the Markov models synthesised and kept up to date by our techniques enables the accurate quantitative verification of these QoS properties as the analysed system evolves over time. For adaptive systems, this enables dynamic reconfiguration actions which ensure that QoS requirements continue to be met after environmental and internal changes.

Our new learning techniques use as input runtime observations of system events associated with the transitions between the states of a model, and with costs/rewards associated with these states and transitions. When the model structure is known, they continually update the state transition probabilities and costs/rewards in line with the observed variations in the behaviour of the system. In scenarios when the model structure is unknown, a Markov chain is synthesised from sequences of such observations.

The two categories of learning techniques underpin the operation of a new toolset for the engineering of self-adaptive service-based systems, which was developed as part of this research. The thesis introduced this software engineering toolset, and showed its effectiveness in a case study that involved the development of a prototype telehealth service-based system capable of continual self-verification.

## 6.1 Summary of contributions

The main contributions of the thesis, their advantages and novel characteristics, and the insights gained from pursuing the research to devise them are summarised below.

Firstly, we developed two online learning techniques that infer the state transition probabilities of a Markov model from observations of the modelled system behaviour. The former, Bayesian derived technique, weighs observations based on their age, to account for the fact that older observations are less relevant than more recent ones. The approach decreases the impact of old observations on the estimates, significantly speeding up the detection of sudden changes in the actual transition probabilities (e.g., due to failures of system components). This is particularly noticeable when such changes occur after long periods of relatively constant behaviour. This first learning technique calculates new transition probability estimates in $O(1)$ time and using $O(1)$ memory, a key advantage for an online learning algorithm. However, the effectiveness of this algorithm depends on the choice of its two parameters, the *smoothing parameter*

$c_i^0$ and the *ageing parameter* $\alpha_i$, and no combination of values for these parameters is suitable for all scenarios. To address this limitation, the latter learning technique selects suitable values for these parameters at runtime, based on the frequency of the observations. This *adaptive learning* leads to a faster and more accurate inference of the transition probabilities than that provided by existing methods. Furthermore, we introduced rigorous theoretical results that link the parameters chosen dynamically to the expected error in the accuracy of the learnt state transition probabilities. This allows the configuration of the adaptive learning method so that it yields results within an acceptable expected error range.

Secondly, we introduced a new technique that uses the Kalman filter and the recursive weighted least-square filter to establish cost/reward structures for Markov chains. These structures are associated with, and support the analysis of, performance-related QoS properties of component-based systems whose instrumentation is not possible or not desirable (e.g., embedded and real-time systems). The approach works for systems that can be continuously monitored as a black box, so that the values of the cumulative costs/rewards for operation sequences can be observed. The technique can be used to identify under-performing components, to predict the system behaviour for infrequent execution paths, and to support reconfiguration decisions in self-adaptive systems. We showed the application of this technique for a case study taken from the telematics domain, evaluated its effectiveness, and compared the two optimal filters for a wide range of experiments. These experiments provided empirical evidence that the Kalman filter is a better choice than the recursive weighted least square filter in terms of both false positives and false negatives. Using the latter filter led to slower but also to "smoother" learning. This could be preferable for systems that operate close to the bounds specified in their requirements, since in these circumstances the Kalman filter generates numerous false positives.

Thirdly, we developed a technique for the synthesis of Markov chains with the desired degree of accuracy for service-based systems. This new technique uses a distance function to compare the model versions learnt after every $N$ observations of system events. When the distance between two successive model versions compared in this way drops below an *acceptable distance* $\epsilon$, the latest model is deemed "stable" enough for use in verification, and ultimately to support adaptation decisions. By varying the acceptable distance parameter $\epsilon$ and the number of observations $N$, we provided insight into their impact on the ability to use the learnt model to accurately analyse the compliance of a system with a range of requirements. In particular, our empirical evidence showed that the accuracy of the learnt model increased when decreasing $\epsilon$, at the expense of a larger number of observations being required before the model could be deemed stable. This trade-off between model accuracy and learning effort is also influenced by $N$, with lower values of $N$ speeding up the learning process. However, caution needs to be exercised, since decreasing $N$ too much leads to the synthesis of models with incomplete states and/or transitions.

Fourthly, we introduced the COntinual VErification (COVE) service-based system development framework that contributes towards narrowing the gap between state-of-the-art research and the state of practice in the engineering of self-adaptive service-based systems (SBSs). Similar to other self-adaptive SBS frameworks, COVE selects the services used to execute each operation of an SBS workflow at runtime, from sets of functionally equivalent services with different levels of reliability and cost. However, COVE has a several advantages over the existing approaches to developing and operating self-adaptive SBSs. Thus, the self-verifying systems developed using this framework employ continual formal verification to select the service combination that guarantees the realisation of their reliability requirements with minimal cost. The underlying model is updated online to reflect changes in the service reliability and in the frequency with

which the SBS operations are invoked. The continual verification, model updating and service selection capabilities are fully automated, and are provided by a combination of reusable and automatically generated software components. Last but not least, the development process supported by COVE resembles the traditional SBS development process, so practitioners can use it with little learning effort.

Finally, we used a range of case studies to evaluate the effectiveness of the new model learning techniques and software tools developed by the project. We envisage that these case studies will serve as exemplars for other researchers and practitioners, helping them gain insight into the engineering of self-adaptive service-based software systems.

## 6.2 Future work

There are multiple ways in which the research presented in this thesis can be refined and extended. The most significant research directions requiring further investigation are covered below.

**Extension to other types of QoS models** In this thesis we described techniques for the online learning of discrete-time Markov chains. These Markov models are widely used to analyse the QoS properties of computer systems. Other important types of models that are used for this purpose include continuous-time Markov chain [13], queueing networks [23], Petri nets [120] and stochastic process algebra [95]. Extending the learning techniques introduced in the thesis to these additional modelling formalisms represents an important area of future research.

**Validation of online learning techniques in new domains** The case studies used to illustrate and evaluate the online learning techniques proposed in the thesis came primarily from the domain of service-based systems. Another area of future research would be to use our techniques in other application domains, such as the Internet of Things (IoT). As self-organisation is a key system-level feature of IoT, the complexity and dynamics that many IoT will be implemented in will likely require continual adaptation that will necessarily have to rely on runtime analysis of accurate system models.

**Exploit less resource-intensive verification techniques** Extending the applicability of our continual verification framework COVE to larger service-based systems requires its integration with recently emerged incremental and compositional verification techniques [33, 73, 102, 112]. Achieving this integration represents an important area of research work. A key target of this work is the incremental verification technique which our group proposed in [33], which we deem particularly suitable for this purpose due to its ability to produce system-level verification results by re-analysing only the parts of the system that were affected by a change.

**Add natural language support to the continual verification framework** Another area of future work is to extend the COVE framework with a plugin that supports the specification of service-based system requirements in a domain-specific natural language, similar to ProProST [90]. Last but not least, future work is required to extend COVE with the ability to handle additional categories of QoS requirements (e.g., performance and energy related), along the lines of the work from [29].

# Appendix A

# Matlab implementation of the Kalman filter

```
function [x1] = KalmanAlgorithm(z)
persistent A H Q R
persistent x P
persistent firstRun
if isempty(firstRun)
    firstRun = 1;
    A = 1;
    H = 1;
    R = 5;
    Q = 0;
    x = 600;
    P = 50;
end
% 1. Prediction of the estimate
xp = A*x;
% 2. Prediction of the error covariance
Pp = A*P*A'+Q;
% 3. Calculate the Kalman gain
K = Pp*H'/(H*Pp*H' + R);
% 4. Calculate the new estimate
x = xp + K*(z - H*xp);
% 5. Calculate the error covariance
P = Pp - K*H*Pp;
% 6. Return the new estimated value
x1 = x(1);
```

# Appendix B

# Test program for the Kalman filter

```
Xsaved = zeros(500, 1);
Msaved = zeros(500, 1);
Psaved = zeros(500,1);
Ksaved = zeros(500,1);
Zsaved = zeros(500, 1);
Xactual = zeros(500,1);
for j = 1:500
    x = 740;
    Xactual(j,:) = x;
    z = x + 50*randn(1,1);
    Msaved(j,:) = z;
    [x1] = KalmanAlgorithm(z);
    Xsaved(j,:) = x1;
end
t = 1:500;
Xactual(:,1);
Msaved (:,1);
Xsaved (:,1);
figure
hold on;
plot(t, Xactual(:,1),'r');
plot(t, Msaved(:,1), 'g');
plot(t, Xsaved(:,1), 'b');
```

# Appendix C

# Matlab implementation of the RWLS filter

```
function [x1] = RecursiveWeightedLeastSqAlgorithm(z)
% 1. Initialize the estimator
persistent  H x P R
persistent firstRun
if isempty(firstRun)
    firstRun = 1;
    H = 1;
    R = 5;
    x = 600;
    P = 50;
    lambda = 0.98;        % -- forgetting function
    laminv  = 1/lambda;
end
% 2.1 Obtain new measurement z_k = H'x + v_k;
z_k = z;
% 2.2 Update sequentially
e = z_k - H*x;            % -- Update measurement residual
S = (H*P*H' + (lambda*R));% -- Update measurement prediction covariance matrix
K = P*H'*inv(S);          % -- Calculate the gain
x = x + K*e;              % -- Update estimate x
P = laminv*P - K*S*K';    % -- Update error covariance estimate
% 3. Return the new estimated value
x1 = x(1);
```

# Appendix D

# Test program for the RWLS filter

```
Xsaved = zeros(500, 1);
Msaved = zeros(500, 1);
Psaved = zeros(500,1);
Ksaved = zeros(500,1);
Zsaved = zeros(500, 1);
Xactual = zeros(500,1);
for j = 1:500
    x = 740;
    Xactual(j,:) = x;
    z = x + 50*randn(1,1);
    Msaved(j,:) = z;
    [x1] = RecursiveWeightedLeastSqAlgorithm(z);
    Xsaved(j,:) = x1;
end
t = 1:500;
Xactual(:,1);
Msaved (:,1);
Xsaved (:,1);
figure
hold on;
plot(t, Xactual(:,1),'r');
plot(t, Msaved(:,1), 'g');
plot(t, Xsaved(:,1), 'b');
```

# Appendix E

# PRISM model for Bioinformatics workflow

```
dtmc
//This is a PRISM model of a Taverna workflow used in studies for
//Graves disease. The workflow can be found at:
//http://www.myexperiment.org/workflows/28.html
//constants p1 to p18 represent (a priori estimates of) the probabilities
//that the 18 web service invocations complete successfully, and a PRISM
//module is used to model each web service. An additional module "Workflow"
//is used to model the workflow as a whole.

const double p1=0.999;
const double p2=0.998;
const double p3=0.999;
const double p4=0.995;
const double p5=0.997;
const double p6=0.997;
const double p7=0.999;
const double p8=0.996;
const double p9=0.998;
const double p10=0.999;
const double p11=0.998;
const double p12=0.999;
const double p13=0.999;
const double p14=0.991; //The UniProt Knowledgebase is a
                         //central database of protein sequence

const double p15;      //(EBI) maintains and distributes the EMBL
```

```
                        //Nucleotide Sequence database

const double p16=0.991;//getMedlineIds

const double p17=0.991;//ebi_medline2007, medline bibliographic database

const double p18=0.99;//calcMeltTemp
const int SUCC=1;
const int FAIL=2;
//------------------------------------------------------------------------

module WorkFlow
wf : [0..2] init 0; // 0 -init; 1 -success; 2 -fail

[] (wf=0)&(
(ebi_uniprot=FAIL)
|(calcMeltTemp=FAIL)
|(ebi_medline2007=FAIL)
|(markPathwayByObjects=FAIL)
|(DDBJBlastn=FAIL)
|(getInterProIds=FAIL)
|(getDotFromViz=FAIL)
) -> 1:(wf'=FAIL);
[] (wf=0)&(
(ebi_uniprot=SUCC)
&(calcMeltTemp=SUCC)
&(ebi_medline2007=SUCC)
&(markPathwayByObjects=SUCC)
&(DDBJBlastn=SUCC)
&(getInterProIds=SUCC)
&(getDotFromViz=SUCC)
) -> 1:(wf'=SUCC);
[] wf=SUCC -> 1:(wf'=SUCC);
[] wf=FAIL -> 1:(wf'=FAIL);
endmodule
//------------------------------------------------------------------------

module ScalcMeltTemp
//Service 18
calcMeltTemp : [0..2] init 0;
[] calcMeltTemp=0 -> p18:(calcMeltTemp'=SUCC)+(1-p18):(calcMeltTemp'=FAIL);
[] (calcMeltTemp=0)&(ebi_embl=FAIL) -> 1:(calcMeltTemp'=FAIL);
```

```
69 endmodule
70 //-------------------------------------------------------------------------
71
72 module Sebi_medline2007
73 //Service 17
74 ebi_medline2007 : [0..2] init 0;
75 [] ebi_medline2007=0 -> p17:(ebi_medline2007'=SUCC)+
76 (1-p17):(ebi_medline2007'=FAIL);
77 [] (ebi_medline2007=0)&(getMedlineIds=FAIL) ->
78 1:(ebi_medline2007'=FAIL);
79 endmodule
80 //-------------------------------------------------------------------------
81
82 module SgetMedlineIds
83 //Service 16
84 getMedlineIds : [0..2] init 0;
85 [] getMedlineIds=0 -> p16:(getMedlineIds'=SUCC)+(1-p16):
86 (getMedlineIds'=FAIL);
87 [] (getMedlineIds=0) & (getEmblld=FAIL) -> 1:
88 (getMedlineIds'=FAIL);
89 endmodule
90 //-------------------------------------------------------------------------
91
92 module Sebi_embl
93 //Service 15
94 ebi_embl : [0..2] init 0;
95 [] ebi_embl=0 -> p15:(ebi_embl'=SUCC)+(1-p15):(ebi_embl'=FAIL);
96 [] (ebi_embl=0) & (getEmblld=FAIL) -> 1:(ebi_embl'=FAIL);
97 endmodule
98 //-------------------------------------------------------------------------
99
100 module Sebi_uniprot
101 //Service 14
102 ebi_uniprot : [0..2] init 0;
103 [] ebi_uniprot=0 -> p14:(ebi_uniprot'=SUCC)+
104 (1-p14):(ebi_uniprot'=FAIL);
105 [] (ebi_uniprot=0)&(getSwissProtId=FAIL) ->
106 1:(ebi_uniprot'=FAIL);
107 endmodule
108 //-------------------------------------------------------------------------
109
110 module SgetEC
```

```
111 //Service 1
112 getEC : [0..2] init 0;
113 [] getEC=0 -> p1:(getEC'=SUCC) + (1-p1):(getEC'=FAIL);
114 endmodule
115 //-------------------------------------------------------------------------
116
117 module SgetEmblld
118 //Service 2
119 getEmblld : [0..2] init 0;
120 [] (getEmblld=0) -> p2:(getEmblld'=SUCC)+
121 (1-p2):(getEmblld'=FAIL);
122 endmodule
123 //-------------------------------------------------------------------------
124
125 module SgetMolFuncGolds
126 getMolFuncGolds : [0..2] init 0;
127 //Service 3
128 [] (getMolFuncGolds=0) -> p3:(getMolFuncGolds'=SUCC) +
129 (1-p3):(getMolFuncGolds'=FAIL);
130 endmodule
131 //-------------------------------------------------------------------------
132
133 module SgetSwissProtId
134 //Service 4
135 getSwissProtId : [0..2] init 0;
136 [] (getSwissProtId=0) -> p4:(getSwissProtId'=SUCC)+
137 (1-p4):(getSwissProtId'=FAIL);
138 endmodule
139 //-------------------------------------------------------------------------
140
141 module SgetPathwaysByECNumbers
142 //Service 5
143 getPathwaysByECNumbers : [0..2] init 0;
144 [] (getPathwaysByECNumbers=0)&(getEC=SUCC) ->
145 p5:(getPathwaysByECNumbers'=SUCC)+
146 (1-p5):(getPathwaysByECNumbers'=FAIL);
147 [] (getPathwaysByECNumbers=0)&(getEC=FAIL) ->
148 1:(getPathwaysByECNumbers'=FAIL);
149 endmodule
150 //-------------------------------------------------------------------------
151
152 module SmarkPathwayByObjects
```

```
153 //Service 7
154 markPathwayByObjects : [0..2] init 0;
155 [] (markPathwayByObjects=0)&(getPathwaysByECNumbers=SUCC) ->
156 p7:(markPathwayByObjects'=SUCC)+ (1p7):(markPathwayByObjects'=FAIL);
157 [] (markPathwayByObjects=0)&(getPathwaysByECNumbers=FAIL) ->
158 1:(markPathwayByObjects'=FAIL);
159 endmodule
160 //-------------------------------------------------------------------
161
162 module SgetTargetSequence
163 //Service 8
164 getTargetSequence: [0..2] init 0;
165 [] (getTargetSequence=0) -> p8:(getTargetSequence'=SUCC)+
166 (1p8):(getTargetSequence'=FAIL);
167 endmodule
168 //-------------------------------------------------------------------
169
170 module SDDBJBlastn
171 //Service 11
172 DDBJBlastn : [0..2] init 0;
173 [] (DDBJBlastn=0)&(getTargetSequence=SUCC) ->
174 p11:(DDBJBlastn'=SUCC)+ (1p11):(DDBJBlastn'=FAIL);
175 [] (DDBJBlastn=0)&(getTargetSequence=FAIL) ->
176 1:(DDBJBlastn'=FAIL);
177 endmodule
178 //-------------------------------------------------------------------
179
180 module SgetInterProIds
181 //Service 9
182 getInterProIds: [0..2] init 0;
183 [] (getInterProIds=0) -> p9:(getInterProIds'=SUCC)+
184 (1-p9):(getInterProIds'=FAIL);
185 endmodule
186 //-------------------------------------------------------------------
187
188 module ScreateVizSession
189 //Service 6
190 createVizSession : [0..2] init 0;
191 [] (createVizSession=0) -> p6:(createVizSession'=SUCC)+
192 (1-p6):(createVizSession'=FAIL);
193 endmodule
194 //-------------------------------------------------------------------
```

```
195
196 module SaddTermToViz
197 //Service 10
198 addTermToViz : [0..2] init 0;
199 [] (addTermToViz=0)&(getMolFuncGolds=SUCC)&(createVizSession=SUCC) ->
200 p10:(addTermToViz'=SUCC)+ (1p10):(addTermToViz'=FAIL);
201 [] (addTermToViz=0)&((getMolFuncGolds=FAIL)|(createVizSession=FAIL)) ->
202 1:(addTermToViz'=FAIL);
203 endmodule
204 //-------------------------------------------------------------------------
205
206 module sgetDotFromViz
207 //Service 12
208 getDotFromViz : [0..2] init 0;
209 [] (getDotFromViz=0)&(createVizSession=SUCC) -> p12:(getDotFromViz'=SUCC)+
210 (1p12):(getDotFromViz'=FAIL);
211 [] (getDotFromViz=0)&(createVizSession=FAIL) -> 1:(getDotFromViz'=FAIL);
212 endmodule
213 //-------------------------------------------------------------------------
214
215 module SdestroyVizSession //success in work flow does not
216                           //depend on this service.
217 //Service 13
218 destroyVizSession : [0..2] init 0;
219 [] (destroyVizSession=0)&(createVizSession=SUCC) ->
220 p13:(destroyVizSession'=SUCC)+ (1p13):(destroyVizSession'=FAIL);
221 [] (destroyVizSession=0)&(createVizSession=FAIL) ->
222 1:(destroyVizSession'=FAIL);
223 endmodule
```

# Appendix F

# E-commerce application domain

We tested the applicability of our approach from Chapter 5 on an e-commerce application introduced in [74]. The workflow represents a system in which new or returning customers log into an ecommerce application to purchase products. Once one or more products have been selected the customer may choose to have their purchases shipped by an express service or a normal service. The customer logs out and their session with the application is terminated successfully.

The system comprises four abstract services:

1. Authentication Service, which provides the operation *Auth*.
2. Payment Service, which provides the operation *Payment*.
3. Normal Shipping Service, which provides the operation *NrmShipping*.
4. Express Shipping Service, which provides the operation *ExpShipping*.

**Requirements** The autonomic manager is used to maintain the following high-level QoS requirements:

1. Probability of success shall be greater than 0.8.
2. Probability of a ExpShipping failure for a user recognized as a returning customer shall be less than 0.035.
3. Probability of an authentication failure shall be less than 0.06.

We formalise these requirements in probabilistic computational tree logic:

1. $P > 0.8[F(s = 16)]$.
2. $filter(forall, P < 0.035[F(s = 13)], s = 1)$.
3. $P < 0.06[Fs = 5]$.

The expected cost of a single invocation of the workflow is determined by the PCTL formula "$R =?[F(s = 16)|(s = 15)|(s = 5)|(s = 13)|(s = 8)]$", relative to the rewards structure associating a monetary value to each abstract operation named in the ecommerce workflow, appropriate to the concrete services provided by the user of COVE to the autonomic manager.

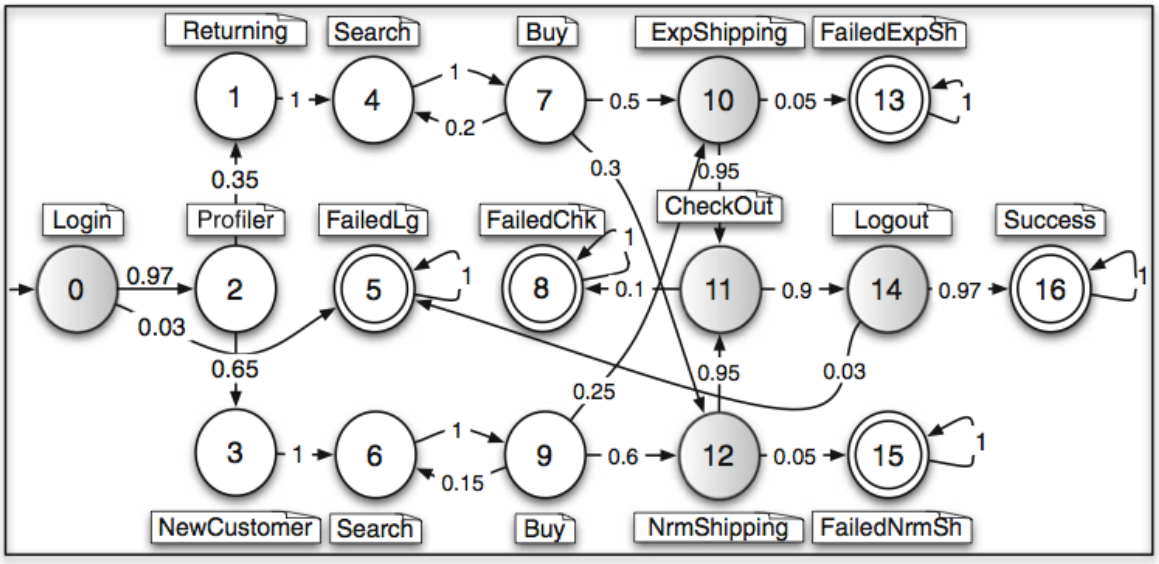


Figure F.1: The parameterised discreat-time Markov chain—modelling the reliability of the ecommerce application—taken from [79]

# Appendix G

# Travel application domain

Our third case study is a travel assistant application whose workflow organises the invocation of services relating to accommodation booking and vehicle rental, based on the travel planner in [164]. Requests arriving to the workflow can either be invocations to search for flights (90%) or search for tourist attractions at a specific destination (10%), when accommodation arrangements have otherwise been made. The drive time between the hotel and the attraction is calculated and either a car (70%) or bicycle (30%) is hired based on the distance between locations.

The workflow comprises the following five services

1. Attraction information service, which provides the operation *Attract*.
2. Flight reservation and booking service, which provides the operation *Flight*.
3. Hotel reservation service, which provides the operation *Hotel*.
4. Bicycle rental service, which provides the operation *Bike*.
5. Car hire service, which provides the operation *Car*.

**Requirements** The autonomic manager is used to maintain the following high-level QoS requirements:

1. More than 80% of users invoke the bike rental service.
2. Less than 25% of invocations end in failure.
3. More than 80% of people who search for attractions rent a bike.

Formulated as PCTL formulae:

1. $P > 0.8[F(s = 11)]$.
2. $P < 0.25[F(s = 4|s = 5|s = 7|s = 9|s = 12)]$.
3. $filter(forall, P > 0.8[F(s = 11)], s = 2)$.

The expected cost of a single invocation of the workflow is determined by the PCTL formula "$R =?[Fs = 13]$", relative to the rewards structure associating a monetary value to each abstract operation named in the travel assistant workflow, appropriate to the concrete services provided by the user of COVE to the autonomic manager.

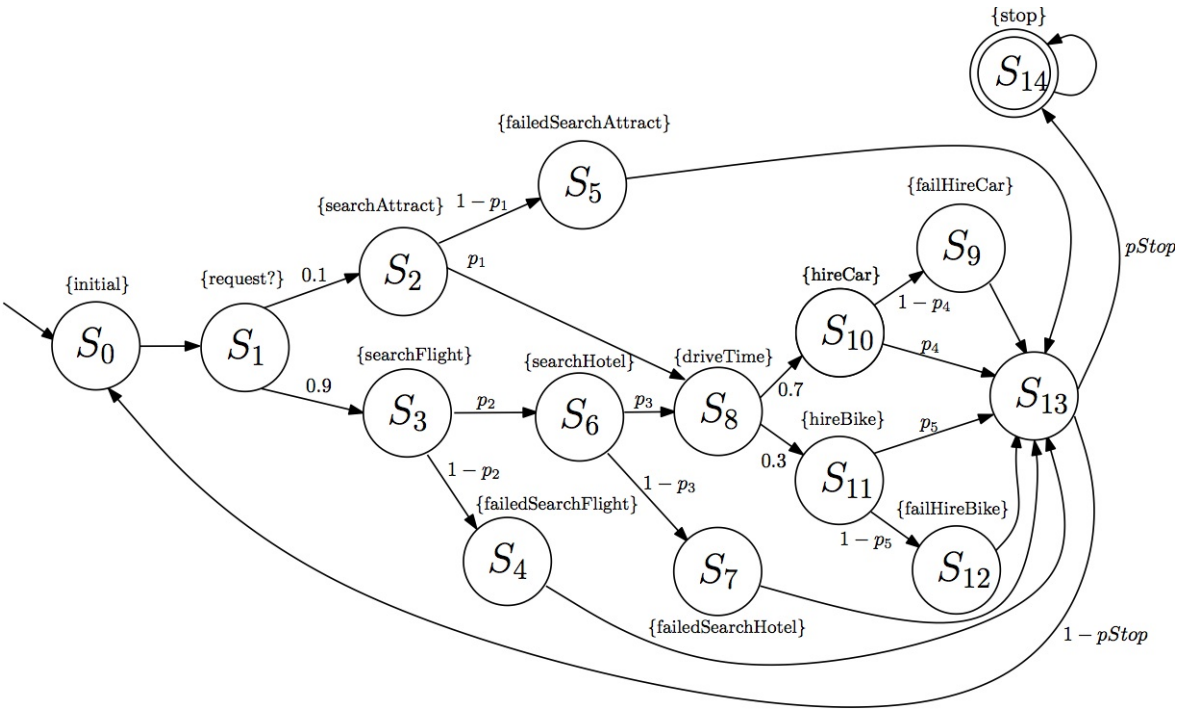


Figure G.1: The parameterised discreat-time Markov chain—modelling the reliability of the travel assistant application

# Chapter 7

# Bibliography


[1] W. Afzal, R. Torkar, and R. Feldt. A Systematic Review of Search-Based Testing for Non-Functional System Properties. *Information and Software Technology*, 51(6):957–976, 2009.

[2] M. Algreer, M. Armstrong, and D. Giaouris. Active Online System Identification of Switch Mode DC–DC Power Converter Based on Efficient Recursive DCD-IIR Adaptive Filter. *IEEE Transactions on Power Electronics*, 27(11):4425–4435, 2012.

[3] C. Am, K. Jihoon, L. Sanghyo, and K. Changdon. Wind Estimation and Airspeed Calibration Using a UAV with a Single-Antenna GPS Receiver and Pitot Tube. *IEEE Transactions on Aerospace and Electronic Systems*, 47(1):109–117, Jan 2011.

[4] U. Amann, S. Gtz, J. Jzquel, B. Morin, and M. Trapp. A Reference Architecture and Roadmap for Models@run.time Systems. In *Models@run.time*, volume 8378, pages 1–18. Springer International Publishing, 2014.

[5] M. Aoyama. Computing for the Next-Generation Automobile. *Computer Society*, 45(6):32–37, 2012.

[6] D. Arcelli, V. Cortellessa, and D. Di Ruscio. Applying Model Differences to Automate Performance-Driven Refactoring of Software Models. In *Computer Performance Engineering*, volume 8168, pages 312–324. Springer Berlin Heidelberg, 2013.

[7] D. Ardagna, L. Baresi, S. Comai, M. Comuzzi, and B. Pernici. A Service-Based Framework for Flexible Business Processes. *IEEE Transactions on Software Engineering*, 28(2):61–67, 2011.

[8] D. Ardagna, E. Di Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C.S. Nechifor, D. Petcu, et al. MODAClouds: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds. In *ICSE Workshop on Modeling in Software Engineering*, pages 50–56, Jun 2012.

[9] D. Ardagna, C. Ghezzi, and R. Mirandola. Rethinking the Use of Models in Software Architecture. In *Proceedings of the 4th International Conference on Quality of Software-Architectures: Models and Architectures*, pages 1–27, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] D. Ardagna and B. Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 33(6):369–384, Jun 2007.

[11] A. S. Ashour. LPA Beamformer for Tracking Nonstationary Accelerated Near-Field Sources. *International Journal of Advanced Computer Science & Applications*, 5(3), 2014.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[13] C. Baier, B. Havekort, H. Hermanns, and J. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, Jun 2003.

[14] C. Baier, B. Haverkort, H. Hermanns, and J. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, Jun 2003.

[15] S. Balsamo, A. Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: a Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[16] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of Web Service Compositions. *IET Software*, 1:219–232(13), Dec 2007.

[17] L. Baresi and C. Ghezzi. The Disappearing Boundary Between Development-Time and Run-Time. In *FSE/SDP Workshop on Future of Software Engineering Research*, pages 17–22, New York, NY, USA, 2010. ACM.

[18] L. Baresi and S. Guinea. Self-Supervising BPEL Processes. *IEEE Transactions on Software Engineering*, 37(2):247–263, 2011.

[19] N. Bencomo and A. Belaggoun. Supporting Decision-Making for Self-Adaptive Systems: From Goal Models to Dynamic Decision Networks. In *Requirements Engineering: Foundation for Software Quality*, volume 7830, pages 221–236. Springer Berlin Heidelberg, 2013.

[20] N. Bencomo, A. Belaggoun, and V. Issarny. Dynamic Decision Networks for Decision-Making in Self-Adaptive Systems: A Case Study. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 113–122. IEEE, 2013.

[21] P. Bocciarelli and A. D´ Ambrogio. A Model-Driven Method for Enacting the Design-Time QoS Analysis of Business Processes. *Software & Systems Modeling*, 13(2):573–598, 2014.

[22] C. Böhm and G. Jacopini. Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[23] G. Bolch, S. Greiner, H. De Meer, and K.S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. John Wiley & Sons, 2006.

[24] A. Bry, A. Bachrach, and N. Roy. State Estimation for Aggressive Flight in GPS-Denied Environments Using Onboard Sensing. In *IEEE International Conference on Robotics and Automation*, pages 1–8, May 2012.

[25] R. Calinescu. Reconfigurable Service-Oriented Architecture for Autonomic Computing. *International Journal on Advances in Intelligent Systems*, 2(1):38–57, 2009.

[26] R. Calinescu. Emerging Techniques for the Engineering of Self-Adaptive High-Integrity Software. In *Assurances for Self-Adaptive Systems*, volume 7740, pages 297–310. Springer Berlin Heidelberg, 2013.

[27] R. Calinescu. Emerging Techniques for the Engineering of Self-Adaptive High-Integrity Software. In *Assurances for Self-Adaptive Systems*, volume 7740, pages 297–310. Springer Berlin Heidelberg, 2013.

[28] R. Calinescu et al. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Transactions on Software Engineering*, 37(3):387 –409, May-Jun 2011.

[29] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. In *IEEE Transactions on Software Engineering*, volume 37, pages 387–409, May 2011.

[30] R. Calinescu, K. Johnson, and Y. Rafiq. Using Observation Ageing to Improve Markovian Model Learning in QoS Engineering. In *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, pages 505–510, New York, NY, USA, 2011. ACM.

[31] R. Calinescu, K. Johnson, and Y. Rafiq. Developing Self-Verifying Service-Based Systems. In *28th IEEE/ACM International Conference on Automated Software Engineering*, pages 734–737, 2013.

[32] R. Calinescu and S. Kikuchi. Formal Methods @ Runtime. In *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662, pages 122–135. Springer Berlin Heidelberg, 2011.

[33] R. Calinescu, S. Kikuchi, and K. Johnson. Compositional Reverification of Probabilistic Safety Properties for Large-Scale Complex IT Systems. In *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539, pages 303–329. Springer Berlin Heidelberg, 2012.

[34] R. Calinescu and M. Kwiatkowska. CADS*: Computer-Aided Development of Self-* Systems. *Fundamental Approaches to Software Engineering*, 5503:421–424, 2009.

[35] R. Calinescu and Y. Rafiq. Using Intelligent Proxies to Develop Self-Adaptive Service-Based Systems. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 131–134, Jul 2013.

[36] R. Calinescu, Y. Rafiq, K. Johnson, and M.E. Bakir. Adaptive Model Learning for Continual Verification of Non-Functional Properties. In *5th ACM/SPEC International Conference on Performance Engineering*, 2014.

[37] G. Canfora, M.D. Penta, R. Esposito, and M.L. Villani. A Framework for QoS-Aware Binding and Re-Binding of Composite Web Services. *Journal of Systems and Software*, 81(10):1754 – 1769, 2008.

[38] L. Cao, J. Cao, and M. Li. Genetic Algorithm Utilized in Cost-Reduction Driven Web Service Selection. In *Computational Intelligence and Security*, volume 3802, pages 679–686. Springer Berlin Heidelberg, 2005.

[39] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola. Moses: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. *IEEE Transactions on Software Engineering*, 38(5):1138–1159, 2012.

[40] I.X. Chen, Y.C. Wu, I.C. Liao, and Y.Y. Hsu. A High-Scalable Core Telematics Platform Design for Intelligent Transport Systems. In *12th International Conference on ITS Telecommunications*, pages 412–417, Nov 2012.

[41] M.C. Chen, J.L. Chen, and T.W. Chang. Android/OSGi-Based Vehicular Network Management System. *Computer Communications*, 34(2):169–183, 2011.

[42] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. Automatic Verification of Competitive Stochastic Systems. *Formal Methods in System Design*, 43(1):61–92, 2013.

[43] T. Chen, M. Kwiatkowska, D. Parker, and A. Simaitis. Verifying Team Formation Protocols with Probabilistic Model Checking. In *Computational Logic in Multi-Agent Systems*, volume 6814, pages 190–207. Springer Berlin Heidelberg, 2011.

[44] Y. Chen, H. Mao, M. Jaeger, T. Nielsen, K. Guldstrand Larsen, and B. Nielsen. Learning Markov Models for Stationary System Behaviors. In *NASA Formal Methods*, volume 7226, pages 216–230. Springer Berlin Heidelberg, 2012.

[45] B.C. Cheng, K. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. Mller, P. Pelliccione, A. Perini, N. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, volume 8378, pages 101–136. Springer International Publishing, 2014.

[46] S.W. Cheng, V. Poladian, D. Garlan, and B. Schmerl. Improving Architecture-Based Self-Adaptation Through Resource Prediction. *Software Engineering for Self-Adaptive Systems*, 5525:71–88, 2009.

[47] S.J. Chern, M.K. Cheng, and P.S. Chao. Blind Capon-Like Adaptive ST-BC MIMO-CDMA Receiver Based on Constant Modulus Criterion. *Digital Signal Processing*, 23(6):1958 – 1966, 2013.

[48] S.J. Chern, W.C. Huang, and R.H.H. Yang. Adaptive Semi-Blind Channel Estimation for ST-BC MIMO-CDMA Systems with Hybrid User Signature. In *International Symposium on Intelligent Signal Processing and Communications Systems*, pages 369–374, Nov 2013.

[49] L. Cheung, S. Banerjee, N. Medvidovic, L. Golubchik, and R. Roshandel. Estimating software component reliability by leveraging architectural models. *Software Engineering, International Conference on*, 0:853–856, 2006.

[50] F. Ciesinski and M. Grer. On Probabilistic Computation Tree Logic. In *Validation of Stochastic Systems*, volume 2925, pages 147–188. Springer Berlin Heidelberg, 2004.

[51] V. Cortellessa. Performance Antipatterns: State-of-Art and Future Perspectives. In M. Balsamo, W.J. Knottenbelt, and A. Marin, editors, *Computer Performance Engineering*, volume 8168 of *Lecture Notes in Computer Science*, pages 1–6. Springer Berlin Heidelberg, 2013.

[52] V. Cortellessa, A. Di Marco, and C. Trubiani. Performance Antipatterns as Logical Predicates. In *15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 146–156, Mar 2010.

[53] V. Cortellessa, A. Di Marco, and C. Trubiani. An Approach for Modeling and Detecting Software Performance Antipatterns Based on First-Order Logics. *Software & Systems Modeling*, 13(1):391–432, 2014.

[54] V. Cortellessa, A.D. Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 1st edition, 2011.

[55] V. Cortellessa, F. Marinelli, R. Mirandola, and P. Potena. Quantifying the Influence of Failure Repair/Mitigation Costs on Service-Based Systems. In *24th IEEE International Symposium on Software Reliability Engineering*, pages 90–99, Nov 2013.

[56] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press Ltd., London, UK, 1972.

[57] R. de Lemos, H. Giese, H. Mller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In *Software Engineering for Self-Adaptive Systems II*, volume 7475, pages 1–32. Springer Berlin Heidelberg, 2013.

[58] C. Delpierre, L. Cuzin, J. Fillaux, M. Alvarez, P. Massip, and T. Lang. A Systematic Review of Computer-Based Patient Record Systems and Quality of Care: More Randomized Clinical Trials or a Broader Approach? *International Journal for Quality in Health Care*, 16(5):407–416, 2004.

[59] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel. Sound Development of Secure Service-Based Systems. In *2nd International Conference on Service Oriented Computing*, pages 115–124, New York, NY, USA, 2004. ACM.

[60] S. Doclo, S. Gannot, M. Moonen, and A. Spriet. Acoustic Beamforming for Hearing Aid Applications. *Handbook on Array Processing and Sensor Networks*, pages 269–302, 2008.

[61] S. Duri, J. Elliott, M. Gruteser, X. Liu, P. Moskowitz, R. Perez, M. Singh, and J.M. Tang. Data Protection and Data Sharing in Telematics. *Mobile Networks and Applications*, 9(6):693–701, 2004.

[62] S. Duri, M. Gruteser, X. Liu, P. Moskowitz, R. Perez, M. Singh, and J.M. Tang. Framework for Security and Privacy in Automotive Telematics. In *2nd International Workshop on Mobile Commerce*, pages 25–32, New York, NY, USA, 2002. ACM.

[63] F. El-Hawary. A Comparison of Recursive Weighted Least Squares Estimation and Kalman Filtering for Source Dynamic Motion Evaluation. *Canadian Journal of Electrical and Computer Engineering*, 17(3):136–145, Jul 1992.

[64] V.C. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar. Low Level Metrics to High Level SLAs - LoM2HiS Framework: Bridging the Gap Between Monitored Metrics and SLA Parameters in Cloud Environments. In *International Conference on High Performance Computing and Simulation*, pages 48–54, Jun 2010.

[65] W. Emfinger, G. Karsai, A. Dubey, and A. Gokhale. Analysis, Verification, and Management Toolsuite for Cyber-Physical Applications on Time-Varying Networks. In *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, pages 44–47, New York, NY, USA, 2014. ACM.

[66] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model Evolution by Run-Time Parameter Adaptation. In *31st IEEE International Conference on Software Engineering*, pages 111–121, May 2009.

[67] C.C. Eriksen, T.J. Osse, R.D. Light, T. Wen, T.W. Lehman, P.L. Sabin, J.W. Ballard, and A.M. Chiodi. Seaglider: A Long-Range Autonomous Underwater Vehicle for Oceanographic Research. *IEEE Journal on Oceanic Engineering*, 26(4):424–436, Oct 2001.

[68] N. Esfahani and S. Malek. Guided Exploration of the Architectural Solution Space in the Face of Uncertainty. Technical report, Department of Computer Science, George Mason University, March, 2011.

[69] N. Esfahani, S. Malek, and K. Razavi. GuideArch: Guiding the Exploration of Architectural Solution Space Under Uncertainty. In *International Conference on Software Engineering*, pages 43–52, Piscataway, NJ, USA, 2013. IEEE Press.

[70] M.A. Esteve, J. Katoen, V.Y. Nguyen, B. Postma, and Y. Yushtein. Formal Correctness, Safety, Dependability, and Performance Analysis of a Satellite. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1022–1031, Piscataway, NJ, USA, 2012. IEEE Press.

[71] A. Filieri and C. Ghezzi. Further Steps Towards Efficient Runtime Verification: Handling Probabilistic Cost Models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 2–8, Jun 2012.

[72] A. Filieri, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Conquering Complexity via Seamless Integration of Design-Time and Run-Time Verification. In *Conquering Complexity*, pages 253–275. Springer London, 2012.

[73] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-Time Efficient Probabilistic Model Checking. In *Proceeding of the 33rd International Conference on Software Engineering*, pages 341–350, New York, NY, USA, 2011. ACM.

[74] A. Filieri, C. Ghezzi, and G. Tamburrelli. A Formal Approach to Adaptive Software: Continuous Assurance of Non-Functional Requirements. In *Formal Aspects of Computing*, volume 24, pages 163–186. Springer-Verlag, 2012.

[75] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated Verification Techniques for Probabilistic Systems. In *Formal Methods for Eternal Networked Software Systems*, volume 6659, pages 53–113. Springer Berlin Heidelberg, 2011.

[76] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. Incremental Runtime Verification of Probabilistic Systems. In *Runtime Verification*, volume 7687, pages 314–319. Springer Berlin Heidelberg, 2013.

[77] R. Frei, G. Di Marzo Serugendo, and J. Barata. Designing Self-Organization for Evolvable Assembly Systems. *2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 97–106, Oct 2008.

[78] B. Friedman and P.H. Kahn Jr. Human Agency and Responsible Computing: Implications for Computer System Design. *Journal of Systems and Software*, 17:7–14, 1992.

[79] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality Prediction of Service Compositions Through Probabilistic Model Checking. In *Quality of Software Architectures. Models and Architectures*, volume 5281, pages 119–134. Springer Berlin Heidelberg, 2008.

[80] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality Prediction of Service Compositions through Probabilistic Model Checking. In S. Becker, F. Plasil, and R. Reussner, editors, *Quality of Software Architectures. Models and Architectures*, volume 5281 of *Lecture Notes in Computer Science*, pages 119–134. Springer Berlin Heidelberg, 2008.

[81] H. Gao, H. Miao, and H. Zeng. Service Reconfiguration Architecture Based on Probabilistic Modeling Checking. In *IEEE International Conference on Web Services*, pages 714–715, Jun 2014.

[82] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 2 of *Series on Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1995.

[83] H. Ghanbari, C. Barna, M. Litoiu, M. Woodside, T. Zheng, J. Wong, and G. Iszlai. Tracking Adaptive Performance Models Using Dynamic Clustering of User Classes. In *ACM SIGSOFT Software Engineering Notes*, volume 36, pages 179–188. ACM, 2011.

[84] C. Ghezzi and A. Molzam Sharifloo. Model-Based Verification of Quantitative Non-Functional Properties for Software Product Lines. *Information and Software Technology*, 55(3):508–524, 2013.

[85] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining Behavior Models from User-Intensive Web Applications. In *International Conference on Software Engineering*, pages 277–287, 2014.

[86] C. Ghezzi and G. Tamburrelli. Predicting Performance Properties for Open Systems with KAMI. In R. Mirandola, I. Gorton, and C. Hofmeister, editors, *Architectures for Adaptive Software Systems*, volume 5581 of *Lecture Notes in Computer Science*, pages 70–85. Springer Berlin Heidelberg, 2009.

[87] C. Girish and J. Ravindra. Aerodynamic Parameter Estimation from Flight Data Applying Extended and Unscented Kalman Filter. *Aerospace Science and Technology*, 14(2):106 – 117, 2010.

[88] A. Gorla, M. Pezze, J. Wuttke, L. Mariani, and F. Pastore. Achieving Cost-Effective Software Reliability Through Self-Healing. *Computing and Informatics*, 29(1):93–115, 2012.

[89] C.M. Grinstead and L.J. Snell. *Grinstead and Snell's Introduction to Probability*. American Mathematical Society, july edition, 2006.

[90] L. Grunske. Specification Patterns for Probabilistic Quality Properties. In *30th ACM/IEEE International Conference on Software Engineering*, pages 31–40, May 2008.

[91] Z. Gu, Z. Wang, S. Li, and H. Cai. Design and Implementation of an Automotive Telematics Gateway Based on Virtualization. In *15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 53–58, Apr 2012.

[92] R. Hamadi and B. Benatallah. A Petri Net-Based Model for Web Service Composition. In *14th Australasian Database Conference - Volume 17*, pages 191–200, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

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

[94] M.T. Haupt, C.E. Bekes, R.J. Brilli, L.C. Carl, A.W. Gray, M.S. Jastremski, D.F. Naylor, A. Spevetz, S.K. Wedel, and M. Horst. Guidelines on Critical Care Services and Personnel: Recommendations Based on a System of Categorization of Three Levels of Care*. *Critical Care Medicine*, 31(11):2677–2683, 2003.

[95] H. Hermanns, U. Herzog, and J. Katoen. Process Algebra for Performance Evaluation. *Theoretical Computer Science*, 274(1):43–87, 2002.

[96] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer Berlin / Heidelberg, 2006.

[97] Z. Huan, W. Xiaofeng, and S. Jinshu. A General Self-Adaptive Reputation System Based on the Kalman Feedback. In *International Conference on Service Sciences*, pages 7–12, Apr 2013.

[98] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M.R. Pocock, P. Li, and T. Oinn. Taverna: A Tool for Building and Running Workflows of Services. *Nucleic Acids Research*, 34(suppl 2):W729–W732, 2006.

[99] M.R. Hussain, A. Zainal, W.M. Elmedany, and M.W. Fakhr. Telematics Business and Management in Bahrain Market. *Transport and Telecommunication*, 14(1):13–19, 2013.

[100] S.Y. Hwang, H. Wang, J. Tang, and J. Srivastava. A Probabilistic Approach to Modeling and Estimating the QoS of Web-Services-Based Workflows. *Information Sciences*, 177(23):5484–5503, 2007.

[101] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. In C. Flanagan and B. Knig, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 188–203. Springer Berlin Heidelberg, 2012.

[102] K. Johnson, R. Calinescu, and S. Kikuchi. An Incremental Verification Framework for Component-Based Software Systems. In *16th International ACM Sigsoft Symposium on Component-based Software Engineering*, pages 33–42, New York, NY, USA, 2013. ACM.

[103] J. Katoen, M. Khattri, and I. Zapreev. A Markov Reward Model Checker. In *2nd International Conference on the Quantitative Evaluation of Systems*, pages 243–244, Sept 2005.

[104] J. Katoen, I.S. Zapreev, E.M. Hahn, H. Hermanns, and D.N. Jansen. The Ins and Outs of the Probabilistic Model Checker. *Performance Evaluation*, 68(2):90–104, 2011.

[105] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan 2003.

[106] M. Kiljima. *Markov Processes for Stochastic Modeling*, volume 6. Cambridge University Press, 1997.

[107] J.C. Knight. Safety Critical Systems: Challenges and Directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–550, May 2002.

[108] D. Kumar, A. Tantawi, and L. Zhang. Estimating model parameters of adaptive software systems in real-time. In *Run-Time Models for Self-Managing Systems and Applications*, pages 45–71. Springer Basel, 2010.

[109] M. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker {PRISM}. *Electronic Notes in Theoretical Computer Science*, 153(2):5 – 31, 2006. Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005) Quantitative Aspects of Programming Languages 2005.

[110] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic Model Checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 220–270. Springer Berlin Heidelberg, 2007.

[111] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer Berlin Heidelberg, 2011.

[112] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-Guarantee Verification for Probabilistic Systems. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin Heidelberg, 2010.

[113] M. Kwiatkowska, D. Parker, and H. Qu. Incremental Quantitative Verification for Markov Decision Processes. *41st IEEE/IFIP International Conference on Dependable Systems Networks*, pages 359–370, Jun 2011.

[114] A.K. Lammas, K. Sammut, and H. Fangpo. A 6 DoF Navigation Algorithm for Autonomous Underwater Vehicles. In *OCEANS - Europe*, pages 1–6, Jun 2007.

[115] E. Letier, D. Stefan, and E.T. Barr. Uncertainty, Risk, and Information Value in Software Requirements and Architecture. In *36th International Conference on Software Engineering*, pages 883–894, New York, NY, USA, 2014. ACM.

[116] Q. Liang, X. Wu, and H.C. Lau. Optimizing Service Systems Based on Application-Level QoS. *IEEE Transactions on Services Computing*, 2(2):108–121, Apr 2009.

[117] H. Liu, F. Zhong, B. Ouyang, and J. Wu. An Approach for QoS-Aware Web Service Composition Based on Improved Genetic Algorithm. In *International Conference on Web Information Systems and Mining*, volume 1, pages 123–128, Oct 2010.

[118] Z. Liu, Z. Jia, X. Xue, and J. An. Reliable Web Service Composition Based on QoS Dynamic Prediction. *Soft Computing*, pages 1–17, 2014.

[119] D. Loebis, R. Sutton, J. Chudley, and W. Naeem. Adaptive Tuning of a Kalman Filter via Fuzzy Logic for an Intelligent AUV Navigation System. *Control Engineering Practice*, 12(12):1531 – 1539, 2004. Guidance and control of underwater vehicles.

[120] J.P. López-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 25–36. ACM, 2004.

[121] Y. Ma and C. Zhang. Quick Convergence of Genetic Algorithm for QoS-Driven Web Service Selection. *Computer Networks*, 52(5):1093 – 1104, 2008.

[122] F. Maggi, M. Westergaard, M. Montali, and W.P. van der Aalst. Runtime Verification of LTL-Based Declarative Process Models. In S. Khurshid and K. Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 131–146. Springer Berlin Heidelberg, 2012.

[123] H. Mao, Y. Chen, M. Jaeger, T.D. Nielsen, K.G. Larsen, and B. Nielsen. Learning Probabilistic Automata for Model Checking. In *8th International Conference on Quantitative Evaluation of Systems*, pages 111–120, Sep 2011.

[124] M. Marzolla and R. Mirandola. Performance Prediction of Web Service Workflows. *Software Architectures, Components, and Applications*, 4880:127–144, 2007.

[125] M. L. Massie, N. B. Chun, and E. D. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[126] J.L. Mathieu, S. Koch, and D.S. Callaway. State Estimation and Control of Electric Loads to Manage Real-Time Energy Imbalance. *IEEE Transactions on Power Systems*, 28(1):430–440, 2013.

[127] D.A. Menasc, H. Ruan, and H. Gomaa. QoS Management in Service-Oriented Architectures. *Performance Evaluation*, 64(78):646 – 663, 2007.

[128] A. Metzger, O. Sammodi, and K. Pohl. Accurate Proactive Adaptation of Service-Oriented Systems. In *Assurances for Self-Adaptive Systems*, volume 7740, pages 240–265. Springer Berlin Heidelberg, 2013.

[129] A. Mosincat and W. Binder. Automated Maintenance of Service Compositions with SLA Violation Detection and Dynamic Binding. *International Journal on Software Tools for Technology Transfer*, 13(2):167–179, 2011.

[130] A. Mosincat, W. Binder, and M. Jazayeri. Achieving Runtime Adaptability Through Automated Model Evolution and Variant Selection. *Enterprise Information Systems*, 8(1):67–83, 2014.

[131] K. Nam, S. Oh, H. Fujimoto, and Y. Hori. Estimation of Sideslip and Roll Angles of Electric Vehicles Using Lateral Tire Force Sensors Through RLS and Kalman Filter Approaches. *IEEE Transactions on Industrial Electronics*, 60(3):988–1000, Mar 2013.

[132] E. Oberortner, S. Sobernig, U. Zdun, and S. Dustdar. Monitoring Performance-Related QoS Properties in Service-Oriented Systems: A Pattern-Based Architectural Decision Model. In *16th European Conference on Pattern Languages of Programs*, pages 13:1–13:37, New York, NY, USA, 2012. ACM.

[133] E. Oberortner, S. Sobernig, U. Zdun, and S. Dustdar. Monitoring Performance-Related QoS Properties in Service-Oriented Systems: A Pattern-Based Architectural Decision Model. In *16th European Conference on Pattern Languages of Programs*, pages 13:1–13:37, New York, NY, USA, 2012. ACM.

[134] J.A. Parejo, S. Segura, P. Fernandez, and A. Ruiz-Cortés. QoS-Aware Web Services Composition Using GRASP with Path Relinking. *Expert Systems with Applications*, 41(9):4211–4223, 2014.

[135] R. Pietrantuono, S. Russo, and K.S. Trivedi. Online Monitoring of Software System Reliability. In *European Dependable Computing Conference*, pages 209–218, Apr 2010.

[136] L. Pike, S. Niller, and N. Wegmann. Runtime Verification for Ultra-Critical Systems. In *Runtime Verification*, volume 7186, pages 310–324. Springer Berlin Heidelberg, 2012.

[137] H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: A Framework for Extrapolating Behavioral Models. *International Journal on Software Tools for Technology Transfer*, 11(5):393–407, 2009.

[138] F. Raimondi, J. Skene, and W. Emmerich. Efficient Online Monitoring of Web-Service SLAs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 170–180, New York, NY, USA, 2008. ACM.

[139] F. Raimondi, J. Skene, and W. Emmerich. Efficient Online Monitoring of Web-Service SLAs. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 170–180, New York, NY, USA, 2008. ACM.

[140] R. Roshandel. Calculating architectural reliability via modeling and analysis. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 69–71, Washington, DC, USA, 2004. IEEE Computer Society.

[141] R. Roshandel and N. Medvidovic. Toward architecture-based reliability estimation. *IET Conference Proceedings*, pages 2–6(4), 2004.

[142] B. Sabata, S. Chatterjee, M. Davis, J.J. Sydir, and T.F. Lawrence. Taxonomy for qos specifications. In *Object-Oriented Real-Time Dependable Systems, 1997. Proceedings., Third International Workshop on*, pages 100–107, Feb 1997.

[143] F. Saffre, R. Tateson, J. Halloy, M. Shackleton, and J.L. Deneubourg. Aggregation Dynamics in Overlay Networks and Their Implications for Self-Organized Distributed Applications. *The Computer Journal*, 52(4):397–412, 2009.

[144] M. Sango, L. Duchien, and C. Gransart. Component-Based Modeling and Observer-Based Verification for Railway Safety-Critical Applications. In *11th International Symposium on Formal Aspects of Component Software*, Bertinoro, Italy, September 2014.

[145] N. Sato and K.S. Trivedi. Stochastic Modeling of Composite Web Services for Closed-Form Analysis of Their Performance and Reliability Bottlenecks. *5th International Conference Service-Oriented Computing*, 4749:107–118, 2007.

[146] K. Sen, M. Viswanathan, and G. Agha. Learning Continuous Time Markov Chains from Sample Executions. In *1st International Conference on the Quantitative Evaluation of Systems)*, pages 146–155, Sep 2004.

[147] H. Song and K. Lee. sPAC (web Services Performance Analysis Center): Performance Analysis and Estimation Tool of Web Services. In W.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 109–119. Springer Berlin Heidelberg, 2005.

[148] G. Spanoudakis and K. Mahbub. Non-Intrusive Monitoring of Service-Based Systems. *International Journal of Cooperative Information Systems*, 15(03):325–358, 2006.

[149] C. Strelioff, J. Crutchfield, and A. Hübler. Inferring Markov Chains: Bayesian Estimation, Model Comparison, Entropy Rate, and Out-of-Class Modeling. *Physical Review E. Statistical, Nonlinear, and Soft Matter Physics*, 76:011106, Jul 2007.

[150] Y. Sun, J. White, and J. Gray. Model transformation by demonstration. In A. Schrr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 712–726. Springer Berlin Heidelberg, 2009.

[151] G. Tamura, N. Villegas, H. Mller, J. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezz, W. Schfer, L. Tahvildari, and K. Wong. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In R. de Lemos, H. Giese, H.A. Mller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 108–132. Springer Berlin Heidelberg, 2013.

[152] M.H. Tran, A. Colman, and J. Han. Service-Based Development of Context-Aware Automotive Telematics Systems. In *15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 53–62, Mar 2010.

[153] M. Trapp and D. Schneider. Safety Assurance of Open Adaptive Systems A Survey. In *Models@run.time*, volume 8378, pages 279–318. Springer International Publishing, 2014.

[154] A. Vaccaro, M. Popov, D. Villacci, and V. Terzija. An Integrated Framework for Smart Microgrids Modeling, Monitoring, Control, Communication, and Verification. *Proceedings of the IEEE*, 99(1):119–132, Jan 2011.

[155] W.M. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(01):21–66, 1998.

[156] N.M. Villegas, H.A. Müller, G. Tamura, L. Duchien, and R. Casallas. A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 80–89, New York, NY, USA, 2011. ACM.

[157] Z.C. Wang, W.X. Ren, and G.D. Chen. Time-Varying System Identification of High Voltage Switches of a Power Substation with Slide-Window Least-Squares Parameter Estimations. *Smart Materials and Structures*, 22(6):065023, 2013.

[158] D. Weyns, M.U. Iftikhar, D.G. de la Iglesia, and T. Ahmad. A Survey of Formal Methods in Self-Adaptive Systems. In *Proceedings of the 5th International C* Conference on Computer Science and Software Engineering*, pages 67–79, New York, NY, USA, 2012. ACM.

[159] S.S. Yau and H.G. An. Adaptive Resource Allocation for Service-Based Systems. *Proceedings of the First Asia-Pacific Symposium on Internetware*, pages 3:1–3:7, 2009.

[160] S.S. Yau and Y. Yin. QoS-Based Service Ranking and Selection for Service-Based Systems. In *IEEE International Conference on Services Computing*, pages 56–63, Jul 2011.

[161] B. Ye, M. Ghavami, A. Pervez, and M. Nekovee. An Automatic Trust Calculation Based on the Improved Kalman Filter Detection Algorithm. In *Trust Management VII*, volume 401, pages 208–222. Springer Berlin Heidelberg, 2013.

[162] H.S. Younes. Ymer: A statistical model checker. In K. Etessami and S.K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 429–433. Springer Berlin Heidelberg, 2005.

[163] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, January 1997.

[164] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Duma, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.

[165] C. Zhang, S. Su, and J. Chen. DiGA: Population Diversity Handling Genetic Algorithm for QoS-Aware Web Services Selection. *Computer Communications*, 30(5):1082 – 1090, 2007.

[166] C. Zhao, W. Zhang, B. Li, Y. Liu, and Z. Li. An Efficient Hybrid Beamforming for Uplink Transmissions of 60GHz Millimeter-Wave Communications. In *Communications, Signal Processing, and Systems*, volume 202, pages 283–291. Springer New York, 2012.

[167] W.X. Zhao and T. Zhou. Weighted Least Squares Based Recursive Parametric Identification for the Submodels of a PWARX system. *Automatica*, 48(6):1190–1196, 2012.

[168] X. Zhao, Z. Wen, and X. Li. QoS-Aware Web Service Selection with Negative Selection Algorithm. *Knowledge and Information Systems*, 40(2):349–373, 2014.

[169] T. Zheng, M. Woodside, and M. Litoiu. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering*, 34(3):391–406, May 2008.

[170] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai. Tracking Time-Varying Parameters in Software Systems with Extended Kalman Filters. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 334–345. IBM Press, 2005.

[171] J. Zhou, Y. Lu, and K. Lundqvist. A TASM-Based Requirements Validation Approach for Safety-Critical Embedded Systems. In L. George and T. Vardanega, editors, *Reliable Software Technologies Ada-Europe*, volume 8454 of *Lecture Notes in Computer Science*, pages 43–57. Springer International Publishing, 2014.

[172] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *Evolutionary Computation, IEEE Transactions on*, 3(4):257–271, Nov 1999.

[173] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, and V.G. da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *Evolutionary Computation, IEEE Transactions on*, 7(2):117–132, April 2003.