Probabilistic Model Checking Techniques for the Verification and Synthesis of Software Systems

Naif Alasmari

DOCTOR OF PHILOSOPHY

UNIVERSITY OF YORK

COMPUTER SCIENCE

FEBRUARY 2021

Abstract

Probabilistic model checking is a mathematically based technique widely used to verify whether systems with stochastic behaviour satisfy their nonfunctional requirements, and to synthesise new system designs that comply with such requirements. Nevertheless, the technique has a number of limitations. First, the validity of the verification depends on the accuracy of the models being verified—with invalid verification results causing wrong software engineering decisions. Second, the variant of the technique used to perform verification under parametric uncertainty is computationally very expensive. Finally, the synthesis of stochastic models (corresponding to desirable system designs/configurations) is typically only possible for simple requirement combinations, and for discrete-time models.

This thesis provides solutions to these limitations. First, we introduce a method for the efficient and accurate verification of nonfunctional requirements under parametric uncertainty. This method collects additional data about the parameters of the verified model by unit-testing specific system components over a series of verification iterations. The components tested in each iteration are decided based on the sensitivity of the model to variations in the parameters of different components, and the overheads (time or cost) of unit-testing these components.

Second, we extend the applicability of probabilistic model checking under parametric uncertainty to larger models than currently possible by leveraging recent advances from the area of parametric model checking.

Third, we propose a new method for synthesising Pareto-optimal Markov decision process (MDP) policies that satisfy complex requirement combinations. These policies correspond to optimal system designs or configurations, and are obtained by translating the MDPs into parametric Markov chains, and using search-based software engineering techniques to synthesise Pareto-optimal parameter values that define optimal policies for the original MDP.

Finally, we present an approach for the synthesis of continuous-time Markov decision process (CTMDP) policies—a problem not addressed by existing probabilistic model checkers. This approach generates policies that define software-system configurations or cyber-physical system controllers that meet predefined nonfunctional constraints and achieve optimal trade-offs between multiple optimisation objectives.

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to thank King Khalid University for granting me this opportunity to pursue my postgraduate studies and for its unwavering support. I also extend my thanks to my colleagues at the College of Arts and Sciences for their encouragement and support during my PhD journey.

I would like to extend my sincere gratitude to my supervisor, Professor Radu Calinescu, for his assistance, guidance, support and feedback as I worked on my PhD. He inspired me to work tirelessly to acquire new skills throughout both my past studies and my future academic life.

I am deeply grateful to my dear mother for her constant encouragement and support with her soothing words and beautiful prayers. She was, and still is, a great motivator for me to achieve my goals.

I want to sincerely thank my dear wife and beloved children for their encouragement and patience. I greatly appreciate their understanding of my research commitments during this wonderful journey.

Lastly, my thanks go to Dr Faisal Alhwikem, Dr Colin Paterson and Dr Simos Gerasimou for their cooperation and assistance during this stage. I also extend my heartfelt gratitude to my dear colleagues Saud Yonbawi, Abdullah Albalawi, Khaled Aldheeif, Emad Alharbi, Misael Alpizar and Ioannis Stefanakos for the great times I spent with them and the unforgettable memories.

Declaration

Except where stated, I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, university. All sources are acknowledged as references.

Parts of the contributions in this thesis have been published in:

- Naif Alasmari, Radu Calinescu, Colin Paterson and Raffaela Mirandola. **Quantitative verification with adaptive uncertainty reduction**. In Journal of Systems and Software, vol. 188, 2022.

- Naif Alasmari and Radu Calinescu. **Efficient formal verification with confidence intervals**. In 13th International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, pages 1–6, 2022.

- Naif Alasmari and Radu Calinescu. **Synthesis of Pareto-optimal policies for continuous-time Markov decision processes**. In 48th EUROMICRO Conference on Software Engineering and Advanced Applications, 2022. To appear.

- Radu Calinescu, Naif Alasmari and Mario Gleirscher. **Maintaining driver attentiveness in shared-control autonomous driving**. In 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pages 90–96, 2021.

- Simos Gerasimou, Javier Cámara, Radu Calinescu, Naif Alasmari, Faisal Alhwikem and Xinwei Fang. **Evolutionary-guided synthesis of verified Pareto-optimal MDP policies**. In 36th IEEE/ACM International Conference on Automated Software Engineering, pages 842–853, 2021.

The evaluation of the thesis contribution from Chapter 5 uses an extension of the EvoChecker probabilistic model synthesis tool that I co-developed with Dr Faisal Alhwikem.

The evaluation of the controller synthesis approach described in Chapter 6 uses a case study adopted from the Assuring Autonomy International Programme project SafeSCAD.

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Probabilistic and parametric model checking

Computer systems and software are increasingly at the core of all human activity. The complex hardware and software components of these systems must be designed with an adequately high level of confidence in their integrity and correctness. Ensuring the correctness and integrity of these systems is a particularly essential task for safety-critical and business-critical systems [1, 2]. System failures can expose human lives to danger or result in high financial losses [3–5]. Furthermore, manual analysis and inspection of those systems are costly, error-prone and take a substantial period of time [6]. One widely used approach to avoiding the limitations of such manual analysis is model checking.

Model checking is a mathematically based technique that assists in assuring the correctness of systems and has the ability to analyse the quality of service properties of such systems [7]. It refers to an automated verification method that systematically verifies that concurrent systems satisfy provided properties [8, 9]. In this method, systems are represented as a state transition model (e.g. Kripke structures or discrete-time Markov chains), and properties are written in propositional temporal logic (e.g. linear temporal logic or probabilistic computation tree logic) [10]. Compared to other approaches, model checking is fully automatic and attempts to expose all possible system situations [11]. It also provides a counterexample to illustrate a behaviour that violates the evaluated property. The counterexample gives engineers a clear perception of the error and often suggests a way to fix the problem [12].

Analysing nonfunctional properties (e.g. reliability and performance) for systems has become crucial. However, traditional model checking approaches and tools are not appropriate for real-world systems [13] because the traditional approaches produce either "yes" or "no" to show that the analysed property is satisfied or violated, respectively, and do not include a quantitative evaluation [14]. Consideration of the random aspects of such systems is essential for fulfilling a quantitative analysis. Probabilistic model checking is a formal verification technique capable of evaluating nonfunctional properties of systems whose behaviour is stochastic [15]. For instance, probabilistic model checking has the ability to verify nonfunctional properties of a robot system, such as “the time within which the robot completes a task should be at most three minutes”. Markov models are used in probabilistic model checking to capture the stochastic behaviour of systems. The most frequently utilised kinds of Markov models are discrete-time Markov chain (DTMC), continuous-time Markov chain (CTMC) and Markov decision process (MDP) [16]. Sometimes the state transition probabilities of these models are unknown during system development, and therefore they are represented as parameters. This sort of model is called a parametric Markov model, in which the analysis is through parametric model checking [17, 18].

In parametric model checking, parameters are used to specify some of the probability values and rewards of the model, as these values may only be known when the modelled system is deployed and its behaviour and environment are monitored. Thus, Kwiatkowska et al. [19] describe parametric model checking as a variant of model checking in which “*One or more values in definition of the model (for example, a transition probability) or in the property to be verified (for example, a time bound) are* *provided as a parameter to the verification problem, rather than being instantiated to a specific value*." These parameters may be unknown before run-time or may be configurable parameters of the system. The result of parametric model checking is an algebraic expression for the analysed nonfunctional property.

1.2 Motivation

Despite its widespread adoption in the verification of software systems ranging from service-based systems [20, 21] and software product lines [22] to robotic system controllers [23, 24], probabilistic model checking (and its parametric model checking variants) have a number of limitations. Several of these limitations are particularly relevant to the use of probabilistic and parametric model checking for the verification of synthesis of software systems, which are the focus of this thesis. These limitations are detailed in the following.

First, the validity of verifying a Markov model depends on how accurately the parameters of the model are estimated, and invalid verification results could cause wrong software engineering decisions. Second, the recently proposed approach for probabilistic model checking with confidence intervals [25] (which aims to address the first limitation by supporting the verification of nonfunctional requirements of a software system under uncertainty) is computationally very expensive, and thus does not scale to larger Markov models.

Third, despite significant recent advances in the synthesis of MDP policies [26] that correspond to, for example, software components that satisfy a set of nonfunctional requirements, these synthesis techniques can only handle simple combinations of such requirements. For instance, the latest version of probabilistic model checkers, such as PRISM [27] and Storm [28], can synthesise Pareto-optimal policies for certain combinations of two nonfunctional requirements, but not for three requirements or more.

Finally, current probabilistic model checkers cannot handle the synthesis of policies for continuous-time Markov decision processes (CTMDPs), which can be used for the synthesis of software components with nonfunctional requirements that refer to timing aspects of the system behaviour. An example of such requirements is ‘Minimise the risk associated with a robotic mission over a one-hour time period, subject to the energy consumption during that time not exceeding the capacity of the robot’s battery’.

The thesis addresses the limitations of probabilistic and parametric model checking detailed above, under the following multi-part hypothesis:

1. We can lower the cost of the component testing required to apply formal verification with confidence intervals to nonfunctional requirements of component-based systems modelled as parametric Markov chains by focusing the testing on components whose parameters have a bigger impact on the relevant system properties (as opposed to testing all components similarly).

2. By integrating the efficient parametric model checking method from [17] into the FACT [25] model checker with confidence intervals, it is possible to support the verification of parametric Markov chain models that the current version of FACT does not support.

3. The synthesis of MDP policies that satisfy combinations of constraints and optimisation objectives currently unsupported by the leading model checkers PRISM and Storm can be handled by mapping the MDP models being analysed to parametric DTMCs and using metaheuristics to search for the required policies.

4. By encoding a CTMDP as a parametric CTMC, it is possible to obtain Pareto-optimal policies corresponding to complex combinations of nonfunctional requirements, either manually by using the probabilistic model checker PRISM to analyse the resulting CTMC, or fully automatically by using the probabilistic model synthesis tool EvoChecker [29].

1.3 Research contributions

The thesis makes four research contributions, summarised as follows.

The first contribution is a tool-supported iterative approach for the efficient and accurate verification of nonfunctional requirements under epistemic parameter uncertainty. The approach is called VERACITY, and integrates confidence-interval quantitative verification with a new *adaptive uncertainty reduction* heuristic that collects additional data about the parameters of the verified model by unit-testing specific system components over a series of verification iterations. VERACITY supports the quantitative verification of Markov chains, deciding the components tested in each iteration based on factors that include the sensitivity of the model to variations in the parameters of different components, and the overheads (e.g. time or cost) of unit-testing each of these components. We show the effectiveness and efficiency of VERACITY by using it for the verification of the nonfunctional requirements of a service-based system and a web application.

The second contribution is a method for efficient formal verification with confidence intervals (eFACT) that extends the applicability of probabilistic model checking under parametric uncertainty to larger models than currently possible by leveraging recent advances from the area of parametric model checking. Furthermore, eFACT can analyse nonfunctional requirements to discover the highest confidence level α_{MAX} at which the requirement can be verified as satisfied or violated. It benefits engineers who are particularly interested in measuring their software's confidence over the analysed requirement. Getting the value of α_{MAX} (or a close approximation to it) enables essential decisions to be made. For example, if a requirement can be demonstrated with the highest confidence level of α_{MAX}, then the system can be deployed with confidence (on the basis that that requirement is met).

The third contribution is a new method for synthesising Pareto-optimal MDP policies that satisfy complex requirement combinations (e.g. the requirement that has more than optimisation objectives with constraints). These policies correspond to optimal sys-

<table>
<tr><td colspan="2"><b>Part I: Introduction and Background</b>
<b>Chapter 1:</b> Introduction
<b>Chapter 2:</b> Background</td></tr>
<tr><td><b>Part II: Verification Techniques</b>
<b>Chapter 3:</b> Quantitative verification with adaptive uncertainty reduction
<b>Chapter 4:</b> Efficient formal verification with confidence intervals</td><td><b>Part III: Synthesis Techniques</b>
<b>Chapter 5:</b> MDP policy synthesis for complex requirement combinations
<b>Chapter 6:</b> Component synthesis for continuous-time stochastic systems</td></tr>
<tr><td colspan="2"><b>Part IV: Conclusion and Future Research Directions</b>
<b>Chapter 7:</b> Conclusion
<b>Chapter 8:</b> Future research directions</td></tr>
</table>

Figure 1.1: Thesis structure

tem designs or configurations, and are obtained by translating the MDPs into parametric Markov chains, and using search-based software engineering techniques to synthesise Pareto-optimal parameter values that define optimal policies for the original MDP.

The final contribution is an approach for the synthesis of CTMDP policies—an important problem not handled by current probabilistic model checkers. The policies synthesised by this approach correspond to configurations of software systems or software controllers of cyber-physical systems (CPS) that satisfy predefined nonfunctional constraints and are Pareto-optimal with respect to a set of optimisation objectives. We illustrate the effectiveness of our method by using it to synthesise optimal configurations for a client-server system, and optimal controllers for a driver-attention management CPS.

1.4 Structure of the thesis

The thesis is divided into four parts, as depicted in Figure 1.1. The first part introduces the research area, motivation and contributions of the thesis in Chapter 1, and the concepts, techniques and knowledge required to understand the thesis contributions in Chapter 2.

The second part, comprising Chapters 3 and 4, presents the verification techniques developed for the first two research contributions mentioned in the previous section. Chapter 3 presents quantitative verification with an adaptive uncertainty reduction technique. The chapter describes the problem statement, the contribution and discusses the results of the evaluation. Chapter 4 introduces efficient formal verification with confidence intervals that enable the analysis of larger models than those handled by current solutions.

The third part presents the synthesis techniques developed for the last two contributions in Chapters 5 and 6. Chapter 5 describes the first synthesis contribution and provides details about the problem being addressed, the new MDP policy synthesis method, and then evaluates it using a suite of MDP benchmarks. Chapter 6 presents a CTMDP policy synthesis approach that represents the last contribution of the thesis. It explains the problem and the suggested approach, and discusses the results of the evaluation of the approach.

In the last part of the thesis, Chapter 7 summarises the contributions of the thesis, and Chapter 8 discusses future research directions.

Chapter 2

Background

Analysing the properties of hardware and software systems is a necessary step towards increasing the correctness of the systems. A widely used method for ensuring system correctness is formal verification. To accomplish the analysis process, formal verification requires a suitable mathematical model for the system being analysed and a language appropriate for describing the properties.

This chapter introduces the fundamental concepts, tools, and approaches that supported the work undertaken to produce this thesis. Section 2.1 describes Markov models, with a focus on discrete-time Markov chains (DTMCs), which are used in Chapters 3 and 4. It also describes Markov decision processes (MDPs), a stochastic modelling paradigm that can be used to represent both probability and nondeterminism, that is used in Chapter 5. Section 2.2 explains the probabilistic computation tree logic that is used to express the nonfunctional properties of software systems whose behaviour is modelled using DTMCs or MDPs. Sections 2.3 and 2.4 provide brief descriptions of probabilistic model checking and parametric model checking. In Section 2.5, we present formal verification with confidence intervals, which analyses parametric DTMCs by computing confidence intervals for properties of interest. Finally, in Section 2.6, we concisely describe EvoChecker [29, 30], a search-based software engineering technique for probabilistic model synthesis used in several of the approaches introduced in the thesis.

2.1 Markov models

A Markov model is a state-transition system employed to model a system whose behaviour can be described as probabilistic (stochastic). Markov models can encode the behaviour of a system into a determined number of states embodying various configurations of this system, and a set of transitions specifying the probabilities of moving between these states. According to [31], stochastic processes can be viewed as extending the concept of the random variable to include time:

$$\{X(t), t \in T\}. \tag{2.1}$$

A Markov process is a special category of stochastic process which meets the Markovian property defined in (2.1). It assumes that the future status of a system can be specified by the current status regardless of previous ones. In other words, the transition into the following event depends only on the system's existing event, not on the whole of the previous events (the so-called *memorylessness property* of Markov chains) [32, 33].

The following formal definition of the Markovian property is adopted from [34].

Definition 2.1 A stochastic process $\{X(t)|t = 0,1,2,3,\dots\}$ fulfils the Markov property if:

$$P\{X_{t+1} = s_{t+1}|X_t = s_t, X_{t-1} = s_{t-1}, \dots, X_1 = s_1, X_0 = s_0\} = P\{X_{t+1} = s_{t+1}|X_t = s_t\}$$

where s_0, s_1,..., s_k are consecutive states of the random process.

2.1.1 Discrete-time Markov chains

A discrete-time Markov chain (DTMC) is a type of probabilistic model used to model discrete probabilistic systems [35]. It comprises a finite number of states describing a system's different phases and several transition possibilities determining the likelihood that a phase moves from one single state to another.

A DTMC can model systems that are observed at discrete time periods because transitions between states occur at discrete intervals. Furthermore, a DTMC can accept a probabilistic choice and calculate the likelihood of building a transition between states of a system [14]. The following formal definition was adapted from [14, 36]:

Definition 2.2 DTMC is a quadruple $D = (S, s_0, P, L)$ where:

- S is a finite set of states;
- $s_0 \in S$ is the initial state;
- $P : S \times S \rightarrow [0,1]$ is the matrix of state transition, where $\sum_{s' \in S} P(s,s') = 1$ for all $s \in S$; and
- $L : S \rightarrow 2^{AP}$ represents a labelling function that allocates a number of AP to each state $s \in S$. AP represents a finite set of atomic propositions.

The matrix element $P(s,s')$ provides the likelihood of a transition from state s to state s'. A path (ω) to the DTMC is a sequence of states ω = s_0,s_1,... with $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. The path (ω) could be either finite or infinite. The first state of a path ω is defined by the expression: ω(1). In general, we can define the nth state of path ω by ω(n). If the path is finite, the last state for this path can be expressed as last(ω_{fin}).

Example 2.1 The tele-assistance system (TAS) [21, 37] is a service-based system application that aims to offer health care to chronically ill patients and elderly people in the comfort of their homes. This system utilises a set of sensors mounted on a wearable

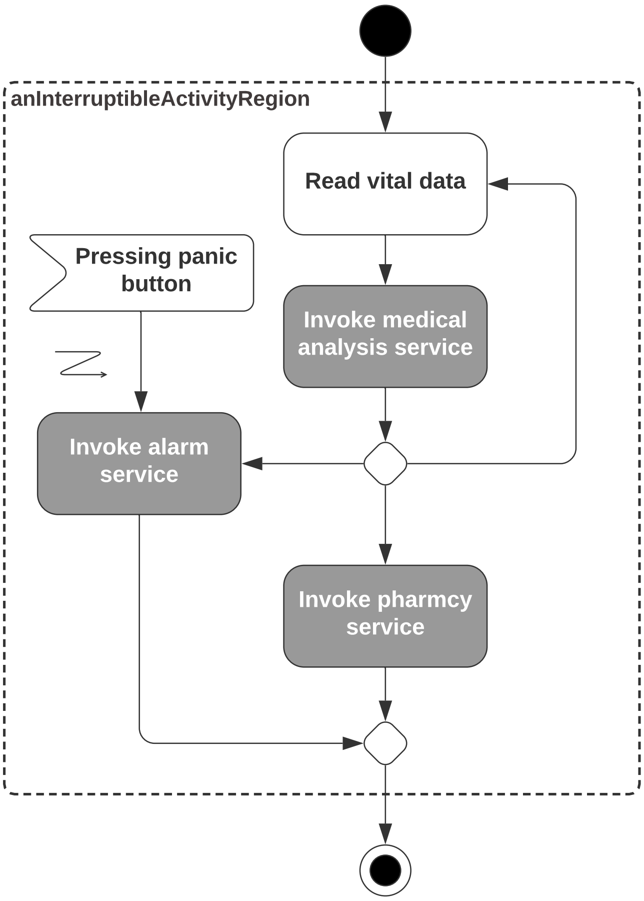


Figure 2.1: Activity diagram for tele-assistance system

device (e.g. a smartwatch) and can provide remote assistance offered by emergency, health-care, and pharmacy services providers. Figure 2.1 shows an activity schematic for the TAS that illustrates the workflow of the system. The TAS regularly takes the patient's vital signs and forwards them to a hired third-party medical service to analyse the data. This system includes three main services:

1. Analysis service: This service analyses the patient's vital and medical data, and on the basis of the outcome of the analysis process, it may trigger an invocation to other services (pharmacy or emergency services) to start their tasks.

2. Pharmacy service: If the analysis shows that the patient must change a dose of

medicine or use a new medicine, the pharmacy service is invoked.

3. Alarm service: The patient can immediately invoke an alarm service by pressing a panic key on the wearable device. The alarm service can also be called by the analysis service if the outcome of the data analysis deems that appropriate.

The TAS will be used as the first case study of our first contribution in Chapter 3. The TAS can involve several scenarios. For instance, the response time of one of the services could vary, a service could fail to respond, or a new feature could be added to the system. By considering various types of actions for the TAS, it can be modelled as a DTMC model, as shown in Figure 2.2. There are 11 states, and the initial state S_0 has a probabilistic branching to S_2 with a probability of 1.0. The state S_2 has a probabilistic branching to S_5 with a probability of 0.1, and to S_3 with a probability of 0.9, and so on. Also, based on this figure, the DTMC elements for this system can be summarised as below:

- Set of states $S = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}$,

- Initial state $s_0 = S_0$,

- Set of atomic propositions $AP = \{initial, request, alarm, analysis, result, pharmacy, failedAlarm, failedAnalysis, failedPharmacy, stop, final\}$,

- Labelling function L: $L(S_0) = \{initial\}, L(S_1) = \{final\}, L(S_2) = request, L(S_3) = \{analysis\}, L(S_4) = \{result\}, L(S_5) = alarm, L(S_6) = \{pharmacy\}, L(S_7) = \{failedAlarm\}, L(S_8) = failedPharmacy, L(S_9) = \{failedAnalysis\}, L(S_{10}) = \{stop\}$.

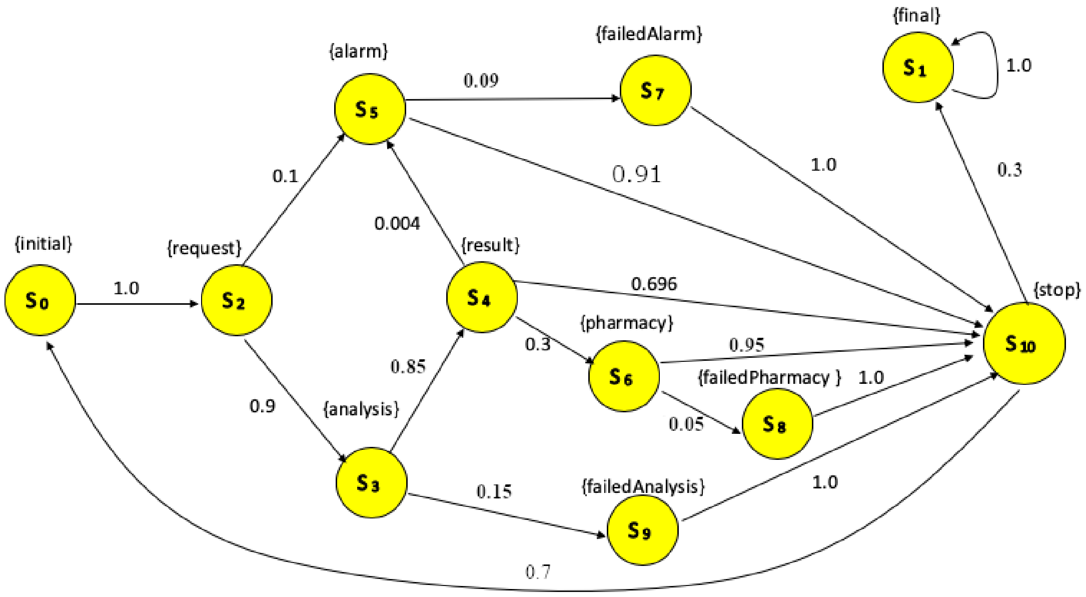


Figure 2.2: DTMC model of the tele-assistance systems (adapted from [38])

The state transition matrix P is:

$$P = \begin{vmatrix}
0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.9 & 0 & 0.1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.85 & 0 & 0 & 0 & 0 & 0.15 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.004 & 0.3 & 0 & 0 & 0 & 0.696 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.09 & 0 & 0 & 0.91 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.05 & 0 & 0.95 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\
0.7 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{vmatrix}$$

2.1.2 Markov decision processes

A Markov decision process (MDP) is a mathematical framework used to solve and model dynamic systems and decision-making problems that exhibit stochastic and non-deterministic behaviour [31, 39]. It consists of five elements: states, transition probabilities, actions, rewards, and decision epochs. It helps a decision-maker (e.g. an agent, or a controller) choose an action depending on the system's existing state (we will assume the decision-maker is a software controller in this thesis). The consequence of this decision is either that the controller gains an immediate reward or incurs an immediate cost. Then the system proceeds to the next state based on the transition probability related to the selected action. At this time point, the controller faces the same situation, but it could be in another state with other actions to select from. The rewards are utilised to guide the synthesis of the controller to the objective, as the values of gained rewards show the objective's significance.

In an MDP, a controller or policy (π) selects the action when moving to the future state. It uses a set of rules to decide which action should be taken at a particular time. MDP policies can be of two types: deterministic and stochastic. A deterministic policy uses the same single action each time when it reaches a specific state [40], i.e. $\pi : S \rightarrow A$. In contrast, a stochastic policy selects one of the available actions a available in state s by using an associated discrete probability distribution; thus, $\pi(s,a)$ is a probability such that $\Sigma_{a \in A(s)} \pi(s,a) = 1 \forall s \in S$, where $A(s)$ is the set of actions available in state s [41, 42]. The optimal policy can be defined as a policy selecting the action that maximises the profits/gains or minimises the cost/risk.

MDPs retain the Markovian property defined in Definition 2.1. The following formal definition of MDPs is adopted from [43, 44]:

Definition 2.3 MDPs can be defined as a tuple $M = (S, s_0, A, \Delta, L)$, where:

- S: is a finite set of states in which every single state $s \in S$;

- $s_0 \in S$: is the initial state;

- A: is a countable set of actions, and each action $a \in A$;

- $\Delta : S \times A \to f(S)$ is a state transition function such that if $\forall s \in S$ and $a \in A$, we have $\sum_{s' \in S} \Delta(s,a)(s') \in \{0,1\}$, where a zero sum indicates that the action a is not available in state s (thus, $f(S)$ is the set of functions containing all discrete probability distributions over S and the "zero" function $\forall s \in S\,.\,zero(s) = 0$); and

- $L : S \to 2^{AP}$ represents a labelling function that allocates a number of atomic propositions from a set AP to each state $s \in S$.

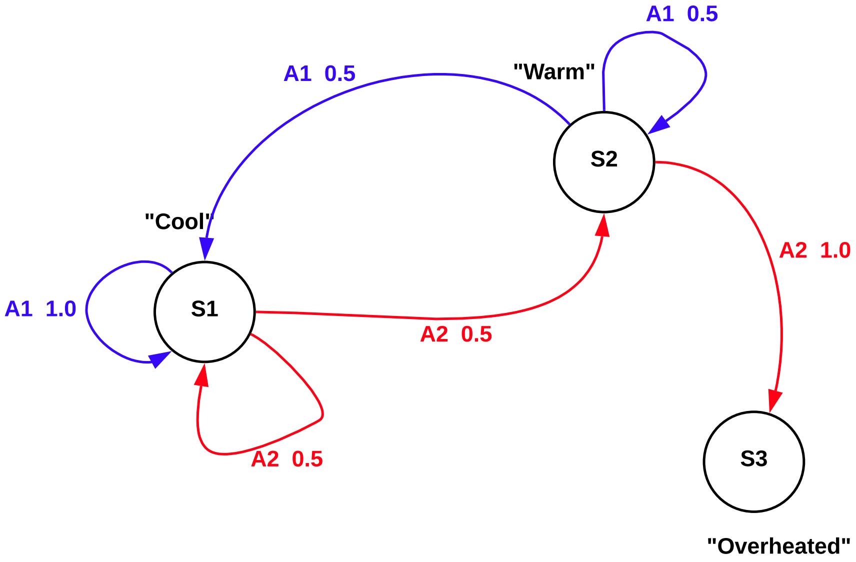


Figure 2.3: Markov decision process example

Example 2.2 Assume that we have a mobile robot that has three states: *cool* ($S1$), *warm* ($S2$), and *overheated* ($S3$). It also has two actions: (moving) *fast* ($A1$) and (moving) *slow* ($A2$). Figure 2.3 shows this example, adopted from [45]. In this example, $S1$ has three outgoing transitions. One transition, designated as action $A1$, enables $S1$ to move to itself, with a probability distribution of one. The other two transitions are designated as action $A2$, which allows $S1$ to either move to itself with a probability of 0.5, or move to $S2$ with a probability of 0.5.

2.2 Probabilistic computation tree logic

Probabilistic Computation Tree Logic (PCTL) [46] is a type of temporal logic that is extended from the computation temporal logic (CTL) [47]. We can use PCTL for reasoning about time and timing events [48] associated with DTMCs. PCTL replaces the path quantifiers used in CTL with the probabilistic operator $P \bowtie \rho(*)$, where $\bowtie \in \{<, \leq, \geq, >\}$, and $\rho \in [\,0,\ 1]$ is a likelihood value that indicates the property requirements, such as "the chance of system failure occurring is at least 0.003". In other words, temporal logic is used with cost, probabilities and rewards to analyse and describe the nonfunctional (or quality of service, QoS) properties of the modelled system. For instance, consider the TAS model explained in Sections 2.1.1 and 2.2. The PCTL formula $P_{\leq 0.05}$ [F "$failedAlarm$"] means: from the initial state of the TAS model, the failure of Alarm service state is reached with a probability that equals at most 0.05. Table 2.1 has additional examples of PCTL formulae for the TAS model.

Definition 2.4 A PCTL formula over an atomic proposition set AP has the general form [35, 49]:

$\Phi ::= \text{true} \mid \text{a} \in \text{AP} \mid \Phi \wedge \Phi \mid \neg\, \Phi \mid \text{P} \bowtie_{\rho} [\alpha]$

$\alpha ::= \text{X}\ \Phi \mid \Phi \cup^{\leq k} \Phi \mid \Phi \cup \Phi$ where:

- Φ: indicates a state-formula;

- AP: is a set of atomic propositions;
  $\bowtie$: is a logical operator and $\in \{<, \leq, \geq, >\}$;

- ρ: is a probability threshold or bound and $\in$ [0, 1];

- α: denotes a path-formula; and

- k: Natural numbers, $k \in \mathbb{N}$.

Definition 2.5 A cost/ reward augmented PCTL state-formula has the general form ([34]):
$\Phi ::= R \bowtie_r [I^{=k}] | R \bowtie_r [C^{\leq k}] | R \bowtie_r [F\Phi]$
where:

- $R \bowtie_r [I^{=k}]$: is the expected state reward at time step k that meets the constraint specified by $\bowtie_r$;
- $R \bowtie_r [C^{\leq k}]$: gives the amount of expected accumulated reward up to time step k that satisfies the constraint specified by $\bowtie_r$; and
- $R \bowtie_r [F\Phi]$: is the expected cumulative reward before reaching the state ϕ that satisfies constraint $\bowtie_r$.

Definition 2.6 The semantics of PCTL over a DTMC model $D = \{S, s', P, L\}$ is defined as follow (adapted from [34]): For any state $s \in S$ (or a path ω) satisfies a formula Φ, the satisfaction relation $\models$ can be defined inductively by:
$s \models true \; \forall \; s \in S$
$s \models a \Leftrightarrow a \in L(s)$
$s \models \neg\Phi \Leftrightarrow s \not\models \Phi$
$s \models \Phi_1 \wedge \Phi_2 \Leftrightarrow s \models \Phi_1$ and $s \models \Phi_2$
$s \models P \bowtie_\rho [\phi] \Leftrightarrow Prob^M(S, \phi) \bowtie_\rho$
$\omega \models X\Phi \Leftrightarrow \omega[1] \models \Phi$
$\omega \models \Phi_1 \cup \Phi_2 \Leftrightarrow \exists i \geq 0(\omega[i] \models \Phi_2$ and $\forall j \leq i.\omega[j] \models \Phi_1$)

The following path-formula ω could be used with a probabilistic path operator:

- XΦ represents the "next" formula that is true when Φ can be met in the next state;
- $\Phi_1 \cup^{\leq k} \Phi_2$ is the time threshold, "bounded until" formula that is satisfied continuously at the time step $t < k$, and Φ_2 is met at the time step $t+1$; and

- $\Phi_1 \cup \Phi_2$ represents "unbounded until" formula. It corresponds to "bounded until", however the time step threshold is equal to infinity $t = \infty$.

Example 2.3 Consider the TAS model shown in Section 2.1.1 and Figure 2.2. The following table illustrates a set of properties defined in PCTL format with their descriptions.

Table 2.1: Some PCTL formulae for the TAS model

PCTL Formula	Description
$P_{\geq 0.03}[\mathsf{F}$ "$failedAlarm$"$]$	The probability that Alarm service will fail is at least 0.03
$P_{\leq 0.15}[\neg$ "$done$" U "$failedAnalysis$"$]$	The probability that the system arrives at "$failedAnalysis$" state before reaching "$done$" state is at most 0.15
$P_{\leq 0.80}[$ true $\mathsf{U}^{\leq 65}$ "$analysis$"$]$	The probability of reaching "$analysis$" state within 65 time units is at most 0.80
$P = ?[!$"$done$" U "$failedService$"$]$	The probability that the system is not done before reaching "$failedService$" state
$P = ?\,[\mathsf{F}$ "$analysis$"$]$	The probability of the system, from the initial state, of reaching "$analysis$" state

2.3 Probabilistic model checking

2.3.1 Overview

In 1981, the concept of model checking was proposed by Clarke and Emerson [50]. In 1982, Queille and Sifakis [51] proposed a similar concept to verify finite-state systems. Model checking is a formal method used to check all potential system states and verify the QoS, which provides significant advantages over traditional testing and simulation methods [52]. It can detect system defects by applying a rigorous procedure in which all possible actions of the system are fully examined. Model checking has become one of the common practices to measure the quality of software systems [53]. The basic aim of the model checking technique is to model a system as a finite number of states machines and describe the specification in a temporal logic formula [54].

Model checking approaches were being implemented to verify systems that exhibit stochastic behaviour, which led to the rise of the concept of probabilistic model checking (PMC). PMC refers to a set of automated verification approaches used to model uncertainties and the probability behaviour of stochastic systems [55]. It is a set of techniques extended from model checking and intended to verify the correctness of probabilistic systems. The systems can be represented in a state transition structure with a set of probability values attached to the transitions [56].

Traditional model checking requires two inputs: captured system behaviour in a high-level modelling formalism and one or more requirement specifications for the system in temporal logic [57]. However, PMC encodes the transition probabilities between states rather than just the existence of transitions [15]. The modelling formalism used for stochastic systems usually includes DTMC, a continuous-time Markov chain (CTMC), MDP, and probabilistic automata. Specification requirements can be expressed in PCTL or CTL. According to [58], the strength of PMC arises from its ability to 1) construct a systematic and exhaustive probabilistic model, 2) discover all potential scenarios of the system (involving best and worst case scenarios), and 3) produce precise quantitative values instead of approximate ones.

Several tools have been invented to uphold PMC algorithms and compute the algebraic expressions for parametric Markov chain properties. Some of these tools are described in the next section.

2.3.2 Probabilistic model checking tools

The probabilistic model checking tools described in this section are included because of their simplicity of use, wide adoption, and the use of some of them in this PhD project.

2.3.2.1 PRISM

PRISM [59, 60] is a frequently used, powerful model checking tool [61] that has been used to analyse several systems in such contexts as power management systems, security protocols, biological systems, communication, networks, multimedia protocols, and randomised distributed algorithms. PRISM is a probabilistic symbolic model checker developed by Kwiatkowska et al. [59] at the University of Birmingham. It can construct and analyse three kinds of models that exhibit probabilistic behaviour: MDPs, DTMCs, and CTMCs. It accepts various property specification languages, including PCTL, PCTL*, and CSL. PRISM is open-source software that can be installed on Macintosh, Linux, and Windows platforms and can be run through either a graphical user interface or a command line [62].

2.3.2.2 Storm

Storm [28] is a PMC tool developed by Dehnert et al. [28] that can analyse four types of models: DTMCs, CTMCs, MDPs, and Markov automata. It depends on symbolic and numerical calculations and aids various types of solvers. Storm supports multiple symbolic and explicit input formats, including probabilistic programs, generalised stochastic Petri nets [63], dynamic fault trees [64], and PRISM [59] and JANI modelling languages [65]. Storm's flexible access provides APIs for C++ and Python, and it can also be run from a command line [66]. It accepts PCTL and CSL specification languages in addition to their extensions with rewards. Storm can be installed on Macintosh and Linux OS platforms or can be used through a Docker container on all platforms.

2.3.2.3 MRMC

The Markov reward model checker (MRMC) [67] is one of the PMC software tools that supports the analysis of discrete-time and continuous-time Markov rewards. MRMC was developed by the RWTH Aachen University's software modelling and verification group and the University of Twente's formal methods and tools group. This tool is implemented in C and can be run from a command line. MRMC supports four kinds of models: DTMCs, CTMCs, discrete Markov reward models, and continuous Markov reward models. It can be easily used as a back-end element for several tools such as the PEPA Workbench [68], the performance modelling tool GreatSPN v2.0 [69], and the STATEMATE toolchain [70].

2.3.2.4 VESTA

VESTA [71] is a probabilistic model checker that utilises statistical algorithms to analyse Markov models. It can analyse two kinds of Markov models, CTMC and DTMC, using properties written in PCTL, CSL or quantitative temporal expressions. VESTA can be run through a command-line or operated using the designed user-interface. The language used to develop this tool is Java.

2.4 Parametric model checking

Parametric model checking [17, 18] can be defined as a formal method to analyse Markov chains with probabilities of transitions identified as rational functions over a collection of continuous variables. These variables are the Markov model customisable parameters that may be unknown before run-time, or may be configurable parameters of the system. Parametric model checking can generate an algebraic expression for a given PCTL-formalised property. It is supported by PMC tools such as PRISM and Storm, and by verification tools like PARAM [72].

Daws [35] has developed a language-theoretic method to produce regular expressions that define the probability of attaining a given set of states by using a state elimination algorithm. The algorithm transforms the parametric model into a finite automaton. However, the regular expression of the finite automaton is restricted to $n^{\Theta(log\, n)}$ where n is the number of states [73]. A study by [18] reduced this limitation by applying a set

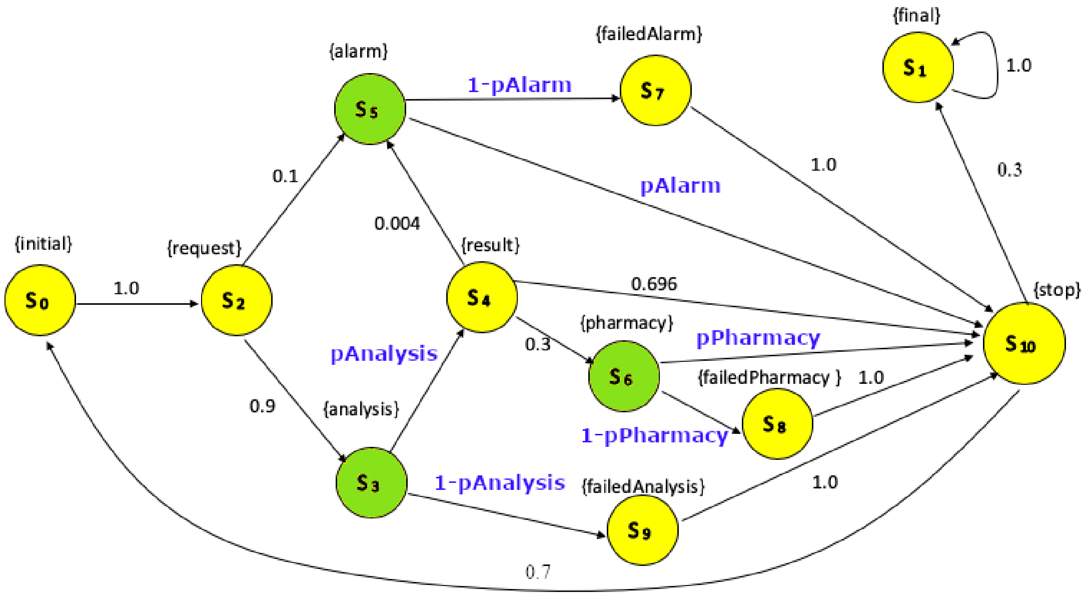


Figure 2.4: TAS parametric model

of techniques to reduce the state space.

Several studies have been done on enhancing the effectiveness of parametric model checking. A study done by [74] proposed an approach that applies a recursively hierarchical decomposition of the analysed parametric model checker to reduce its computation. This approach accelerates parametric model checking by different orders-of-magnitude to previous approaches.

Example 2.4 Considering the TAS system example shown in Figure 2.2, if some states have undetermined probabilities for their transitions to other states, we can redraw the model as a parametric model, as shown in Figure 2.4.

Figure 2.4 shows a parametric model for the TAS in which the probabilities of three states (green circles) are given as parameters, and they remain unknown until run-time. The parameters *pAlarm*, *pAnalysis*, and *pPharmacy* indicate to the customisable parameters that they are unidentified before run-time. For instance, if the desired property to analyse is “the probability of the system returning a failed service”, which is expressed in PCTL as: $P=?\,[F\text{“}failedService\text{”}]$, the PMC (such as PRISM) can produce the algebraic expression for this property as

(450 pAnalysis * pAlarm + 33750 pAnalysis * pPharmacy + 12500 pAlarm + 78300 pAnalysis - 125000 | 441 pAnalysis * pAlarm + 33075 pAnalysis * pPharmacy + 12250 pAlarm + 76734 pAnalysis - 125000)

2.5 Formal verification with confidence intervals

Formal verification with confidence intervals [75] is a mathematically based technique for the computation of confidence intervals for nonfunctional properties of systems with stochastic behaviour. Given a parametric Markov chain $M=(S,s_0,\mathbf{P},L)$ that models the behaviour of an SUV, a PCTL formula $P_{=?}[\cdot]$ or $R_{=?}[\cdot]$ corresponding to a nonfunctional property of the SUV, and a confidence level $\alpha\in(0,1)$, the technique computes an α confidence interval for the property.

To perform this computation, the technique also requires observations of the outgoing transitions from all states with outgoing transition probabilities specified as rational functions over unknown parameters of the SUV components. Assuming that $Z\subseteq S$ is the subset of all these states, the required observations are provided by a function

$$O: Z\times S\to\mathbb{N} \tag{2.2}$$

that maps each pair of states $(z,s)\in Z\times S$ to the number $O(z,s)$ of transitions from state z to s within a fixed period of time during which all outgoing transitions from z were observed and counted. Given such observations, the confidence interval computation is done in three stages. In the first stage, a confidence interval is calculated for each parameter of the Markov chain. In the next stage, parametric model checking is used to obtain a closed-form expression for the nonfunctional property.[1] This expression is a

[1] Parametric model checking is supported by a growing number of model checkers, including PARAM [72], PRISM [27], Storm [28] and ePMC/fPMC [17, 76].

rational function over the SUV parameters, and is a byproduct of the technique exploited by our VERACITY approach. Finally, in the third stage, the parameter confidence intervals and the property expression are used to establish the confidence interval for the nonfunctional property of interest. For a detailed description of the technique and of a model checker that implements it see [75] and [25], respectively.

The number of available observations and the confidence level α used by the technique influence the width of the confidence interval. In particular, few observations and large α values yield wide confidence intervals which may contain the lower or upper bound that a nonfunctional requirement specifies for the property. In this case, the verification of the requirement is not possible, and additional observations need to be collected to allow the computation of a narrower confidence interval that does not contain the bound.

Example 2.5 Consider the pDTMC model of the TAS shown in Figure 2.4. Assume that the user wants to evaluate the probability of arriving at the ”$failedAlarm$” state (S_7), having started in the ”*initial*” state (S_0). First, the user can provide the model for the TAS in FACT-encoded format as follows:

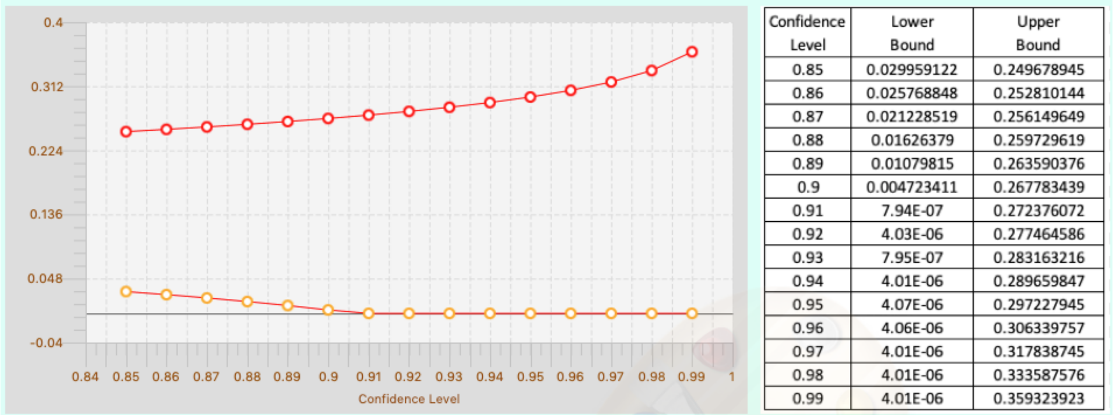
```
dtmc

// Probabilities of successful service invocations
param double pAlarm = 520 17;
param double pPharmacy = 9500 170;
param double pAnalysis = 73820 1350;
// known probabilities of different outcomes
const double p_analyse_alarm = 0.004;
const double p_analyse_Drug = 0.3;
const double p_analyse_skip = 0.696;
const double p_alarmReq = 0.1;
const double p_analyseReq = 0.9;

```

```
14 module TeleAssistance
15  a : [0..10] init 0;
16  [initial]  (a=0) -> 1.0:(a'=2); //request
17  [final]    (a=1) -> true;       //FINAL
18  [request]  (a=2) -> p_alarmReq:(a'=5) + p_analyseReq:(a'=3);
19  [analysis] (a=3) -> pAnalysis1:(a'=4)
20                        + (1-pAnalysis1):(a'=9);
21  [result]   (a=4) -> p_analyse_alarm:(a'=5)
22                        + p_analyse_Drug:(a'=6)
23                        + p_analyse_skip:(a'=10);
24  [alarm]    (a=5) -> pAlarm1:(a'=10) + (1-pAlarm1):(a'=7);
25  [pharmacy] (a=6) -> pPharm1:(a'=10) + (1-pPharm1):(a'=8);
26  [failAl]   (a=7) -> 1.0:(a'=10); // failed send alarm
27  [failPh]   (a=8) -> 1.0:(a'=10); // failed changed drug
28  [failAn]   (a=9) -> 1.0:(a'=10); // failed analysis
29  [done]     (a=10)-> 0.02:(a'=1) + 0.98:(a'=0);
30 endmodule
31
32 rewards "cost"
33  (a=5) : 2.7;  //cost of invoking alarm
34  (a=3) : 0.03; //cost of invoking analysis
35  (a=6) : 0.24; //cost of invoking drug
36 endrewards
37
38 // labels
39 label "done" = a=10;
40 label "failedAlarm" = a=7;
41 label "failedService" = a=7|a=8|a=9;
42 label "final" = a=1;
43 label "analysis" = a=3;
```

Note that in the above-mentioned format, the FACT parameter sets from lines 4 to 6, *pAlarm*, *pPharmacy*, and *pAnalysis*, each have two transitions. Then the user writes



Confidence Level	Lower Bound	Upper Bound
0.85	0.029959122	0.249678945
0.86	0.025768848	0.252810144
0.87	0.021228519	0.256149649
0.88	0.01626379	0.259729619
0.89	0.01079815	0.263590376
0.9	0.004723411	0.267783439
0.91	7.94E-07	0.272376072
0.92	4.03E-06	0.277464586
0.93	7.95E-07	0.283163216
0.94	4.01E-06	0.289659847
0.95	4.07E-06	0.297227945
0.96	4.06E-06	0.306339757
0.97	4.01E-06	0.317838745
0.98	4.01E-06	0.333587576
0.99	4.01E-06	0.359323923

Figure 2.5: Screenshot of the FACT result for the TAS model and a list of confidence levels with their bounds

the property in a PCTL formula and provides the lowest and highest confidence levels. The PCTL formula is written as:

$$P =?[F\text{”}failedAlarm\text{”}]$$

With the above-mentioned inputs, FACT can obtain the following algebraic expression for this property by using PRISM:

$$(129050 * pAlarm1 - 129050/126469 * pAlarm1 - 151469)$$

Also, FACT can use the transition observations to compute the confidence intervals for the expression parameters ($pAlarm1$ in this example). Figure 2.5 shows the result of the FACT for this example. The table on its right is a list of confidence levels with their bounds for the evaluated property, and the left-hand side is a graphical representation of the result.

2.6 EvoChecker probabilistic model synthesis

EvoChecker [29, 30] is a search-based technique and tool that automates the process of verifying probabilistic models for various alternative structures and instances of the system parameters. EvoChecker's primary purpose is to investigate the space of alternatives to obtain a set of Pareto-optimal solutions for the tested system by using PMC and multi-objective optimisation genetic algorithms. EvoChecker takes two inputs:

1. A probabilistic model in the PRISM modelling language extended with the following EvoChecker-specific constructs:

   - '`evolve` $\langle$`int or double`$\rangle$ $\langle$`parameter-name`$\rangle$$[min..max]$', which is used to define either integer or double parameters for the model with their values. For example,

     ```
     evolve int x [1..10];
     ```

     The above declaration defines an integer parameter called x with range of values from 1 to 10. Therefore, during the execution of EvoChecker, x will assigned a value between 1 and 10.

   - '`evolve distribution` $\langle$`dist-name`$\rangle$$[min_1, max_1] \ldots [min_n, max_n]$', which is used to define an m-element discrete probability distribution, and declares the ranges of acceptable values for the elements of this distribution, where

     $$\forall j = 1, 2, ..., m.[min_j, max_j] \subseteq [0, 1].$$

     For example, the declaration below defines a discrete probability distribution with two elements, y_1 and y_2:

     ```
     evolve distribution y [0.3..0.5][0.7..0.9];
     ```

   - '`evolve` $\langle$`module-name`$\rangle$', which is used to define different possible designs for the model being analysed.

```
evolve module test
  m : [0..4];
      ...
  [sleep2idle] m = 3 -> (m’=m);
  [idle2sleep] m = 3 -> (m’=m);
      ...
endmodule
```

2. A set of QoS requirements comprising $n \geq 0$ constraints of the form

   $$prop_i \bowtie_i bound_i$$

   where $\bowtie \in \{<,>,=,\leq,\geq\}$ and $i = 1,2,\ldots n$, and a set of $m \geq 1$ optimisation objectives of the form

   $$M\ prop_j$$

   where $M \in \{$minimise, maximise$\}$, and $j = 1,2,\ldots m$.

Given these inputs, EvoChecker uses multi-objective genetic algorithms such as NSGA-II [77] and a probabilistic model checker such as PRISM or Storm to synthesise Pareto-optimal sets of ‘evolve’ parameter values (and thus sets of probabilistic models corresponding to designs or configurations of the modelled software system).

We do not provide full examples of EvoChecker-encoded sets of alternative probabilistic models and Pareto-optimal models synthesised using EvoChecker in this section, but multiple such examples can be found in Chapters 5 and 6.

Part II

Verification Techniques

Chapter 3

Quantitative verification with adaptive uncertainty reduction

3.1 Introduction

The verification of dependability, performance, cost and other nonfunctional requirements of software systems [78, 79] needs to consider the stochastic nature of software characteristics such as inputs, workloads, timeouts and failures. As such, stochastic modelling paradigms ranging from Markov chains [20, 80] and probabilistic automata [81, 82] to stochastic Petri nets [83, 84] are widely used to perform this verification. However, ensuring that stochastic models are sufficiently accurate for the verification results to be valid is very challenging. While the structure of the models can be extracted from the actual code [85] or from software artefacts such as activity diagrams [38, 80], model parameters such as the probabilities, timing and other quantitative information annotating the model states and state transitions are affected by uncertainty.

These parameters need to be estimated using data obtained, for instance, from testing the system components individually, or (for systems already in use) from system logs. Point estimators such as the mean of the observed parameter values are typically used for this purpose. However, the point estimation of the uncertain model parameters produces imprecise verification results, and risks causing invalid engineering decisions [75, 86], especially when only few observations are available. In mature subjects like medicine [87, 88] and in established engineering disciplines like civil [89] and mechanical [90] engineering, this risk is deemed unacceptable, and is mitigated by computing *confidence intervals* for the model parameters and the verified properties [91, 92]. In contrast, this risk is rarely considered in software performance and dependability engineering. Instead, the research in this area focuses on devising new techniques, tools and applications for the verification of stochastic models, under the strong assumption that using point estimates for the model parameters is sufficiently accurate.

To address the risk of invalid decisions associated with this assumption, we introduce VERACITY,[1] a tool-supported approach for the quantitative verification of Markov chains under epistemic parametric uncertainty[2].

VERACITY builds on the previous research on computing confidence intervals for the reliability, performance and other nonfunctional properties of a system [25, 75]. This computation uses a *parametric Markov chain* (i.e. a Markov chain with unknown state transition probabilities) that models the system behaviour, and observations of the system behaviour available from component unit testing, runtime monitoring or system logs. However, when insufficient observations are available, these confidence intervals are too wide to verify whether nonfunctional requirements that impose constraints on such properties are satisfied. For example, as shown in Figure 3.5 later in this chapter, the width of confidence intervals obtained at confidence level 95% for verifying nonfunctional requirements of TAS is too large when the given number of observations is low. This width becomes narrower as additional observations are acquired. To handle this frequently encountered problem efficiently, VERACITY obtains additional observations by unit-testing specific system components over a series of *adaptive un-*

[1] quantitative VERification with Adaptive unCertaInTY reduction

[2] Uncertainty is termed epistemic when it is due to insufficient data (and therefore reducible by gathering additional data), and aleatory when it is intrinsic to the analysed system (and therefore irreducible)

certainty reduction iterations. The components tested in each iteration are decided using a heuristic that takes into account multiple factors. These factors are detailed later in this chapter, and include the sensitivity of the nonfunctional properties to variations in the parameters of different components, the overheads (e.g. time or cost) of testing each of these components, and the number of observations already available for each component.

VERACITY supports both the verification of new system designs, and the verification of planned updates to existing systems. Using VERACITY to decide whether a new system should be deployed or not involves applying our approach with few or no initial observations of the system parameters. Multiple uncertainty reduction iterations are typically required to acquire sufficient observations of the parameters of the system and to reach a decision in this case. In contrast, when deciding whether updated versions of specific system components should be adopted, VERACITY can exploit a large number of initial observations of the parameters associated with the components not being updated. Accordingly, fewer uncertainty reduction iterations are typically necessary in this case, primarily to acquire observations of the parameters associated with the updated components.

The contributions of the chapter are threefold. First, we introduce a new heuristic for the efficient reduction of epistemic parametric uncertainty of Markov chains used in dependability and performance software engineering. Second, we present a new approach that integrates this heuristic with a recently proposed method for formal verification with confidence intervals [25, 75], and a tool that implements the approach, automating the verification of nonfunctional requirements under parametric uncertainty. Finally, we present extensive experimental results showing: (i) the effectiveness of the VERACITY verification approach during initial software development and software updating; and (ii) the efficiency of the VERACITY uncertainty reduction compared to uncertainty reduction by uniformly testing all the components of the system under verification (SUV).

We organised the remainder of the chapter as follows. Section 3.2 introduces a motivating example that we then use to present our quantitative verification approach in Section 3.3. Sections 3.4 and 3.5 describe the tool support we implemented for approach, and the case studies we carried out to evaluate VERACITY, respectively. Finally, we discuss related work in Section 3.6, and we conclude with a brief summary and we suggest directions for future work in Section 3.7.

3.2 Motivating example

To motivate our VERACITY approach, we use a tele-assistance service-based system (TAS) initially introduced in [37]. TAS aims to support a patient suffering from a chronic condition in the comfort of their home by using: (i) a set of vital-sign monitoring sensors mounted on a medical device worn by the patient; and (ii) remote assistance services offered by emergency, medical and pharmacy service providers. Periodically, the patient's vital signs are measured by the wearable device, and a third-party medical analysis service is invoked to analyse them in conjunction with the patient's medical record. Depending on the results of this analysis, TAS may confirm that the patient is fine, may invoke a pharmacy service to request the delivery of different medication to the patient's home, or may invoke an alarm service. The invocation of the alarm service is also triggered when the patient presses a panic button on the wearable device, and results in a medical team being dispatched to provide emergency assistance to the patient. The activity diagram for the TAS workflow is shown in Figure 3.1, where we assume that the operational profile of the system is known (e.g. from previous deployments) and is given by the probabilities annotating the decision points from the diagram.

We suppose that a team of software engineers wants to verify whether the third-party services they consider for the implementation of the TAS system satisfy—at 95% confidence level—the nonfunctional requirements from the first two columns of Table 3.1. We assume that the three services are yet to be tested, and therefore the success probabilities p_{ma}, p_{ph} and p_{al} for the medical analysis service, pharmacy service and alarm

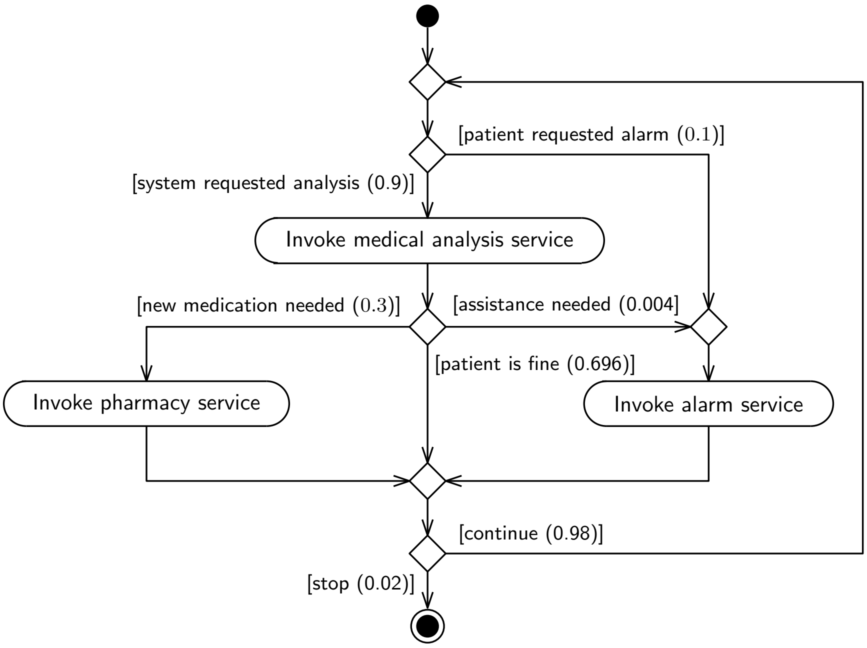

[patient requested alarm (0.1)]
[system requested analysis (0.9)]
Invoke medical analysis service
[new medication needed (0.3)]
[assistance needed (0.004]
[patient is fine (0.696)]
Invoke pharmacy service
Invoke alarm service
[continue (0.98)]
[stop (0.02)]

Figure 3.1: TAS activity diagram, with the execution probability of each branch provided in brackets after the guard expression for the branch.

Table 3.1: Nonfunctional requirements for the TAS system

ID	Requirement	PCTL formula
R1	The probability that an alarm failure ever occurs during the lifetime of the TAS system shall be below 0.26.	$P_{<0.26}[\mathrm{F}\ alarmFail]$
R2	The probability that the handling of a request by the TAS workflow ends with a service failure shall be below 0.04.	$P_{<0.04}[\neg done\ \mathrm{U}\ serviceFail]$
R3	The probability that an invocation of the medical analysis service is followed by an alarm failure shall be below 0.0003.	$P_{<0.0003}[\neg done\ \mathrm{U}\ alarmFail\{analysis\}]$

service, respectively, are unknown. As such, the Markov chain that the engineers can use to verify the TAS requirements is parametric (Figure 3.2), and the three services must be tested to observe how many of their executions succeed and how many fail (e.g. by not finishing timely). With these observations, the engineers can use formal verifi-

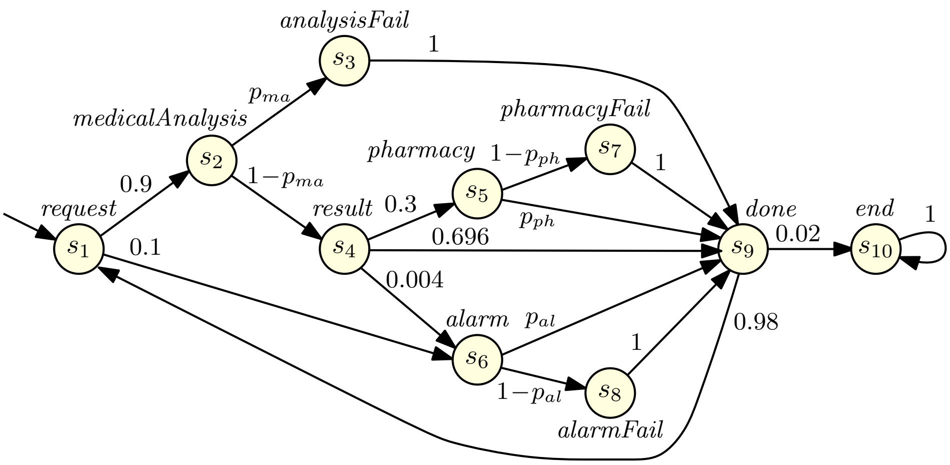


Figure 3.2: Parametric Markov chain modelling the TAS workflow (adapted from [38])

cation with confidence intervals (cf. Section 2.5) to compute 95% confidence intervals for the probabilities from the three TAS requirements, which are formally expressed in PCTL in the last column from Table 3.1. Furthermore, once enough observations are available, these confidence intervals will be sufficiently narrow to ensure that the upper bounds from the requirements in Table 3.1 (i.e. 0.26 for requirement R1, 0.04 for R2, and 0.0003 for R3) fall outside the intervals, allowing the engineers to verify whether the requirements are satisfied or not.

However, under the realistic assumption that service invocations take non-negligible time, the engineers will want to complete this verification with as few invocations (i.e. observations) of each service as possible.[3] Deciding how many observations to obtain for each service in order to complete the verification of the requirements with minimal testing effort is very challenging. Our VERACITY verification approach addresses this challenge as described in the next section.

[3] Minimising this testing effort is particularly important when the verification needs to be performed at runtime, e.g. to find a suitable replacement for a failed component of a system, or when testing a system component has some other cost associated with it (e.g. an invocation charge paid to the provider of a service, or using battery energy on an embedded system).

3.3 The VERACITY verification approach

3.3.1 Problem definition

Our VERACITY verification approach is applicable to systems comprising $m > 1$ components that can be tested independently. We consider a component-based system whose $n \geq 1$ nonfunctional requirements are of the form

$$prop_i \bowtie_i bound_i, \tag{3.1}$$

where, for all $i \in \{1, 2, \ldots, n\}$, $prop_i$ is a nonfunctional system property (e.g. reliability or response time), $\bowtie_i \in \{<, \leq, \geq, >\}$, $bound_i \in \mathbb{R}$ places a constraint on the acceptable values of $prop_i$, and the i-th requirement can be expressed as a PCTL formula $P_{\bowtie_i bound_i}[\,\cdot\,]$ or $R_{\bowtie_i bound_i}[\,\cdot\,]$ over a parametric Markov chain $M = (S, s_0, \mathbf{P}, L)$. Given such a system, the verification problem addressed by VERACITY is to verify whether the n nonfunctional requirements (3.1) are satisfied at confidence level $\alpha \in (0, 1)$:

1. with an overall testing cost that is as low as possible, and not exceeding a predefined testing budget $budget \in \mathbb{N}$;
2. by using a (possibly empty) initial set of observations given by an observation function $O_0 : Z \times S \to \mathbb{N}$ with the semantics from (2.2); and
3. by obtaining additional observations through unit-testing the m system components as required, where each unit test of the j-th component:
    - generates one additional observation of an outgoing transition for every state in a non-empty set $Z_j \subset Z$, such that the state sets $Z_1, Z_2, \ldots, Z_m$ are disjoint and $\bigcup_{j=1}^{m} Z_j = Z$,
    - has an associated cost $cost_j$, that may represent testing time, resources, price, or a combination thereof.

Due to the epistemic uncertainty associated with this verification problem and to the stochastic nature of the component-testing results, a strategy guaranteed to achieve a minimum overall testing cost does not exist. We illustrate this limitation with an example. Consider a system that uses two web services, A and B. This system implements a simple workflow: first, it invokes service A, which is available with probability p_A; next, it invokes service B, which is available with probability p_B; next, it stops. Suppose that we want to establish whether the probability of successful invocation of both services is at least 0.9 (i.e. whether $p_A p_B \geq 0.9$) at confidence level $\alpha = 0.95$. If the two unknown probabilities are $p_A = 0$ (i.e. service A is never available) and $p_B = 0.95$, then allocating all the testing effort to unit-testing service A is the cheapest strategy for establishing that the requirement is violated, as this strategy will quickly show that p_A cannot be large enough for the requirement to be satisfied. Conversely, if $p_A = 0.95$ and $p_B = 0$, unit-testing only service B is the cheapest strategy. However, with no prior knowledge about p_A and p_B, it is impossible to always choose the best testing strategy. Therefore, our objective is to achieve an overall testing cost that is, on average, significantly lower than the cost associated with uniformly testing all components. Furthermore, for practical reasons, we added the constraint that the overall cost does not exceed a predefined testing budget *budget* $\in \mathbb{N}$.

Example 3.1 Consider the TAS system from our motivating example. Its $n = 3$ non-functional requirements R1–R3 from Table 3.1 are of the form in (3.1), are expressed as PCTL formulae over the parametric Markov chain from Figure 3.2, and need to be verified at confidence level $\alpha = 0.95$. The set of Markov chain states with unknown outgoing transition probabilities is $Z = \{s_2, s_5, s_6\}$, and the (empty) initial set of observations is defined by the function $O_0(z,s) = 0$ for any $(z,s) \in Z \times S$. The system comprises $m = 3$ components that can be tested independently: the medical analysis service (component 1), the pharmacy service (component 2) and the alarm service (component 3). Additionally, invoking one of these services once provides an additional observation of an outgoing transition for the state in one of the disjoint sets $Z_1 = \{s_2\}$, $Z_2 = \{s_5\}$ and

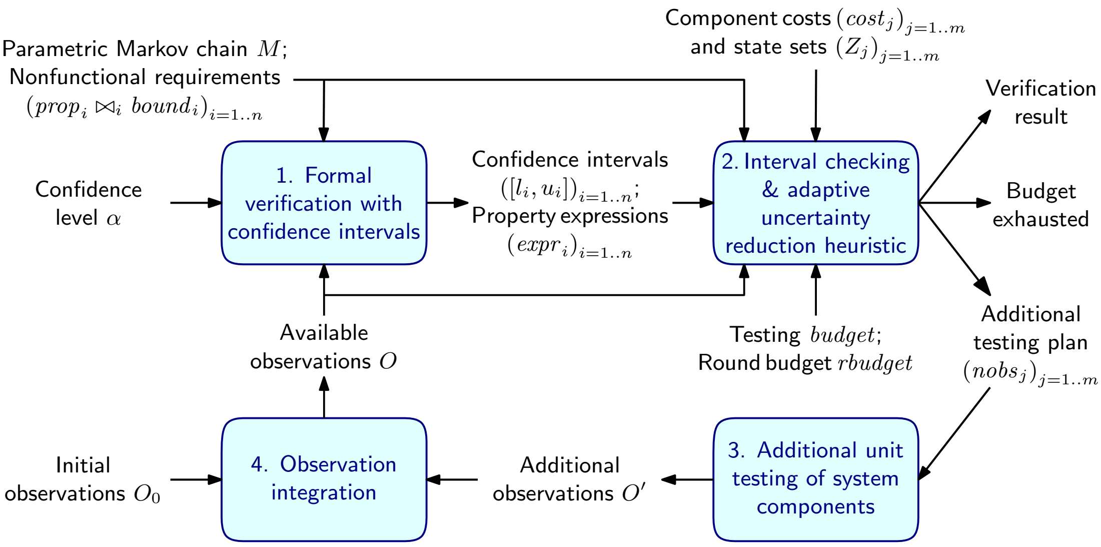

Parametric Markov chain M;
Nonfunctional requirements
$(prop_i \bowtie_i bound_i)_{i=1..n}$
Component costs $(cost_j)_{j=1..m}$
and state sets $(Z_j)_{j=1..m}$
Verification result
Confidence level α
1. Formal verification with confidence intervals
Confidence intervals
$([l_i, u_i])_{i=1..n}$;
Property expressions
$(expr_i)_{i=1..n}$
2. Interval checking & adaptive uncertainty reduction heuristic
Budget exhausted
Available observations O
Testing $budget$;
Round budget $rbudget$
Additional testing plan
$(nobs_j)_{j=1..m}$
Initial observations O_0
4. Observation integration
Additional observations O'
3. Additional unit testing of system components

Figure 3.3: Iterative four-step process of the VERACITY quantitative verification with adaptive uncertainty reduction

$Z_3 = \{s_6\}$, where $Z_1 \cup Z_2 \cup Z_3 = Z$. Finally, to fully cast the task of verifying requirements R1–R3 in the format from our problem definition, we assume that a testing budget $budget = 150000$ is available to complete the verification, and that the per-invocation costs of testing the three services are $cost_1 = 1$, $cost_2 = 1$ and $cost_3 = 2$, e.g. based on the ratios between their mean execution times.

3.3.2 VERACITY verification process

To solve the problem from Section 3.3.1, VERACITY employs the iterative verification process depicted in Figure 3.3. Each *round* (i.e. iteration) of this process comprises the four steps described below.

In the first step, formal verification with confidence intervals [75] is used to compute confidence intervals $[l_1, u_1]$, $[l_2, u_2]$, ..., $[l_n, u_n]$ at confidence level α and (as a byproduct, as explained in Section 2.5) closed-form expressions $expr_1$, $expr_2$, ..., $expr_n$ for the n properties from the nonfunctional requirements (3.1). The observations O used to compute the n confidence intervals include the initial observations O_0 and, starting with the second iteration, all the additional observations O' obtained in the previous iterations of the process. If the observation set O_0 is empty, then the confidence interval $[l_i, u_i]$ computed for the i-th property in the first iteration is $[0,1]$ or $[0,\infty)$, depending on whether the i-th requirement is of the form $P_{\bowtie_i bound_i}[\,\cdot\,]$ or $R_{\bowtie_i bound_i}[\,\cdot\,]$.

In the second step, VERACITY checks whether the n confidence intervals are sufficiently narrow to allow the verification of the nonfunctional requirements, and, if necessary, plans additional testing of the m system components. The function $sat : 1..n \rightarrow \{\mathsf{true}, \mathsf{false}, \mathsf{undecidable}\}$ defined below is used to verify if the i-th requirement is satisfied, violated or insufficient observations are available to reach a decision (at confidence level α):

$$sat(i) = \begin{cases} \mathsf{true}, & \text{if } (\bowtie_i \in \{<,\leq\} \wedge u_i \bowtie_i bound_i) \vee \\ & \qquad\qquad (\bowtie_i \in \{\geq,>\} \wedge l_i \bowtie_i bound_i) \\ \mathsf{false}, & \text{if } (\bowtie_i \in \{<,\leq\} \wedge bound_i \bowtie_i l_i) \vee \\ & \qquad\qquad (\bowtie_i \in \{\geq,>\} \wedge bound_i \bowtie_i u_i) \\ \mathsf{undecidable}, & \text{otherwise (i.e. if } bound_i \in [l_i, u_i]) \end{cases} \tag{3.2}$$

Figure 3.4 summarises the rationale for this definition, for the case $\bowtie_i \in \{<,\leq\}$. Using this function, the following decisions are made:

1. If $\forall i = 1..n : sat(i) = \mathsf{true}$, then all requirements are satisfied, and the verification process terminates with a positive result.

2. Otherwise, if $\exists i = 1..n : sat(i) = \mathsf{false}$, then at least one requirement is violated, and the verification process terminates with a negative result. The verification process is ended as soon as a violated requirement is identified under the assumption that the component modifications needed to resolve the violation will invalidate the verification results, so further testing effort before these modifications are completed would be unjustified. Instead, all available observations of any unmodified components can be exploited to re-verify all the requirements once

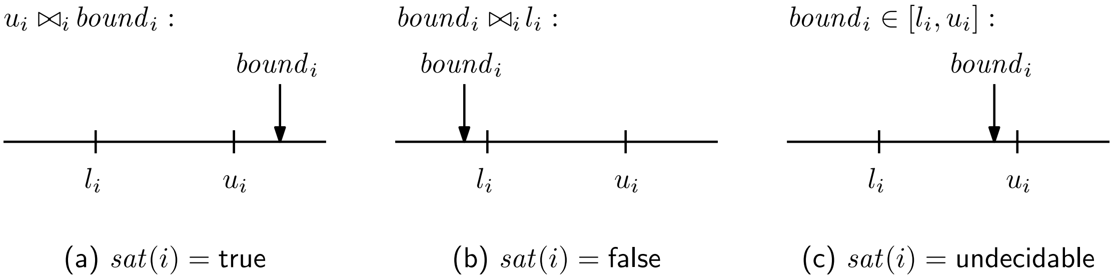

$u_i \bowtie_i bound_i$:
$bound_i$
l_i
u_i
(a) $sat(i)$ = true
$bound_i \bowtie_i l_i$:
$bound_i$
l_i
u_i
(b) $sat(i)$ = false
$bound_i \in [l_i, u_i]$:
$bound_i$
l_i
u_i
(c) $sat(i)$ = undecidable

Figure 3.4: When $\bowtie_i \in \{<, \leq\}$, the i-th requirement (i.e. $prop_i \bowtie_i bound_i$) is: (a) satisfied if $u_i \bowtie_i bound_i$; (b) violated, if $bound_i \bowtie_i l_i$; and (c) undecidable, if $bound_i \in [l_i, u_i]$.

the modifications are in place. However, our VERACITY approach can be easily adjusted to only deem the verification complete when a decision was reached on every requirement, i.e. when $\forall i = 1..n : sat(i) \in \{true, false\}$.

3. Finally, if neither of the previous termination conditions apply, then additional observations are needed to complete the verification. Two cases are possible in this situation. First, in the case when the testing budget *budget* was fully utilised in the previous VERACITY rounds, the process terminates with an inconclusive 'budget exhausted' result. Otherwise, testing budget is still available, and the VERACITY adaptive uncertainty reduction heuristic detailed in Section 3.3.3 is used to calculate the numbers of additional component observations $nobs_1$, $nobs_2$, $\ldots$, $nobs_m$ needed for the next round of the verification process, where

$$\sum_{j=1}^{m} nobs_j cost_j \approx rbudget \tag{3.3}$$

and $rbudget \in [0, budget]$ is a parameter of the VERACITY approach called the *round budget*.[4] The heuristic is adaptive in the sense that these numbers of additional observations vary from round to round, as the heuristic takes into account the actual observations from all the previous rounds (and any initial observations

[4] Equality cannot be always achieved in (3.3) because $nobs_1, nobs_2, \ldots, nobs_m$ must take non-negative integer values.

that may be available). The maximum number of rounds for the verification process is $\lceil budget/rbudget \rceil$. Accordingly, larger round budgets yield fewer rounds, and therefore less opportunity for adaptation but lower overheads (due to the fewer rounds). In contrast, smaller round budgets lead to more rounds, which offer more opportunity for adaptation but also come with higher overheads.

In the third step of the VERACITY process, $nobs_j$ tests of components j are carried out for $j = 1..m$. As we will explain in Section 3.4, these tests can be fully automated, or can be performed by a software engineer when requested by the VERACITY verification tool. The results of these tests are then encoded as an observation function O' with the format from (2.2).

Finally, in the fourth and last step of VERACITY, the new observations O' are integrated with all the observations obtained in the previous rounds of the process and the initial observations O_0, and the combined set of all available observations O is used in the next round of the verification process.

Example 3.2 For the three requirements for the TAS system from our motivating example, $\bowtie_1 = \bowtie_2 = \bowtie_3 = <$. Accordingly, the requirements will be verified as satisfied at 99% confidence level if the observations acquired over successive rounds of the VERACITY verification process (and within the available *budget*) lead to the calculation of 99% confidence intervals $([l_i, u_i])_{i=1..3}$ that satisfy $u_1 < bound_1$, $u_2 < bound_2$ and $u_3 < bound_3$. If, on the other hand, $l_i \geq bound_i$ for any $i \in \{1,2,3\}$ in one of the verification rounds, the verification process will decide that requirement i is violated at 99% confidence level, and will terminate in that round. Finally, if the testing *budget* is used up before sufficient observations of the medical analysis, pharmacy and alarm services are obtained to allow either of these decisions to be made, the verification process will terminate with a 'budget exhausted' outcome.

3.3.3 Adaptive uncertainty reduction heuristic

3.3.3.1 Desiderata

Before describing the VERACITY heuristic for partitioning the round testing budget *rbudget* among the *m* system components, we present a set of desirable properties (i.e. *desiderata*) that we propose for any such heuristic:

D1. The requirements verified as satisfied in previous rounds should not influence the partition of the round budget.

D2. Reaching a resolution on undecidable requirements (i.e. on requirements with $bound_i \in [l_i, u_i]$, cf. Figure 3.4(c)) that are *likely* to be violated should be prioritised when the round budget is partitioned. This desideratum captures the fact that verifying a requirement as violated ends the verification process immediately. Such requirements can be identified by noting that their $bound_i$ is very close to the "wrong" end of the confidence interval $[l_i, u_i]$. For instance, if $bound_i$ were much closer to l_i that to u_i in Figure 3.4(c), narrowing down the confidence interval $[l_i, u_i]$ even slightly has a good chance (but is, of course, not certain) of showing that requirement *i* is violated, and of terminating the verification process.

D3. If several undecidable requirements influence the partition of *rbudget*, undecidable requirements whose $bound_i$ value is closer to the middle of the confidence interval $[l_i, u_i]$ should have a bigger influence. This desideratum reflects the fact that the verification of such requirements is particularly affected by epistemic uncertainty, as a significant narrowing of their confidence intervals is likely to be needed in order to decide whether they are satisfied or violated.

D4. The *rbudget* fraction allocated to each component should reflect the sensitivity of the properties $prop_1$ to $prop_n$ from (3.1) to the parameters of that component. For instance, if the closed-form expression for the *i*-th requirement is $expr_i =$

$p_1 + 0.01p_2$, where p_1 and p_2 are probabilities associated with components C_1 and C_2, respectively, component C_1 should be allocated a larger *rbudget* fraction than C_2 (all other factors being equal).

3.3.3.2 Algorithm

The numbers of new component observations $(nobs_j)_{j=1..m}$, for each round of the VERACITY verification process are computed by function NEWOBS from Algorithm 1. This function takes the following arguments (cf. Figure 3.3):

- the round testing budget *rbudget*;
- the Markov chain M and the requirements $(prop_i \bowtie_i bound_i)_{i=1..n}$;
- the property confidence intervals $([l_i,u_i])_{i=1..n}$ and expressions $(expr_i)_{i=1..n}$ obtained in the previous step of the round;
- the component testing costs $(cost_j)_{j=1..m}$ and associated state sets with unknown transition probabilities $(Z_j)_{j=1..m}$; and
- the observations O available at the start of the round.

The algorithm has three parts. In the first part (lines 2–7), it identifies the set of *relevant requirements R* that will influence the partitioning of the round budget. According to desideratum D1, the set of undecidable requirements U is obtained in line 2. Next, the if statement from lines 3–7 checks whether $bound_i$ of any undecidable requirement is much closer (i.e. $1/\varepsilon_1$ times closer) to the "wrong" end of the confidence interval $[l_i, u_i]$ than to the "right" end, where:

- the "wrong" end of $[l_i, u_i]$ is the end beyond which requirement i is violated, i.e. l_i if $\bowtie_i \in \{<, \leq\}$, and u_i otherwise, cf. Figure 3.4;
- the helper functions WRONG, RIGHT return the respective ends of $[l_i, u_i]$; and

Algorithm 1 Adaptive uncertainty reduction heuristic

```
 1: function NEWOBS($rbudget$, $M$, $(prop_i \bowtie_i bound_i)_{i=1..n}$, $([l_i, u_i])_{i=1..n}$,
                      $(expr_i)_{i=1..n}$, $(cost_j)_{j=1..m}$, $(Z_j)_{j=1..m}$, $O$)
 2:     $U = \{i \in 1..n \mid bound_i \in [l_i, u_i]\}$                          ▷ desideratum D1
 3:     if $\exists i \in U : \frac{|bound_i - \textsc{Wrong}(\bowtie_i, l_i, u_i)|}{|bound_i - \textsc{Right}(\bowtie_i, l_i, u_i)|} < \varepsilon_1$ then          ▷ desideratum D2
 4:         $R \leftarrow \left\{ \mathrm{argmin}_{i \in U} \frac{|bound_i - \textsc{Wrong}(\bowtie_i, l_i, u_i)|}{|bound_i - \textsc{Right}(\bowtie_i, l_i, u_i)|} \right\}$
 5:     else
 6:         $R \leftarrow U$
 7:     end if
 8:     $paramEstimate \leftarrow \textsc{EstimateParams}(M, O)$
 9:     $(relevance_j \leftarrow 0)_{j=1..m}$
10:     for $i \in R$ do
11:         $weight = \frac{u_i - l_i}{\max\{|bound_i - (l_i + u_i)/2|, \varepsilon_2\}}$              ▷ desideratum D3
12:         for $j = 1$ to $m$ do
13:             $sens \leftarrow \sum_{p \in \textsc{Params}(M, Z_j)} \left| \frac{\partial expr_i(paramEstimate)}{\partial p} \right|$    ▷ desideratum D4
14:             $relevance_j \leftarrow relevance_j + weight \cdot sens$
15:         end for
16:     end for
17:     for $j = 1$ to $m$ do
18:         $nobs_j \leftarrow \left\lfloor \left(rbudget \cdot relevance_j\right) / \left(cost_j \cdot \sum_{k=1}^{m} relevance_k\right) \right\rfloor$
19:     end for
20:     return $(nobs)_{j=1..m}$
21: end function
```

- $\varepsilon_1 \in (0, 1)$ is an VERACITY configuration parameter.

As explained in desideratum D2, requirements with this property are likely to be violated. Therefore, if any such requirements exist, only the requirement most likely to be violated is retained in the relevant requirement set R (line 4). Otherwise, R is initialised to include all the undecidable requirements (line 6).

The second part of the algorithm (lines 8–16) starts by using the observations O to calculate estimates for each unknown transition probability (i.e. parameter) associated with a state from $Z = \bigcup_{j=1}^{m} Z_j$ (line 8). This calculation is performed by the auxiliary function ESTIMATEPARAMS, which estimates the unknown transition probabilities between each state in $z \in Z$ and each state $s \in S$ using the observed transition frequency

$O(z,s)/\sum_{s' \in S} O(z,s')$. The special case when $\sum_{s' \in S} O(z,s') = 0$ for one or more states $z \in Z$ may be encountered in the first round, as we allow an empty initial set of observations O_0 (cf. Section 3.3.1). In this case, which we do not show in Algorithm 1 in order to keep the pseudocode simple, ESTIMATEPARAMS raises an exception and the round budget is split uniformly between the components whose state sets Z_j include states with zero observations.

Next in this part of the algorithm, a component relevance measure $relevance_j$ is first initialised in line 9, and then updated with contributions corresponding to the relevant requirements R by the for loop from lines 10–16. Each such contribution is the product of two factors (line 14) that correspond to desiderata D3 and D4, respectively:

- *weight*, a factor calculated as the ratio between the width of the confidence interval $[l_i, u_i]$ and the distance between $bound_i$ and the middle of the interval $[l_i, u_i]$ (line 11, where a small VERACITY configuration parameter $0 < \varepsilon_2 \ll 1$ is used to prevent a division by zero); and

- *sens*, a measure of the sensitivity of expression $expr_i$ to the epistemic uncertainty affecting the parameters of component j, calculated by summing the absolute value of the partial derivatives of $expr_i$ with respect to every parameter of component j (line 13), where the set of all such parameters is provided by the auxiliary function PARAMS, and the partial derivatives are evaluated for the parameter values estimated in line 8.

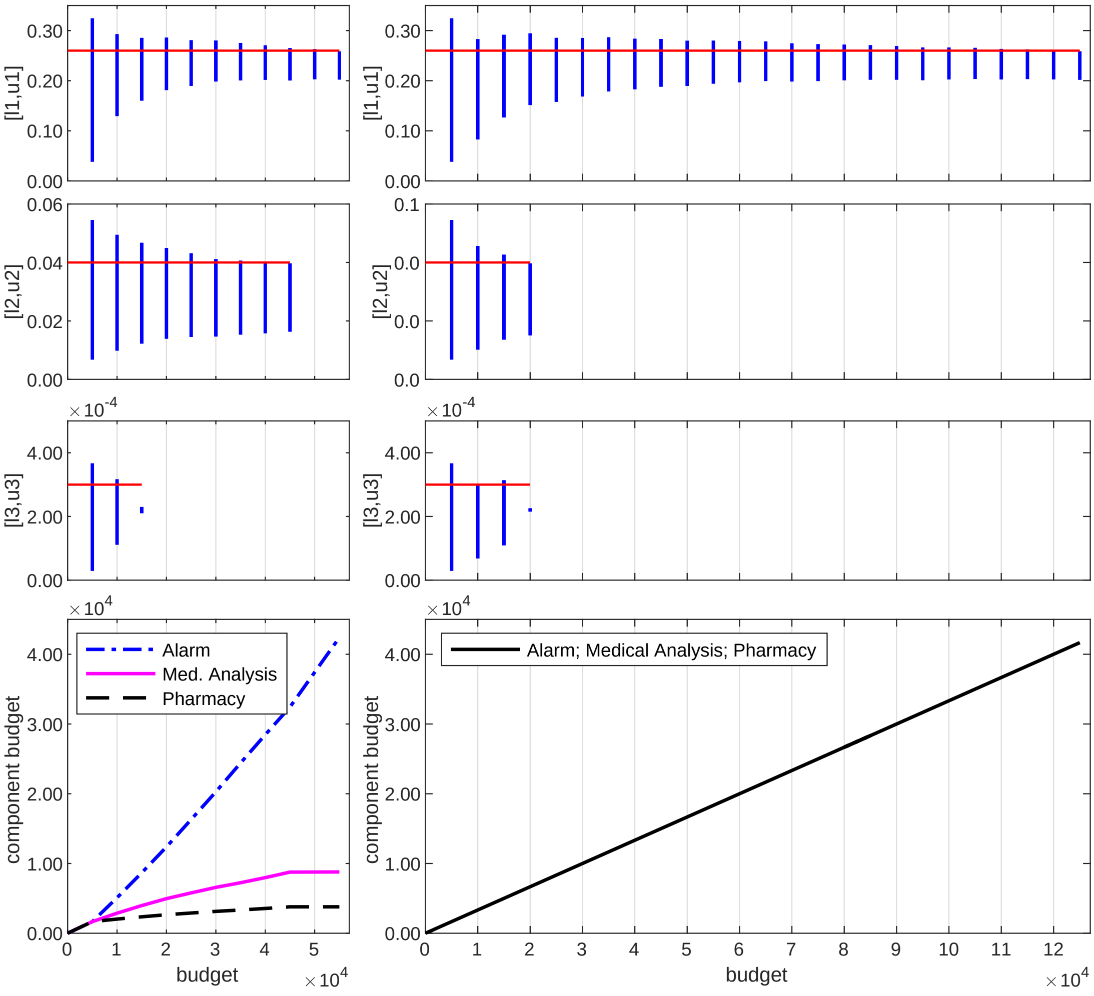
The third and final part of the algorithm (lines 17–19) partitions the round budget *rbudget* based on the relevance of each component. Thus, component j is allocated a fraction of $relevance_j / \sum_{k=1}^{m} relevance_k$ of the round budget; the number of new observations for component j is then calculated by dividing this *rbudget* fraction by the cost $cost_j$ of testing the component once.

For improved readability, a couple of efficiency improvements are not included in Algorithm 1. First, the function PARAMS and the partial derivatives required for factor *sens* (line 13) can be precomputed once (in the first round of the VERACITY verification process), as the SUV parameters associated with a component do not change; only the evaluations of the precomputed partial derivatives need to be done in each round, for the new *paramEstimate* from that round. Second, $\sum_{k=1}^{m} relevance_k$ from line 18 only needs to be calculated once, e.g. immediately before the for loop that uses it.

With these improvements in place, the complexity of algorithm is $\mathsf{O}(mn)$, because of the two nested for loops from lines 10–16 and 12–15, respectively. This is typically negligible compared to the complexity of the formal verification with confidence intervals and the additional unit testing from steps one and three of the VERACITY verification process, respectively.

Example 3.3 Figure 3.5 shows the difference between the verification of the TAS nonfunctional requirements from Table 3.1 using the VERACITY verification process from Figure 3.3 with: (a) our adaptive uncertainty reduction heuristic from Algorithm 1; and (b) our heuristic replaced with a uniform splitting of the round testing budget among the three system components. These results were obtained assuming that the unknown probabilities from the Markov chain in Figure 3.2 were $p_{al=0.94}$, $p_{ma=0.99}$ and $p_{ph=0.95}$, and using random number generators to synthesise additional observations O' based on these probabilities in the additional unit testing step of the VERACITY verification process from Figure 3.3. The verification was performed with a round budget $rbudget = 5000$, unlimited overall testing *budget*, and the default values $\varepsilon_1 = 0.15$ and $\varepsilon_2 = 10^{-6}$ for the two parameters of the VERACITY heuristic from Algorithm 1.

The top three pairs of graphs from Figure 3.5 show how the 95% confidence intervals $([l_i, u_i])_{i=1..3}$ (depicted as vertical lines) for the nonfunctional properties from the three TAS requirements from Table 3.1. These confidence intervals become narrower as additional observations are obtained in each round of the verification process, until they are narrow enough to fit completely under the $bound_i$ threshold (drawn as a horizontal line) from their associated requirement, i.e. until $u_i < bound_i$. As soon as this condition is met for a confidence interval $[l_i, u_i]$, that interval is not longer calculated in subsequent



(a) VERACITY (b) uniform uncertainty reduction

Figure 3.5: Verification of the nonfunctional requirements for the TAS system from the motivating example using (a) VERACITY adaptive vs. (b) uniform uncertainty reduction

verification rounds. When the condition is met for all three confidence intervals, the epistemic uncertainty was reduced sufficiently to conclude that all requirements are satisfied, and the verification process terminates successfully. As shown by these graphs, VERACITY and the uniform uncertainty reduction method finish the verification of each requirement after a different number of verification rounds, and VERACITY completes the verification of the entire set of requirements with an overall testing cost of $55,000$ compared to a 127% higher overall testing cost of $125,000$ for the approach based on uniform uncertainty reduction.

The bottom pair of graphs from Figure 3.5 shows the cumulative testing cost per system component. When uniform uncertainty reduction is used, this cost is identical for all components, and is growing linearly with the number of verification rounds. In contrast, for the VERACITY approach, the cumulative cost grows at different rates for different components. Furthermore, the rate of growth for any single component varies across verification rounds because VERACITY continually *adapts* its partitioning of the round budget to the observations acquired in previous rounds, and to the effect that these observations have on confidence intervals $([l_i, u_i])_{i=1..3}$.

3.4 Implementation

We implemented the VERACITY verification process as a Java tool built on top of the existing FACT model checker [25]. The source code for the VERACITY tool is available at `https://gitlab.com/nnma500/veracity/-/tree/main`.

3.4.1 High level diagram

VERACITY receives the model (i.e. the pDTMC model), requirements and confidence level, as depicted in Figure 3.6. In addition, VERACITY can read the cost of each component and initial observations from the model and reads the budget and round budget from the configuration file. Next, VERACITY sends the model, the equivalent properties of the requirements and the confidence level to FACT to obtain confidence intervals and algebraic expressions for each requirement. After that, VERACITY checks whether the bound specified in each requirement is located inside the associated confidence interval. If the bounds lies inside the intervals, VERACITY performs sensitive analyses for each component and uses the result of these analysis and factors such as the cost of testing the component and the round budget to suggest numbers of additional observa-

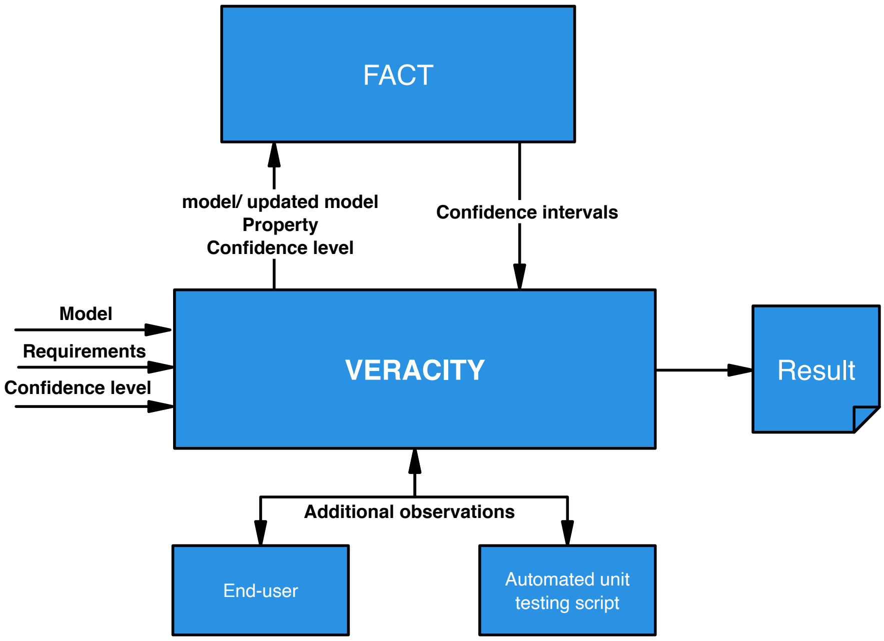


Figure 3.6: High level diagram of the VERACITY

tions for each component. The suggested observations are obtained either by running the automated unit testing script (if such a script is provided through the configuration file) or from the end user. In the latter case, the user needs to run the tests manually and to supply the outcome of the tests to VERACITY. Subsequently, VERACITY updates the model with the new observations and sends the updated model to FACT to produce new (narrower) confidence intervals. This process continues until one of the following conditions is met: (i) one requirement is violated, (ii) all the requirements are satisfied, or (iii) the total budget is exhausted.

3.4.2 Class diagram

We implemented VERACITY using seven classes that work together with FACT to carry out the verification. These classes are depicted in Figure 3.7 and described as follows:

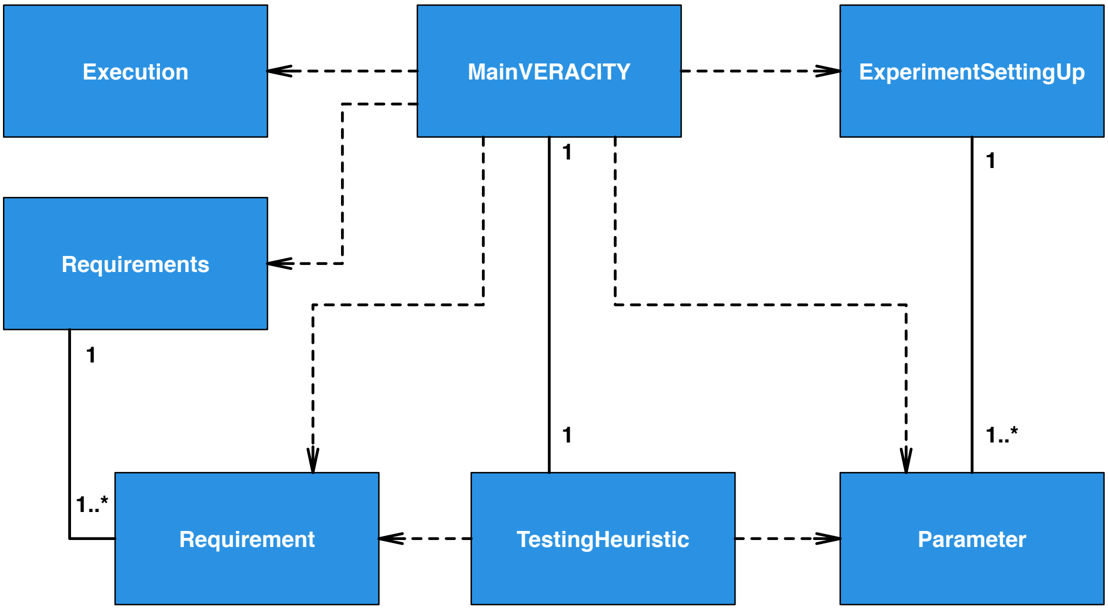


Figure 3.7: Class Diagram of the VERACITY

- MainVERACITY: This class represents the main class of our tool and receives inputs, such as the pDTMC model and its requirements. It also accepts testing parameters, such as round budget, budget and observation script, as inputs. This class runs FACT for each property that appears in a requirement, manages the rounds and budget, receives additional observations from the TestHeuristic class and then updates the model with the new observations.

- ExperimentSettingUp: This class handles components with specific characteristics, such as ID, cost and observations; an array list is used for this purpose.

- Execution: This class finds the location of the PRISM model checker on the local machine, invokes MATLAB and assists in computing the confidence interval of each property. The class is modified from the existing FactExecution class in FACT. We removed unnecessary FACT code related to the GUI to ensure that the tool supports a command line interface.

- Parameter: This class holds and manages the model components and their characteristics and assists in updating the observation of each component during the execution of the VERACITY heuristic.

- Requirement: This class stores and provides access to information about a requirement of the verified system.
- Requirements: This class stores and provides access to information about the requirements of the verified system.
- TestingHeuristic: This core VERACITY class implements the heuristic that determines the additional observations required for each component. It performs sensitive analyses for each component to compute its importance and then uses this information to calculate the number of additional required observations.

3.4.3 Use of the tool

The VERACITY tool takes as input: a parametric Markov chain M expressed in the PRISM modelling language [27] and annotated with the component costs $\left(cost_j\right)_{j=1..m}$ and state sets $\left(Z_j\right)_{j=1..m}$, and with the initial observations O_0; a set of PCTL-encoded nonfunctional requirements; and a confidence level α.

The overall testing *budget* and round testing budget *rbudget* are specified via a configuration file. In addition, this configuration file allows the user to optionally specify a component testing script that the tool can execute with the command

`testing-script` j $nobs_j$

in order to obtain $nobs_j$ additional observations for component j automatically in the third step of the VERACITY verification process (cf. Figure 3.3). If provided, this script needs to run $nobs_j$ unit-test against component j (e.g. by invoking the appropriate third-party service for the TAS system from our motivating example), and to return the $nobs_j$

observations from these tests as a list of numbers of transitions from the states in Z_j to other states of the Markov chain M. Alternatively (i.e. if the testing script is not provided), the tool asks the user to supply the required $nobs_j$ observations interactively at each round of the verification process.

Our VERACITY tool uses the model checkers FACT [25] and PRISM [27], along with MATLAB[5], to compute the confidence intervals and property expressions in the first step of the verification process. The heuristic employs the Java mathematical library mXparser [93] to assist in computing the derivative of algebraic expressions for the sensitivity analysis. The tool is freely available on our project website `https://www.cs.york.ac.uk/tasp/VERACITY`, along with detailed instructions and all models, requirements and results of this chapter.

3.5 Evaluation

We evaluated VERACITY by performing an extensive set of experiments aimed at answering the following research questions.

RQ1 (Effectiveness) Does VERACITY reduce the testing budget needed to verify a set of nonfunctional requirements compared to the baseline approach that partitions the testing budget of each verification round equally among the components of the SUV?

RQ2 (Cost awareness) How effective is VERACITY at reducing the testing budget in scenarios where the SUV components have different testing costs?

[5] `http://www.mathworks.co.uk/products/matlab`

RQ3 (Configurability) What effect does adjusting the round budget have on the overall testing cost[6] and verification time of VERACITY?

To assess the generality of VERACITY, we performed our experiments within two case studies that used software systems from different domains. The first case study was based on the TAS system from our motivating example. In the second case study, we applied VERACITY to the verification of an online shopping web application. This system is introduced in Section 3.5.1, followed by descriptions of the experiments carried out to address the three research questions in Sections 3.5.2, 3.5.3 and 3.5.4. To enable the reproducibility of our results, we made all the models, properties and data from our experiments available on the VERACITY project website `https://www.cs.york.ac.uk/tasp/VERACITY`.

3.5.1 Online shopping web application

The system we used for the second case study is an online shopping application adapted from [94]. We modelled the shopping process implemented by this application using a parametric Markov chain that comprises a combination of known and unknown transition probabilities.[7] The known transition probabilities correspond to application components that have been in use for a long time, and for which the values of these probabilities can be determined from application logs. In contrast, the unknown transition probabilities correspond to new versions of several components that the online shopping company's developers have re-implemented and want to evaluate through *A/B testing*.

A/B testing [95–97] is a method for testing a new online application feature, or a new implementation of an existing feature. Frequently used by leading companies like

[6] Note that this corresponds the minimum testing budget for which the verification completes with a conclusive result as opposed to the budget being exhausted.

[7] Note that what we model here is the stochastic behaviour of the customer, not the stateful process implemented by the web application. As such, using a Markov model for this purpose is suitable, as further shown by the use of Markov chains to model web applications in recent projects including [25, 75].

Table 3.2: Nonfunctional requirements for the online shopping application

ID	Requirement	PCTL formula
R1	The probability that customers complete the shopping process successfully shall be above $bound_1$.	$P_{>bound_1}[\mathrm{F}\, success]$
R2	The probability that the authentication component fails shall be below $bound_2$.	$P_{<bound_2}[\mathrm{F}\, authFail]$
R3	The average number of successful uses of new components per shopping session shall exceed $bound_3$.	$R_{>bound_3}[\mathrm{F}\, done]$

Amazon, Facebook, Google and Microsoft, the method involves splitting the users of a web application into two sets, such that one set of users is given access to a version of the application that includes the new feature (or the new implementation of a feature), while the other set continues to use the standard version of the application. In this way, A/B testing allows companies to evaluate new features and components, and to decide whether they should be included in the default version of online applications or not.

For our case study, we assume that the online shopping company wants to verify whether the nonfunctional requirements from Table 3.2 would be satisfied if several application components were to be replaced with new variants. Furthermore, we assume that in order to limit the business loss that may occur if these requirements are in fact violated, the company wants to perform this verification with as little A/B testing of each of four new component implementations as possible.

The parametric Markov chain modelling the operation of the online shopping application is shown in Figure 3.8. In the initial state (s_0) of this Markov chain, a customer attempts to login. We assume that the authentication web page is one of the components for which a new implementation needs to be tested, and therefore the probability that the customer can follow the authentication instructions and succeeds to login, denoted p_a, is unknown. If the authentication succeeds (state s_2), the customer is identified either as a returning customer (whose settings from the previous shopping session are restored, state s_1) or as a new customer (for whom default settings are used, state s_3). In both cases, the customer searches for items to purchase (states s_4/s_6) and adds them to the shopping basket (states s_7/s_9), until eventually all the required items are in the shopping basket and the customer moves to checkout where he or she selects between two shipping options: fast shipping (state s_{10}) or standard shipping (state s_{12}). We assume that the probabilities of the incoming transitions into states s_1, s_3, s_4, s_6, s_7, s_9, s_{10} and s_{12} are known (from the previous use of the web application) and have the values from Figure 3.8.

However, we assume that the web application components for selecting the two shipping options have been re-implemented, and therefore the probabilities p_{fs} and p_{ss} that the customer manages to use them successfully and to reach the payment state s_{11} are unknown. Likewise, we consider that a new version of the payment component has been implemented, and that the probability p_p that the customer manages to use it successfully (and to move to the logout state s_{14}) is unknown. Finally, we assume that the logout involves the use of the same new authentication component that was used for

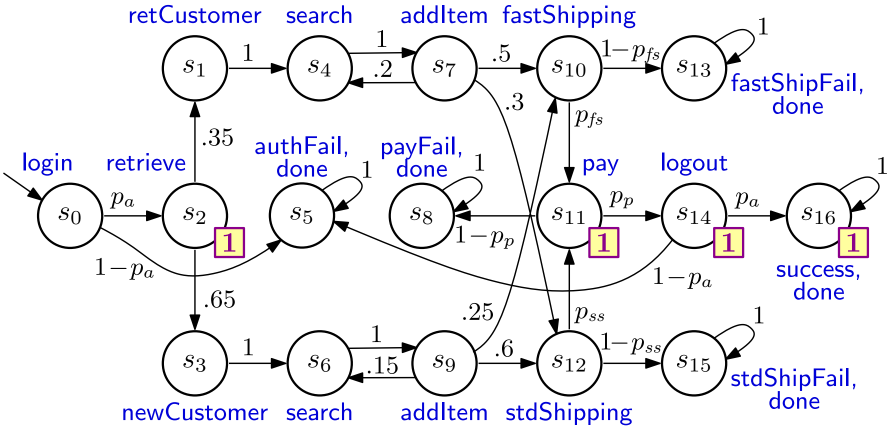


Figure 3.8: Parametric Markov chain modelling the online shopping application. To enable the verification of requirement R3 from Table 3.2, the model is augmented with a reward structure that "counts" the successful uses of new components; this reward structure associates a reward $\rho(s_2) = \rho(s_{11}) = \rho(s_{14}) = \rho(s_{16}) = 1$ with each state that follows after a successful use of a new component (as shown in small squares next to these states) and zero rewards with the other states and the state transitions of the Markov chain.

login, and therefore its (unknown) probability of succeeding is p_a.

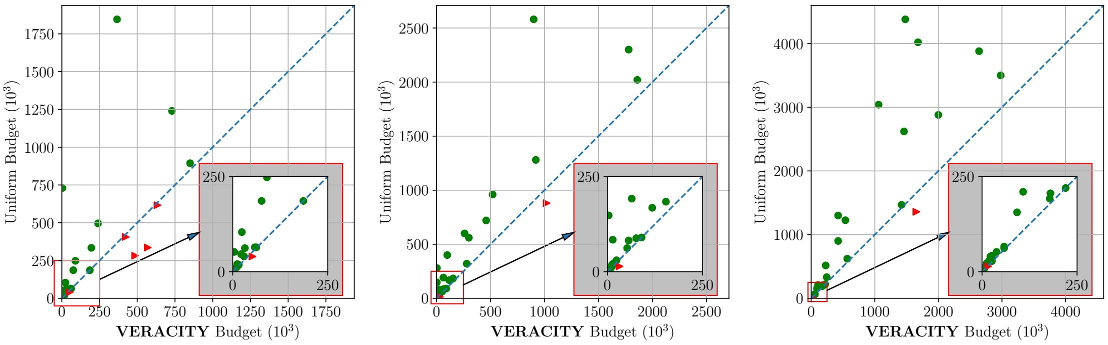
3.5.2 RQ1 (Effectiveness)

For each of the two systems used in our evaluation, we examined a broad range of simulated scenarios in which both the values of the unknown probabilities of the parametric Markov chain M from Figure 3.3 and the bounds from the nonfunctional requirements (3.1) were randomly generated. In doing so, we ensured that the evaluation covered a combination of:

1. scenarios in which all requirements were satisfied: (i) by a narrow margin, i.e. the actual values of the properties from the nonfunctional requirements (3.1) were close to their associated bounds; (ii) by a wide margin; and (iii) some by a narrow margin and the others by a wide margin;
2. scenarios in which some of the requirements were satisfied and the remaining requirements were violated by: (i) a narrow margin; (ii) a wide margin; and (iii) some by a narrow margin and the others by a wide margin; and
3. scenarios in which all requirements were violated by: (i) a narrow margin; (ii) a wide margin; and (iii) some by a narrow margin and the others by a wide margin.

We note that the values for the unknown probabilities of the parametric Markov chain were required to establish the ground truth for the verification. These values remained unknown to the verification process.

In all the experiments, we used the VERACITY tool in conjunction with a simulated component-testing script with the characteristics described in Section 3.4. This script emulated the outcome of unit testing the SUV components by using a separate Java pseudorandom number generator for each component. To avoid any bias in the comparison of VERACITY with the baseline approach mentioned in research question



(a) $\alpha = 0.90$ (b) $\alpha = 0.95$ (c) $\alpha = 0.99$

Figure 3.9: Testing budgets required to complete the verification of the TAS nonfunctional requirements using the VERACITY and the uniform methods for partitioning the testing round budget among the components of the TAS system. The wide range of budgets required to complete the verification process for different scenarios reflects the variety of these scenarios: in some scenarios, the TAS requirements are satisfied or violated by a wide margin (so less testing is needed, as shown by the inset diagrams), whereas in others some or all of the requirements are satisfied or violated by a narrow margin (so much more testing is needed).

RQ1, we used the same pseudorandom number generator seeds in the corresponding experiments for the two approaches.

3.5.2.0.1 Case Study 1 (TAS) For the TAS system, we carried out experiments that examined the effectiveness of VERACITY for a set of 33 scenarios that were randomly generated as described at the beginning of this section. For each of these scenarios, the verification of the TAS nonfunctional requirements was carried out at three confidence levels: $\alpha = 0.90$, $\alpha = 0.95$ and $\alpha = 0.99$. Finally, to answer research question RQ1, two experiments were performed for each scenario and each confidence level α: one in which we used the VERACITY uncertainty reduction heuristic, and one in which we used the baseline approach that partitions the testing budget of each verification round equally among the TAS components. In total, we performed 198 verification experiments, corresponding to 33 scenarios × 3 confidence levels × 2 uncertainty reduction methods.

The experimental results are shown in Figures 3.9 and 3.10, which compare the minimum testing budgets required to complete the verification of the TAS requirements using the two uncertainty reduction methods.

Figure 3.9 shows the minimum testing budget consumed to complete the verification of the nonfunctional requirements for the TAS system using both VERACITY and uniform (baseline) methods. The experiments where VERACITY consumed a lower testing budget than the baseline method appear above the diagonal (shown as a dashed line) and are depicted as green dots, whereas the experiments where VERACITY required a higher testing budget appear under the diagonal and are depicted as red triangles. We can see that (i) the number of experiments above the diagonal line is greater than the number of experiments below it, indicating that the consumed testing budget is, in most cases, reduced by using the VERACITY method, and (ii) most experiments below the diagonal are close to it, indicating that even when VERACITY is occasionally outperformed by the baseline method, it is only by a small margin.

Figure 3.10 contains box plots for the (minimum) additional testing budget required by the baseline method for three different confidence levels: 90%, 95% and 99%. We notice from these box plots that the median (the line dividing each blue box) is always

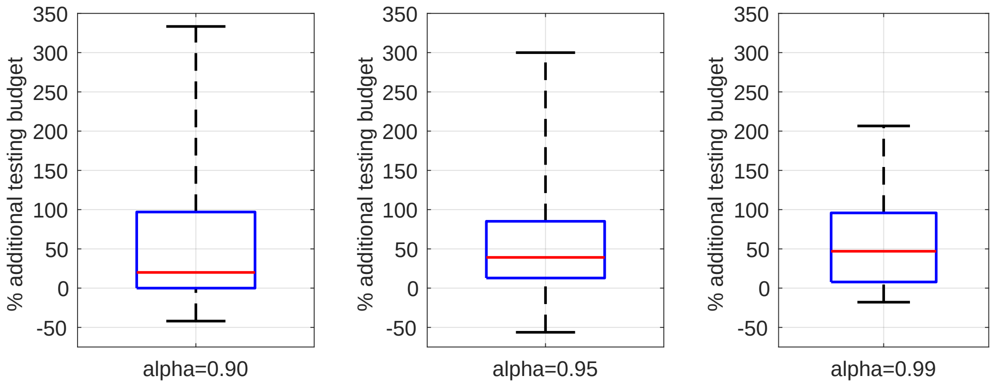
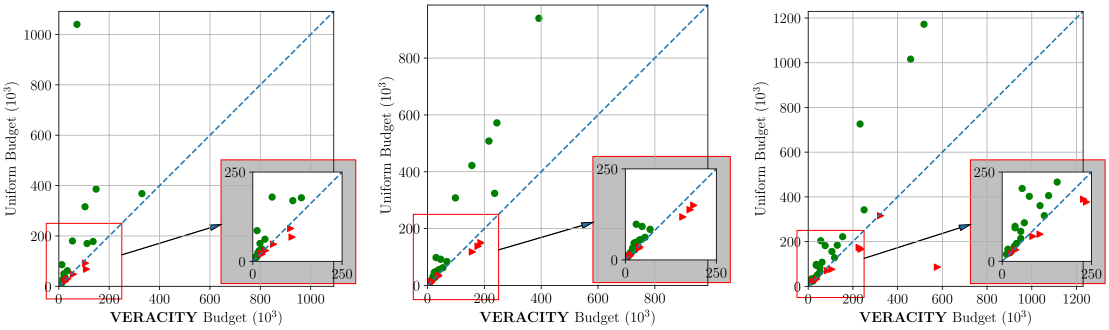


Figure 3.10: Additional testing budget required to complete the verification of the TAS nonfunctional requirements when the round budget is partitioned using the uniform method instead of the VERACITY method. To ensure readability, the upper part of the boxplots is truncated, meaning that the outliers at 404%, 1200% and 18150% (for $\alpha = 0.90$), and at 4252% and 6900% (for $\alpha = 0.95$) are not shown. No outliers exist below the bottom whisker of any of the boxplots.

above 0 and increases when the confidence level is increased. That is, at 90%, 95% and 99% confidence levels, the uniform method requires more than 20%, 40% and 50% of the testing budget used by VERACITY to carry out the verification.

These results show that VERACITY consistently outperforms the baseline method by completing the verification process with smaller testing budgets for a great majority of the scenarios and at all confidence levels. In a few scenarios, the baseline method performs better than VERACITY (typically only marginally better). This is expected given the stochastic nature of the verified system, and the fact that the verification starts with no knowledge about the behaviour of the three TAS components. Finally, in a small number of additional scenarios, VERACITY achieves only modest testing cost savings. This is also expected, as the best way to reduce epistemic uncertainty in some verification scenarios is to partition the round testing budget approximately equally among the tested system components, and our approach manages to do this well.

The experimental results show that the testing budget reductions enabled by VERACITY are particularly significant when the verification is carried out at higher confidence levels. This is extremely useful for two reasons. First, in real-world scenarios, the nonfunctional requirements of software systems should be verified with high levels of confidence (e.g. $\alpha = 0.95$ or even $\alpha = 0.99$); deploying a system whose requirements were only verified at a low confidence level introduces significant risks. Second, the testing budget needed to complete the verification increases with the confidence level α, as the epistemic uncertainty needs to be reduced much more in order to make decisions with higher confidence. This increase of the required testing budget for larger α values is clearly visible in the scales of the graphs from Figure 3.9. As such, the scenarios in which VERACITYreduces the cost of testing the most are: (i) of particular practical importance; and (ii) characterised by high testing costs, so the cost reductions achieved by our uncertainty reduction method are especially beneficial.



(a) $\alpha = 0.90$ (b) $\alpha = 0.95$ (c) $\alpha = 0.99$

Figure 3.11: Testing budgets required to complete the verification of the WebApp nonfunctional requirements using the VERACITY and the uniform methods for partitioning the testing round budget among the components of the online shopping system.

3.5.2.0.2 Case Study 2 (Online shopping web application) To assess the effectiveness of VERACITY for the online shopping web application (WebApp), we performed a similar suite of experiments to those described for the TAS case study. This time, we examined the ability of VERACITY to reduce testing costs compared to the baseline uncertainty reduction method for the verification of 30 randomly generated scenarios. In each of the 30 scenarios, the verification of the WebApp requirements was carried out at three confidence levels ($\alpha = 0.90$, $\alpha = 0.95$ and $\alpha = 0.99$), for both the VERACITY and the uniform uncertainty reduction methods, giving a total of 180 experiments.

Figures 3.11 and 3.12 summarise the results of these experiments. Figure 3.11 shows the minimum testing budget necessary to complete the verification process for the WebApp when using VERACITY and when using baseline methods. The green dots represent the experiments where VERACITY outperformed the baseline method and, hence, consumed lower testing budget, while the red triangles depict the experiments where the baseline completed the verification process with less testing budget than VERACITY. The box plots in Figure 3.12 represent the (minimum) additional testing budget needed by the baseline method for verification in order to be equal to VERACITY. We notice that to outperform VERACITY, the baseline method required more than 30%, 40% and 35% of the testing budget used by VERACITY at the confidence levels 90%, 95% and 99%, respectively.

We notice that as for the TAS system, VERACITY successfully reduces the testing cost required to complete the verification of the non-functional requirements, across a wide range of testing cost needs (where small testing cost are needed when the requirements are satisfied or violated by a wide margin, and large costs when some or all of the requirements are narrowly satisfied/violated). In the small number of scenarios where the uniform round budget partitioning method achieves better results, the overall testing cost is small, and the VERACITY-based verification is typically only marginally more expensive. Again, many significant cost reductions occur when (i) the baseline method budget is high and (ii) the requirements are verified at higher confidence levels. For instance, all of the baseline method budgets above 400,000 from Figure 3.11 (one for $\alpha = 0.90$, four for $\alpha = 0.95$, and three for $\alpha = 0.99$) are at least halved by VERACITY.

3.5.3 RQ2 (Cost awareness)

In many practical situations, the costs of testing different components of a system are different. This is likely to be true, for instance, when these costs represent the times

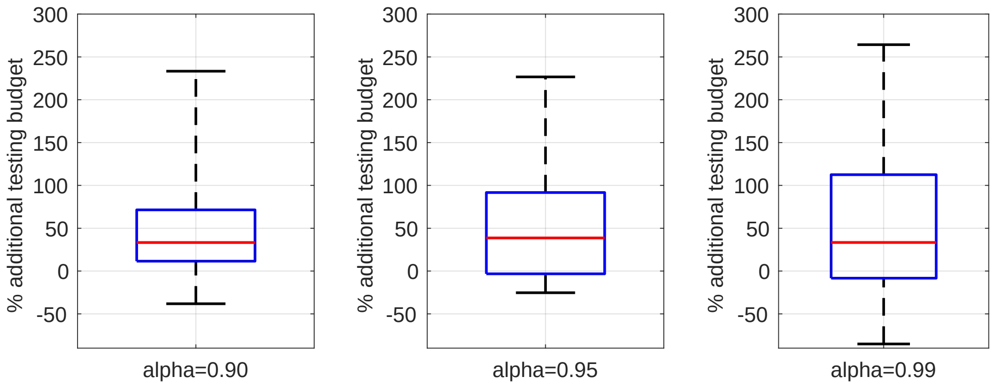
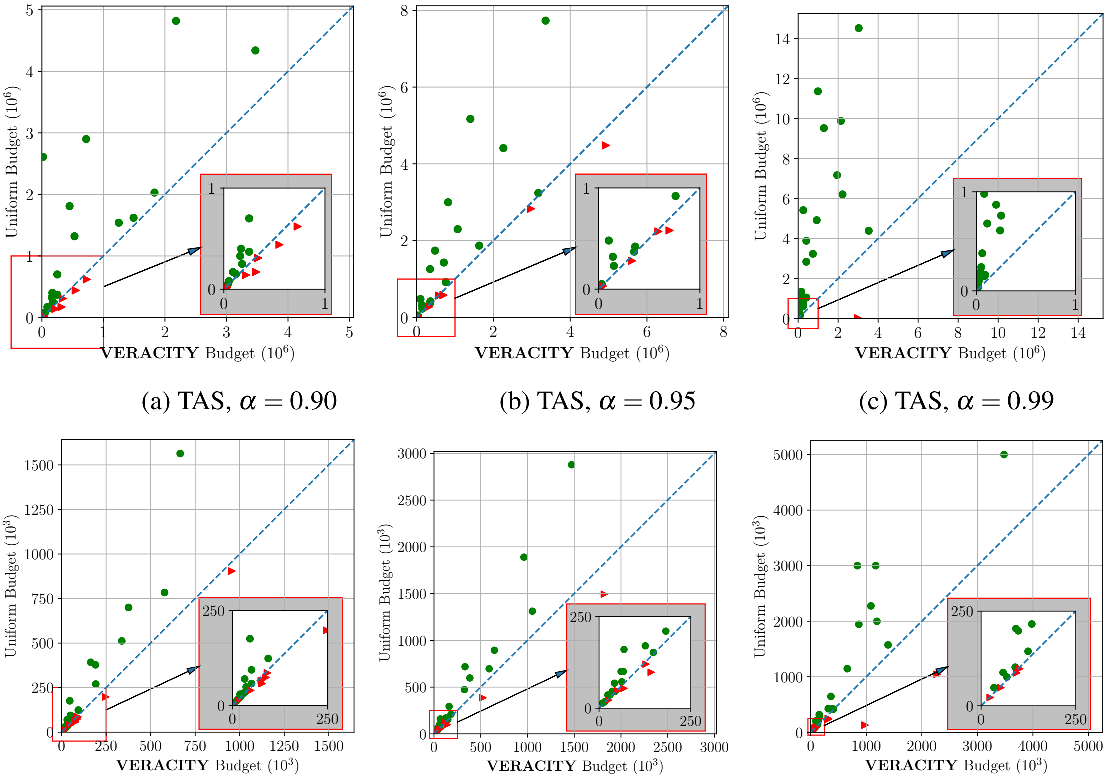


Figure 3.12: Additional testing budget required to complete the verification of the WebApp nonfunctional requirements when the round budget is partitioned using the uniform method instead of the VERACITY method. To ensure readability, the upper part of the boxplots is truncated at 300%, meaning that the outliers at 616% and 1344% (for $\alpha = 0.90$) are not shown. No outliers exist below the bottom whisker of any of the boxplots.

required for the regression testing of a software system with several modified components [98] or for testing different web services at runtime, and a limited overall time (i.e. testing budget) is available to verify whether using these services as part of a service-based system like TAS satisfies a set of nonfunctional requirements [20]. This is also likely to be true the A/B testing of new features of an online application [95–97] like the shopping application from Section 3.5.1, where these costs may represent the different (expected) business impact of each of the new features not working as intended. While a specific testing budget may be difficult to define in this second situation, it is easy to see that the verification should be completed with as low a testing budget as possible.

To evaluate the usefulness of VERACITY in such situations, we repeated all the experiments from Section 3.5.2 assuming different testing costs for the components of the TAS and WebApp systems from our two case studies. To this end, we took each of the 33 TAS verification scenarios and of the 30 WebApp verification scenarios, and we assigned randomly generated testing costs in the interval $[1,5]$ to the three TAS components and the four WebApp components, respectively. The testing budgets required to complete the verification process using the VERACITY and the baseline round-budget partitioning methods in these scenarios with different component testing costs are compared in Figures 3.13 and 3.14. As in the scenarios with the same testing costs for all components, our VERACITY verification approach outperforms the baseline verification approach in the majority of the examined scenarios, often by a large margin. As shown in Figure 3.14, this margin increases for larger confidence level value. This increase is particularly significant for the TAS system, where the median additional testing budget required by the baseline verification approach grows from 33% at $\alpha = 0.90$ to 42% at $\alpha = 0.95$, and 335% at $\alpha = 0.99$. This growth is less pronounced but still present for the WebApp system, where the median additional testing budget increases from 16% at $\alpha = 0.90$ to 26% at $\alpha = 0.95$, and 38% at $\alpha = 0.99$.

In the small number of scenarios where the baseline approach completes the verification within a smaller testing budget, the difference between this approach and VERAC-



(a) TAS, $\alpha = 0.90$ (b) TAS, $\alpha = 0.95$ (c) TAS, $\alpha = 0.99$

(d) WebApp, $\alpha = 0.90$ (e) WebApp, $\alpha = 0.95$ (f) WebApp, $\alpha = 0.99$

Figure 3.13: Testing budgets required to complete the verification process using the VERACITY and the uniform methods for partitioning the testing round budget among the components of the TAS and WebApp systems.

ITY is typically modest, and/or occurs for scenarios where both approaches perform the verification with relatively small overall testing budgets.

3.5.4 RQ3 (Configurability)

The round testing budget *rbudget* is a key parameter of VERACITY. For very large *rbudget* values, all the component observations needed to complete the verification of the nonfunctional requirements are acquired in a small number of verification rounds, or even in a single round. This is undesirable for two reasons. First, with only a few verification rounds, VERACITY has limited opportunity to meaningfully adapt its par-

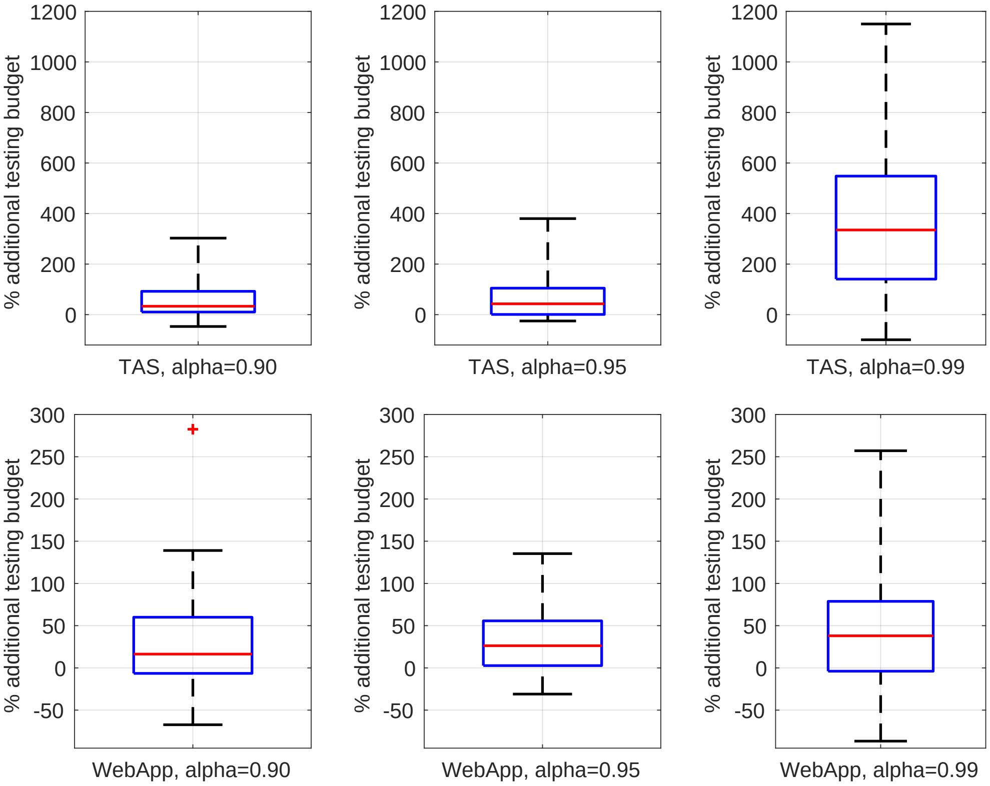
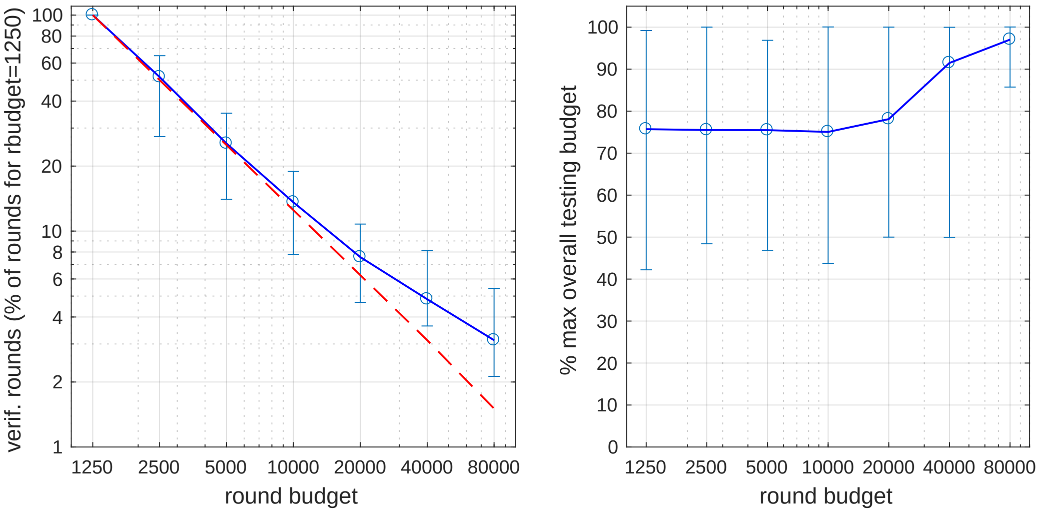


Figure 3.14: Additional testing budget required to complete the verification of the TAS and WebApp nonfunctional requirements when the round budget is partitioned using the uniform method instead of the VERACITY method. To ensure readability, the upper part of the TAS boxplots is truncated at 1200%, meaning that two TAS outliers at 12950% (for $\alpha = 0.90$) and at 1984% (for $\alpha = 0.99$) are not shown. No other hidden outliers exist for any of the boxplots.

titioning of the round budget to the system and requirements being verified. Second, with extremely large verification rounds, many more observations than strictly needed are likely to be acquired in the last verification round. Both of these drawbacks of very large *rbudget* values can lead to VERACITY using larger overall testing budgets than necessary.

Very small *rbudget* values are equally undesirable, also for two reasons. First, such round budgets yield only a few additional observations in each round, so only small sets of observations are available to guide the VERACITY round-budget partitioning in the early verification rounds. As such, the adaptive partitioning of the round budget is



(a)

(b)

Figure 3.15: Effect of varying the VERACITY round budget on (a) the number of verification rounds; and (b) a normalised measure of the overall testing budget (see main text for details). The plots show mean values and ranges over 10 randomly selected verification scenarios.

likely to be sub-optimal early in the verification process, and larger testing budgets may be required overall. Second, very small round budgets require large numbers of verification rounds, and these rounds can be computationally very expensive because of the formal verification with confidence intervals step of VERACITY˜(cf. Figure 3.3). For example, for the TAS and WebApp parametric Markov chains and requirements from our case studies, the mean execution time for this step was 9.8s and 12s, respectively, on a c5.2xlarge Windows Server 2019 Amazon EC2 instance with 3.00GHz Intel(R) Xeon(R) Platinum 8124M CPU, and 16 GB of memory (further details about the VERACITYẽxecution time are provided later in this section). Therefore, using many tens or hundreds of rounds to complete the verification process could be unacceptable in some scenarios (e.g. when the verification is done at runtime [94, 99, 100]).

To analyse these effects of *rbudget*, we randomly selected five of the TAS verification scenarios and five of the WebApp verification scenarios from Section 3.5.2, and we used VERACITY to verify the nonfunctional requirements of the two systems for each round budget value in $RB \in \{1250, 2500, 5000, 10000, 20000, 40000, 80000\}$. The experimental results are presented in Figure 3.15. First, the graph from Figure 3.15(a) shows the effect of repeatedly doubling the round budget from an initial value $\mathit{rbudget} = 1250$ until a final value $\mathit{rbudget} = 80000$. The dashed line from this graph shows what the "ideal" effect of increasing the round budget would look like, i.e. a halving of the number of verification rounds each time when *rbudget* is doubled, from the baseline of 100% for $\mathit{rbudget} = 1250$ to 50% of that baseline for $\mathit{rbudget} = 2500$, 25% for $\mathit{rbudget} = 5000$, etc. In reality, the number of verification rounds is increasingly above the ideal value as *rbudget* grows, until it is above this ideal value for all 10 verification scenarios both for $\mathit{rbudget} = 40000$ and for $\mathit{rbudget} = 80000$. This indicates that very large *rbudget* values increase the overall testing budgets required by VERACITY—a finding that is further confirmed by Figure 3.15(b), which shows how the overall testing budget necessary to complete the verification process increases with *rbudget*.

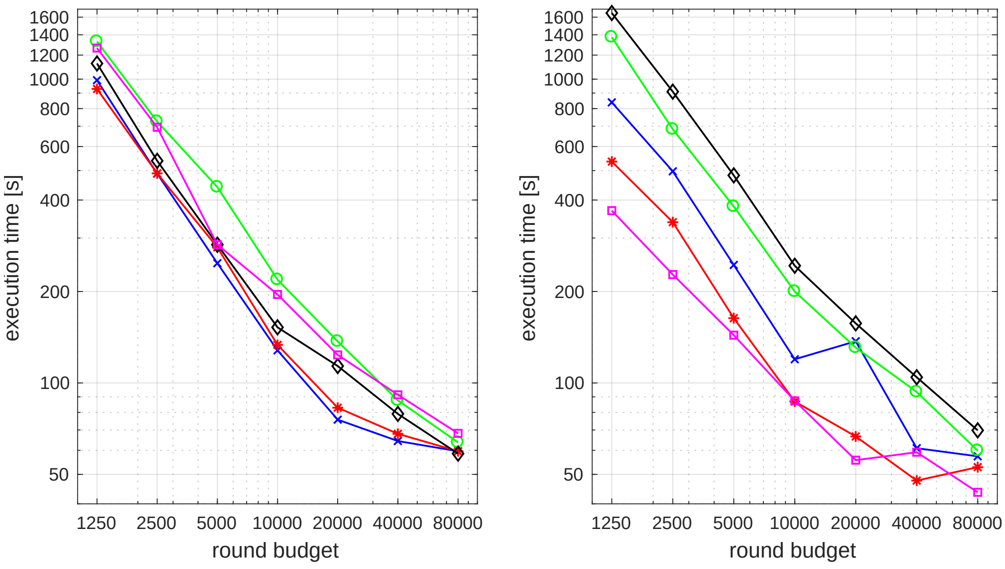
To summarise the very different overall testing budgets required for our 10 randomly selected verification scenarios in a consistent way, Figure 3.15(b) considers the budgets b_1, b_2, …, b_7 associated with each verification scenario and the seven *rbudget* values from *RB*, finds $b_{\mathsf{max}} = \max\{b_1, b_2, \ldots, b_7\}$, and computes the percentages of b_{max} that b_1, b_2, …, b_7 correspond to, i.e. $pb_1 = 100b_1/b_{\mathsf{max}}$, $pb_2 = 100b_2/b_{\mathsf{max}}$, …, $pb_7 = 100b_7/b_{\mathsf{max}}$. These "normalised" budgets show the round budget for which VERACITY requires the highest overall testing budget (e.g. $pb_7 = 100$ means that the highest overall testing budget is needed when $\mathit{rbudget} = 80000$), and how the testing budgets for other *rbudget* values compare to that (e.g. $pb_1 = 80$ means that the overall testing budget for $\mathit{rbudget} = 1250$ is 75% of the highest overall testing budget). Figure 3.15(b) shows how the mean of these normalised budgets increases from $pb_1 = 75.5\%$ for $\mathit{rbudget} = 1250$ to $pb_7 = 97\%$ for $\mathit{rbudget} = 80000$. The variability of the budget values is very large across the 10 verification scenarios from our experiments, except for the largest round budget $\mathit{rbudget} = 80000$, which indicates that this round budget is consistently too large across the majority of the scenarios.

While the effects of using very large *rbuget* values are clear in Figure 3.15, noticing the effects of small *rbuget* values requires a more careful analysis of the experimental results. A first observation we can make is that the experiments with the smallest *rbudget* values of 1250, 2500 and 5000 used the largest number of verification rounds (as expected, see Figure 3.15a) without delivering smaller mean overall testing budgets than the experiments for *rbudget* = 10000 (see Figure 3.15b). In fact, the numerical results show a very slight decrease in the mean overall testing budgets from 75.7% for *rbudget* = 1250 to 75.5% for *rbudget* = 2500, 75.47% for *rbudget* = 5000 and 75.05% for *rbudget* = 10000. Thus, very small *rbudget* values increase the cost of the component testing without enabling VERACITY to adapt its partition of the round budget more effectively.

The second undesirable effect of using small *rbudget* values is visible in Figure 3.16, which depicts the end-to-end verification times for each of the five TAS verification scenarios and each of the five WebApp verification scenarios we used for the experiments described in this section. As shown by the logarithmic-scale graphs from this figure, the VERACITY execution times approximately double each time the round budget is halved from *rbudget* = 10000 to *rbudget* = 5000, to *rbudget* = 2500 and, finally, to *rbudget* = 1250. Even when VERACITY is used at design time and execution times of close to 30 minutes (for *rbudget* = 1250) are acceptable, the results from Figure 3.15b show that such long execution times yield no benefit, so very small *rbudget* are not recommended.

3.6 Related work

The term uncertainty in software analysis and modelling has been greatly studied during the past years, resulting in various kinds of uncertainty now broadly acknowledged [101–105]. Studies such as [101, 105–107] provide multiple definitions of uncertainty, which were categorized based on source (e.g. data and model structure) or nature (epis-



(a) TAS

(b) WebApp

Figure 3.16: Effect of varying the round budget on the VERACITY execution time (experiments carried out on a c5.2xlarge Windows Server 2019 Amazon EC2 instance with 3.00GHz Intel(R) Xeon(R) Platinum 8124M CPU, and 16 GB of memory).

temic or aleatory) level (e.g. statistical, recognised). When dealing with the verification of performance, reliability, or other non-functional requirements, the term uncertainty is often described as aleatory or epistemic uncertainty. The aleatory variability of parameters and indices is typically captured using stochastic modelling notations, while the epistemic uncertainty, which refers to the behavior of system portions that are intrinsically unknown, requires ad-hoc methods.

The common goal of these approaches is to introduce analysis methodologies able to produce satisfactory results even in presence of such type of uncertainty. For example, [108–110] propose methods to be implemented at software design time to explore the software compositions that satisfy the analysed properties. They used probability distribution functions to model epistemic uncertainty and then evaluated the software system's robustness under uncertainty.

More recently, [111] focuses on understanding the influence of configuration options on performance and proposes an approach based on probabilistic programming that explicitly models uncertainty for option influences and provides both a scalar and a confidence interval for each prediction of a configuration's performance. [112] presents a method -based on closed mathematical formulas- for incorporating and evaluating epistemic uncertainty of the input parameters of queueing models. A similar approach to the study of uncertainty propagation in reliability models has been presented in [113].

A different philosophy in dealing with uncertainty involves the adoption of self-adaptive systems. The amount of research that considers uncertainty in self-adaptive systems has continuously grown over the last few years. For instance, the works [114–116] on adaptive systems applied probabilistic models for reasoning about the changes. Works in [117, 118] make use of self-adaptation to cope with uncertainty. [117] proposes a combination of adaptation and evolution of software to make its behavior resilient to uncertainty, which in turn entails that the software system is *sustainable*, while [118] focuses on the uncertainty surrounding the execution of cyber-physical production systems. A different approach can be found in [119], where a control-theoretic approach is adopted to handle uncertainty in self-adaptive software systems. Furthermore, the need for software systems to operate well under the existing uncertainties is among the main waves that have pushed the research on self-adaptive systems [120], although a *perpetual assurance* of goal satisfaction in self-adaptive systems is still an open research challenge [121]. Most of these works consider uncertainty in the decision-making process and proposes adaptation approaches that are able to guarantee the quality requirements under different and (possibly) unknown types of changes.

Our work lies in the area of the reduction of parametric epistemic uncertainty and introduces an adaptive uncertainty reduction heuristic for performance and dependability software engineering. The proposed heuristic is integrated into a new iterative approach that exploits the adoption of formal verification with confidence intervals.

One of the key aspects of the proposed approach consists of the identification of the system component for which additional data -to be obtained through testing- are needed.

The selection of components to be tested in each iteration is based on a combination of factors that include the sensitivity of the model to variations in the parameters of different components, and the overheads of unit-testing each of these components. Reducing the cost of the (reliability) testing phase by selecting key components to test is a topic that has been analysed in the literature. For example, [122] tackles the question "When to stop testing" by focusing on reliability and discussing the challenges and the potentials related to existing software reliability models. Classical approaches in this domain are based on operational profile [123], however operational profile is often unknown and subject to changes. To overcome this problem, [124] proposes an adapting testing schema that iteratively learns from test execution results as they become available, and, based on them, allocates test cases to the most sensible parts. The assessment is then performed adopting a second sampling strategy that provides the interval estimate of the reliability computed during testing. A different approach that focuses on the allocation of testing resources under uncertain conditions is presented in [125]. A multi-objective debug-aware and robust optimization problem under uncertainty of data is proposed that allows the evaluation of alternative trade-offs among reliability, cost, and release time.

3.7 Summary

We presented VERACITY, a tool-supported approach for the efficient verification of nonfunctional requirements under uncertainty. VERACITY operates by acquiring information about the components of the verified system through testing them individually over a number of verification rounds. A user-defined testing budget specifies the amount of testing performed in each round, and the partition of this budget among system components is adapted from one round to the next in order complete the verification process with a low overall testing cost. The heuristic used to compute this adaptive partition considers factors such as the sensitivity of the verified requirements to the parameters associated with different components, and the different cost (e.g. time, price or risk) of testing these components. The evaluation of VERACITY in case studies from the areas of service-based systems and web applications showed that, on average, it significantly reduces the overall testing cost required to complete the verification process compared to uniformly the testing budget across all system components. This result confirms part 1 of our hypothesis from Section 1.2.

Chapter 4

Efficient formal verification with confidence intervals

4.1 Introduction

Over the years, quantitative verification has been a powerful means of analysing the performance, reliability, and other nonfunctional properties of systems. However, the analysed system should be modelled carefully and accurately as a Markov model in order to obtain a precise verification result. Building a Markov model for the system is a time-consuming task because it requires determining the system's states and the transitions between them, as well as the probabilities of these transitions. Establishing the precise probabilities of transition is challenging [25] since probabilities can only be estimated, with error margins, from run-time observations of the system, from system logs, or based on input obtained from domain experts using a point estimate. Therefore, errors in the estimation of probabilities could be cummulated by quantitative verification, and can produce inaccurate outcomes due to the non-linearity of Markovian models [75]. Formal verification with confidence intervals (FACT) [25] resolves this limitation by providing a confidence interval for the verification result rather than a single value.

FACT is a probabilistic model checker that calculates confidence intervals for properties of parametric Markov chains that have observations for unknown transition probabilities. The current FACT version invokes PRISM [59] to get the algebraic expression for the targeted property of a parametric discrete-time Markov chain (pDTMC) model. However, for many nontrivial models (i.e. models with more than a few states, transitions and parameters), the algebraic expression is too large or too complex for FACT to analyse successfully (i.e. the tool fails with an out of memory or timeout error). Thus, FACT does not scale well to nontrivial pDTMCs.

To extend the capability of FACT, this chapter introduces *e*fficient *F*ormal verific*A*tion with *C*onfidence in*T*ervals (eFACT), a new model checker that is able to calculate such confidence intervals for nontrivial pDTMC models. eFACT exploits efficient parametric model checking (ePMC) [17, 114], which uses domain-specific modelling patterns in order to produce sets of closed-form subexpressions of the analysed properties, and uses these subexpressions as terms in the main formula for the analysis of the whole pDTMC model. ePMC derives an abstraction model from the original pDTMC. The abstraction model consists of fragments, and each fragment represents a single state in the abstraction model and a subset of states in the original model. The main formula is the abstraction model's algebraic expression, and the component formula is a formula related to the fragment. In this way, eFACT computes the confidence intervals for each closed-form expression, and then uses the obtained results to calculate the confidence interval for the main formula. Furthermore, eFACT exploits a binary search technique to enable engineers to efficiently obtain the highest confidence level at which a nonfunctional requirement of a system can be confirmed as violated or satisfied—an important feature unavailable in the FACT tool.

The objectives of this chapter are: (i) to integrate ePMC with FACT, (ii) to augment the resulting solution through adding binary search to eFACT, and (iii) to perform experiments within two case studies in order to evaluate eFACT and compare it to FACT. The chapter is organised as follows. Section 4.2 explains how eFACT computes the confidence intervals for the properties of pDTMC models, then demonstrates the use of binary search to find the required confidence level efficiently. Section 4.3 describes the case studies used to evaluate eFACT. Next, Section 4.4 discusses the experimental results, and Section 4.5 compares our solution to other related work. Finally, Section 4.6 briefly summarises this work.

4.2 Efficient formal verification with confidence intervals

4.2.1 Computing confidence intervals for large pDTMC

To compute confidence intervals for a given pDTMC model and property, FACT obtains algebraic expressions from PRISM and completes its process until confidence intervals are produced. However, when the model is nontrivial (as explained in the previous section), FACT cannot produce confidence intervals. eFACT aims to analyse nontrivial pDTMC models with at least one unknown transition probability, provided that observations of unknown transitions exist. To achieve this purpose, we exploit a recent advance in probabilistic model checking ePMC that produces closed-form expressions (i.e. component formulae) for the property being analysed, and then combines them into one main formula. eFACT analyses each expression separately to produce its confidence intervals for the provided confidence levels. The confidence intervals of the expression then substitute into the main formula. Therefore the outcomes of all expressions contribute to calculating the confidence intervals for the analysed property of a given pDTMC.

At a given confidence level α, computing confidence intervals for a large pDTMC model consists of three main components: confidence interval quantitative verification, ePMC [114], and substitution. The confidence interval quantitative verification is used

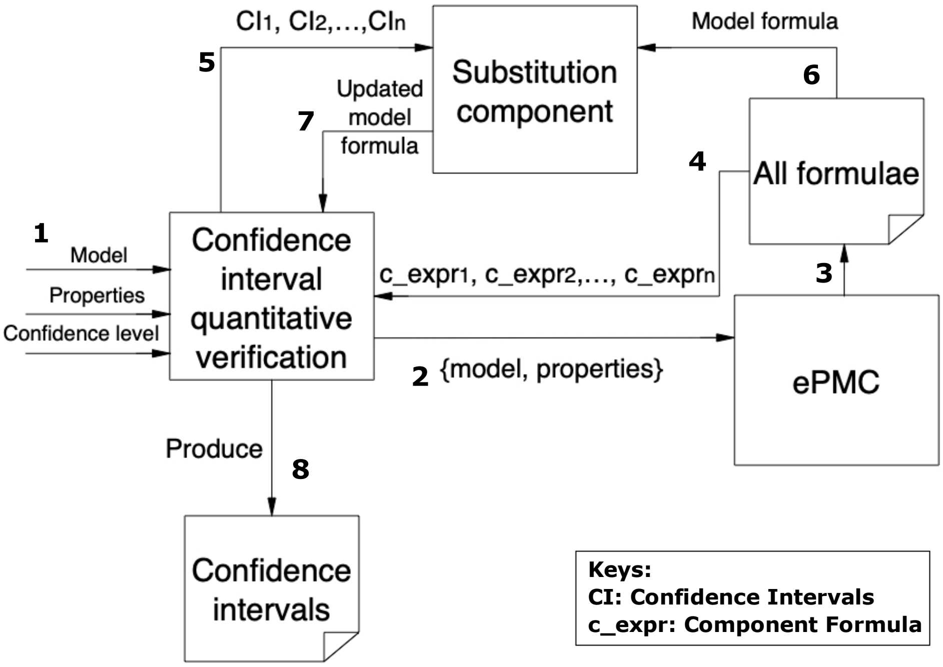
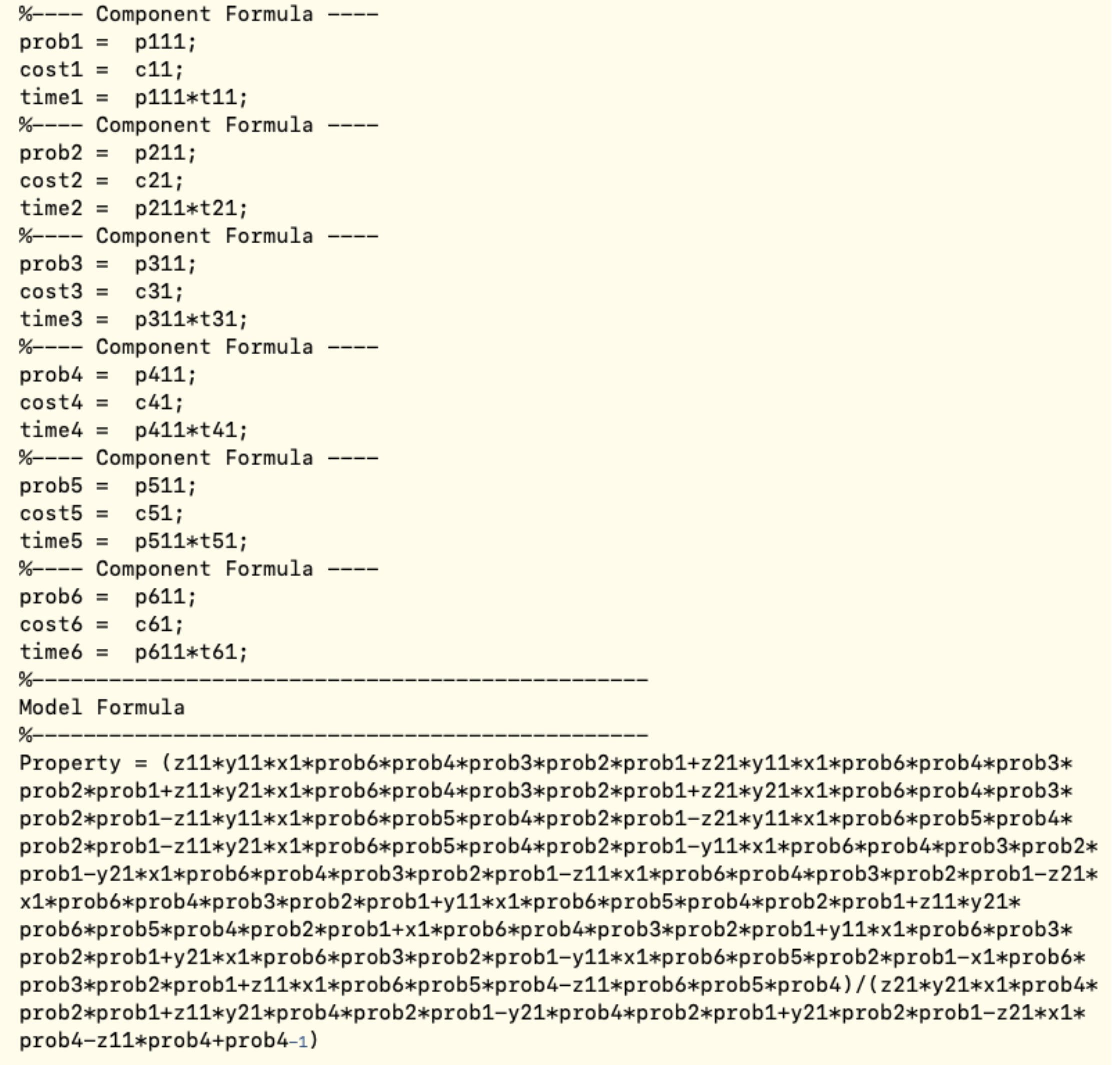


Figure 4.1: eFACT structure

to compute the confidence intervals for each fragment. ePMC is employed to decompose the model. The substitution component is used to substitute all fragment outcomes into the main formula of the original model.

Figure 4.1 illustrates in detail the steps followed to compute the confidence intervals for a nontrivial pDTMC. First, the confidence interval quantitative verification will receive the pDTMC model, property and a range of confidence intervals as inputs. Following this, the model and property are sent to ePMC to produce all possible formulae (Steps 2 and 3 in the figure). There are two kinds of produced formulae: the component formula (a closed-form expression) and the model formula. In the latter, the component formula is a part of the model formula. In general, the formula represents an algebraic expression related to the analysed property of the model. Figure 4.2 shows an example of the two types of formulae. After that, the component formulae (denoted as c_expr_1,c_expr_2,...,c_expr_n in the figure) are sent to confidence interval quantitative verification to compute their confidence intervals sequentially (Step 4). The outcomes



```
%---- Component Formula ----
prob1 =  p111;
cost1 =  c11;
time1 =  p111*t11;
%---- Component Formula ----
prob2 =  p211;
cost2 =  c21;
time2 =  p211*t21;
%---- Component Formula ----
prob3 =  p311;
cost3 =  c31;
time3 =  p311*t31;
%---- Component Formula ----
prob4 =  p411;
cost4 =  c41;
time4 =  p411*t41;
%---- Component Formula ----
prob5 =  p511;
cost5 =  c51;
time5 =  p511*t51;
%---- Component Formula ----
prob6 =  p611;
cost6 =  c61;
time6 =  p611*t61;
%-----------------------------------------------
Model Formula
%-----------------------------------------------
Property = (z11*y11*x1*prob6*prob4*prob3*prob2*prob1+z21*y11*x1*prob6*prob4*prob3*
prob2*prob1+z11*y21*x1*prob6*prob4*prob3*prob2*prob1+z21*y21*x1*prob6*prob4*prob3*
prob2*prob1-z11*y11*x1*prob6*prob5*prob4*prob2*prob1-z21*y11*x1*prob6*prob5*prob4*
prob2*prob1-z11*y21*x1*prob6*prob5*prob4*prob2*prob1-y11*x1*prob6*prob4*prob3*prob2*
prob1-y21*x1*prob6*prob4*prob3*prob2*prob1-z11*x1*prob6*prob4*prob3*prob2*prob1-z21*
x1*prob6*prob4*prob3*prob2*prob1+y11*x1*prob6*prob5*prob4*prob2*prob1+z11*y21*
prob6*prob5*prob4*prob2*prob1+x1*prob6*prob4*prob3*prob2*prob1+y11*x1*prob6*prob3*
prob2*prob1+y21*x1*prob6*prob3*prob2*prob1-y11*x1*prob6*prob5*prob2*prob1-x1*prob6*
prob3*prob2*prob1+z11*x1*prob6*prob5*prob4-z11*prob6*prob5*prob4)/(z21*y21*x1*prob4*
prob2*prob1+z11*y21*prob4*prob2*prob1-y21*prob4*prob2*prob1+y21*prob2*prob1-z21*x1*
prob4-z11*prob4+prob4-1)
```

Figure 4.2: A simple example of components and model formulae

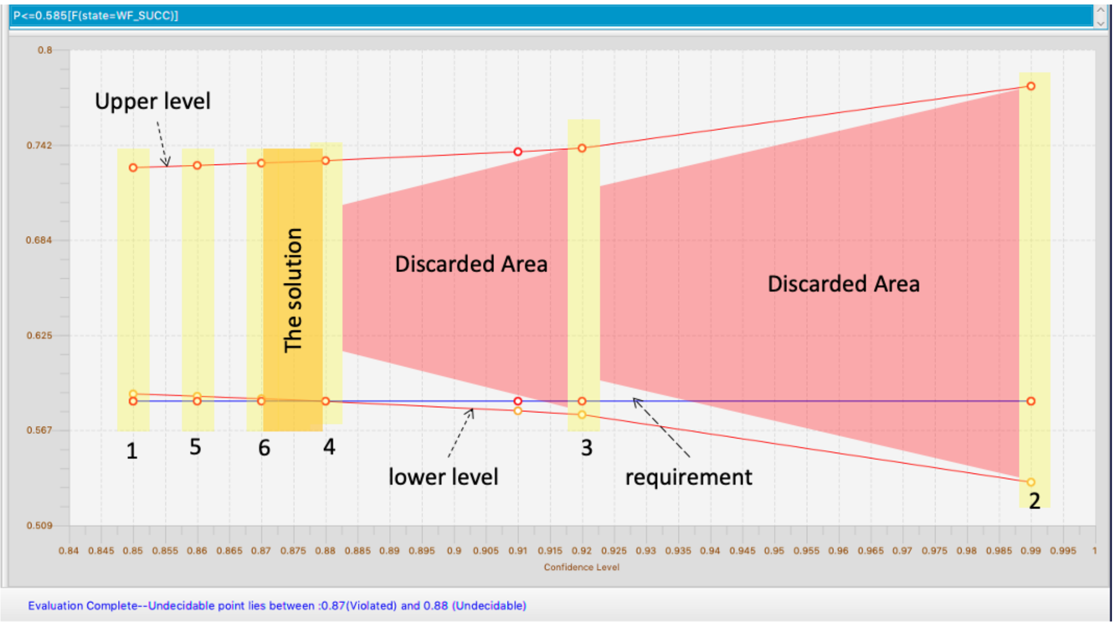
of component formulae are sent to the substitution component to substitute their results into the model formula (as shown in Steps 5 and 6). Finally, the model formula is sent to the confidence interval quantitative verification unit to calculate the final confidence intervals that will appear to the end-user.

4.2.2 Determining the highest confidence level at which a requirement can be verified as satisfied or violated

As described in the previous chapter, when engineers use eFACT to compute confidence intervals for a PCTL-encoded pDTMC property, they are often interested in comparing these intervals with a bound that the property must satisfy as per the analysed system's nonfunctional requirement. Furthermore, they are particularly interested in finding the *highest confidence level* α_{MAX} at which the requirement can be shown as violated or satisfied, given the available set of observations of the unknown pDTMC transitions. For confidence levels $\alpha > \alpha_{MAX}$, the observations available are insufficient to decide whether the requirement is satisfied. Finding the value of α_{MAX} (or a close approximation of it) enables important decision-making. For instance, if a requirement can be shown to be satisfied at the highest confidence level $\alpha_{MAX} = 0.99$, then the system can be confidently deployed (based on the requirement being met). In contrast, if a requirement can only be shown as satisfied at the highest confidence level $\alpha_{MAX} = 0.75$, the decision of whether to deploy the system cannot be made. Further observations should be obtained (e.g. by testing the relevant system components).

eFACT must compute a potentially very large number of confidence intervals at different confidence levels α to find a close approximation of α_{MAX}. eFACT is highly inefficient in achieving this, given the overheads of formal verification with confidence intervals. Therefore, we developed an efficient method (implemented in eFACT) for computing this close approximation. This method employs a binary search to efficiently approximate the highest confidence level α_{MAX}.

Instead of slowly performing verification of each confidence level to find where the requirement is satisfied or violated, the binary search algorithm will speed up the process of achieving this. When the user inserts the model (with its nonfunctional requirement and range of confidence levels), eFACT starts its work by verifying the first inserted confidence level and computing its confidence intervals. Following this, it moves to the



P<=0.585[F(state=WF_SUCC)]

Evaluation Complete--Undecidable point lies between :0.87(Violated) and 0.88 (Undecidable)

Figure 4.3: An example of using binary search in eFACT

last-inserted confidence level to calculate its confidence intervals. Now, there are two confidence levels with their intervals, enabling eFACT to check whether the analysed property requirement is located inside those intervals. If the requirement is located inside all intervals, the process will terminate with a message stated that the requirement is undecidable for the given range of confidence levels. Otherwise, eFACT moves to the middle confidence level (e.g. if the range of confidence levels is between 89 and 99, then the middle level is 94) and computes its confidence intervals. eFACT then checks the requirement's position over the current confidence intervals and compares it with the obtained ones over the confidence intervals from the first and last levels. The confidence levels that lie between the middle confidence level and the other confidence level, in which the requirement's position matches its place in the middle level, will be discarded. Again, eFACT moves to the middle of the remaining confidence levels and repeats the same procedure until it determines the highest confidence level α_{MAX}.

Figure 4.3 shows the result of the verification test, where eFACT is looking for the confidence level at which the requirement is violated or satisfied. The test was conducted between confidence levels 0.85 and 0.99, where the increment step was 0.01. Instead of completing 15 verification tests to find the required confidence level, we performed six verification tests until the required result was found. The discarded area (red area) has a list of confidence levels with intervals containing the requirement; therefore performing additional tests in this area is useless. The solution area (orange area) is where the requirement's position moves from outside the confidence intervals to be inside the next intervals.

The pseudocode for the eFACT binary search is presented in Algorithm 2. Its EFFICIENTFACT method receives five inputs: the pDTMC model ($\mathscr{M}$), the nonfunctional requirement (*req*), the first confidence level (α_{start}), the last confidence level (α_{end}) and the distance between successive confidence levels (*step*). The algorithm begins with the verification of the nonfunctional requirement at confidence level α_{start} (line 2). The result of this verification could have one of three values: satisfied, unsatisfied (violated), or undecidable (cf. Figure 3.4). After that, the verification result at confidence level α_{end} is obtained and compared to the verification result at confidence level α_{start} (line 3). If the results are equal, the process ends with a decision showing that the maximum confidence level for which a conclusive result can be obtained is α_{end}, and returns this value and the verification result (line 4).

If the results are not equal, and the difference between α_{start} and α_{end} is greater than *step* (line 6), the "middle" confidence level (α_{mid}) will be calculated as shown in line 7. In the next line, the algorithm performs verification at confidence level α_{mid}, and if the result is undecidable, the α_{end} value will be updated to the value of α_{mid}. Otherwise, the α_{start} will be updated to be α_{mid}. This process will be repeated until the difference between α_{start} and α_{end} is less that *step*, and thus the condition of the "while" loop becomes false. At this point, the algorithm will return the result (either satisfied or violated) along with the last value of the confidence level α_{start} (line 14).

Algorithm 2 Computing the verification *result* $\in \{\mathsf{SAT}, \mathsf{UNSAT}, \mathsf{UNDECIDABLE}\}$ and—when *result* $\in \{\mathsf{SAT}, \mathsf{UNSAT}\}$—the maximum confidence level for which it is achieved, where the confidence level bounds α_{start} and α_{end} are both multiples of *step*

```
function EfficientFACT($\mathscr{M}, req, \alpha_{\mathsf{start}}, \alpha_{\mathsf{end}}, step$)
    $result \leftarrow$ Verify($\mathscr{M}, req, \alpha_{\mathsf{start}}$)
    if $result =$ Verify($\mathscr{M}, req, \alpha_{\mathsf{end}}$) then
        return $(result, \alpha_{\mathsf{end}})$
    end if
    while $\alpha_{\mathsf{end}} - \alpha_{\mathsf{start}} > step$ do
        $\alpha_{\mathsf{mid}} \leftarrow \alpha_{\mathsf{start}} + \left\lfloor \frac{\alpha_{\mathsf{end}} - \alpha_{\mathsf{start}}}{2 \cdot step} \right\rfloor \cdot step$
        if Verify($\mathscr{M}, req, \alpha_{\mathsf{mid}}$) $= \mathsf{UNDECIDABLE}$ then
            $\alpha_{\mathsf{end}} \leftarrow \alpha_{\mathsf{mid}}$
        else
            $\alpha_{\mathsf{start}} \leftarrow \alpha_{\mathsf{mid}}$
        end if
    end while
    return $(result, \alpha_{\mathsf{start}})$
end function
```

4.3 Case studies

4.3.1 Service-based systems

Service-based systems (SBSs) are applications that provide services dependent on or connected to one another [126]. SBSs comprise internal system components and possible independent third-party components implemented as services. There are different ways in which services can conduct operations similar to those of SBSs, with different probabilities in their execution time (t_1,..,t_n), costs (c_1,...,c_n) and successes (p_1,...,p_n). The following patterns are adopted from [17] and used to implement the SBS operations with n services equivalent to those operations:

1. SEQ $(p_1,t_1,c_1,...,p_n,t_n,c_n)$: There are n services invoked in order, terminated after the last service or upon the first successful request.

2. SEQ-R $(p_1,t_1,c_1,...,p_n,t_n,c_n,r)$: This is similar to SEQ. However, if all service invocations fail, the operation is re-executed from the first service with probability *r*, or it fails with probability 1-*r*.

3. SEQ-R1 $(p_1,t_1,c_1,r_1,...,p_n,t_n,c_n,r_n)$: This is similar to SEQ. However, service *i* will be re-invoked with probability r_i if the invocation of this service fails.

4. PAR$(p_1,t_1,c_1,...,p_n,t_n,c_n)$: There are *n* services invoked simultaneously. The operation will use the output of the first successful invocation.

5. PAR-R $(p_1,t_1,c_1,...,p_n,t_n,c_n,r)$: This is similar to PAR. However, if all service invocations fail, the operation is re-executed with probability *r*, or it fails with probability 1-*r*.

6. PROB $(x_1,p_1,t_1,c_1,...,x_n,p_n,t_n,c_n)$: There is a single service to request. The probability that indicates the service *i* is x_i, where $\Sigma_{i=1}^{n}$ x_i =1.

7. PROB-R$(x_1,p_1,t_1,c_1,...,x_n,p_n,t_n,c_n,r)$: This is similar to PROB. However, if the service invocations fail, the operation is re-executed with probability *r*, or it fails with probability 1-*r*.

8. PROB-R1$(x_1,p_1,t_1,c_1,r_1,...,x_n,p_n,t_n,c_n,r_n)$: This is similar to PROB. However, if the service invocations fail, the service is re-invoked with the probability of *r*, or it fails with probability 1-*r*.

9. Combination: This is a combination of the above patterns.

To evaluate eFACT, a foreign exchange system (FX system) from the SBS area that aims to assists the trader is adopted from [29]. As shown in Figure 4.4, the FX system offers the trader two operational modes: expert or normal. The expert mode executes the trade automatically when the transaction meets the customer's objectives. It begins with the market-watch component to obtain the current price of the chosen currency,

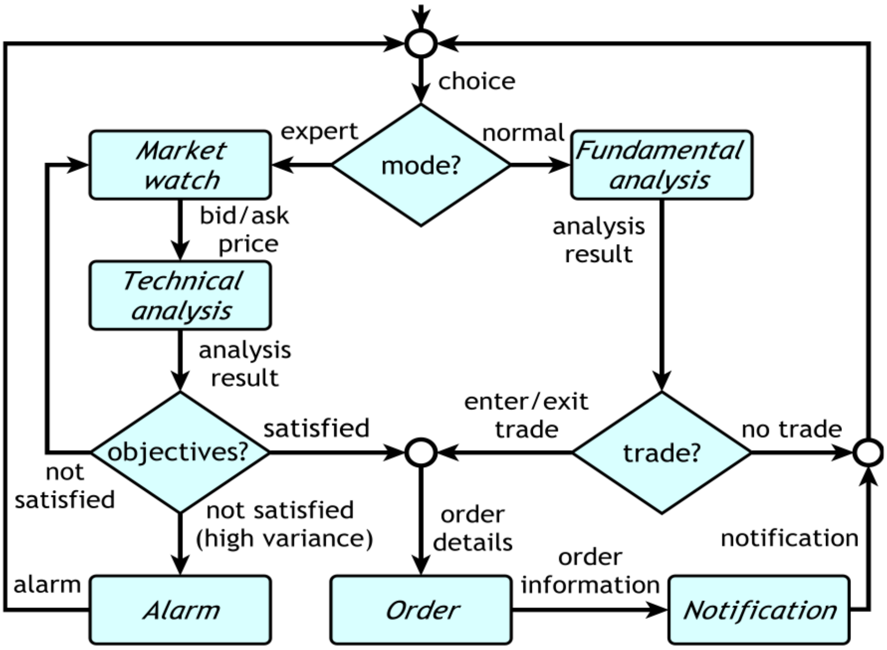


Figure 4.4: Foreign Exchange System Workflow, from [29]

then it uses the technical-analysis component to assess the market and estimate the price movement. The analysed outputs could be one of three options:

1. The transaction can be performed because the objectives that the traders set up are satisfied;
2. The market watch component is re-invoked since the objectives were not met; or
3. The objectives are incorrect, and the Alarm unit will be triggered to warn the trader.

Conversely, the FX system utilises the fundamental-analysis component in its normal mode to determine whether to conduct a transaction, retry the analysis or end the session.

eFACT aims to analyse the following properties of the pDTMC model of the FX system with multiple services (from 1 to 6), and under different patterns:

1. $P1$: The possibility of completing a transaction successfully, written in the PCTL format as $P =? [F(state = WF\_SUCC)]$;

2. $P2$: The estimated time to execute the transaction, written in the PCTL format as $R\{\text{“}time\text{”}\} =? [F((state = WF\_SUCC)|(state = WF\_FAIL))]$; and

3. $P3$: The estimated cost of running the transaction successfully, written in the PCTL format as $R\{\text{“}cost\text{”}\} =? [F((state = WF\_SUCC)|(state = WF\_FAIL))]$.

4.3.2 Three-tier software architectures

The three-tier server [127], as shown in Figure 4.5, provides three services: web, database and application. The services are hosted on four different physical servers (A, B, C, D) and operate on different virtual machines (VMs). The system can be scaled-up to include more servers, VMs and service instances. This case study presents the following three patterns:

1. Basic (B): Several tier instances are running on a server. If the server crashes, the running tier instances are lost.

2. Virtualised (V): There are a number of tier instances, and each one is running on its own virtual machine on a server.

3. Virtualised-M (VM): This is similar to the virtualised pattern. However, when the server crashes, a monitor component can detect the crash before it occurs. Therefore, the virtual machine can be migrated to other running servers.

If the engineers intend to evaluate the reliability of deploying options for the three-tier software on different servers, they could evaluate the following properties:

1. $P1$: This measures the likelihood of the system failing within a determined time due to all tier instances failing. It can be written in PCTL as $P =? [F\ done\ \&\ fail]$; and

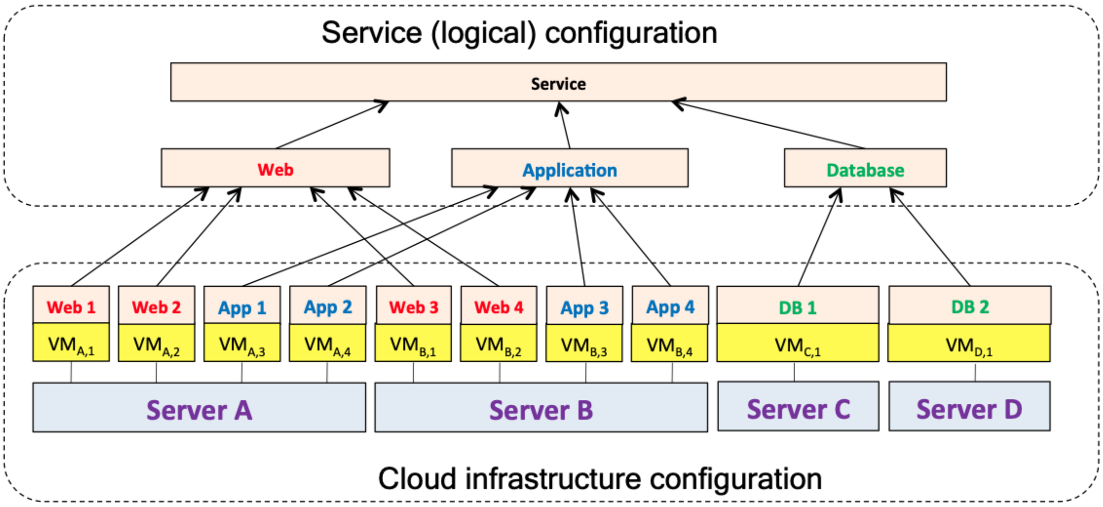


Figure 4.5: Three-tier architecture with three services deployed on Cloud, from [127]

2. $P2$: This assesses the possibility of a single failure point during the analysis. The PCTL encoded for this property is $P = ?[F\ done\ \&\ spf]$.

4.4 Evaluation

We performed a set of experiments to compare eFACT and FACT using two case studies from different areas. These case studies are described in Section 4.3. All experiments were conducted on an OSX 10.14.6 MacBook Pro laptop with 8 GB 1600 MHz DDR3 RAM and CPU 2.5 GHz Intel Core i5 processor.

4.4.1 Experimental environment

eFACT was developed using Java and required installing the following tools and applications:

1. PRISM/Storm are model checkers used to analyse properties and produce algebraic expressions. eFACT was tested using PRISM v4.4 and Storm v1.5.1.

2. MATLAB is used for computing confidence intervals, and the version used was R2019a.

3. YALIMP [128, 129] is a MATLAB-based modelling language that was developed by Johan Lofberg and contains several free and commercial solvers. It is used to model and formulate both convex and non-convex optimisation problems. It is invoked in the background by eFACT/FACT to obtain confidence intervals for the verified requirement by solving a convex optimisation problem. Our experiments were carried out using version 20210331 of YALMIP.

4. Gurobi [130] is an optimisation solver that YALMIP can invoke to solve the optimisation problem.

5. ePMC repository defines the model's patterns and contains the expressions related to the properties of the model.

4.4.2 Results

For the first case study of an SBS (as explained in Section 4.3.1), we carried out several experiments to analyse three properties ($P1$, $P2$, $P3$), to produce the confidence intervals at confidence levels from $\alpha = 0.90$ to $\alpha = 0.99$, and to record the execution time. The analysis was carried out under different patterns and with different services. Table 4.1 summarises the results and shows the execution time (in seconds) taken to analyse each property using eFACT and FACT. The table contains the following symbols:

- (T) denotes a timeout, which means that the execution time exceeded the predefined time of 1800 seconds without completing the analysis.

- (T*) means the tool failed to produce an algebraic expression for the property being analysed during the pre-defined time.

- (-) indicates that we the experiment was skipped because the previous model was smaller than the current one, and it could not be analysed (i.e., the tool failed to compute the confidence intervals in the determined time frame).

As shown in Table 4.1, the execution time recorded for eFACT is better than FACT's execution time, except for the first row, where the model has a single service (SEQ pattern with one service) and the algebraic expressions produced by PRISM for the evaluated properties are simple. We note that to analyse the model from the first row, eFACT requires more time because it needs to compute confidence intervals for the component expressions (more than one expression) before substituting their results into the model formula. Moreover, we notice that the difference is not so significant. The table shows that eFACT took less time than FACT for the analysis of other patterns and services. The last row in the table reports a minimum, maximum, mean and standard deviation of the execution time for twenty models that had different combinations of patterns with a variety of services. The results show that eFACT can produce confidence intervals for all properties of the evaluated models, whereas FACT cannot.

In general, FACT fails to compute the confidence intervals because PRISM can not produce the algebraic expressions of these models, or produces expressions that are too large for the computation of confidence intervals to succeed. In contrast, eFACT invokes ePMC to obtain multiple simpler expressions, evaluates each such expression separately, and substitute the results of these multiple evaluations in the main formula (which is also simpler than the monolithic formula that FACT operates with).

For the second case study, mentioned in Section 4.3.2, several experiments were performed to evaluate two properties ($P1,P2$) and calculate the confidence intervals from α=0.90 to $\alpha = 0.99$. Table 4.2 illustrates the results for four models of four servers using different deployment patterns (D). FACT takes less execution time to analyse the model of deployment D1, which is found in the first row. The model is simple and produces a small expression that FACT can handle. In deployment D2, the model has some complexity (loop), and the expressions for both properties are too large. There-

Table 4.1: The results of FX system, (the execution time is in seconds).

Pattern	Services	eFACT			FACT		
		P1	P2	P3	P1	P2	P3
SEQ	1	141.405	175.357	188.243	98.817	105.328	100.098
	2	147.276	194.089	194.503	T	T	T
	3	188.993	225.028	227.659	-	-	-
	4	286.416	354.003	353.514	-	-	-
SEQ-R	2	197.213	284.967	282.978	T	T	T
	3	253.189	348.103	355.679	-	-	-
	4	1507.257	1314.996	1285.241	-	-	-
SEQ-R1	2	187.587	270.305	272.994	T*	T	T
	3	222.844	322.971	322.634	-	-	-
	4	423.71	501.281	510.492	-	-	-
PAR	2	141.299	189.637	185.0	T	T	T
	3	186.318	269.309	221.306	-	-	-
	4	290.874	384.679	296.634	-	-	-
PAR-R	2	199.079	299.229	280.785	T	T	T
	3	248.545	380.596	337.698	-	-	-
	4	1487.141	1787.865	1352.024	-	-	-
PROB	2	138.287	183.473	182.499	182.595	1130.434	1025.849
	3	143.724	187.631	192.325	T	T	T
	4	148.537	197.401	200.723	-	-	-
PROB-R	2	197.09	262.869	263.637	T	T	T
	3	220.979	294.346	295.803	-	-	-
	4	238.253	339.933	334.826	-	-	-
PROB-R1	2	186.974	262.863	260.842	T	T	T
	3	216.312	293.574	294.891	-	-	-
	4	230.583	337.127	345.044	-	-	-
Combination	Min	155.351	199.017	197.774	T	T	T
	Max	292.297	290.975	286.386	-	-	-
	Mean	177.582	231.562	222.059	-	-	-
	Stdev	30.139	24.23	24.952	-	-	-

fore, FACT failed to analyse them before the allocated time ran out. eFACT is able to handle this model because it deals with small component-level expressions. The third row is for deployment D3, which is a loop-free model. We note that FACT can analyse this model but requires a longer analysis time than eFACT. The last row shows the superiority of eFACT, as FACT cannot analyse the properties of this model because the algebraic expression was not produced within 1800 seconds.

Table 4.2: The results of the multi-tiered system.

D	Number of instances	Server type				eFACT		FACT	
		Server A	Server B	Server C	Server D	P1	P2	P1	P
D1	6	V	V	B	B	203.266	214.387	94.088	86.317
D2	6	VM	VM	B	B	331.342	359.419	T	T
D3	10	V	V	V	V	337.281	365.966	687.554	730.999
D4	10	VM	VM	VM	VM	824.153	873.638	T*	T*

In conclusion, eFACT is useful for analysing a pDTMC model in two situations. First, when PRISM fails to produce the algebraic expression for the analysed model, FACT also fails to produce a result since FACT uses PRISM as a back-end tool to extract the algebraic expression. Second, when there is a time limit for obtaining the confidence intervals, PRISM, MATLAB with YALMIP, or both may take too long to complete their computations due to a large algebraic expression. As a result, FACT may not meet this time limit. In contrast, eFACT has the ability to handle larger models, and thus, it extends the use of quantitative verification of such models to fields in which the usage of estimation error is not acceptable for non-trivial models.

4.5 Related work

Software engineers can exploit probabilistic model checking to analyse and assess the reliability, correctness, potential performance and other key attributes of systems with probabilistic behaviour. However, the model can be affected by the unquantified estimation errors of transition probabilities, leading to uncertainty. Specifically, the probabilities of transitions from one state to another in DTMC could be unrealistic since statistical experiments calculate them. Multiple studies have been conducted to diminish the uncertainty that arises in DTMC models. The studies accomplished by [131, 132] sought to capture this kind of problem. Kozine et al.[131] supposed that the probability value should be included between two bounds (upper and lower) instead of being a specific value. They exploited the theory of interval-valued coherent prevision to generalise discrete Markov chains and introduce interval-valued, discrete-time Markov chains (IDTMCs). Skulj [132] attempted to refine the IDTMCs and develop consecutive steps to make the IDTMCs suitable for models with generic convex sets of probabilities. Benedikt et al. [133] applied upper and lower bounds on the complexity of calculating values for undetermined probabilities in the model checking of an interval Markov chains that increase the likelihood of satisfying ω-regular specification. This work focuses on the pDTMC model, where some of their transitions are unknown but have observations. FACT [25] has the same scope of our work but cannot handle nontrivial models.

4.6 Summary

This chapter has introduced *e*FACT, a new model checker with confidence intervals that utilise ePMC to compute the confidence intervals for nontrivial pDTMCs models. In addition, eFACT can benefit engineers who want to establish the analysed pDTMC model's confidence level in the satisfaction or violation of a given nonfunctional requirement. Our experimental results show that *e*FACT has better execution times than the model checker FACT that it builds on, outperforming FACT in most cases, and therefore confirming part 2 of our hypothesis from Section 1.2. One of our work's limitations is that the model requires a repository of components' equations and an abstract model that needs a domain expert. However, this limitation can be resolved using a recently introduced generic method for efficient parametric model checking [76].

Part III

Synthesis Techniques

Chapter 5

Markov decision process policy synthesis for complex requirement combinations

5.1 Introduction

As described earlier in Section 2.1.2, MDPs are widely utilised to support decision-making in multiple domains, including in the development of software components (e.g. software controllers) for a wide range of systems [134–138]. This chapter presents an approach for synthesising Pareto-optimal MDP policies for software components with nonfunctional requirements that include multiple constraints and optimisation objectives. The rest of the chapter is organised as follows. Section 5.2 defines the problem and describes the proposed approach for the synthesis, and the implementation steps are provided in Section 5.3. Section 5.4 describes the MDP benchmarks we employed to evaluate the approach. Finally, we discuss the related work in Section 5.5 and conclude the chapter with a brief summary in Section 5.6.

5.2 MDP policy synthesis approach

5.2.1 Problem definition

The nonfunctional requirements of a software system are often comprised of a combination of $n \geq 0$ *constraints* and $m \geq 0$ optimisation objectives. Constraints are nonfunctional requirements with the format from Chapter 3, namely:

$$prop_i \bowtie_i bound_i, \tag{5.1}$$

where $\bowtie \in \{<,>,=,\leq,\geq\}$ and $i = 1,2,\ldots n$. Optimisation objectives are nonfunctional requirements of the form

$$M\, prop_j \tag{5.2}$$

where $M \in \{$minimise, maximise$\}$, and $j = 1,2,\ldots m$.

When MDPs are used to model a software system's behaviour under development, the MDP actions often correspond to the system's alternative design options. In this case, synthesising MDP policies corresponds to identifying designs that meet the n constraints and m optimisation objectives for the system (i.e. the system's nonfunctional requirements). With a single optimisation objective, this synthesis produces one such design. Meanwhile, with multiple optimisation objectives, a Pareto-optimal set of policies is synthesised.

Modern probabilistic model checkers, such as PRISM, Storm and others, support this process for simple combinations of PCTL-encoded requirements. In particular, PRISM supports a simple combination of requirements. This is very useful but cannot support MDP policies' synthesis for the more complex combinations of requirements encountered with many software systems. For instance, to find a Pareto curve for two-objective properties using PRISM, the user should select the right maximum value of iterations and/or correct numerical method. If this is not done, an error message will appear asking to change the configurations. PRISM will consume non-negligible time to test each inserted configuration before the error message pops up.

Thus, the problem to be solved is the synthesis that the Pareto-optimal MDP policy sets for systems with nonfunctional requirements comprising:

1. $m \geq 2$ optimisation objectives;

2. $n \geq 0$ constraints; and

3. the expected rewards that must be encoded using eventually operator (F) PCTL properties.

5.2.2 Search-based software engineering approach to MDP policy synthesis

In this section, we explain our approach in detail. The purpose of our approach to maintain the quantitative properties format used for a single-objective verification in MDP and make them usable for multi-objective verification upon the user's selection. Therefore, we propose a transformation approach that can convert a given MDP model into a parametric discrete-time Markov chain (pDTMC). Following this, a search-based technique will be conducted with verification to produce a Pareto front for various optimal solutions.

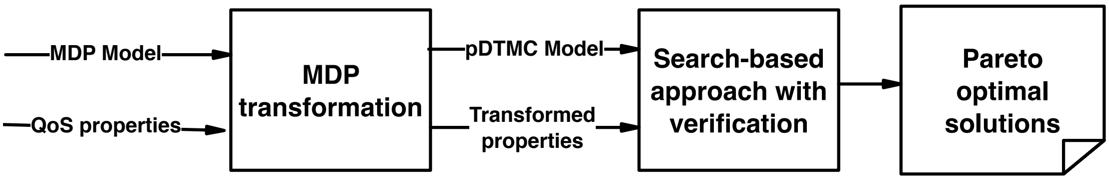


Figure 5.1: Overview of MDP transformation approach

Figure 5.1 depicts the main components of the MDP policy synthesis approach. It comprises two main components: MDP transformation and search-based technique with verification (EvoChecker is utilised for this purpose). As shown in this figure, there are two inputs: the MDP model and two or more conflicting QoS properties. For example, one property is for maximising success, and the other is for minimising the cost. These inputs are transformed by MDP transformation into a readable format of both model and properties to be read by EvoChecker to use them as inputs for producing the Pareto-optimal policies or their approximations.

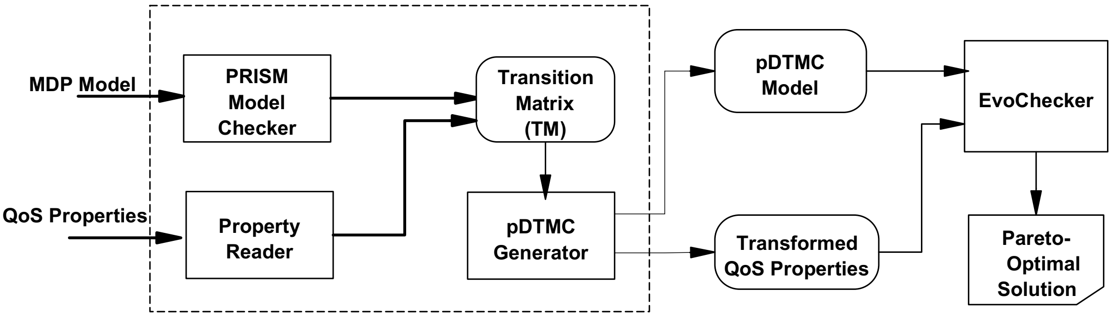


Figure 5.2: Detailed MDP transformation approach

As illustrated in Figure 5.2, our approach requires determining all of the possible *Steps* from an initial state to end states of a given MDP model. In other words, we resolve nondeterministic choices (*Steps*) in which each state in the model is essentially a probabilistic choice over successor states. To do that, we employ the PRISM model checker tool, which receives the MDP model as input to obtain choices in the term of the *transition matrix (TM)*, which contains a finite number of rows and columns. Figure 5.3 shows simple example of an MDP, its *TM*, and its equivalent pDTMC after the transformation process. In this example, PRISM receives the shown MDP model (left side of the figure) and produces its matrix with five columns for states, choices, new states, probabilities and labels. The pDTMC generator can form four primary keys taken from the first two columns (S and C) of this matrix. These primary keys are 00, 01, 10 and 20. Each key constructs a line in the equivalent pDTMC model, which corresponds to lines 5–8 in the figure. For instance, line (1): $[\quad]u = 0\&o1 = 0- > 0.1 : (u' = 1) + 0.9 : (u' = 2)$; is built from the first primary key 00 (taken from the first two rows of the matrix). Line (1) is built as described in the following:

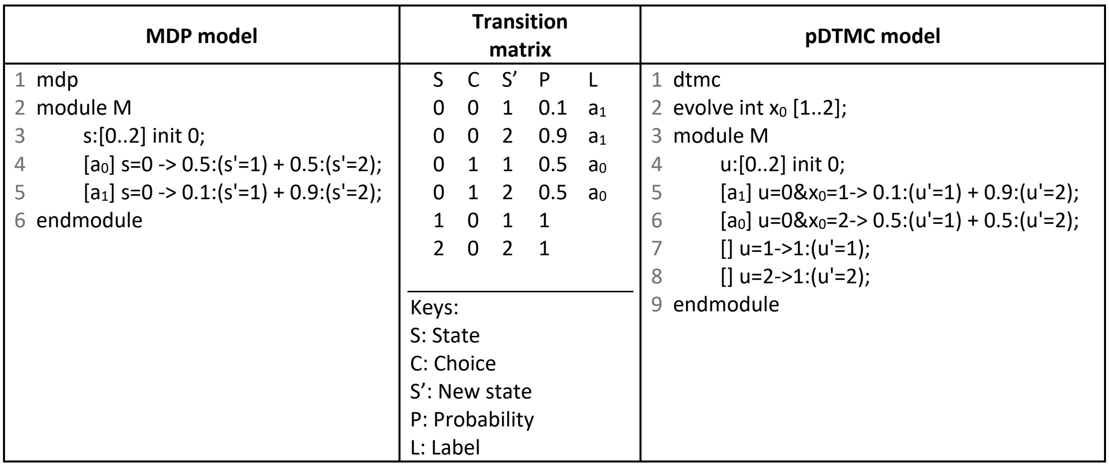
1. u is a symbol representing the name of a state in the equivalent pDTMC model. In this line, u equals 0 (0 is the first digit of the primary key 00).

2. $o1$ is taken from the choice column (can be called option) at the matrix. In this line, $o1$ equals 0 (the second digit of the primary key 00).

3. $0.1 : (u' = 1)$ is taken from the first row of the matrix, which means moving from $s = 0$ to $s' = 1$ with $p = 0.1$. It is reflected in this line to capture the transition probability (0.1) from the current state ($u = 0$) to the new one ($u' = 1$).

4. $0.9 : (u' = 2)$ is taken from the second row of the matrix, which means moving from $s = 0$ to $s' = 2$ with $p = 0.9$. It represents the transition probability (0.9) to reach the new state ($u' = 2$).

The second line in the equivalent pDTMC model (*evolve int* $o1$ [0..1]) represents an EvoChecker statement that means the range of integer values assigned to $o1$ will be 0 or 1. In this example, there is only one option $o1$ because the matrix has only one state for all available choices (i.e. $s = 0\ \forall$ choices: 0 and 1). The fourth line in the equivalent pDTMC model (u : [0..2] *init* 0;) means that we have three states from 0 to 2 taken from the first column in the matrix ($s = \{0, 1, 2\}$).



MDP model	Transition matrix	pDTMC model

mdp
module M
s:[0..2] init 0;
[a_0] s=0 -> 0.5:(s'=1) + 0.5:(s'=2);
[a_1] s=0 -> 0.1:(s'=1) + 0.9:(s'=2);
endmodule
	S C S′ P L
	0 0 1 0.1 a_1
	0 0 2 0.9 a_1
	0 1 1 0.5 a_0
	0 1 2 0.5 a_0
	1 0 1 1
	2 0 2 1
	Keys:
	S: State
	C: Choice
	S′: New state
	P: Probability
	L: Label
		dtmc
		evolve int x_0 [1..2];
		module M
		u:[0..2] init 0;
		[a_1] u=0&x_0=1-> 0.1:(u'=1) + 0.9:(u'=2);
		[a_0] u=0&x_0=2-> 0.5:(u'=1) + 0.5:(u'=2);
		[] u=1->1:(u'=1);
		[] u=2->1:(u'=2);
		endmodule

Figure 5.3: Example of an MDP, its TM and its equivalent pDTMC as used by the MDP policy synthesis approach

Finally, the MDP model properties are updated to replace their original states with matched states in the equivalent pDTMC model. Once the pDTMC and the transformed properties are available, they will be inserted into EvoChecker to explore alternative model designs and generate the Pareto-optimal policies.

5.2.2.1 Algorithm

In this section, we present the algorithm used to transform the MDP model into the equivalent pDTMC. The transformation is performed by the function MDPTOPDTMC from Algorithm 3, which takes as inputs the five elements of the MDP model being transformed. Line 2 goes through each MDP state s and stores the indices of the actions available in this state into a set X_s, as shown in line 3. Next, the for loop in lines 4–6 assembles the probabilities $P(s,s')$ of transitioning from state s to every state $s' \in S$ as a sum parameterised by a variable x_s with value domain X_s. This sum is constructed such that, for any possible value $x_s = i \in X_s$, it reduces to the probability of transition between the same states (i.e., s and s') within the original MDP when action a_i is selected in state s. To this end, the sum uses the ternary-operator expression $(x_s = i)?1 : 0$, which evaluates to 1 if $x_s = i$ or 0 otherwise. The tuple (S, s_0, P, L) corresponding to the assembled pDTMC is returned in line 8. We note that this is a valid pDTMC because, for any state $s \in S$ and any $x_s = i$, we have:

$$\sum_{s' \in S} P(s,s') = \sum_{s' \in S} \Delta(s,a_i,s') = 1.$$

To establish the complexity of our algorithm, we note that the for loop in lines 2–7 is executed once for each MDP state, and the body of the loop first assembles the set X_s in linear, $O(n)$, time and then iterates over each state s' of the MDP in line 5, with each such iteration requiring $O(N)$ operations in the worst case (i.e. when all actions are available in a state). Thus, this $O(N)$ computation from line 5 is performed n^2 times, where n is the number of states for the MDP model, giving an overall worst-case complexity of $O(n^2N)$.

Algorithm 3 Algorithm for transforming an MDP (S,s_0,A,Δ,L) with action set $A = \{a_0,a_1,a_2,\ldots,a_N\}$ into a pDTMC (S,s_0,P,L) with parameters $\{x_s \mid s \in S\}$; for any $s \in S$, the parameter x_s can take discrete values in $X_s = \{i \in \{0,1,2,\ldots,N\} \mid \Delta(s,a_i) \neq zero\}$, which represents the set of action indices available in state s.

```
1: function MDPToPDTMC($S,s_0,A,\Delta,L$)
2:     for $s \in S$ do
3:         $X_s = \{i \in \{0,1,2,\ldots,N\} \mid \Delta(s,a_i) \neq zero\}$
4:         for $s' \in S$ do
5:             $P(s,s') = \sum_{i \in X_s}[((x_s = i)?1:0) \cdot \Delta(s,a_i,s')]$
6:         end for
7:     end for
8:     return $(S,s_0,P,L)$
9: end function
```

5.3 Experimental setup

We carried out a set of experiments to evaluate our approach using different MDP models. These experiments were carried out on a Mac mini with the operating system MacOS Catalina, 3.2 GHz 6-Core Intel Core i7 CPU, and 32 GB of 2667 MHz DDR4 memory. Installing the following tools is essential to run the experiments:

1. The PRISM model checker, which is required to get the transition matrix of the MDP model, assists in the transformation process.

2. EvoChecker[1], which is a tool that is built on top of PRISM and that combines verification and optimisation of probabilistic models. The pDTMC produced by the transformation in Algorithm 3 is provided as input to EvoChecker to find the optimal solutions or their approximations.

The experiments presented in this chapter can be reproduced by following the steps below:

1. Run the command

   ```
   java -jar MDPtransformation.jar MDP_model_file MDP_properties_file
   ```

   in a terminal to obtain the pDTMC model and associated property files;

2. Use the produced files from the previous step as inputs to EVoChecker to get the Pareto-optimal policies for the original MDP.

3. Use Python3 and the Panda library[2] to plot the Pareto diagram for the EvoChecker results.

All the required files and case studies, along with detailed instructions to reproduce the results, are available at `https://gitlab.com/nnma500/mdptransformation`.

5.4 Evaluation

We performed several experiments to evaluate our approach using MDP case studies obtained from the PRISM benchmark suite website[3]. The experiments are divided into two

[1] https://www-users.cs.york.ac.uk/simos/EvoChecker/

[2] https://mode.com/python-tutorial/libraries/pandas

[3] www.prismmodelchecker.org/benchmarks/models.php#mdps

Table 5.1: Output of PRISM and MDP transformation method

MDP model	#States	#Transitions	Decision space size	Property	PRISM output	Our approach output
CSMA	1036	1282	2^{16}	Pmin=? [!"collision_max_backoff" U "all_delivered"]	0.875	0.875
				R{"time"}min=? [F "all_delivered"]	66.99	66.99
				Pmax=? [F "all_delivered"]	1.0	1.0
Coin	272	492	2^{128}	Pmax=? [F "finished"]	1.0	1.0
				R{"steps"}max=? [F "finished"]	74.99	74.99
				R{"steps"}min=? [F "finished"]	47.99	47.99
Firewire_abst	611	718	$2^{43} * 3^{20}$	R{"time"}max=? [F "done"]	298.961	280.473
				R{"time"}min=? [F "done"]	135.25	135.25
				R{"rounds"}min=? [F "done"]	1.0	1.0
Zeroconf	670	977	$2^{149} * 3^{4}$	Pmax=? [F (l=4 & ip=1)]	0.001	0.001
				R{"cost"}min=?[F l=4]	9.019	9.02
				R{"cost"}max=?[F l=4]	9.059	9.058

sets: one set to validate the correctness of the MDP transformation and make comparisons between the results from PRISM and our approach. The other set of experiments is to evaluate our approach for combinations of requirements that the model checker PRISM cannot handle.

5.4.1 The first set of experiments

To compare the PRISM verification results for the original MDP models to the results produced by our approach, we used several MDP models whose number of states and transitions did not exceed 5000. We chose this threshold to ensure that the size of the decision space (i.e. the number of combinations of parameter values possible for the pDTMC) could be handled by our approach in a reasonable time since an even larger decision space would be computationally expensive or infeasible because of the iterative nature of the evolutionary approach. The assessment we carried out in [139] shows that software engineering often requires the use of probabilistic models of this size or smaller.

We ran the experiments with a population size of 100 and an evaluation size of 100 for the genetic algorithm used by EvoChecker. Table 5.1 shows the comparison between the verification results of various properties for different MDP models. The first row of the table indicates the model name, the number of states, the number of transitions, the decision space size, the property and the obtained results in the last two columns. The results show that our approach produces similar results to PRISM. Our approach deals with the problem as a search-based problem, as it employs EvoChecker (which uses evolutionary algorithms) to find the results. Therefore, getting identical results to PRISM is not always guaranteed. We notice that in a small decision space, such as the CSMA model, the results are identical. That means the approach has enough opportunity to explore the decision space. However, with a large decision space, obtaining identical results is not guaranteed (e.g. firewire_abst), and this is because the approach does not have enough chance to explore the whole solution space and needs more iterations and more time. In general, the results of our approach are close to the results produced by PRISM.

5.4.2 The second set of experiments

The second set of experiments were conducted to produce Pareto-optimal policies for requirement sets with two and three optimisation objectives. These experiments are described below.

CSMA/CD MDP benchmark

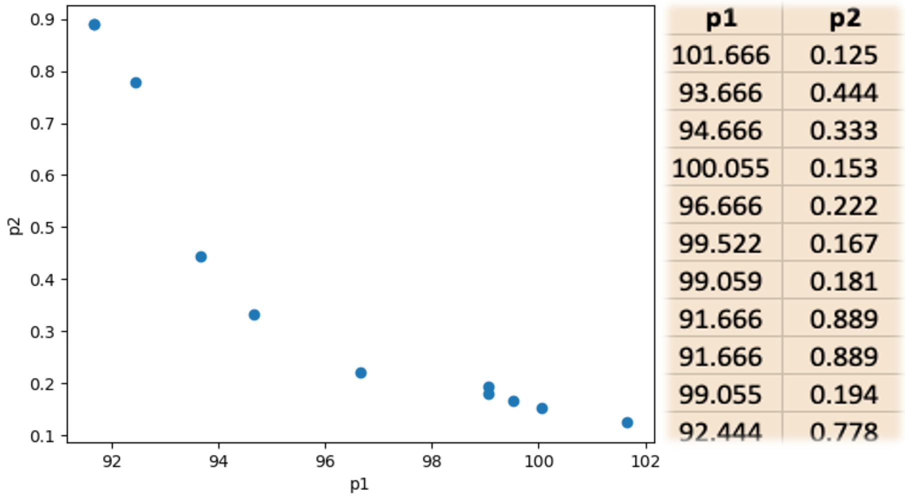
Carrier Sense Multiple Access/Collision Detection (CSMA/CD) is a protocol devised to mitigate data collisions in the traffic sent by multiple stations within a network. It enforces a pause at each station for a period before sending the data again [140]. We performed experiments to obtain the optimal solution for two conflicting properties that need to be optimised. Figure 5.4 depicts the Pareto front for the optimal policies for the CSMA/CD MDP benchmark. The following are the properties that were analysed:

1. P1 – minimise the expected time for all messages to be sent:

   $R\{\text{“}time\text{”}\}min=?\,[F\ \text{“}all\_delivered\text{”}]$

2. P2 – minimisr collision until all messages are delivered:

$$Pmin = ?\,[!\,``collision\_max\_backoff"\ U\ ``all\_delivered"]$$

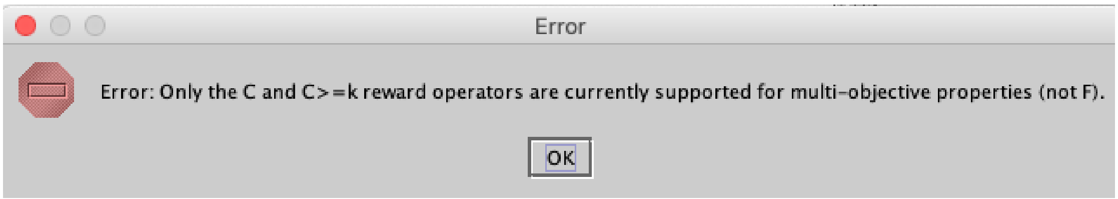


p1	p2
101.666	0.125
93.666	0.444
94.666	0.333
100.055	0.153
96.666	0.222
99.522	0.167
99.059	0.181
91.666	0.889
91.666	0.889
99.055	0.194
92.444	0.778

Figure 5.4: Pareto front optimal policies for CSMA/CD MDP benchmark

Compared to PRISM, our approach produces the Pareto front curve, while PRISM cannot. The error message produced by PRISM when we attempt to obtain the multi-objective for this benchmark's properties is shown in Figure 5.5. PRISM uses the *multi* operator to compute the multi-objective of the two properties, as shown by:

$$multi(R\{``time"\}min = ?\,[F\ ``all\_delivered"],$$
$$Pmin = ?\,[``collision\_max\_backoff"\ U\ ``all\_delivered"])$$



Error: Only the C and C>=k reward operators are currently supported for multi-objective properties (not F).

Figure 5.5: Error message in PRISM when attempting to compute multi-objectives

WLAN MDP benchmark

IEEE 802.11 Wireless LAN (WLAN) is used to link devices via high-frequency radio waves. The WLAN MDP models the competition between two devices to send data simultaneously over the same channel [141]. Figure 5.6 illustrates Pareto-optimal policies for three properties as optimisation objectives and two properties as constraints related to the WLAN MDP model benchmark, which cannot be obtained by PRISM. The evaluated properties are:

1. P1: maximise the expected number of collisions before the two stations send their messages correctly. The minimum expected number is always equal to zero. The property is written in the PCTL format as:

   $R\{\text{``}collisions\text{''}\}max =?\,[F\ s1 = 12\ \&\ s2 = 12]$

2. P2: minimise the expected time for both stations to send their messages correctly.

   $R\{\text{``}time\text{''}\}min =?\,[F\ s1 = 12\ |\ s2 = 12]$

3. P3: minimise the expected cost for both stations to send their messages correctly.

   $R\{\text{``}cost\text{''}\}min =?\,[F\ s1 = 12\ \&\ s2 = 12]$

4. P4: the maximum expected number of collisions must be less than or equal to 3.0. The property is written in the PCTL format as:

   $R\{\text{``}collisions\text{''}\}max <= 3.0[F\ s1 = 12\ \&\ s2 = 12]$

5. P5: with probability greater than or equal to one, the two stations send their messages correctly. The property is written in the PCTL format as:

   $P >= 1.0[F\ s1 = 12\ \&\ s2 = 12]$

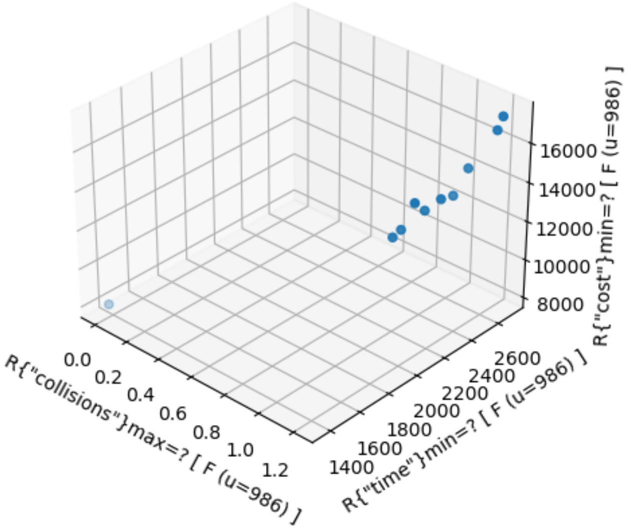


Figure 5.6: Pareto front associated with the optimal policies for the WLAN MDP benchmark

Team formation protocol model

This represents a model that captures the collaboration protocol for a multi-agent system [142]. Figure 5.7 shows the Pareto-optimal policies for three properties associated with the team formation protocol model. These properties specify conflicting optimisation objectives, and the aim is to achieve optimal trade-offs among them. The properties are as follows:

1. P1: maximise the probability of completing the first task successfully.

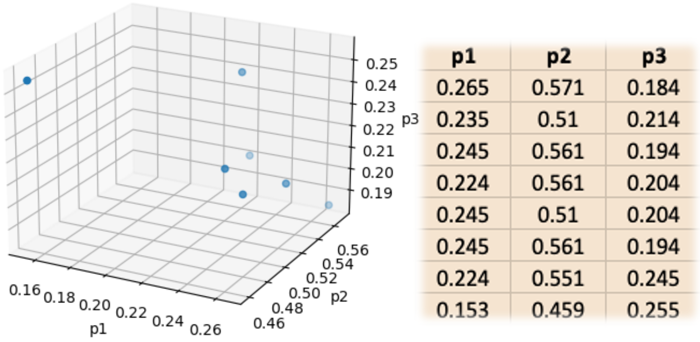
   $Pmax = ?[F\ task1\_completed]$

2. P2: maximise the probability of successful team size until it completes all tasks.

   $R\{"w\_1\_total"\}max = ?[C]$

3. P3: maximise the probability of completing the second task successfully.

   $Pmax = ?[F\ task2\_completed]$

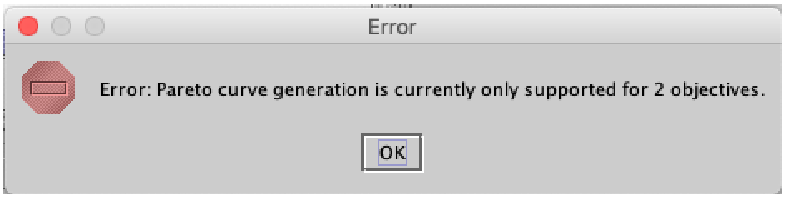


p1	p2	p3
0.265	0.571	0.184
0.235	0.51	0.214
0.245	0.561	0.194
0.224	0.561	0.204
0.245	0.51	0.204
0.245	0.561	0.194
0.224	0.551	0.245
0.153	0.459	0.255

Figure 5.7: Pareto front associated with the optimal policies for the team formation protocol

Generating a Pareto curve for this benchmark and the above properties using the PRISM model checker is not currently possible. The error message shown in Figure 5.8 appeared when we attempted to obtain the Pareto front using the *multi* PRISM operator:

$multi(Pmax =? \ [\ F \ task1\_completed\ ], R"w\_1\_total"max =? \ [\ C\ ], Pmax =? \ [\ F$ $task2\_completed\ ]\ ).$



Error: Pareto curve generation is currently only supported for 2 objectives.

Figure 5.8: PRISM error message

5.5 Related work

The optimisation of multi-objective problems is actively researched in multiple domains, such as artificial intelligence, operation research, and formal methods, involving the MDP. General methods are applied in stochastic systems to discover optimal solutions for conflicting objectives (e.g. colony optimisation [143]; simulated annealing [144]; tabu search [145] and evolutionary algorithms [146]).

Several techniques and approaches have been proposed to perform the verification for MDP properties and analyse them to obtain a set of optimal solutions [147–151]. However, these studies focused on single-objective optimisation.

Several works have been conducted to synthesise the MDP policy and make trade-offs between the analysed properties. Work done by [152] applied payoff value methods on discount reward objectives and reduced the problem into linear programming (LP) to generate the Pareto-optimal policies. Similarly, Brázdil et al. [153] utilised the payoff methods and computed the average of a long run for expected rewards and satisfaction properties to optimise the objectives. Ogryczak et al. [154] provided an approach relying on optimising a weighted ordered, weighted average of objectives to find the optimal policy. Following this, they reformulated this problem by utilising an approach based on LP. Forejt et al. [26] proposed a technique that utilised value iteration to produce Pareto-optimal policies or their approximations. Finally, the recent work in [155] introduced mixed-integer linear programming to obtain the approximation of the Pareto curve for multi-objective requirements of a pure stationary policy with bounded memory. However, this work considers only requirements that have multiple optimisation objectives and does not support constraints. In other words, this work considered unconstrained optimisation problems, while our work supports constrained optimisation problems, which are important in many real-world situations.

5.6 Summary

This chapter presented a new approach for synthesising MDP policies corresponding to software system configurations that satisfy complex combinations of nonfunctional requirements. The requirement combinations supported by the new approach can include both constraint-type requirements and optimisation objectives. Further, many of them cannot be handled by existing probabilistic model checkers. In particular, our new approach to the MDP policy synthesis can generate Pareto-optimal MDP policies for requirement combinations with more than two optimisation objectives. It can also be applied for requirement combinations that include constraints. Neither of these complex combinations of requirements is supported by existing model checkers. This confirms part 3 of our hypothesis from Section 1.2.

Chapter 6

Component synthesis for continuous-time stochastic systems

6.1 Introduction

In the previous chapter, we introduced an approach for the synthesis of MDP policies corresponding to system configurations that satisfy complex combinations of nonfunctional requirements. This approach supports such a synthesis for systems whose stochastic behaviour can be modelled using discrete-time Markov models extended with nondeterminism (i.e. MDPs). However, for many systems of practical importance, time appears explicitly in the nonfunctional requirements, and therefore, it is not possible to use MDPs for their modelling and synthesis. For these systems, the modelling of the timing aspects is important, and this can only be achieved using continuous-time models. In this chapter, we present a method that extends the synthesis approach from the previous chapter to continuous-time Markov models, supporting systems whose timing properties need to be taken into account. Note that these continuous-time models are akin to continuous-time Markov decision processes (CTMDPs) [156, 157]. However, existing probabilistic model checkers do not support CTMDPs. Thus, the method introduced in this chapter encodes a CTMDP as a parametric continuous-time Markov chain (pCTMC) that can then be employed to synthesise the required CTMDP policies either (i) manually, by using a probabilistic model checker, or (ii) in a fully automated way, by using the EvoChecker probabilistic model synthesis tool.

Another new aspect of the method provided in this chapter is its application to the synthesis of: (i) Pareto-optimal configurations for a queue whose requests are handled by a server with dual operating mode, and (ii) a software controller for a realistic human-in-the-loop self-adaptive system. The latter system comes from the autonomous driving domain and was adopted from the existing work in the Safety of Shared Control in Autonomous Driving (SafeSCAD) project[1].

The chapter is organised as follows. Section 6.2 introduces a running example that is used to illustrate our new synthesis method. This is followed by a formal description of CTMDPs in Section 6.3. Section 6.4 presents the synthesis approach, and Section 6.5 describes the utilised case studies. Finally, the chapter concludes with a brief summary in Section 6.6.

6.2 Running example

We will illustrate the new method for the synthesis of CTMDP policies using a simple queue as a running example. As shown in Figure 6.1, this queue consists of a single server that services incoming requests. This system has a state space $q = \{0, 1, \ldots, N\}$, where q denotes the number of requests that are waiting for service or being served, and $N > 0$ is the maximum number of requests that the queue can process. Each request arrives at a $\mu > 0$ rate. Assume the server has two states: ready $(s = 0)$ when the server is free and ready to process a request from the queue, or busy $(s = 1)$ when the server has just serviced a request and needs to perform a clean-up operation before servicing to another request. Each request can be processed by the server with one of two service

[1] `www.york.ac.uk/assuring-autonomy/projects/safe-scad`

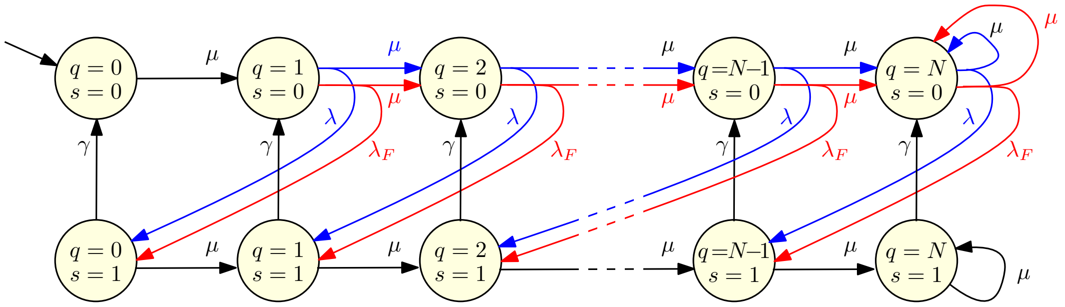


Figure 6.1: CTMDP model of a queue of size $N > 0$ with request arrival rate μ, and a server that can process each request using one of two modes of operation: a "standard" mode with service rate λ and a "fast" mode with service rate $\lambda_F > \lambda$; the server needs to perform a clean-up operation (with rate γ) after each serviced request. Note how two actions (corresponding to the blue and red state transitions) are available in each state when the queue contains $q > 0$ requests, and the server is ready to process a request (i.e. $s = 0$).

rates: $\lambda_F > 0$ for fast mode or $\lambda > 0$ for standard mode. When the server is busy, the arriving requests join the queue and wait until the server becomes available. If the queue is full, the arriving request is dropped from the queue. Finally, we assume that the cost of serving a request using the standard and fast modes of operation is $c_1 > 0$ and $c_2 > c_1$, respectively.

Given this queue, we suppose that the mode of operation (i.e. standard or fast) used by its server for each queue size $q \in \{1, 2, \ldots, N\}$ needs to be determined such that the following two constraints and two optimisation criteria are satisfied:

1. the number of requests dropped within a time period of length T should not exceed a given bound *MaxDropped*,

2. the total cost for serving requests over a time period of length T should not exceed a given bound *MaxCost*,

3. the expected queue length at time T should be minimised, and

4. the total costs for serving requests over a time period of length T should be minimised,

where $T > 0$ is a predefined period of time.

6.3 Continuous-time Markov decision processes

Similar to MDPs, each state of a CTMDP has one or several associated *actions* such that the outgoing transitions and for the state and their rates depend on the action *selected* in that state. We use the following formal definition adapted from [156, 157].

Definition 6.1 A CTMDP is a tuple $M = (S, s_0, A, R)$, where:

- S is a countable set of states;
- $s_0 \in S$ represents the initial state;
- A is a finite set of actions; and
- $R : S \times A \times S \to \mathbb{R}_{\geq 0}$ is a transition rate function such that, for any states $s_i, s_j \in S$ and any action $a \in A$, $R(s_i, a, s_j)$, specifies the rate of transition from state s_i to state s_j when action a is selected in state s_i.

We have $R(s_i, a, s_j) = 0$ when $s_j = s_i$, and $\sum_{s_j \in S} R(s_i, a, s_j) = 0$ if action a is not available in state s_i. Finally, for any action $a \in A$ available in state s_i, the probability that the CTMC transitions leaves state s_i within $t > 0$ time units is given by $1 - e^{-t \cdot \sum_{s_k \in S} R(s_i, a, s_k)}$, and the probability that this transition is to state s_j is given by $R(s_i, a, s_j) / \sum_{s_k \in S} R(s_i, a, s_k)$.

Example 6.1 The queueing system from the running example introduced in Section 6.2 can be modelled as a CTMDP $M = (S, s_0, A, R)$ with:

- state set $S = \{0, 1, \ldots, N\} \times \{0, 1\}$;
- initial state $s_0 = (0, 0)$;
- action set $A = \{\mathsf{standard}, \mathsf{fast}\}$; and

- transition rate function

$$R(s_i,a,s_j)=\begin{cases}\mu, & \text{if } (s_i=(q,0)\wedge s_j=(q+1,0))\vee\\ & \quad (s_i=(q,1)\wedge s_j=(q+1,1)\wedge a=\mathsf{standard})\vee\\ & \quad s_i=s_j=(N,0)\vee(s_i=s_j=(N,1)\wedge a=\mathsf{standard})\\ \lambda, & \text{if } s_i=(q,0)\wedge s_j=(q-1,1)\wedge a=\mathsf{standard}\\ \lambda_F, & \text{if } s_i=(q,0)\wedge s_j=(q-1,1)\wedge a=\mathsf{fast}\\ \gamma, & \text{if } s_i=(q,1)\wedge s_j=(q,0)\\ 0, & \text{otherwise}\end{cases}$$

  for any $s_i,s_j\in S$ and $a\in A$.

To enable the analysis of a broader range of pCTMDP properties, the states and transitions of a CTMDP can be annotated with *rewards*.

Definition 6.2 A *reward structure* over a CTMDP with state set S is a pair of real-valued functions $r^X=(r_1,r_2)$, where $r_1:S\rightarrow\mathbb{R}_{\geq 0}$ is a *state reward function* that determines the rate $r_1(s)$ at which the reward is acquired while the Markov model remains in state s; and $r_2:S\times A\times S\rightarrow\mathbb{R}_{\geq 0}$ is a *transition reward function* that describes the reward $r_2(s_i,s_j)$ obtained each time a transition occurs from state s_i to state s_j following the selection of action $a\in A$ in state s_i.

Example 6.2 Three reward structures need to be defined over the CTMDP from Example 6.1 in order to analyse the properties associated with the two constraints and two optimisation criteria from Section 6.2:

1. A "dropped" requests reward structure $r^{dropped}=(r_1^d,r_2^d)$ where

$$r_1^d(s)=0$$

for all states CTMDP states $s \in S$ and

$$r_2^d(s_i, a, s_j) = \begin{cases} 1, & \text{if } s_i = s_j = (N,0) \vee s_i = s_j = (N,1) \\ 0, & \text{otherwise} \end{cases}$$

for all $s_j, s_j \in S$ and $a \in A$;

2. A "cost" rewards structure $r^{cost} = (r_1^c, r_2^c)$ where

   $$r_1^c(s) = 0$$

   for all states CTMDP states $s \in S$ and

   $$r_2^c(s_i, a, s_j) = \begin{cases} c_1, & \text{if } s_i = (q,0) \wedge s_j = (q-1,1) \wedge a = \text{standard} \\ c_2, & \text{if } s_i = (q,0) \wedge s_j = (q-1,1) \wedge a = \text{fast} \\ 0, & \text{otherwise} \end{cases}$$

   for all $s_j, s_j \in S$ and $a \in A$;

3. a "queue length" rewards structure $r^{length} = (r_1^l, r_2^l)$ where

   $$r_1^l((q,s)) = q$$

   for all $(q,s) \in S$ and

   $$r_2^l(s_i, a, s_j) = 0$$

   for all $s_j, s_j \in S$ and $a \in A$.

As for Markov decision processes, the choice of which action from A to take in every state $s \in S$ of the CTMDP is assumed to be nondeterministic, and reasoning about the behaviour of CTMDPs involves the use of *policies*. A policy resolves the nondeterministic choices of a CTMDP by choosing the action taken in every state. CTMDP policies can be *finite-memory*, *infinite-memory* and *memoryless*. In this chapter, we consider *deterministic memoryless policies*, meaning policies for which the same action is chosen each time when a CTMDP state is reached. We use the term "policy" to refer to this class of CTMDP policies for the remainder the chapter.

Definition 6.3 A (deterministic memoryless) policy of a CTMDP is a function $\sigma : S \to A$ that maps each CTMPD state $s \in S$ to an action from A that is available in state s.

Note that each such policy maps the CTMDP over which it is defined to a standard CTMC with a transition rate matrix defined by $\mathbf{R}(s_i, s_j) = R(s_i, \sigma(s_i), s_j)$ for any states $s_i, s_j \in S$.

Finally, we use *continuous stochastic logic* (CSL) augmented with rewards [14] to express the requirements (including constraints and optimisation objectives) for the CTMDP policies to synthesise. These requirements include bounded and unbounded probabilistic reachability, and several types of reward properties.

The following definition is adopted from [14], as below:

Definition 6.4 The syntax of CSL state-formulae Φ and path-formulae α over an atomic proposition set a is described as

$$\Phi ::= true \mid a \mid \neg\, \Phi \mid \Phi \wedge \Phi \mid P \bowtie_{\rho} [\alpha] \mid S \bowtie_{\rho} [\Phi]$$
$$\alpha ::= X\Phi \mid \Phi \cup^{I} \Phi$$

The cost/reward augmented CSL state formulae are described as

$$R \bowtie_{r} [C^{\leq T}] \mid R \bowtie_{r} [I^{=T}] \mid R \bowtie_{r} [F\Phi] \mid R \bowtie_{r} [S]$$

where:

$\bowtie \in \{<, \leq, \geq, >\}$: is a logical operator;

ρ : is a probability threshold or bound and $\in$ [0, 1];

S: represents the CTMC's steady-state behaviour;

I: is a time interval that belongs to non-negative reals, where $I = [0, \infty)$;

r: a reward constraint; and

T: is a time interval that belongs to non-negative reals.

According to [14, 158], the semantics of CSL over a CTMC model $C = (S, s_0, \mathbf{R}, \mathbf{L})$ is defined as follows:

Definition 6.5 Let s $\in$ S and $\pi \in Paths^C(s)$ in a CTMC model, the satisfaction relation $\models$ can be defined as follows:

s $\models$ true $\forall$ s $\in$ S

s $\models a \Leftrightarrow a \in$ L(s)

s $\models \neg\Phi \Leftrightarrow s \not\models \Phi$

s $\models \Phi_1 \wedge \Phi_2 \Leftrightarrow s \models \Phi_i$ *such that* $i = 1,2$

s $\models P \bowtie_\rho [\alpha] \Leftrightarrow Prob^C(s \models \alpha) \bowtie \rho$

$\pi \models X\Phi \Leftrightarrow \pi[1] \models \Phi$

$\pi \models \Phi_1 \cup^I \Phi_2 \Leftrightarrow \exists t \in I.(\pi@t \models \alpha \wedge (\forall t' \in [0,t).\pi@t' \models \Phi)).$

Example 6.3 The two constraints and two optimisation criteria for the queueing system from our running example can be defined using rewards-extended CSL as follows:

1. $R^{dropped}_{=?}[C^{\leq T}] \leq MaxDropped$

2. $R^{cost}_{=?}[C^{\leq T}] \leq MaxCost$

3. minimise $R^{length}_{=?}[I^{=T}]$

4. minimise $R^{cost}_{=?}[C^{\leq T}]$

6.4 CTMDP policy synthesis approach

Our approach for (deterministic memoryless) CTMDP policy synthesis comprises two steps. The input for the first step is a K-action CTMDP $M = (S, s_0, A, R)$ with $A = \{a_0, a_1, \ldots, a_{K-1}\}$. Using the notation $I_{s_i} = \{k \in \{0, 1, \ldots, K-1\} \mid \sum_{s_j \in S} R(s_i, a_k, s_j) \neq$

$0\}$ to denote the set of indices of the actions available in state $s_i \in S$, this step builds a pCTMC $M' = (S, s_0, \mathbf{R})$ with a transition rate between states $s_i \in S$ and $s_j \in S$ that is given by

$$\mathbf{R}(s_i, s_j) = \sum_{k \in I_{s_i}} equal(x_{s_i}, k) R(s_i, a_k, s_j), \tag{6.1}$$

where $x_{s_i} \in I_{s_i}$ is a parameter, and $equal(a,b) = 1$ if $a = b$ and zero otherwise. Note that fixing the value of each parameter x_{s_i} for the pCTMC M' reduces it to a (non-parametric) CTMC identical to the CTMC obtained for the policy of the original CTMDP that selects action $a_{x_{s_i}}$ for each state $s_i \in S$. This process of obtaining a pCTMC from the original CTMDP is summarised by the function CTMDPtoPCTMC from Algorithm 4. This algorithm takes as input the elements of the CTMDP, and returns an equivalent pCTMC in line 8. This pCTMC has the same state space S and initial state s_0 as the initial CTMDP, and the elements of its transition rate matrix $\mathbf{R}$ are assembled, for each pair of states $s_i, s_j \in S$, in line 5. The sum from this line corresponds to (6.1), and makes use of the set of actions I_{s_i} available in state s_i, where this action set is obtained in line3.

Algorithm 4 CTMDP to pCTMC

```
1: function CTMDPtoPCTMC($S, s_0, A, R$)
2:     for $s_i \in S$ do
3:         $I_{s_i} = \{k \in \{0, 1, \ldots, K-1\} \mid \sum_{s_j \in S} R(s_i, a_k, s_j) \neq 0\}$
4:         for $s_j \in S$ do
5:             $\mathbf{R}(s_i, s_j) = \sum_{k \in I_{s_i}} equal(x_{s_i}, k) R(s_i, a_k, s_j)$
6:         end for
7:     end for
8:     return $(S, s_0, \mathbf{R})$
9: end function
```

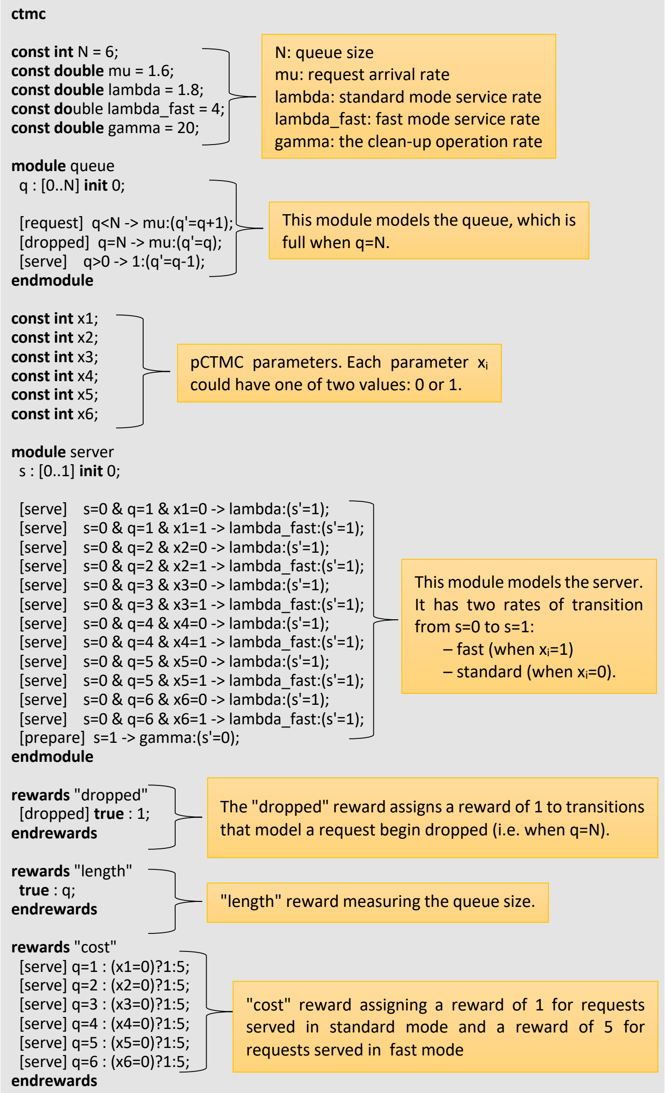
In the second step of our method, we search the parameter space $\bigtimes_{s_i \in S} I_{s_i}$ of the pCTMC from step 1 for combinations of parameter values $\{x_{s_i}\}_{s_i \in S}$, which reduce the pCTMC to non-parametric CTMCs that satisfy a set of CSL-encoded requirements of interest. Two options are available for performing this search. First, the pCTMC can be encoded in the modelling language of the probabilistic model checking PRISM [27] for a manual analysis of the different combinations of parameter values using this model checker. This option is suitable when only a small number of such combinations are possible. Alternatively, the pCTMC can be encoded in the extended PRISM modelling language used by the probabilistic model synthesis tool EvoChecker [159], and the multiobjective genetic algorithm search engine provided by this tool can be used to perform an automated search for parameter value combinations that satisfy the requirements. This option can handle very large search spaces. We illustrate the use of both options in the next section.

6.5 Case studies

To evaluate the effectiveness of the CTMDP policy synthesis approach introduced in the previous section, we applied it for two case studies. The first case study is based on the simple queueing system from our running example. The second case study involves the synthesis of a controller for managing the attentiveness of drivers of vehicles with Level 3 automated driving systems. We present these case studies in the following sections.

6.5.1 CTMDP policy synthesis for the queueing system

For this case study, we considered a version of the queueing system from our running example with the parameter values $N = 6$, $\mu = 1.6$, $\lambda = 1.8$, $\lambda_F = 4$, $\gamma = 20$, $c_1 = 1$ and $c_2 = 5$. Figure 6.2 shows the pCTMC model obtained by applying our approach from the previous section to the CTMDP from Example 6.1. To find the Pareto-optimal policies that satisfy the objectives and constraints mentioned in Section 6.2 and encoded in CSL in Example 6.3, we manually ran PRISM experiments covering all possible policies for the CTMDP, meaning all combinations of $(x_1, x_2, \ldots, x_6) \in \{0,1\}^6$. The size of this search space is $2^6 = 64$ because we have six parameters with two types of service rates (fast or standard). The Pareto front associated with the set of Pareto-optimal policies for



```
ctmc

const int N = 6;
const double mu = 1.6;
const double lambda = 1.8;
const double lambda_fast = 4;
const double gamma = 20;

module queue
  q : [0..N] init 0;

  [request]  q<N -> mu:(q'=q+1);
  [dropped]  q=N -> mu:(q'=q);
  [serve]    q>0 -> 1:(q'=q-1);
endmodule

const int x1;
const int x2;
const int x3;
const int x4;
const int x5;
const int x6;

module server
  s : [0..1] init 0;

  [serve]    s=0 & q=1 & x1=0 -> lambda:(s'=1);
  [serve]    s=0 & q=1 & x1=1 -> lambda_fast:(s'=1);
  [serve]    s=0 & q=2 & x2=0 -> lambda:(s'=1);
  [serve]    s=0 & q=2 & x2=1 -> lambda_fast:(s'=1);
  [serve]    s=0 & q=3 & x3=0 -> lambda:(s'=1);
  [serve]    s=0 & q=3 & x3=1 -> lambda_fast:(s'=1);
  [serve]    s=0 & q=4 & x4=0 -> lambda:(s'=1);
  [serve]    s=0 & q=4 & x4=1 -> lambda_fast:(s'=1);
  [serve]    s=0 & q=5 & x5=0 -> lambda:(s'=1);
  [serve]    s=0 & q=5 & x5=1 -> lambda_fast:(s'=1);
  [serve]    s=0 & q=6 & x6=0 -> lambda:(s'=1);
  [serve]    s=0 & q=6 & x6=1 -> lambda_fast:(s'=1);
  [prepare]  s=1 -> gamma:(s'=0);
endmodule

rewards "dropped"
  [dropped] true : 1;
endrewards

rewards "length"
  true : q;
endrewards

rewards "cost"
  [serve] q=1 : (x1=0)?1:5;
  [serve] q=2 : (x2=0)?1:5;
  [serve] q=3 : (x3=0)?1:5;
  [serve] q=4 : (x4=0)?1:5;
  [serve] q=5 : (x5=0)?1:5;
  [serve] q=6 : (x6=0)?1:5;
endrewards
```

N: queue size
mu: request arrival rate
lambda: standard mode service rate
lambda_fast: fast mode service rate
gamma: the clean-up operation rate

This module models the queue, which is full when q=N.

pCTMC parameters. Each parameter x_i could have one of two values: 0 or 1.

This module models the server. It has two rates of transition from s=0 to s=1:
- fast (when x_i=1)
- standard (when x_i=0).

The "dropped" reward assigns a reward of 1 to transitions that model a request begin dropped (i.e. when q=N).

"length" reward measuring the queue size.

"cost" reward assigning a reward of 1 for requests served in standard mode and a reward of 5 for requests served in fast mode

Figure 6.2: The PRISM-encoded pCTMC model for the queuing system

the queuing system is shown in Figure 6.3. Obtaining this Pareto front through using PRISM to analyse all 64 possible combinations of parameter values took 9.53 seconds on a MacBook Pro computer with a 2.5 GHz Dual-Core Intel Core i5 processor and 8 GB of memory.

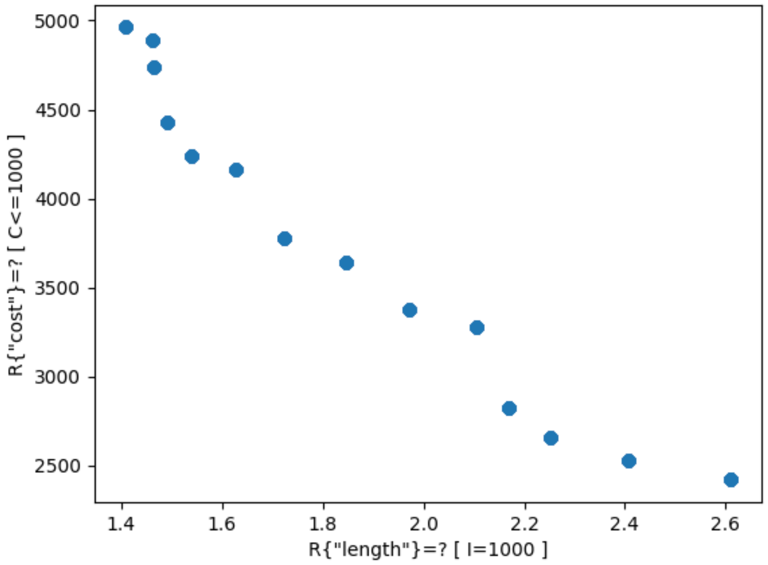


Figure 6.3: Pareto front associated with the set of Pareto-optimal policies for the queuing system

6.5.2 CTMDP policy synthesis for the driver attentiveness management controller

6.5.2.1 Background

Based on the J3016 standard [160], the automated driving system (ADS) is categorised into six levels, ranging from level 0 (no automation) to level 5 (complete automation). Vehicles equipped with ADS that provide partial automation (i.e. level 2) are currently available from manufacturers. At the same time, regulators worldwide are considering permissions for vehicles that offer levels 3 and 4. ADS requires that the driver fasten the seat belt and be attentive enough to respond to commands and share control of the vehicle running at autonomy levels 2 and 3. In addition, at level 4, the driver must intervene in critical situations. The driver must be alert enough to moderate and control the vehicle on time to avoid accidents at all levels. However, it is challenging to keep people mindful when they supervise the operation of automated and autonomous systems.

To improve drivers' attentiveness levels while operating vehicles equipped with ADS, [161] proposed an approach that utilises four phases, monitor–analysis–plan–execute (MAPE), in a control loop. Our case study focuses on the planning phase, where we utilised probabilistic model checking to synthesise the controller of this phase.

The ADS considered in this problem is subject to the automated lane-keeping system (ALKS) regulations imposed by the United Nations. Drivers who fasten their seat belts can activate the ALKS, which once activated can control the car's speed, keep the vehicle in its lane, detect the risk of collision and make an emergency manoeuvre to avoid a collision. In addition, the ALKS regularly issues a transition demand that requires the driver to take control of the vehicle. If there is no response, the system extends the response time and reduces the vehicle speed. If the driver still does not respond, they are deemed inattentive, and the ALKS will stop the vehicle safely. To this end, the ALKS evaluates the driver's attentiveness and availability continually. When the driver is distracted or unavailable, the ADS can use haptic, optical and acoustic means to warn the driver of the transition demand and enhance attentiveness.

6.5.2.2 Problem definition

The ADS from our case study considers $n \geq 2$ levels of driver attentiveness, and can employ two techniques when the driver is insufficiently attentive. The first technique activates one or multiple alerts (out of $m \geq 1$ alerts), and the second reduces the vehicle speed to one of $q \geq 1$ available speed levels. As a result, the ALKS state at any given time is defined by the following components:

1. The level of driver attentiveness $l \in \{0, 1, 2, \ldots, n-1\}$, where the driver is attentive when $l = 0$ and inattentive when $l = n-1$.

2. The active alerts $a = (a_1, a_2, \ldots, a_m) \in \{0, 1\}^m$, where, for all $i = 1, 2, \ldots, m$, $a_i = 1$ if the i-th alert is active and $a_i = 0$ if the i-th alert is inactive.

3. The vehicle speed $v \in \{0, 1, \ldots, q-1\}$, where $v = 0$ corresponds to the vehicle travelling at nominal speed, and $v > 0$ to lower vehicle speeds such that $v = q-1$ is the slowest speed for the car.

Using the notation $L = \{0, 1, \ldots, n-1\}$, $Al = \{0, 1\}^m$ and $V = \{0, 1, \ldots, q-1\}$, the ALKS state space is given by $L \times Al \times V$. Furthermore, consider the following measures described over the ALKS state space:

1. *nuisance* $\in \mathbb{R}_{\geq 0}$: defines the nuisance undergone by the driver due to the active alerts.

2. *progress* $\in \mathbb{R}_{\geq 0}$: represents the trip's progress, which depends on the vehicle's speed.

3. *risk* $\in \mathbb{R}_{\geq 0}$: measures the risk incurred during the trip, which depends on the driver's attentiveness level and the vehicle's speed.

Given this formalisation, the purpose of solving the driver attentiveness management problem is obtaining the set of alerts to use in each ALKS state such that a Pareto-optimal trade-off is achieved between minimising *nuisance*, maximising *progress*, and minimising *risk* over a T-hour journey, where $T > 0$ is a parameter of the problem. A controller that enforces these alerts dependent on the current ALKS state can then be implemented.

6.5.2.3 CTMDP for the driver attentiveness management problem

We formalised the driver attentiveness management problem as a CTMDP policy synthesis problem over the continuous-time MDP $M = (S, s_0, A, R)$ defined by:

- state set $S = L \times Al \times V \times C$, where $C = \{0, 1\}$ is a state component that indicates when it is the controller's turn to select a new combination of alerts to be activated and the new speed to be used (when this component has a value of 1);

- initial state $s_0 = (l_0, a_0, v_0, c_0) = (0, (0, 0, \ldots, 0), 0, 0)$, i.e. the driver is initially attentive ($l_0 = 0$), no alert is switched on initially ($a_0 = (0, 0, \ldots, 0)$), the car starts at nominal speed ($v_0 = 0$), and the controller is not active ($c_0 = 0$);
- action set $A = Al \times V$, so that the selection of an action (a, v) will take the CTMDP from its current state to the state in which the alerts specified by a are activated and the speed v is adopted by the car; and
- transition rate function such that, for any CTMDP states $s_i = (l_i, a_i, v_i, c_i), s_j = (l_j, a_j, v_j, c_j) \in S$ and any action $(a, v) \in Al \times V$, we have

  $$R(s_i, (a, v), s_j) = \begin{cases} r_{l_i l_j a_i}, & \text{if } c_i = 0 \wedge l_j \neq l_i \wedge a_j = a_i \wedge v_j = v_i \wedge c_j = 1 \\ r_c, & \text{if } c_i = 1 \wedge l_j = l_i \wedge (a_j, v_j) = (a, v) \wedge c_j = 0 \\ 0, & \text{otherwise} \end{cases}$$

  where $r_{l_i l_j a_i} > 0$ represents the rate at which the driver attentiveness level changes from l_i to $l_j \neq l_i$ when the active alerts are a_i, and $r_c > 0$ is the (fast) rate at which the controller switches on a new combination of alerts a and/or decides a new speed v for the car.

We note that the rates $r_{l_i l_j a_i} > 0$ must be estimated by, for example, using data sources such as:

1) the numerous available studies and surveys of driver attentiveness (e.g. [162–165]);
2) additional data from controlled experiments with drivers of ALKS vehicles; and
3) driver data collected during the actual driving of ALKS vehicles, such as by using a black-box solution similar to that already employed by many insurers of new drivers [166, 167], either across a fleet of vehicles or for a specific driver.

Using the last data source enables both (i) the definition of personalised controller design spaces for each driver and (ii) the continual updating of these design spaces to support the runtime synthesis of new SafeSCAD controllers when the transition rates for a driver change significantly [168].

The three measures that appear in the problem definition from the previous section (i.e. nuisance, progress and risk) are then defined as reward structures over this CTMDP:

1. A "nuisance" reward structure $r^{nuisance} = (r_1^n, r_2^n)$, where $r_1^n((l,a,v,c)) = nuisance(a) > 0$, and $r_2^n(s_1,s_2) = 0$ for all $s_1,s_2 \in S$.
2. A "progress" reward structure $r^{progress} = (r_1^p, r_2^p)$, where $r_1^p((l,a,v,c)) = progress(v)$, and $r_2^p(s_1,s_2) = 0$ for all $s_1,s_2 \in S$.
3. A "risk" reward structure $r^{risk} = (r_1^r, r_2^r)$, where $r_1^r((l,a,v,c)) = risk(l,v)$, and $r_2^r(s_1,s_2) = 0$ for all $s_1,s_2 \in S$.

Finally, the controllers that we want to obtain correspond to the CTMDP policies that achieve Pareto-optimal trade-offs between the optimisation objectives defined by the following CSL reward formulae:

1. minimise $R^{nuisance}_{=?}[C^{\leq T}]$
2. maximise $R^{progress}_{=?}[C^{\leq T}]$
3. minimise $R^{risk}_{=?}[C^{\leq T}]$

where T is the duration of the car journey (in hours).

6.5.2.4 Synthesis of Pareto-optimal controllers

To synthesise Pareto-optimal policies for the CTMDP and optimisation objectives from the previous section, we first built an equivalent pCTMC as described in Section 6.4, and then used the search-based software engineering tool EvoChecker [29, 159], which:

1. obtains the precise values of the three reward properties for any given CTMC from the controller design space using a probabilistic model checker (the tool can be configured to use PRISM [27] or Storm [28]); and

```
// Controller options specifying the next $a\_v \in \{000_{(2)}, 001_{(2)}, \dots, 111_{(2)}\}$ value
evolve int $option_{\mathsf{S},0}$ [0..7]; // when driver is semi-attentive and $a\_v = 000_{(2)}$
...
evolve int $option_{\mathsf{S},7}$ [0..7]; // when driver is semi-attentive and $a\_v = 111_{(2)}$
evolve int $option_{\mathsf{I},0}$ [0..7]; // when driver is inattentive and $a\_v = 000_{(2)}$
...
evolve int $option_{\mathsf{I},7}$ [0..7]; // when driver is inattentive and $a\_v = 111_{(2)}$

module Controller
  $c$ : [0..1] init 0;    // 0 = controller inactive, 1 = controller active
  $a\_v$ : [0..7] init 0; // current alerts $a$ and speed level $v$

  // activate controller
  [driver_change] $c$=0 $\rightarrow$ $(c' = 1)$;           // when driver state changes
  [ ] $c = 0 \wedge l \neq 0 \rightarrow timerRate : (c' = 1)$; // periodically if driver not attentive

  // switch off alerts and use nominal speed if the driver is attentive ($l = 0$)
  [ ] $c = 1 \wedge l = 0 \rightarrow controllerRate : (a\_v' = 0)\&(c' = 0)$;

  // controller actions for driver attentiveness level $l = 1$ (semi-attentive)
  [ ] $c = 1 \wedge l = 1 \wedge a\_v = 0 \rightarrow controllerRate : (a\_v' = option_{\mathsf{S},0})\&(c' = 0)$;
  ...
  [ ] $c = 1 \wedge l = 1 \wedge a\_v = 7 \rightarrow controllerRate : (a\_v' = option_{\mathsf{S},7})\&(c' = 0)$;

  // controller actions for driver attentiveness level $l = 2$ (inattentive)
  [ ] $c = 1 \wedge l = 2 \wedge a\_v = 0 \rightarrow controllerRate : (a\_v' = option_{\mathsf{I},0})\&(c' = 0)$;
  ...
  [ ] $c = 1 \wedge l = 2 \wedge a\_v = 7 \rightarrow controllerRate : (a\_v' = option_{\mathsf{I},7})\&(c' = 0)$;
endmodule
```

Figure 6.4: Fragment of EvoChecker-encoded controller design space for $m = 2$ independent alerts and $q = 2$ speed levels

2. synthesises a close approximation of the Pareto-optimal set of alert-speed combinations by using a multi-objective genetic algorithm (MOGA) optimisation (the tool can be configured to work with any of the NGSA-II [77], SPEA2 [169] or MOCell [170] MOGAs).

To this end, we supplied EvoChecker with: (i) the pCTMC obtained from the CTMDP defined in the previous section and encoded in the high-level PRISM modelling language [27] extended with EvoChecker constructs (these constructs are used to specify the possible values for the CTMC parameters), and (ii) the three CSL reward prop-

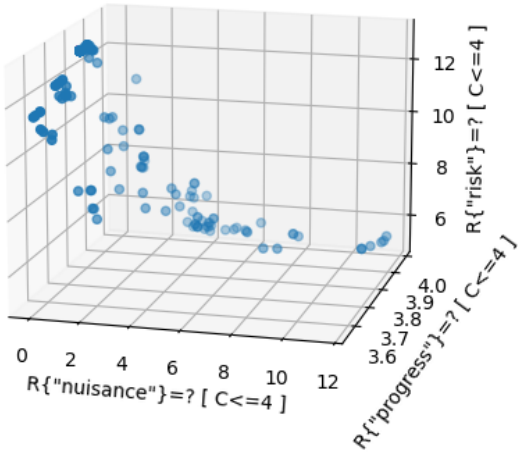


Figure 6.5: Pareto front associated with the set of Pareto-optimal SafeSCAD controllers for the controller design space instance, as synthesised in 98.58 s by EvoChecker configured to use PRISM [27] and NSGA-II [77] (population size 7000 × 1000 iterations) and running on a 3.6 GHz Intel Core i3 Mac OSX 10.14.6 Mac mini computer with 16 GB of memory

erties specifying the optimisation objectives from our problem.

Figure 6.4 shows how the controller design space is expressed in this encoding. Only a fragment of the encoding is shown, but we have made the entire encoding (and all artifacts from this section) available for inspection at `https://www.cs.york.ac.uk/tasp/SafeSCAD/SEAMS21`. Given the controller design space and the optimisation objectives, EvoChecker synthesises a close approximation of the Pareto-optimal set of SafeSCAD controllers, as well as the Pareto front associated with this set. Figure 6.5 shows the Pareto front obtained for the instance of the driver attentiveness management problem with $n = 3$ levels of driver attentiveness, $m = 2$ alerts and $q = 2$ speed levels, along with a driving time of $T = 4$ hours. Each element of this Pareto front corresponds to a controller variant whose nuisance, risk and progress values from Figure 6.5 were obtained by EvoChecker through formal verification using the Storm model checker.

6.6 Summary

This chapter presented a synthesis approach for systems in which timing aspects need to be considered, as time occurs explicitly in their nonfunctional requirements. These systems can be modelled using CTMDPs. However, the current model checkers do not support this type of Markov model. Thus, our method employs paramteric CTMCs to encode CTMDPs, and then synthesises the Pareto policies either manually using PRISM or in a fully automated manner using Evochecker. We used two case studies to produce the Pareto-optimal associated with them, confirming part 4 of our hypothesis from Section 1.2. The first is a simple queueing system with two rates, while the second is a software controller from the self-adaptive area.

Part IV

Conclusion and Future Research Directions

Chapter 7

Conclusion

Probabilistic and parametric model checking are widely adopted to verify software systems in various domains. However, current techniques have several limitations, including those mentioned in Section 1.2. This chapter provides a brief summary and discussion of the four contributions laid out in this thesis that were developed to alleviate these limitations.

7.1 The VERACITY verification approach

The VERACITY verification introduced in Chapter 3 enables the efficient and accurate verification of nonfunctional requirements under epistemic parameter uncertainty. VERACITY integrates confidence interval quantitative verification with a new *adaptive uncertainty reduction* heuristic that collects additional data about the parameters of the verified model by unit-testing specific system components over a series of verification iterations. VERACITY supports the quantitative verification of Markov chains, deciding the components tested in each iteration based on factors that include the sensitivity of the model to variations in the parameters of different components, and the overheads (e.g. time or cost) of unit-testing each of these components.

To evaluate the VERACITY method, we carried out an extensive set of experiments with different scenarios (cf. Section 3.5.2) to assess its effectiveness and usefulness, and we applied this method to two case studies from different fields to confirm its generality. In most experimental cases, VERACITY was successful at effectively reducing the overall total testing cost compared to the baseline method. Furthermore, the experimental results showed that VERACITY is particularly effective at completing the verification results with lower testing costs when confidence levels are high (e.g. 0.95 or 0.99). This is particularly useful because in the real world engineers strive to verify a system's nonfunctional requirements with a high degree of confidence before deploying it, as deploying such a system with a low level of confidence in its compliance with requirements constitutes a possible risk. Furthermore, verifying the nonfunctional requirements of a system under certainty at a high confidence level requires a high testing cost, and reducing this cost is beneficial.

7.2 Efficient formal verification with confidence intervals

In Chapter 4, we introduced efficient formal verification with confidence intervals (eFACT). eFACT increases the applicability of probabilistic model checking under parametric uncertainty to larger models than currently handled by FACT. Moreover, eFACT allows engineers to analyse the nonfunctional requirement to specify the highest confidence level α_{MAX} at which the requirement can be verified as satisfied or violated. This feature means that engineers can measure the confidence of their software system over the analysed requirement and, based on that, decide whether to deploy the system. For instance, the system can be deployed with confidence if the nonfunctional requirement analysis has a high level of α_{MAX}.

To evaluate eFACT, we used multiple pDTMC models. The experiments showed that eFACT can handle larger models than FACT, which is useful in two situations: when the PRISM model checker cannot provide an algebraic expression for the evaluated property (because FACT uses PRISM as a back-end to obtain the expression and accomplish the verification), or when there is a time limit imposed for the verification and FACT cannot complete the verification in a timely manner. Thus, the ability of FACT integrated with ePMC to analyse larger models helps broaden the domain of applicability of quantitative verification to include fields where the use of an estimation error is unacceptable for non-trivial models.

7.3 MDP policy synthesis approach

The MDP policy synthesis approach developed in Chapter 5 addresses an important limitation of current MDP synthesis techniques, namely their inability to synthesise Pareto-optimal policies for certain combinations of three or more nonfunctional requirements. The set of requirements can include multiple optimisation objectives, and it could have one or more constraints. We proposed a new approach to synthesise MDP policies corresponding to software system configurations that meet complex combinations of nonfunctional requirements. In particular, our new approach to MDP policy synthesis can generate Pareto-optimal MDP policies for requirement combinations with more than two optimisation objectives, and for requirements combinations that include constraints such as the expected rewards to be encoded using the “eventually” operator PCTL properties, neither of which are supported by existing model checkers.

We first applied our approach to a range of MDP models and requirements that PRISM can handle, in order to compare the results of our approach to the PRISM results, and found them to be similar. We then successfully obtained the Pareto-optimal policies for multiple MDP models and complex combinations of requirements. The policies for these complex requirements cannot be produced by the current leading probabilistic model checker, PRISM.

7.4 Component synthesis for continuous-time stochastic systems

In Chapter 6, we presented a CTMDP policy synthesis approach that addresses an important limitation of current probabilistic model checkers, i.e. their inability to handle the problem of synthesing CTMDP policies. The proposed approach obtains Pareto-optimal policies for complex nonfunctional requirements that correspond to configurations of software systems or software controllers of cyber-physical systems.

We evaluated our approach using two case studies from different application domains. The first case study was based on a simple queueing system with two service rates to synthesise the complex requirements associated with Pareto-optimal policies. The second case study came from a cyber-physical systems aiming to synthesise the system controller that manages driver attentiveness. Our evaluation of both case studies demonstrated that the approach can successfully obtain the Pareto front associated with the optimal policies of those case studies. The results also showed that our approach can handle synthesis problems from different application domains.

Chapter 8

Future research directions

Multiple extensions can be explored and developed to further strengthen the probabilistic model checking verification and synthesis techniques presented in the thesis.

8.1 Verification techniques

Multiple extensions of the VERACITY approach for the verification of nonfunctional requirements under uncertainty are possible, including those summarised below.

A research direction worth exploring is to expand the set of factors underpinning our VERACITY test-budget partitioning heuristic, in order to further improve its efficiency. One such additional factor that may be particularly beneficial is the level of epistemic uncertainty associated with each component: in each round, a larger fraction of the testing budget should be allocated to components with higher levels of epistemic uncertainty, i.e. to those for which fewer observations are already available. We envisage that augmenting our heuristic with this factor will extend the applicability of VERACITY to verification scenarios in which observations about a subset of the system components are already available at the beginning of the verification process (e.g. from previous testing of those components), but additional case studies need to be carried out to validate this hypothesis.

Another important direction of future research for improving the applicability of VERACITY is to consider the scenario in which some of the parameters that the verified requirements depend on are associated with the operational profile of the system, i.e. with parameters whose epistemic uncertainty cannot be lowered by testing the components of the system “at will”. Examples of such parameters include the number of requests received by a web server in one hour, and the probabilities of these requests being of different types. To some extent, VERACITY could handle this scenario by associating such parameters with an “operational profile component” that is assigned an infinite testing cost. Because this “component” will never be tested, the verification problem may be undecidable, in which case VERACITY will (correctly) terminate with a ‘budget exhausted’ outcome. However, this outcome will only be produced after significant testing effort, some of which could be avoidable by noticing—before using all the testing budget—that the operational profile uncertainty renders the verification problem undecidable. It is therefore worthwhile extending VERACITY with the ability to report an ‘undecidable’ outcome (without exhausting the testing budget) in this important verification scenario.

The efficient formal verification with confidence intervals model checking approach described in Chapter 4 can handle larger parametric DTMCs than was previously possible, but requires the use of repositories of domain-specific modelling patterns. Currently, such models are only available for service-based systems and multi-tier software architectures [114]. To extend the applicability of our approach, it can be integrated with the recently introduced generic method for parametric model checking [76].

8.2 Synthesis techniques

The MDP and CTMDP policy synthesis approaches introduced in the thesis allow the generation of *deterministic policies* for complex nonfunctional requirement combinations. However, the use of *randomised policies* (where the action selected in each MDP or CTMDP state is chosen according to a discrete probability distribution over the available actions) can yield better system configurations and software controllers in many scenarios. Further research is required to extend our policy synthesis approaches to support the generation of randomised policies for both types of Markov decision processes.

For the approach for synthesising Pareto-optimal CTMDP corresponding to software system or discrete-event software controllers described in Chapter 6, it is important to fully automate the application of the new method, and to evaluate it in a broader range of case studies and scenarios.

Another research direction worth exploring is the integration of our MDP policy synthesis approach with the safe reinforcement learning method proposed in [171]. Given a very large MDP corresponding to a safety-critical planning or navigation problem, the first step of this method requires the synthesis of *safe abstract policies*, i.e., policies that satisfy a set of strict constraints for a much smaller, *abstract MPD* obtained by eliminating the safety-irrelevant parts of the original MDP. Using our policy synthesis approach for this abstract MDP would result in a safe reinforcement learning solution that can handle combinations of requirements of greater complexity than those supported by the method from [171].

Finally, additional research is required to assess whether the good EvoChecker scalability reported in [159] extends to our MDP and CTMDP policy synthesis approaches. If necessary, one way to improve this scalability is to parallelise the execution of the multi-objective genetic algorithms used by EvoChecker, so that the analyses of different policies is carried out concurrently in every iteration of these algorithms, and the Pareto fronts for each iteration are assembled through combining the results of these separate analyses.

Bibliography

[1] P. A. Hsiung, Y. R. Chen, and Y. H. Lin, “Model checking safety-critical systems using safecharts,” *IEEE Transactions on Computers*, vol. 56, no. 5, pp. 692–705, 2007.

[2] J. C. Knight, “Safety critical systems: challenges and directions,” in *Proceedings of the 24th International Conference on Software Engineering*, pp. 547–550, 2002.

[3] M. Ben-Ari, “A primer on model checking,” *ACM Inroads*, vol. 1, no. 1, pp. 40–47, 2010.

[4] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese, “Model checking large software specifications,” *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 498–520, 1998.

[5] N. G. Leveson, *Safeware: system safety and computers*. ACM, 1995.

[6] V. D’silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[7] E. M. Clarke, T. A. Henzinger, and H. Veith, “Introduction to model checking,” in *Handbook of Model Checking*, pp. 1–26, Springer, 2018.

[8] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT Press, 2008.

[9] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*. MIT Press, 2018.

[10] A. Hartmanns and H. Hermanns, "In the quantitative automata zoo," *Science of Computer Programming*, vol. 112, pp. 3–23, 2015.

[11] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 224–233, 2003.

[12] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 97–105, 2003.

[13] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 341–350, IEEE, 2011.

[14] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation*, pp. 220–270, Springer, 2007.

[15] M. Kwiatkowska, G. Norman, and D. Parker, "Advances and challenges of probabilistic model checking," in *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 1691–1698, IEEE, 2010.

[16] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic model checking in practice: Case studies with prism," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 4, pp. 16–21, 2005.

[17] R. Calinescu, C. A. Paterson, and K. Johnson, "Efficient parametric model checking using domain knowledge," *IEEE Transactions on Software Engineering*, 2019.

[18] E. M. Hahn, H. Hermanns, and L. Zhang, “Probabilistic reachability for parametric Markov models,” *STTT*, vol. 13, no. 1, pp. 3–19, 2011.

[19] M. Kwiatkowska, G. Norman, and D. Parker, “Probabilistic model checking: Advances and applications,” in *Formal System Verification: State-of the-Art and Future Trends* (R. Drechsler, ed.), pp. 73–121, Cham: Springer International Publishing, 2018.

[20] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, “Dynamic QoS management and optimization in service-based systems,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 387–409, 2011.

[21] D. Weyns and R. Calinescu, “Tele assistance: a self-adaptive service-based system examplar,” in *10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 88–92, IEEE Press, 2015.

[22] M.-Y. Huang and Y.-M. Liu, “Model checking software product lines based on feature slicing,” *International Journal of Computational Science and Engineering*, vol. 18, no. 4, pp. 340–348, 2019.

[23] X. Zhao, V. Robu, D. Flynn, F. Dinmohammadi, M. Fisher, and M. Webster, “Probabilistic model checking of robots deployed in extreme environments,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 8066–8074, 2019.

[24] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, and J. Timmis, “Automatic property checking of robotic applications,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3869–3876, IEEE, 2017.

[25] R. Calinescu, K. Johnson, and C. Paterson, “FACT: A probabilistic model checker for formal verification with confidence intervals,” in *International Con-*

ference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 540–546, Springer, 2016.

[26] V. Forejt, M. Kwiatkowska, and D. Parker, “Pareto curves for probabilistic model checking,” in *International Symposium on Automated Technology for Verification and Analysis*, pp. 317–332, Springer, 2012.

[27] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in *Computer Aided Verification*, pp. 585–591, Springer, 2011.

[28] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, “A storm is coming: A modern probabilistic model checker,” in *29th International Conference on Computer Aided Verification (CAV)*, pp. 592–600, 2017.

[29] S. Gerasimou, G. Tamburrelli, and R. Calinescu, “Search-based synthesis of probabilistic models for quality-of-service software engineering,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 319–330, IEEE, 2015.

[30] R. Calinescu, M. Češka, S. Gerasimou, M. Kwiatkowska, and N. Paoletti, “Designing robust software systems through parametric markov chain synthesis,” in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 131–140, IEEE, 2017.

[31] O. Ibe, *Markov processes for stochastic modeling*. Newnes, 2013.

[32] R. Serfozo, “Markov chains,” in *Basics of Applied Stochastic Processes*, pp. 1–98, Berlin, Heidelberg: Springer, 2009. `https://doi.org/10.1007/978-3-540-89332-5_1`.

[33] C. G. Cassandras and S. Lafortune, "Markov chains," in *Introduction to Discrete Event Systems*, pp. 405–464, Cham: Springer, 2021. `https://doi.org/10.1007/978-3-030-72274-6_7`.

[34] C. Paterson, *Observation-enhanced verification of operational processes*. PhD thesis, University of York, 2018.

[35] C. Daws, "Symbolic and parametric model checking of discrete-time Markov chains," in *ICTAC*, pp. 280–294, Springer, 2004.

[36] M. Kwiatkowska, "Model checking for probability and time: from theory to practice," in *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.*, pp. 351–360, IEEE, 2003.

[37] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini, "Validation of web service compositions," *IET Software*, vol. 1, no. 6, pp. 219–232, 2007.

[38] R. Calinescu, K. Johnson, and Y. Rafiq, "Developing self-verifying service-based systems," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 734–737, 2013.

[39] Q. Hu and W. Yue, *Markov decision processes with their applications*, vol. 14. Springer Science & Business Media, 2007.

[40] M. M. Fard and J. Pineau, "Non-deterministic policies in Markovian decision processes," *Journal of Artificial Intelligence Research*, vol. 40, pp. 1–24, 2011.

[41] M. Panfili, A. Pietrabissa, G. Oddi, and V. Suraci, "A lexicographic approach to constrained MDP admission control," *International Journal of Control*, vol. 89, no. 2, pp. 235–247, 2016.

[42] S. Pathak, L. Pulina, and A. Tacchella, "Verification and repair of control policies

for safe reinforcement learning," *Applied Intelligence*, vol. 48, no. 4, pp. 886–908, 2018.

[43] M. A. Alsheikh, D. T. Hoang, D. Niyato, H.-P. Tan, and S. Lin, "Markov decision processes with applications in wireless sensor networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1239–1267, 2015.

[44] O. Alagoz, H. Hsu, A. J. Schaefer, and M. S. Roberts, "Markov decision processes: a tool for sequential decision making under uncertainty," *Medical Decision Making*, vol. 30, no. 4, pp. 474–483, 2010.

[45] A. Farhadi, "CSEP 573: Artificial Intelligence." `https://courses.cs.washington.edu/courses/csep573/14sp/slides/6_MDP.pdf`, 2014. Accessed: 2020-10-11.

[46] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal aspects of computing*, vol. 6, no. 5, pp. 512–535, 1994.

[47] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[48] P. Schnoebelen, "The complexity of temporal logic model checking.," *Advances in modal logic*, vol. 4, no. 393-436, p. 35, 2002.

[49] M. Kwiatkowska, "Quantitative verification: models techniques and tools," in *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pp. 449–458, 2007.

[50] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on Logic of Programs*, pp. 52–71, Springer, 1981.

[51] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *International Symposium on programming*, pp. 337–351, Springer, 1982.

[52] K. Sakib, Z. Tari, and P. Bertok, *Verification of Communication Protocols in Web Services: Model-checking Service Compositions*, vol. 83. John Wiley & Sons, 2013.

[53] S. Merz, "Model checking: A tutorial overview," in *Summer School on Modeling and Verification of Parallel Processes*, pp. 3–38, Springer, 2000.

[54] M. Mohsin, M. U. Sardar, O. Hasan, and Z. Anwar, "Iotriskanalyzer: a probabilistic model checking based framework for formal risk analytics of the internet of things," *IEEE Access*, vol. 5, pp. 5494–5505, 2017.

[55] T. Nipkow *et al.*, "Advances in probabilistic model checking," *Software Safety and Security: Tools for Analysis and Verification*, vol. 33, no. 126, 2012.

[56] H. L. Younes, M. Kwiatkowska, G. Norman, and D. Parker, "Numerical vs. statistical probabilistic model checking," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 216–228, 2006.

[57] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic model checking for systems biology," 2010.

[58] X. Guo and Z. Yang, "Analyzing leader election protocol by probabilistic model checking," in *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pp. 564–567, IEEE, 2016.

[59] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 200–204, Springer, 2002.

[60] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.

[61] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 441–444, Springer, 2006.

[62] M. E. Bakir, M. Gheorghe, S. Konur, and M. Stannett, "Comparative analysis of statistical model checking tools," in *International Conference on Membrane Computing*, pp. 119–135, Springer, 2016.

[63] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, "Modelling with generalized stochastic Petri nets," *ACM SIGMETRICS performance evaluation review*, vol. 26, no. 2, p. 2, 1998.

[64] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Computer science review*, vol. 15, pp. 29–62, 2015.

[65] C. E. Budde, C. Dehnert, E. M. Hahn, A. Hartmanns, S. Junges, and A. Turrini, "Jani: quantitative model and tool interaction," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 151–168, Springer, 2017.

[66] E. M. Hahn, A. Hartmanns, C. Hensel, M. Klauck, J. Klein, J. Křetínskỳ, D. Parker, T. Quatmann, E. Ruijters, and M. Steinmetz, "The 2019 comparison of

tools for the analysis of quantitative formal models," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 69–92, Springer, 2019.

[67] J.-P. Katoen, M. Khattri, and I. Zapreevt, "A Markov reward model checker," in *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pp. 243–244, IEEE, 2005.

[68] S. Gilmore and J. Hillston, "The PEPA workbench: A tool to support a process algebra-based approach to performance modelling," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 353–368, Springer, 1994.

[69] S. Baarir, M. Beccuti, D. Cerotti, M. De Pierro, S. Donatelli, and G. Franceschinis, "The GreatSPN tool: Recent enhancements," *SIGMETRICS Perform. Eval. Rev.*, vol. 36, p. 4–9, mar 2009.

[70] B. Becker, R. Wimmer, R. Pulungan, T. Peikenkamp, S. Johr, H. Hermanns, M. Herbstritt, and E. Bode, "Compositional performability evaluation for statemate," in *Third International Conference on the Quantitative Evaluation of Systems-(QEST'06)*, pp. 167–178, IEEE, 2006.

[71] K. Sen, M. Viswanathan, and G. Agha, "Vesta: A statistical model-checker and analyzer for probabilistic systems," in *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, pp. 251–252, IEEE, 2005.

[72] E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang, "PARAM: A model checker for parametric Markov models," in *Computer Aided Verification*, pp. 660–664, Springer, 2010.

[73] H. Gruber and J. Johannsen, "Optimal lower bounds on regular expression size

using communication complexity," in *International Conference on Foundations of Software Science and Computational Structures*, pp. 273–286, Springer, 2008.

[74] N. Jansen, F. Corzilius, M. Volk, R. Wimmer, E. Ábrahám, J.-P. Katoen, and B. Becker, "Accelerating parametric probabilistic verification," in *QEST*, pp. 404–420, Springer, 2014.

[75] R. Calinescu, C. Ghezzi, K. Johnson, M. Pezzé, Y. Rafiq, and G. Tamburrelli, "Formal verification with confidence intervals to establish quality of service properties of software systems," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 107–125, 2015.

[76] X. Fang, R. Calinescu, S. Gerasimou, and F. Alhwikem, "Fast parametric model checking through model fragmentation," in *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021. To appear.

[77] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[78] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araujo, S. Biffl, J. Cabot, V. Cortellessa, D. Méndez, A. Moreira, H. Muccini, A. Vallecillo, M. Wimmer, V. Amaral, W. Bühm, H. Bruneliere, L. Burgueño, M. Goulão, S. Teufl, and L. Berardinelli, "Dealing with non-functional requirements in model-driven development: A survey," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[79] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*, vol. 5. Springer Science & Business Media, 2012.

[80] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Quality prediction of service compositions through probabilistic model checking," in *International*

Conference on the Quality of Software Architectures, pp. 119–134, Springer, 2008.

[81] I. Krka, L. Golubchik, and N. Medvidovic, "Probabilistic automata for architecture-based reliability assessment," in *ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, pp. 17–24, 2010.

[82] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, pp. 33–42, ACM, 2013.

[83] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, "Modelling with generalized stochastic Petri nets," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, p. 2, Aug. 1998.

[84] D. Perez-Palacin, R. Mirandola, and J. Merseguer, "QoS and energy management with Petri nets: A Self-adaptive framework," *Journal of Systems and Software*, vol. 85, pp. 2796–2811, Dec. 2012.

[85] A. Filieri, C. S. Pasareanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *35th International Conference on Software Engineering (ICSE)*, pp. 622–631, IEEE, 2013.

[86] C. Paterson and R. Calinescu, "Observation-enhanced QoS analysis of component-based systems," *IEEE Transactions on Software Engineering*, vol. 46, no. 05, pp. 526–548, 2020.

[87] A. K. Akobeng, "Confidence intervals and p-values in clinical decision making," *Acta Pædiatrica*, vol. 97, no. 8, pp. 1004–1007, 2008.

[88] B. R. Kirkwood and J. A. Sterne, *Essential medical statistics*. John Wiley & Sons, 2010.

[89] J. R. Benjamin and C. A. Cornell, *Probability, statistics, and decision for civil engineers*. Courier Corporation, 2014.

[90] J. M. Aughenbaugh and C. J. Paredis, “The value of using imprecise probabilities in engineering design,” *Journal of Mechanical Design*, vol. 128, no. 4, pp. 969–979, 2006.

[91] J.-B. du Prel, G. Hommel, B. Röhrig, and M. Blettner, “Confidence interval or p-value? (part 4 of a series on evaluation of scientific publications),” *Deutsches Ärzteblatt International*, vol. 106, no. 19, p. 335, 2009.

[92] M. J. Gardner and D. G. Altman, “Confidence intervals rather than P values: estimation rather than hypothesis testing,” *British Medical Journal*, vol. 292, no. 6522, pp. 746–750, 1986.

[93] M. GROMADA, “mXparser - Math Expressions Parser for JAVA Android C .NET/MONO/Xamarin - Mathematical Formula Parser / Evaluator Library.” `http://mathparser.org`, 2010. Accessed: 2019-08-20.

[94] A. Filieri, C. Ghezzi, and G. Tamburrelli, “A formal approach to adaptive software: continuous assurance of non-functional requirements,” *Formal Aspects of Computing*, vol. 24, no. 2, pp. 163–186, 2012.

[95] A. Fabijan, P. Dmitriev, C. McFarland, L. Vermeer, H. Holmström Olsson, and J. Bosch, “Experimentation growth: Evolving trustworthy A/B testing capabilities in online software companies,” *Journal of Software: Evolution and Process*, vol. 30, no. 12, p. e2113, 2018.

[96] R. Kohavi and R. Longbotham, "Online controlled experiments and A/B testing," *Encyclopedia of Machine Learning and Data Mining*, vol. 7, no. 8, pp. 922–929, 2017.

[97] D. Siroker and P. Koomen, *A/B testing: The most powerful way to turn clicks into customers*. John Wiley & Sons, 2013.

[98] H. Muccini, M. Dias, and D. J. Richardson, "Software architecture-based regression testing," *Journal of Systems and Software*, vol. 79, no. 10, pp. 1379–1396, 2006. Architecting Dependable Systems.

[99] R. Calinescu and S. Kikuchi, "Formal methods@ runtime," in *Monterey Workshop*, pp. 122–135, Springer, 2010.

[100] R. Calinescu and M. Kwiatkowska, "CADS*: Computer-aided development of self-* systems," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 421–424, Springer, 2009.

[101] W. Walker, P. Harremoes, J. Romans, J. van der Sluus, M. van Asselt, P. Janssen, and M. Krauss, "Defining uncertainty. a conceptual basis for uncertainty management in model-based decision support," *Integrated Assessment*, vol. 4, no. 1, pp. 5–17, 2003.

[102] R. Calinescu, R. Mirandola, D. Perez-Palacin, and D. Weyns, "Understanding uncertainty in self-adaptive systems," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 242–251, IEEE Computer Society, 2020.

[103] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng, "A taxonomy of uncertainty for dynamically adaptive systems," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '12, (Piscataway, NJ, USA), pp. 99–108, IEEE Press, 2012.

[104] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke, "Living with uncertainty in the age of runtime models," in *Models@ run. time*, pp. 47–100, Springer, 2014.

[105] D. Perez-Palacin and R. Mirandola, "Uncertainties in the modeling of self-adaptive systems: A taxonomy and an example of availability evaluation," in *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pp. 3–14, ACM, 2014.

[106] D. Garlan, "Software engineering in an uncertain world," in *Future of Software Engineering Research*, ACM, 2010.

[107] N. Esfahani and S. Malek, "Uncertainty in self-adaptive software systems," in *Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol 7475*, Springer, 2013.

[108] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, and L. Grunske, "Model-based performance analysis of software architectures under uncertainty," in *the 9th International ACM SIGSOFT Conference on Quality of Software Architectures*, pp. 69–78, ACM, 2013.

[109] I. Meedeniya, I. Moser, A. Aleti, and L. Grunske, "Architecture-based reliability evaluation under uncertainty," in *the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS*, pp. 85–94, ACM, 2011.

[110] I. Meedeniya, A. Aleti, and L. Grunske, "Architecture-driven reliability optimization with uncertain model parameters," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2340–2355, 2012.

[111] J. Dorn, S. Apel, and N. Siegmund, "Mastering uncertainty in performance estimations of configurable software systems," in *2020 35th IEEE/ACM Inter-*

national Conference on Automated Software Engineering (ASE), pp. 684–696, 2020.

[112] F. Antonelli, V. Cortellessa, M. Gribaudo, R. Pinciroli, K. S. Trivedi, and C. Trubiani, “Analytical modeling of performance indices under epistemic uncertainty applied to cloud computing systems,” *Future Generation Computer Systems*, vol. 102, pp. 746 – 761, 2020.

[113] K. Mishra and K. S. Trivedi, “Closed-form approach for epistemic uncertainty propagation in analytic models,” in *Stochastic Reliability and Maintenance Modeling: Essays in Honor of Professor Shunji Osaki on his 70th Birthday* (T. Dohi and T. Nakagawa, eds.), pp. 315–332, London: Springer London, 2013.

[114] R. Calinescu, K. Johnson, and C. Paterson, “Efficient parametric model checking using domain-specific modelling patterns,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pp. 61–64, IEEE, 2018.

[115] R. Calinescu, M. Autili, J. Cámara, A. Di Marco, S. Gerasimou, P. Inverardi, A. Perucci, N. Jansen, J.-P. Katoen, M. Kwiatkowska, O. J. Mengshoel, R. Spalazzese, and M. Tivoli, “Synthesis and verification of self-aware computing systems,” in *Self-Aware Computing Systems* (S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu, eds.), pp. 337–373, Springer, 2017.

[116] G. A. Moreno, J. Camara, D. Garlan, and B. Schmerl, “Proactive self-adaptation under uncertainty: A probabilistic model checking approach,” in *Foundations of Software Engineering*, ACM, 2015.

[117] D. Weyns, M. Caporuscio, B. Vogel, and A. Kurti, “Design for sustainability = runtime adaptation ∪ evolution,” in *the 2015 European Conference on Software Architecture Workshops*, pp. 62:1–62:7, ACM, 2015.

[118] A. Musil, J. Musil, D. Weyns, T. Bures, H. Muccini, and M. Sharaf, “Patterns for self-adaptation in cyber-physical systems,” in *Multi-Disciplinary Engineering for Cyber-Physical Production Systems*, pp. 331–368, Springer, 2017.

[119] S. Shevtsov, D. Weyns, and M. Maggio, “Simca*: A control-theoretic approach to handle uncertainty in self-adaptive systems with guarantees,” *ACM Trans. Auton. Adapt. Syst.*, vol. 13, jul 2019.

[120] D. Weyns, “Software engineering of self-adaptive systems: an organised tour and future challenges,” *Chapter in Handbook of Software Engineering*, 2017.

[121] D. Weyns, N. Bencomo, R. Calinescu, J. Camara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jezequel, S. Malek, *et al.*, “Perpetual assurances for self-adaptive systems,” in *Software Engineering for Self-Adaptive Systems III. Assurances*, pp. 31–63, Springer, 2017.

[122] M. Garg, R. Lai, and S. J. Huang, “When to stop testing: a study from the perspective of software reliability models,” *IET Software*, vol. 5, pp. 263–273(10), June 2011.

[123] J. D. Musa, “The operational profile,” in *Reliability and Maintenance of Complex Systems* (S. Özekici, ed.), (Berlin, Heidelberg), pp. 333–344, Springer Berlin Heidelberg, 1996.

[124] D. Cotroneo, R. Pietrantuono, and S. Russo, “RELAI testing: A technique to assess and improve software reliability,” *IEEE Trans. Software Eng.*, vol. 42, no. 5, pp. 452–475, 2016.

[125] R. Pietrantuono, P. Potena, A. Pecchia, D. Rodríguez, S. Russo, and L. F. Sanz, “Multiobjective testing resource allocation under uncertainty,” *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 347–362, 2018.

[126] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel, "Sound development of secure service-based systems," in *Proceedings of the 2nd International Conference on Service Oriented Computing*, pp. 115–124, 2004.

[127] R. Calinescu, S. Kikuchi, and K. Johnson, "Compositional reverification of probabilistic safety properties for large-scale complex IT systems," in *Monterey Workshop*, pp. 303–329, Springer, 2012.

[128] J. Löfberg, "Modeling and solving uncertain optimization problems in yalmip," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 1337–1341, 2008.

[129] J. Lofberg, "YALMIP: A toolbox for modeling and optimization in MATLAB," in *International Conference on Robotics and Automation (IEEE Cat. No. 04CH37508)*, pp. 284–289, IEEE, 2004.

[130] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021.

[131] I. O. Kozine and L. V. Utkin, "Interval-valued finite Markov chains," *Reliable computing*, vol. 8, no. 2, pp. 97–113, 2002.

[132] D. Škulj, "Discrete time Markov chains with interval probabilities," *International Journal of Approximate Reasoning*, vol. 50, no. 8, pp. 1314–1329, 2009.

[133] M. Benedikt, R. Lenhardt, and J. Worrell, "LTL model checking of interval Markov chains," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 32–46, Springer, 2013.

[134] E. Altman, *Constrained Markov decision processes*, vol. 7. CRC Press, 1999.

[135] A. Munir and A. Gordon-Ross, "An MDP-based dynamic optimization methodology for wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 4, pp. 616–625, 2011.

[136] A. J. Schaefer, M. D. Bailey, S. M. Shechter, and M. S. Roberts, "Modeling medical treatment using Markov decision processes," in *Operations Research and Health Care*, pp. 593–612, Springer, 2005.

[137] M. Gleirscher and R. Calinescu, "Safety controller synthesis for collaborative robots," in *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 83–92, IEEE, 2020.

[138] M. Gleirscher, R. Calinescu, J. Douthwaite, B. Lesage, C. Paterson, J. Aitken, R. Alexander, and J. Law, "Verified synthesis of optimal safety controllers for human-robot collaboration," *arXiv preprint arXiv:2106.06604*, 2021.

[139] N. Alasmari, R. Calinescu, C. Paterson, and R. Mirandola, "Quantitative verification with adaptive uncertainty reduction," *Journal of Systems and Software*, vol. 188, p. 111275, 2022.

[140] M. Duflot, L. Fribourg, T. Herault, R. Lassaigne, F. Magniette, S. Messika, S. Peyronnet, and C. Picaronny, "Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC," *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 195–214, 2005.

[141] M. Kwiatkowska, G. Norman, and J. Sproston, "Probabilistic model checking of the IEEE 802.11 wireless local area network protocol," in *Joint International Workshop von Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pp. 169–187, Springer, 2002.

[142] T. Chen, M. Kwiatkowska, D. Parker, and A. Simaitis, "Verifying team formation protocols with probabilistic model checking," in *Computational Logic in Multi-Agent Systems* (J. Leite, P. Torroni, T. Ågotnes, G. Boella, and L. van der Torre, eds.), (Berlin, Heidelberg), pp. 190–207, Springer Berlin Heidelberg, 2011.

[143] S. Mirjalili, “Ant colony optimisation,” in *Evolutionary Algorithms and Neural Networks*, pp. 33–42, Springer, 2019.

[144] S. Kirkpatrick, “Optimization by simulated annealing: Quantitative studies,” *Journal of Statistical Physics*, vol. 34, no. 5-6, pp. 975–986, 1984.

[145] F. Glover and M. Laguna, “Tabu search,” in *Handbook of Combinatorial Optimization*, pp. 2093–2229, Springer, 1998.

[146] G. Fleury, P. Lacomme, and C. Prins, “Evolutionary algorithms for stochastic arc routing problems,” in *Workshops on Applications of Evolutionary Computation*, pp. 501–512, Springer, 2004.

[147] E. M. Hahn and A. Hartmanns, “A comparison of time-and reward-bounded probabilistic model checking techniques,” in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 85–100, Springer, 2016.

[148] P. Hou, W. Yeoh, and P. Varakantham, “Revisiting risk-sensitive MDPs: New algorithms and results,” in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 24, 2014.

[149] M. Ummels and C. Baier, “Computing quantiles in Markov reward models,” in *International Conference on Foundations of Software Science and Computational Structures*, pp. 353–368, Springer, 2013.

[150] Z. Cao, H. Guo, J. Zhang, F. Oliehoek, and U. Fastenrath, “Maximizing the probability of arriving on time: A practical q-learning method,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.

[151] A. Christman and J. Cassamano, “Maximizing the probability of arriving on time,” in *International Conference on Analytical and Stochastic Modeling Techniques and Applications*, pp. 142–157, Springer, 2013.

[152] K. Chatterjee, R. Majumdar, and T. A. Henzinger, “Markov decision processes with multiple objectives,” in *Annual symposium on theoretical aspects of computer science*, pp. 325–336, Springer, 2006.

[153] T. Brázdil, V. Brozek, K. Chatterjee, V. Forejt, and A. Kucera, “Two views on multiple mean-payoff objectives in Markov decision processes.,” in *LICS*, pp. 33–42, 2011.

[154] W. Ogryczak, P. Perny, and P. Weng, “A compromise programming approach to multiobjective Markov decision processes,” *International Journal of Information Technology & Decision Making*, vol. 12, no. 05, pp. 1021–1053, 2013.

[155] F. Delgrange, J.-P. Katoen, T. Quatmann, and M. Randour, “Simple strategies in multi-objective MDPs,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 346–364, Springer, 2020.

[156] P. Buchholz, E. M. Hahn, H. Hermanns, and L. Zhang, “Model checking algorithms for CTMDPs,” in *International Conference on Computer Aided Verification*, pp. 225–242, Springer, 2011.

[157] Q. Qiu and M. Pedram, “Dynamic power management based on continuous-time Markov decision processes,” in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, pp. 555–561, 1999.

[158] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, “Model-checking algorithms for continuous-time Markov chains,” *IEEE Transactions on software engineering*, vol. 29, no. 6, pp. 524–541, 2003.

[159] S. Gerasimou, R. Calinescu, and G. Tamburrelli, “Synthesis of probabilistic models for quality-of-service software engineering,” *Automated Software Engineering*, vol. 25, no. 4, pp. 785–831, 2018.

[160] On-Road Automated Driving (ORAD) committee, “Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles,” Standard J3016_201806, SAE International, 2018.

[161] R. Calinescu, N. Alasmari, and M. Gleirscher, “Maintaining driver attentiveness in shared-control autonomous driving,” in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 90–96, IEEE, 2021.

[162] M. Körber, L. Prasch, and K. Bengler, “Why do I have to drive now? Post hoc explanations of takeover requests,” *Human Factors*, vol. 60, no. 3, pp. 305–323, 2018.

[163] A. Lotz and S. Weissenberger, “Predicting take-over times of truck drivers in conditional autonomous driving,” in *International Conference on Applied Human Factors and Ergonomics*, pp. 329–338, 2018.

[164] Q. Maia, M. A. Grandner, *et al.*, “Short and long sleep duration and risk of drowsy driving and the role of subjective sleep insufficiency,” *Accident Analysis & Prevention*, vol. 59, pp. 618–622, 2013.

[165] W. Vanlaar, H. Simpson, D. Mayhew, and R. Robertson, “Fatigued and drowsy driving: A survey of attitudes, opinions and behaviors,” *Journal of Safety Research*, vol. 39, no. 3, pp. 303–309, 2008.

[166] A. Kassem, R. Jabr, G. Salamouni, and Z. K. Maalouf, “Vehicle black box system,” in *2nd IEEE Systems Conference*, pp. 1–6, 2008.

[167] M. A. Kumar, M. V. Suman, Y. Misra, and M. G. Pratyusha, “Intelligent vehicle black box using IoT,” *Int. J. Eng. Technol*, vol. 7, no. 2, pp. 215–218, 2018.

[168] X. Zhao, R. Calinescu, S. Gerasimou, V. Robu, and D. Flynn, “Interval change-point detection for runtime probabilistic model checking,” in *35th IEEE/ACM*

International Conference on Automated Software Engineering (ASE), pp. 163–174, IEEE, 2020.

[169] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength Pareto evolutionary algorithm," *TIK-report*, vol. 103, 2001.

[170] A. J. Nebro, J. J. Durillo, F. Luna, *et al.*, "MOCell: A cellular genetic algorithm for multiobjective optimization," *International Journal of Intelligent Systems*, vol. 24, no. 7, pp. 726–746, 2009.

[171] G. Mason, R. Calinescu, D. Kudenko, and A. Banks, "Assured reinforcement learning with formally verified abstract policies," in *9th International Conference on Agents and Artificial Intelligence (ICAART)*, 2017.