Project Work

Sascha Lehmann

Online Model Checking with UPPAAL SMC

February 4, 2016

supervised by:
Prof. Dr. Sibylle Schupp
M.Sc. Xintao Ma

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Projektarbeit selbstständig verfasst
und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die
Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 04. Februar 2016

Sascha Lehmann
# Contents

1 Introduction 1

2 Conceptual Foundations 3
   2.1 Model Verification 3
   2.2 Model Checking 3
   2.3 Explicit-State / Symbolic Model Checking 5
   2.4 Online Model Checking 6
   2.5 Statistical Model Checking 7
      2.5.1 DTMC (Discrete-Time Markov Chain) and PCTL 8
      2.5.2 CTMC (Continuous-Time Markov Chain) and CSL 9
      2.5.3 NSTA (Network of Stochastic Timed Automata) and MITL 10

3 Existing Solutions 13
   3.1 Available Model Checking Environments 13
      3.1.1 UPPAAL 13
      3.1.2 PRISM 14
   3.2 UPPAAL Solutions 15
      3.2.1 UPPAAL-SMC 15

4 The Statistical Online Model Checking (SOMC) Approach 17
   4.1 Applicable OMC Concepts 17
   4.2 New SOMC Concepts 18
   4.3 Implementation and Integration 23

5 SOMC Experimental Applications 27
   5.1 Proof-of-Concept Sample Models 27
      5.1.1 The Dice Game 27
      5.1.2 The Call Centre 28

6 Evaluation of Analysis Results 31
   6.1 Experiment Results 31
   6.2 Influencing Factors 32
   6.3 Computation Times 36
      6.3.1 Impact of the Query Properties 36
      6.3.2 Impact of the Java Implementation 37
   6.4 SOMC Capabilities and Limitations 38

7 Conclusion 43
## List of Figures

2.1 The basic model checking workflow ........................................... 4  
2.2 A comparison of the static and online verification approaches .......... 7  
2.3 A sample model of a Discrete-Time Markov Chain .......................... 8  
2.4 A sample model of a Continuous-Time Markov Chain ...................... 9  
2.5 A sample model of a Network of Priced Timed Automata ................. 11  
3.1 The (graphical) model editor view of the UPPAAL model checker ........ 13  
3.2 The (code-based) model editor view of the PRISM model checker .......... 14  
3.3 The model verification view of the UPPAAL model checker ............... 15  
4.1 Examples of probabilistic branching constructs ............................ 18  
4.2 Supported model elements for distributions over time .................... 20  
4.3 A basic example for using rates as differential equations ............... 22  
4.4 The parent and child models for dynamic instance spawning .......... 22  
4.5 General structure of the original OMC framework .......................... 23  
4.6 General structure of the modified SOMC framework ....................... 24  
5.1 UPPAAL model of the dice game example .................................. 28  
5.2 UPPAAL model of the call centre example .................................. 29  
6.1 Dice game distribution developing over time ................................ 31  
6.2 Dice game verification results of various queries .......................... 32  
6.3 Call centre rate developing over time ....................................... 32  
6.4 Call centre verification results of various queries ........................ 33  
6.5 Impacts of the data buffer size ............................................... 33  
6.6 Developing of parameters with great value difference .................... 34  
6.7 Impact of different UPPAAL parameters on the verification process .... 35  
6.8 Impact of the probability comparison query (Binary event with varied close probabilities) ........................................ 35  
6.9 Duration times of different SMC queries .................................... 36  
6.10 Duration impact of different query scope values ........................... 37  
6.11 Duration times of data generation and approximation .................... 38  
6.12 Comparison of system and model data ..................................... 39  
6.13 Comparison of real and simulated data distributions (branch point) .... 40  
6.14 Comparison of real and simulated data distributions (exponential rate) .. 40
List of Tables

4.1 Possible query functions with parameters .................................. 25

List of Algorithms

1 Routine for probabilistic branches ............................................ 19

List of Code Snippets

1 Java implementation of the exponential data generation routine ........ 20
2 Java implementation of the parameter approximation routine .......... 21
Abstract – Based on the general principles of a newly developed “Online Model Checking” framework for real-time, interval-based and temporary model verification utilizing the interface with the model checker UPPAAL, we extend the system with statistical functionalities to allow online verification for a broader range of models and weaken the impact of state space explosion. For this purpose, we implement a Java add-on for the OMC framework taking advantage of the integrated Uppaal SMC functions, evaluate its capabilities and finally use it for two experiments including components for both the online model checking and the statistical model checking approach, in order to derive statements about its performance, benefits and limitations in comparison to the original framework. While extending the range of models and components enabled for verification, the project results show that the interpretation of “online” may need to be altered, and the main focus needs to lie on the real-time constraint and - for further research - on the model simulation synchronously to the real system.
1 Introduction

The application of model checking to verify both software and hardware systems and show the satisfiability of propositions derived from given specifications has been a major topic in the process of system engineering for several decades. In the beginning – at the time when many systems were still mostly separated from their surrounding and did not need to react to environment changes extensively as their design was rather single purpose – it was sufficient to analyse a system once at the end of its development phase in order to guarantee its intended behaviour. But over time, systems became more complex and connected, and at the latest with the emergence of cyber-physical systems, they finally came to a point where it was no longer enough to follow this separated and static view. In fact, a static model checking approach could now lead to false negative results, as it supposes specific global constraints to hold forever without adjusting the model parameters to the dynamic and non-deterministic system environment. This fact led to a growing need for a new approach that involves the consideration of unpredictable parameter data and therefore derives only temporarily valid satisfiability statements.

One solution for this problem is called “Online Model Checking” (OMC) and it provides the basis for the following framework additions that we develop in the course of this project. While the OMC approach itself provides the necessary functionality to verify a real-time system for overlapping time intervals and therefore provides sufficient validity information during system observations, it builds up on methods of explicit symbolic model checking for the time-limited paths and may still suffer from state space explosion. In order to overcome some of the resultant restrictions of this approach, we will extend the OMC framework with methods of statistical model checking, and finally combine them to a new statistical online model checking framework (SOMC). This means that instead of deriving logic formulas from the system, we will subsequently simulate the model with multiple runs at given times simultaneously to the observed real-time system, and only estimate the validity of proportions instead of providing a mathematically derived guarantee. Besides lessening the influence of state space explosion in certain cases, the SOMC approach also allows us to incorporate statistical elements like probabilistic branches into the model.

We start this project thesis with a broad overview of conceptual foundations in Chapter 2, which are necessary to understand the SOMC approach, including general explanations of model verification and model checking as well as more detailed information about online model checking and the differences of symbolic and statistical approaches. In Chapter 3, we have a look at already existing solutions of the statistical model checking approach as well as their software environments, and show the potential of an online variant of it. Afterwards we develop the actual statistical online model checking framework in Chapter 4, covering its mathematical background as well as the algorithmic concepts for its implementation. Given the new system, we apply the principles to analyse two different
1 Introduction

proof-of-concept models - an experiment to observe variably loaded dice games and a call centre system with varying agent amounts and call parameters - in Chapter 5. After the application, Chapter 6 provides the evaluation results of the simulation processes and compares the performance, complexity and overall usability of the SOMC approach to the basic OMC and SMC systems. At last, we come to a conclusion in Chapter 7, assessing the general advantages of the new system, and discussing prospects for further development of the SOMC framework as well as the field of online model checking.
2 Conceptual Foundations

We begin this project on statistical online model checking with the most common concepts of the model verification process. This chapter introduces into the field of model verification and its subtopic of model checking with general definitions, logics, and constructs. After that, we get involved into the two more specific concepts of Online Model Checking (OMC) and Statistical Model Checking (SMC), which we want to bring together to a single approach in the subsequent chapters. For a better understanding of the SMC technique, a short explanation of the mostly opposing mechanism of explicit-state and symbolic model checking is provided as well.

2.1 Model Verification

The superordinate discipline of **Verification** covers a large range of techniques to mathematically deduce the correctness of a system or entity by derivation from underlying attributes or axiomatic assumptions. These assumptions may be part of a proof construct - as in the formal verification approach for algorithms - or originate from a given specification during software or model verification. The verification process can either be of static or dynamic nature: The former one - also containing the formal verification technique - is used to analyse the system using logics and mathematical methods, whereas the latter one is realised by embedding and checking the system in a testing environment.

To assure that a model represents a real world system with adequate precision, it is necessary to extract the system-relevant attributes from the real world, and to show that these attributes hold for the model as well. The verification technique applied for that purpose will be explained next.

2.2 Model Checking

**Model checking** describes a technique in the range of model verification that consists of mainly two components: A formal description \( M \) of the model that we want to analyse, and a logical property derived from the specification of the system, normally denominated as \( \phi \). In case that the given model is an actual model for the postulated proposition - meaning that the property holds for it - we denote it as \( M \models \phi \). Otherwise it is denoted as \( M \not\models \phi \). With both the model and the properties being built up on formal requirements, the process of model checking is completely automated in comparison to deductive techniques. The general procedure of the model checking approach with its input and output information can be seen in Figure 2.1.

As shown in the diagram, the process can finally lead to two different results. In case that the required proposition holds for the model, the verification system provides this information as output. Otherwise, the model checking technique is designed to return a
2 Conceptual Foundations

counterexample, which leads to some state or path of the model at which the properties cannot be guaranteed to hold anymore.

While the model itself can be represented as a finite state machine, transition system or programmatic procedure, we need a specific symbolic logic to formulate the specifications of a system. The most widespread logics among them are the Computation Tree Logic (CTL) and the Linear Time Temporal Logic (LTL), which we will cover in the following subsections. But before that, we have a look at the general satisfiability problem (SAT) that forms the basis of every logical verification of a model.

**SAT**

The **Boolean Satisfiability Problem** - commonly abbreviated as SAT - occurs each time we are given an atomic or composed logical formula and want to check whether there exists an assignment of the involved set of Boolean variables to make the complete expression evaluate to true. In this case, the expression is called *satisfiable*. It is already proved that the problem is NP-complete. In terms of model checking, these Boolean expressions are derived from the specification, and the practical aim is to both determine if the expression can be satisfied at all, meaning that the formula actually evaluates to true in at least one case, and - if that is possible - go a step further from this SAT problem and evaluate if, based on system observations, the environment provides the necessary conditions to satisfy the property. Possible logics to formulate these verification expressions are described in the following.

**CTL**

A very common and basic logic for verification processes is the **Computation Tree Logic**. The following definition describes its primary and derived syntactical features.

**Definition 2.1** The Computation Tree Logic (CTL) is inductively defined through the following syntax:

\[
\phi ::= \text{true} \mid a \mid \phi_1 \land \phi_2 \mid \neg \phi \mid X\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2
\]  

(2.1)
By combining the basic connectives and adding path quantifiers, we can furthermore derive the following adequate set of expressions:

\[
\phi ::= \text{false} \mid \phi_1 \lor \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \Leftrightarrow \phi_2 \quad (2.2)
\]

\[
AX \phi \mid EX \phi \mid AF \phi \mid EF \phi \\
AG \phi \mid EG \phi \mid A[\phi_1 U \phi_2] \mid E[\phi_1 U \phi_2]
\]

At this point, \( a \) represents an arbitrary atomic proposition, and \( \phi \) is any CTL formula composed from these propositions. As CTL falls in the category of branching-time logics, the logical expressions cover all possible paths at every given point of time. This fact enables the use of path-spanning quantifiers - \( A \) (all) or \( E \) (exists) - to specify on which paths a certain property needs to hold. The possible combinations of temporal operators are listed in the last two rows of Definition 2.2.

### LTL

The second commonly used temporal logic is the **Linear Time Temporal Logic**, which is - similar to CTL - a sub-logic of the subordinate CTL* set. The following definition gives an overview of the syntactical elements included in LTL.

**Definition 2.2** The Linear Time Temporal Logic (LTL) is inductively defined through the following syntax:

\[
\phi ::= \text{true} \mid a \mid \phi_1 \land \phi_2 \mid \neg \phi \\
X \phi \mid F \phi \mid G \phi \mid \phi_1 U \phi_2
\]

By combining the basic connectives, we can furthermore derive the following adequate set of expressions:

\[
\phi ::= \text{false} \mid \phi_1 \lor \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \Leftrightarrow \phi_2 \quad (2.4)
\]

In contrast to CTL, the LTL logic is designed to be applied to individual program executions and therefore only takes single paths into account. For that reason, the definition does not include any \( A \) (all) or \( E \) (exists) path quantifiers.

### 2.3 Explicit-State / Symbolic Model Checking

The most basic approaches to model checking - the so called **Explicit-State and Symbolic Model Checking** - have already been used for decades and are the most widespread ones in the range of model verification. Provided some properties built up with temporal operators, the Explicit-State Model Checking performs a depth-first exhaustive search through the complete state space for each of those operators in order to derive a logical statement about their satisfiability. This explicit traversal through the state space leads to one of the most limiting problems of model checking: The complexity
increase of the state space, which is mostly exponential in regards to the amount of states and transition relations.

The first attempt to attenuate the impact of state space explosion was made by using Symbolic Model Checking, a fixed-point based method that treats the states of a model as sets instead of analysing them individually. This iterative process, combined with a suitable data structure such as Binary Decision Diagrams (BDD), allowed a more efficient model checking process, even though the core state space limitations remained unchanged. Therefore, in order to tackle this core problem - paired with other limiting factors like its missing capability to deal with dynamically changing systems - new approaches needed to be introduced, which we will investigate in the following.

### 2.4 Online Model Checking

As a relatively new approach - a technique which was not examined in detail until the beginning of the twenty-first century - Online Model Checking was introduced to deal with the emerging need for a concept which can be applied to the growing amount of cyber-physical systems embedded into a dynamically changing environment. A first mentioning of real-time and concurrent model checking can be found in publications of R. Alur in the early 90s, with the very basics covered in their work “Model-checking for real-time systems” [1]. Although it could be used for a first shallow analysis of real-time systems by involving dynamic time delays, it treated the environment in a static manner, meaning that it was still applied to a system model before deployment and the actual run-time situation.

In the early 2000s, the concept of Runtime Verification came up - introduced by the correspondent annual workshop with a monitoring concept by Geilen [2] amongst others - and initially moved the verification focus from the pre-deployment to the run-time phase of a system cycle. For the first time, it was then possible to turn the formal verification process into a dynamic technique which could be continuously adapted to observations of a real system in order to respond to property violations. Nevertheless, even though this concept allowed possibly preventing the recurrence of failures, it was still following a post-checking approach, only reacting to violations instead of fully preventing them. This kind of system view changed essentially with the emergence of the online approach. In a publication by Rammig et al. [3], this technique was brought to a first application as an operating system service. It was now possible to check a system during its execution, observing tasks at run-time and applying necessary adjustments to the system model in case that its configuration gets changed. Instead of relying on a plain check of already given model execution traces to adjust the system model, this new approach allowed a sort of pre-checking, verifying the possible model behaviour in the immediate future before it takes place in the observed system, and hereby actively preventing malfunctioning. In the scope of that publication, the technique was only applied to the rather specific field of task management in operating systems, so it still did not deliver a general framework
for online model checking which could be applied to any sort of state machine. The development of such a framework was finally started by Rinast [4], and in the course of this elaboration it forms the basis of the statistical extension of the online approach. Figure 2.2 illustrates the technique behind Online Model Checking as pointed out by the author: The original procedure of model checking (2.2a) takes up the complete state space of the model execution at once. In case that a single state in this total space does not satisfy any of the safety conditions, the property is considered to be false for the whole model and countermeasures need to be taken directly. A less restrictive concept is given by the OMC approach (2.2b), where the current state space in focus - represented by the overlapping triangles - is reduced to a fixed time interval on each verification step. Besides the fact that we can now progress with the first two execution steps without even encountering a potentially non-satisfying state, the overlapping intervals also guarantee that the following two execution steps will still satisfy the conditions and hence provide a time interval during which we can safely initiate the necessary countermeasures.

2.5 Statistical Model Checking

The second technique which we need to compose our new approach is the one of Statistical Model Checking (SMC). It is a static approach like Explicit Model Checking, but in contrast to that, it utilises statistical methods rather than exhaustive exploration of the complete state space. The benefits gained with this approach are apparent: By simulating the system model several times and combining the Boolean results of these simulation runs, it becomes unnecessary to review the complete state space, as we only need to focus on the current path each time. Therefore, the limiting factor for many verification attempts - some case of state space explosion - is eliminated. With this property, the SMC approach is predestined to complement the real-time capabilities of the previously described Online Model Checking attempt.

Figure 2.2: A comparison of the static and online verification approaches
In the following, we will have a look at different model and logic types which are commonly used to describe probabilistic systems. The first two specification and verification logics - PCTL and CSL - are extensively used by the model checker PRISM [5], while the last ones - MITL and its reduced form WMTL - provide the basis for the UPPAAL model checking process.

2.5.1 DTMC (Discrete-Time Markov Chain) and PCTL

A very basic kind of probabilistic model with elements that are used in all statistical model checkers, is the so called Discrete-Time Markov Chain. An example of such models is provided in Figure 2.3. While its general structure is similar to a finite state machine, the transitions are not guarded by an invariant, but are weighted with probability values, summing up to 1 for all outgoing transitions of a state. Corresponding to these relative probabilities, one transition is randomly chosen and triggered. For example, when state $s_2$ is active, the transitions $s_2 \rightarrow s_3$, $s_2 \rightarrow s_4$ and $s_2 \rightarrow s_5$ will be triggered with probabilities of 40%, 20% and 40%.

To actually take advantage of the additional capabilities of the model, we need to use an extended logic which can also handle probabilistic properties. One way to achieve this is the use of the Probabilistic Computation Tree Logic, which adds exactly these probability queries to the standard CTL language.

**Definition 2.3** The Probabilistic Computation Tree Logic (PCTL) is inductively defined through the following syntax:

\[
\phi \ ::= \ true \mid a \mid \phi_1 \land \phi_2 \mid \neg \phi \mid P_{\sim p}[\psi] \quad (2.5)
\]

\[
\psi \ ::= \ X\phi \mid \phi_1 U^k \phi_2 \mid \phi_1 U \phi_2 \mid F\phi \mid G\phi \quad (2.6)
\]

In the given definition, $a$ represents an arbitrary atomic proposition, $p \in [0,1]$ a path probability value, $\sim \in \{<,\leq,\geq,>\}$ one of the possible inequalities, and $k \in \mathbb{N}$ an
arbitrary natural number. The formulas $\phi$ in Expression 2.5 describe properties on states, whereas the formulas $\psi$ in Expression 2.6 can be used to establish the satisfactory constraints of a path.

Comparing the PCTL logic to the basic CTL version reveals that they accord with each other in most definitions, except for two expressions: The state formula definition now provides an additional probabilistic operator $P_{\gamma p}[\psi]$, which is applicable to describe and determine if the probability of some path formula lies above or below a certain threshold value. In the logics of PCTL, this formula provides the basis for all probabilistic verifications. Besides that, a more general bounded-until expression $\phi U \leq k \phi_2$ is introduced, extending the usability of the former unbounded one. These definitions allow us to formulate different semantic properties for PCTL verification:

\[
\begin{align*}
\text{s} & \models \phi \quad \text{(Example: } \text{s} \models \text{fair} \lor \text{loaded}) \\
\omega & \models \phi_1 U \leq k \phi_2 \quad \text{(Example: } \omega \models \text{fair} U \leq 50 \text{loaded}) \\
\text{s} & \models P_{\gamma p}[\psi] \quad \text{(Example: } \text{s} \models P_{\geq 0.7}[\text{six}])
\end{align*}
\]

The first formula (2.7) represents the fact that a die is either fair or loaded at every point in time. The second formula (2.8) states that the die will be fair until it finally switches to a loaded state, which needs to take place at some point within 50 time units. An example for a probabilistic constraint is given in the last formula (2.9), expressing that the probability of getting a “six” in the future is greater than or equal to 0.7. Additional information on the semantics of PCTL can be found in Principles of Model Checking [6].

### 2.5.2 CTMC (Continuous-Time Markov Chain) and CSL

In those cases where a discrete view of our system clocks is not sufficient, we can utilise a Continuous-Time Markov Chain (CTMC) to model a system. In contrast to the discrete version, the transitions are not triggered by probabilities, but they provide certain rate values with which the individual transitions are triggered. Figure 2.4 shows a sample model with these rates. In each of the intermediate states, the transition leading back to the predecessor is triggered at rate 1, while the transition to the subsequent state will be activated periodically at a rate of 1/2.

Just as with DTMC models, we want to formulate probabilistic properties for the verification process, and therefore need a logic that is also capable of describing the eventual

![Figure 2.4: A sample model of a Continuous-Time Markov Chain](image)
validity of a property in a more general sense. In this case, Continuous Stochastic Logic
is one possible logic to describe the properties properly.

**Definition 2.4** The Continuous Stochastic Logic (CSL) is inductively defined through
the following syntax:

\[
\begin{align*}
\phi & ::= \text{true} \mid a \mid \phi_1 \land \phi_2 \mid \neg \phi \mid P_{\sim p}[\psi] \mid S_{\sim p}[\phi] \\
\psi & ::= X\phi \mid \phi_1 U^I \phi_2 \mid F\phi \mid G\phi
\end{align*}
\] (2.10)

Again, we use an arbitrary atomic proposition \(a\), the path probability \(p \in [0, 1]\), an
inequality denotation \(\sim \in \{<, \leq, \geq, >\}\), and - within this definition - \(I\) as an arbitrary
interval of \(\mathbb{R}_{\geq 0}\). The formulas \(\phi\) in Expression 2.10 are used for states, and the formulas \(\psi\) in Expression 2.11 are applicable to paths.

The CSL syntax provides even more expressions than the previously reviewed PCTL logic.
By using \(S_{\sim p}[\phi]\) instead of the path-based \(P_{\sim p}[\psi]\), we can determine the probability that
a certain (state) property finally holds in the long term. Regarding path formulas, we
can use \(\phi_1 U^I \phi_2\) to assure that a certain property \(\phi_1\) holds until another property \(\phi_2\)
holds, which has to occur within the given interval \(I\). With these definitions, we can now
semantically formulate the desired properties of the system:

\[
\begin{align*}
s & |\phi \quad \text{(Example: } s |\phi = \text{fair} \lor \text{loaded}) \quad (2.12) \\
\omega & |\phi_1 U^I \phi_2 \quad \text{(Example: } \omega |\phi_1 U^I [0,50] \phi_2 \text{ loaded}) \quad (2.13) \\
s & |P_{\sim p}[\psi] \quad \text{(Example: } s |P_{\sim p}[F [0,50] \text{ blocked}] \text{))} \quad (2.14) \\
s & |S_{\sim p}[\phi] \quad \text{(Example: } s |S_{\sim p}[\text{ blocked}] \text{)} \quad (2.15)
\end{align*}
\]

The first formula (2.12) expresses that a die is either fair or loaded at every point in
time. The second formula (2.13) states that the die will be fair until it finally switches
to a loaded state, which needs to take place within a given time interval (in this case:
\([0, 50]\)). Two formulae that can be used to express the evaluation of probabilistic values
are shown in the last two examples (2.14) and (2.15), referring to the probabilities that
the blocked state will eventually be reached within the time interval of \([0, 50]\), or finally
be reached in the long term. Additional information on the semantics of CSL can be
found in the article *Model-Checking Algorithms for Continuous-Time Markov Chains* [7].

**2.5.3 NSTA (Network of Stochastic Timed Automata) and MITL**

Finally, we will have a look at the sort of model which is especially important for the
UPPAAL model checker system that we use throughout this project. A Stochastic Timed
Automata (STA) combines the general structure of a timed automaton with underlying
stochastic processes. Figure 2.5 illustrates how a composition of these automata - a
Network of Stochastic Timed Automata (NSTA) - may look like. Besides the standard
states and transitions, an NSTA provides broadcast synchronisation channels to synchro-
nise the transition triggering of different STAs. We should note that even though binary
synchronisation channels are generally supported as well, they are not applied in order keep all components in an unblocked state.

The most important feature in terms of clock implementations are the different rates that can be applied to each clock variable. In the example, the rate of \( x \) starts with a value 4, meaning that the clock will proceed by 4 steps during each universal time step, and this rate will be gradually decremented in the following states.

The logic which was chosen to describe the validation properties of those NPTAs considering the different clock variables is a form of Metric Interval Temporal Logic (MITL).

**Definition 2.5** The Metric Interval Temporal Logic (MITL) is inductively defined through the following syntax:

\[
\phi ::= a \mid \phi_1 \land \phi_2 \mid \neg \phi \mid O\phi \mid \phi_1 U^d \phi_2 \quad (2.16)
\]

This definition uses an arbitrary atomic proposition \( a \), a clock variable \( x \) and an arbitrary natural number \( k \in \mathbb{N} \). Besides the usual logical operators used in the other logic specifications as well, MITL offers an extended bounded-until operator \( \phi_1 U^d \phi_2 \) which makes it possible to not only define the maximum time value until which \( \phi_2 \) must eventually be satisfied, but also the actual clock variable we are referring to. Hence, we are allowed to use multiple different clocks in parallel.

As it is not decidable whether the probability of a run of the system model \( M \) satisfying \( \psi \) lies above a given threshold value, i.e. \( P_M(\psi) \geq p \), we need to restrict the logic to the cost-bounded case, so we are geared to the principles of Weighted Metric Temporal Logic (WMTL).

**Definition 2.6** The cost-bounded form of MITL is expressed as: \( P_M(\phi, s \leq c \phi) \)
Here, $x$ again represents a clock variable, and the arbitrary natural number $C \in \mathbb{N}$ is generally bound. Using this representation, we are able to formulate the following different queries regarding probability values:

**Probability Evaluation:** $P_M(\phi \leq C \phi)$\(^{(2.17)}\)

**Hypothesis Testing:** $P_M(\phi \leq C \phi) \geq p \in [0, 1]$\(^{(2.18)}\)

**Probability Comparison:** $P_M(\phi_1 \leq C \phi_1) > P_M(\phi_2 \leq C \phi_2)$\(^{(2.19)}\)

All of the expressions in Eq. 2.17, Eq. 2.18 and Eq. 2.19 can be approximated by simulation runs, whose Boolean outcomes are finally gathered and statistically analysed.
3 Existing Solutions

In this chapter, we will focus on two different model checkers - Uppaal and Prism - that provide the necessary functionality to create and verify probabilistic models. Afterwards, we will take a look at the modular software solutions of Uppaal that actually provide the necessary verification query routines used in this project to integrate stochastic capabilities into the OMC framework.

3.1 Available Model Checking Environments

3.1.1 UPPAAL

The Uppaal model checker - developed in a cooperative project at the Uppsala and Aalborg universities by Larsen, Yi et al. - forms the central modelling and verification component of our work, and is integrated into the framework via several interfaces provided by the integrated tool environment to communicate with the model construction, simulation and verification processes. As the Uppaal environment was already used by the original OMC framework, we use it for our extended approach as well, including the Uppaal SMC extension.

The general user interface together with the model editor section is shown in Figure 3.1. A model can be created in both a code-based (as "processes") or graphical manner, and can be extended by custom function calls and data type declarations created in a C-like syntax. By using states and transitions with corresponding invariants and guards, models in Uppaal can represent several types of state machines, and different sub-model templates can be instantiated as often as desired to build up systems of arbitrary complexity. Multiple clock variables for global and local processes can be used to create a form of

Figure 3.1: The (graphical) model editor view of the UPPAAL model checker
independence between the sub-systems, and the synchronization of processes is achieved via either binary or broadcast synchronization channels. A more detailed description of the components and capabilities of Uppaal can be found in the corresponding section about Uppaal provided in the thesis covering the OMC framework [4] or in the more extensive main tutorial of the program [8].

3.1.2 PRISM

Another integrated environment for the model checking process that we could have used for the SOMC approach is the probabilistic symbolic model checker Prism [9], which is specifically designed to deal with models containing stochastic elements. As we pointed out in the Conceptual Foundations chapter, the model checker can be used to build a broad range of probabilistic model types, including the described DTMCs and CTMCs as well as the completely non-deterministic probabilistic (timed) automata and Markov decision processes.

Figure 3.2 shows the basic GUI of Prism as well the specific view in which the models are described. In contrast to Uppaal, the model creation process of Prism completely relies on a modelling language without providing a graphical modelling tool. Structured in modules, the states (e.g., \( p1: [0..15] \)) and transitions paired with transition conditions (e.g., \( \text{[] } p1=2 \land (\text{none}_lht \lor \text{some}_a) \rightarrow (p1'=3) \)) are defined. Additional formulae (e.g., \( \text{formula none}_lht = !(p2>=4\land p2<=13)\land !(p3>=4\land p3<=13) \)) can be used to express more complex conditions for the transition process.

Especially for the fact that Prism supports more probabilistic model types than Uppaal, it could be a reasonable addition to the SOMC framework for prospective research. As the user interface itself as well as the parsers are written in Java, the program could easily be integrated in the existing workflow. Additional information about the tool and its supported models and concepts can be found in the manual distributed with the program.
3 Existing Solutions

3.2 UPPAAL Solutions

3.2.1 UPPAAL-SMC

The Uppaal module we will use exclusively for the framework add-on is the Uppaal SMC extension, which adds probabilistic modelling and verification capabilities to the tool environment. The formerly described Figure 3.1 already shows an example of a probabilistically extended model, and a more in-depth investigation of the added components regarding its use for our purpose will be provided in the next chapter.

The graphical user interface for accessing the stochastic verification tools from within Uppaal is shown in Figure 3.3. At this point, we can insert our own queries - composed of logical properties regarding the variables of template instances or global model parameters - and evaluate them to determine several types of bounds and probability values using methods such as *sequential hypothesis testing* and *monte carlo simulation*. The most important query types among them are the following:

- **Simulation:** \(\text{simulate} \ 1 \ [<=\ 100] \ \{D.\text{One}, D.\text{Six}\}\) \hfill (3.1)
- **Probability Evaluation:** \(\text{Pr}[<\ =\ 100] \ (<>\ D.\text{One})\) \hfill (3.2)
- **Hypothesis Testing:** \(\text{Pr}[<\ =\ 100] \ (<>\ D.\text{One}) > 0.6\) \hfill (3.3)
- **Probability Comparison:** \(\text{Pr}[<\ =\ 100] \ (<>\ D.\text{One}) \geq \text{Pr}[<\ =\ 100] \ (<>\ D.\text{Six})\) \hfill (3.4)
- **Value Bound Determination:** \(E[<=\ 100; 1000] \ (\max : D.\text{RollCount})\) \hfill (3.5)

The aspect that all of these queries have in common is the scope value, defining the time interval - starting from the moment of query evaluation - in which the property should be considered. The first query (3.1) initiates a single simulation run of the model.
(referring to a dice game example we will introduce later on) and tracks the activity
data of the “One” and “Six” states, which can be graphically reviewed afterwards. The
second query (3.2) performs a basic probability evaluation step. In this case, we try to
find out how likely it is to reach the state “One” at least once within the next 100 time
units. Using the result evaluation tools provided by Uppaal SMC, it is possible to further
analyse the probability, probability density, and cumulative probability distributions of
the simulation result.
The next two queries (3.3) and (3.4) allow us to review probabilities in comparison to
other fixed or event-bound probability values. In contrast to the concrete data of the first
query and the probability data of the second query, the evaluation of these two formulae
return the Boolean result data indicating whether the hypothesis is confirmed by the
performed simulation runs or not.
Finally, Uppaal SMC also allows determining the expected (abbreviated as \(E\)) value
bounds that are reached throughout the simulation runs. The last query (3.5) shows
how such queries are formulated for a selected number of 1000 runs. In this case, we look
for the maximum value of the roll counter. A more detailed explanation of the query
types and the methods behind them can be found in the documentation [10].

The behaviour of the stochastic verification process including the time and probability
precision can be adjusted via global statistic parameters, which are the following:

\[
\begin{align*}
\text{Probabilistic Deviation: } & lower(-\delta), upper(+\delta) & (3.6) \\
\text{Probability of false results: } & negatives(\alpha), positives(\beta) & (3.7) \\
\text{Probability Uncertainty: } & \epsilon & (3.8) \\
\text{Ratio Bounds: } & lower(u_0), upper(u_1) & (3.9)
\end{align*}
\]

The probabilistic deviation values (3.6) as well as the probability of false positives and
negatives (3.7) are mainly used for hypothesis testing, defining the region of indifference
and the probabilities that the alternative hypothesis is accepted by mistake. The prob-
ability uncertainty (3.8) defines the interval width around the determined probability
value of a basic probability evaluation step. It is highly probable that the real probabil-
ity of the specific event lies within that interval. Finally, the lower and upper ratio bound
(3.9) influence the probability comparison step by providing the bound values which the
ratio of the two probabilities needs to lie above or below to deduce a clear result state-
ment. Similarly to the query explanations, further information about these parameters
as well as additional ones (e.g., histogram, trace resolution, and discretisation settings)
is provided in the Uppaal SMC documentation [10].
4 The Statistical Online Model Checking (SOMC) Approach

Now that we gathered all necessary information on the systematic background of different model checking techniques for both static and real-time verification, and concluded the benefits of using UPPAAL for our purpose, we first take a deeper look at those specific concepts of online model checking which can be applied to our new statistical online model checking approach as well. Afterwards, we complement the given OMC approach with additional concepts that allow us to not only deal with OMC enabled models, but also with a broader range of probabilistic models containing statistical and non-deterministic elements. We support the most relevant SOMC specific constructs with an underlying algorithmic construct, before we finally transfer the complete framework to an actual implementation – an add-on for the already existent OMC framework developed by Rinast [4].

4.1 Applicable OMC Concepts

In order to provide online capabilities built up on continuous up-to-date real world parameters, two basic processing steps need to be applied to the real-time model: The State Space Reconstruction (SSR) process tracks the simulation trace of the current model execution run, and reduces it to a minimal path by deleting looped and redundant sub-paths as well as those paths without further effect on reaching the current state caused by subsequent variable resets. After deriving a valid and minimized path to the current state, the OMC Framework performs State Space Adaption (SSA) steps on existing or newly inserted transitions within the path to finally reach the desired state that matches real system observations. Rinast describes two concepts to implement these steps - a graph-based and a use-def-chain-based reconstruction and adaption procedure - which are further explained in his thesis [4].

In their basic form, these techniques need to be applied for the SOMC concept as well. But as we will see in the next Section 4.2, some features enabled by the statistical approach such as varying clock rates and non-deterministic stay times of unconstrained states prohibit a direct use of their OMC implementation for our purpose.

The limitations regarding the general structure of the model that are already mentioned for the original OMC framework hold for our concept as well. We still need to restrict our adaption step to data variables which are not constrained in any location of the model, as this would require us to perform additional checks on whether the constrains are still met and the currently active state is actually allowed.

Even though the SOMC approach does not require viewing the complete state space for its queries, the state space itself still plays a role during the SSR process. For that reason, we are limited to models where the clocks in question are restricted, i.e., are reset
4.2 New SOMC Concepts

The statistic model checking approach itself together with its concrete implementation in Uppaal SMC provides five different main conceptions, which we could consider to transfer to the online framework: Probabilistic branching, exponential rates, dynamic pattern creation, differential equations using clock rates, and user-defined custom delay distributions. In the following, we give an overview of each of these elements and evaluate their individual use and importance for the SOMC framework.

Probabilistic Branches

The concept of Probabilistic Branches can be found at every place in a model where a transition leads to a branching point which in turn provides multiple outgoing edges, each of which is not guarded by some invariant, but parameterised with an integral weight, representing their relative trigger probability. A basic example of such a non-deterministic model component can be seen in Figure 4.1a, with one transition for decrementing a variable having an 80% and the other transition for incrementation having a 20% probability. In a probabilistic system, these branches form one of the two additional main components and need to be further investigated at this point.

The probabilistic branches influence the state space reconstruction (SSR) process as well as the variable developing throughout a path, so we need to consider both in order to integrate them properly into the workflow of the OMC framework.

We already saw that the crucial requirement for the SSR is the reset of each involved clock variable at some point of a path inside the model to run inside desired time bounds. Figure 4.1b illustrates the case that could possibly occur in a branched system: One possible path – in this example the one with the greater probability – will be traversed without resetting the clock variable, while the less probable path finally triggers a reset of the clock. For the purpose of our implementation, we will choose the most restrictive
Algorithm 1: Routine for probabilistic branches

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>newExecState = execState;</td>
</tr>
<tr>
<td>2</td>
<td>if type == 'predominantlyBranched' then /* Freq. branching and little impact */</td>
</tr>
<tr>
<td>3</td>
<td>Wait fixed time interval while accumulating data (accumData);</td>
</tr>
<tr>
<td>4</td>
<td>foreach {Stochastic parameter SP ∈ execState} do /* Update parameters */</td>
</tr>
<tr>
<td>5</td>
<td>SP = deriveParameters(accumData);</td>
</tr>
<tr>
<td>6</td>
<td>else if type == 'predominantlyDeterminate' then /* Infreq. branching and great impact */</td>
</tr>
<tr>
<td>7</td>
<td>if (BP ∈ Bset: (execState.currLoc, BP) ∈ Tset) then /* Branch point following? */</td>
</tr>
<tr>
<td>8</td>
<td>Observe decision in real system (singleSysObs);</td>
</tr>
<tr>
<td>9</td>
<td>foreach {Stochastic parameter SP ∈ execState} do /* Update parameters */</td>
</tr>
<tr>
<td>10</td>
<td>SP = adjustParameters(singleSysObs);</td>
</tr>
<tr>
<td>11</td>
<td>newExecState.currLoc = singleSysObs.location; /* Set loc to observed state */</td>
</tr>
</tbody>
</table>

In this work, we will therefore mainly focus on the adjustment of branch probabilities derived from observation of the real-time system. A possible routine for dealing with branches at the different points of execution is presented in Algorithm 1. It distinguished two model cases: If the branching element is the central part of the model and continuously traversed in a high frequency ("predominantlyBranched"), we can rely on observation of the real system distribution over a fixed period of time, and then derive new probability parameters for the branching transitions from that data. This concept is less useful if we analyse a model with rather infrequent branching points, where most of the process is taking place along a determinate, non-stochastic path ("predominantlyDeterminate"). Here it might be useful to adjust the model to every individual branching point traversal. This can be accomplished by observing the decision in the real system, afterwards only adjust the probability parameters by that single event outcome, and manually set the active model location to the observed one.

Exponential Rates

The second important element in the conception of a probabilistic model regards the staying time in each state in case that either the combination of state invariants and outgoing transition guards provide a time interval for triggering or no invariants are given at all. In the former case, the triggering probabilities will be uniformly distributed throughout the time interval. The latter case however requires additional information...
due to the unbounded time interval \([0, \infty]\) and can be addressed with the so-called **Exponential Rates**, a property defined for a state whenever a uniform distribution cannot be applied. An example of this model element is shown in Figure 4.2a.

As the name implies, the parameter determines the exponential factor with which the probability of staying in the currently active state is decreased over time. This probability data is represented through the cumulative distribution function \(1 - e^{-\alpha t}\), where \(\alpha\) is the exponential rate value.

In contrast to the introduced probabilistic branching, exponential rates do not influence the simulation trace in any way under the assumption that we view the trace just as a chronological chain of model states being activated during execution. Only the variable changes during the active time of a state are influenced this way. For an extension of the OMC framework, this means that the state space reconstruction method, which is already formally defined in [4], can be left unchanged in principle. Nevertheless, the way in which the Exponential Rates are implemented in Uppaal still induces a problem for our implementation: While such a state remains active, the duration - which is supposed to be exponentially distributed - is only internally available and not accessible for the used interfaces to Uppaal. This means that the corresponding clock variables cannot be considered for the SSR and its DBMs directly. In consequence, every performed SSR step would force the exponentially rated state to be 'reset', i.e. being treated as if it became active just at the current time. A possible solution for this problem could be the replacement of each exponentially rated state with an equivalent Custom Delay Distribution structure with explicitly declared clock variables, which will be explained next.

---

**Code-Snippet 1**: Java implementation of the exponential data generation routine

```java
public double[] generateExpDistriDoubleSamples(double lowerBound, double expRate, int sampleNum) {
    double[] randomDoubles = generateEqDistriDoubleSamples(0, 1, sampleNum);

    // Return exponentially distributed values with the inversion method
    double[] randomExpDoubles = new double[sampleNum];
    for (int i = 0; i < sampleNum; i++) {
        randomExpDoubles[i] = (Math.log(1 - randomDoubles[i]) / (-expRate)) + lowerBound;
    }
    return randomExpDoubles;
}
```
Our contribution at this point will rather be a method to measure the potentially active
time of each state during run-time and to use the gathered data for an adjustment of
exponential rates of the model to fit the current behaviour of the real system.
The two code snippets show the basic procedures for exponential data generation as well
as exponential parameter approximation. In the first code section (Code-Snippet 1), we
use the inversion method to generate exponential data from a formerly equally distributed
data series. This method is applicable to different distributions with a defined inverse
or quantile function. The second code section (Code-Snippet 2) shows how the observed
data series can then be used to derive approximated distribution parameters. In the case
of Exponential Rates, we focus on the routine for exponential distributions. The inverse
of the arithmetic mean of observed values already provides a feasible approximation of the
\( \lambda \)-parameter (in our notation the exponential rate value \( a \)) for the cumulative distribution
function.

**Custom Delay Distributions**

The next element enabled through our use of SMC - which is related to the explained
Exponential Rate states and can be seen as a generalization of the principle behind it - is the Custom Delay Distribution. The general structure of this construct is shown in Figure 4.2b.

Using this element, we are allowed to calculate values with any desired distribution
by providing a custom delay function. An initially reset and explicitly given clock then
counts up to this delay value, and finally a leaving transition is triggered. In case that the
custom delay function produces exponentially distributed data, this structure is similar
to Exponential Rates, with the difference that here the clock variables can be externally
accessed.

**Clock Rates and Differential Equations**

Another new concept allows us to use multiple time domains progressing independently
at different speed levels: The Clock Rates. Figure 4.3 illustrates how these rates are
implemented. By manually setting the first derivation of a clock with the instruction
$x' = 0$ in the final location of the example, we force the clock to stay at its current value, as it will now advance by 0 time units per global tick. In the same way, the clock progression rate is set to 1 in the initial state, and therefore acts like a clock without a custom clock rate at this point. In this manner, we can move one step further from the original synchronized clock rate concept and even use the rates to represent Differential Equations of arbitrary order by chaining the first order differential rates. It is hence the central component if a system modelling process involves the need for hybrid automata, and extends the lineup of potential use cases.

But even though this concept may be of great use for a comprehensive SOMC approach, a differential equation requires much effort on several levels. For one respect, it may infinitely extend the state space during a simulation, as its size is directly dependent on the possible interval size and granularity of the clock variables involved. This, in turn, has a direct impact on the state space reconstruction process, and may break the real-time constraint which the framework is based on. Besides that, we would need to extend the concept of DBM transformations in order to cover clocks with different progression behaviour. For that reason, this concept is only described in this project and will require further mathematical investigation before being fully usable.

**Dynamic Instance Spawning**

The last element that we want to consider for the SOMC framework is the concept of Dynamic Instance Spawning. Its general structure, consisting of a parent automata that triggers the spawning of child model instances with a lifetime lasting until a child's final call of the `exit()` instruction, is shown in Figure 4.4. As long as the limit of `spawn-Num` children is not reached, the parent model in this example continuously creates child instances in an exponentially distributed manner (caused by the exponential rate of 1 in the initial state as previously explained). These instances will now increment a variable - again with an exponential time distribution - until reaching the `spawnLT` value and eventually initiating their own deletion.

![Figure 4.4: The parent and child models for dynamic instance spawning](image-url)
This concept would be suitable in those cases where we deal with changing amounts of individual and rather complex entities, which cannot be represented by a plain array entry anymore, but which manifest their own behaviour. Groups of visitors in a museum, scheduled and executed threads, or - as we will pick up again in one of our proof-of-concept experiments - the number of agents in a call centre could be modelled this way.

We may then consider to frequently update parameters for the number of entities and their probability for creation and elimination during an experiment. We need to keep in mind that the varying entity amount will directly influence the state space size and therefore its reconstruction time.

### 4.3 Implementation and Integration

The implementation of the derived requirements for the framework takes place at different parts of the software system. In general, the OMC framework itself is divided into several modules responsible for distinct tasks of the application workflow. Figure 4.5 gives an overview of the main components which the OMC framework consists of.

The upper half shows the Data Acquisition section. It contains all the necessary steps of collecting, structuring, and providing data. It ranges from modules for system observation (SensorAdapter, FileProvider), to data processing steps like linear approximation of data sets (LinearRegressor), and interfaces usable for applications to access the prepared data (ProviderAdapter).
The lower part consists of the Model Processing and concrete Application Development section, which is used to implement custom experiments utilizing the OMC capabilities. For that purpose, custom experiments inherit the base \textit{OMCApplication} class. The model is processed via the \textit{UppaalVerifier}, \textit{UppaalReconstructor} and \textit{UppaalSimulator} classes, which provide the functionality to formulate queries, derive initial paths for state space reconstruction and keep the model execution synchronized with the observed system. A variety of different Printer classes allow the output of data depending on the current simulation phase, and finally the \textit{VisualizationGUI} provides a graphical front end for the underlying framework.

Both the necessary and potentially required changes and additions to the framework are shown in Figure 4.6. The red-coloured elements indicate the classes that are already existent and need to be adapted to the SOMC add-on. On the one hand, the \textit{OMCApplication} class itself obviously need to be extended to handle the newly integrated interfaces for SOMC capabilities. We should note that even though the SOMC functions could have been directly integrated into the existing verifier classes, we decided to handle them in distinct classes in order to keep the methods separated according to the individual approach they belong to. Similar changes need to be applied to the \textit{UppaalSimulator}, which also directly contains instances of the Verifier classes.

At last, the \textit{VisualizationGUI} class needs to be extended in order to support the visualization of SOMC specific elements (e.g., branching points) and the associated label data (e.g., exponential rates, transition probabilities, . . . ).
The Statistical Online Model Checking (SOMC) Approach

<table>
<thead>
<tr>
<th>Query function</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>All functions</td>
<td>verification property, scope</td>
<td>query success</td>
</tr>
<tr>
<td>performSimulationRuns(...)</td>
<td>data names</td>
<td>tracked data</td>
</tr>
<tr>
<td>determineValueBounds(...)</td>
<td></td>
<td>value bounds</td>
</tr>
<tr>
<td>determineProbability(...)</td>
<td></td>
<td>prob. bounds</td>
</tr>
<tr>
<td>determineTimeRangeProbability(...)</td>
<td>lower time bound</td>
<td>prob. bounds</td>
</tr>
<tr>
<td>testHypothesis(...)</td>
<td>prob. threshold, inequality</td>
<td>boolean eval</td>
</tr>
<tr>
<td>compareProbability(...)</td>
<td>prob. 1, prob. 2, inequality</td>
<td>boolean eval</td>
</tr>
<tr>
<td>compareIntervalProbability(...)</td>
<td>prob. 1, prob. 2, diff. limit</td>
<td>intersection info</td>
</tr>
</tbody>
</table>

Table 4.1: Possible query functions with parameters

The newly added classes are indicated as blue-coloured elements. The core part of the SOMC add-on is contained in the section formed by the *UppaalSMCVerifier*, *SMCVerifierHook* and *SMCVerificationHook*.

In the *UppaalSMCVerifier* class, we define methods that formulate proper SMC queries consistent with the corresponding Uppaal API, and cover both atomic queries (e.g., Hypothesis Testing, Probability Comparison, ...) as well as composed ones for bundled data (e.g., determination of minimum and maximum values, comparison of probability intervals, ...). Additionally we adapt the query function itself to extract the desired SMC data from the received result strings. An overview of possible query functions with corresponding input and output data is given in Table 4.1.

The *SMCVerifierHook* interface - similar to the original *VerifierHook* of the OMC framework - allows the different components of the framework to trigger procedures during execution of each verification step like its starting or finishing phase. As an example, a concrete implementation of the *OMCAppliation* class using the Hook interface can update its internal parameters with data provided as soon as the Verifier object finishes the query and delivers the result.

Finally, the *SMCVerificationHook* - again similar to the existing *VerificationHook* - provides the same Hook functionality as explained before to be used by the *VisualizationGUI*, which needs to update synchronously to the verification and simulation steps as well.

On the data acquisition side, the *StochasticTools* class provides common functions for the statistical generation, gathering and interpretation of data sets. This includes sample generators for several distributions (e.g., uniform, exponential, normal, ...), distribution parameter approximation functions to update the exponential rates of the system, and methods to discretise and group collected data. These stochastic tools can then be used by concrete SMC experiments, which will be covered in the following chapter.
5 SOMC Experimental Applications

In this chapter, we apply the implemented SOMC framework to multiple models in order to show its general applicability to different scenarios as well as its benefits and limitations for an online model checking approach.

5.1 Proof-of-Concept Sample Models

The focus of this section lies on two specific models - a DiceGame and a CallCentre experiment which show the utilisation of the stochastic branching point and exponential rate components explained in Chapter 4 in more tangible scenarios. We will have a deeper look at the model components and their interaction, the relevant parameters for the experiments and possible queries for the verification step.

5.1.1 The Dice Game

The first sample model represents a basic form of a dice game. The main Uppaal component of the system is shown in Figure 5.1. It consists of an initial state which also serves as the idle state between the dice rolls, a branched, stochastic section for the actual roll result decision, and a final state which is reached as soon as the desired number of rolls has been performed.

As we can see, the different transitions emerging from the central branching point are not weighted with fixed probabilities. In fact, we want to be able to adapt the variable probabilities to the actual probability distribution obtained from ongoing observations of the real system. In our case, this "real system" is an instance of another class, DiceGame, which continuously generates dice roll data with a variable discrete value distribution. This way, we can simulate an exchange of the dice from a formerly fair one to a loaded one, and analyse how the model and the SOMC verification step react to the change.

For our tests, the sets of relevant parameters have the following form:

- **Possible dice values:** [1 2 3 4 5 6] (5.1)
- **Fair distribution:** [0.167, 0.167, 0.167, 0.167, 0.167, 0.167] (5.2)
- **Loaded distribution:** [0.05, 0.05, 0.05, 0.05, 0.05, 0.75] (5.3)

The parameters indicate that we are using a normal 6-sided die with uniform value distribution, and change to a loaded one that results in the value of 6 with a 75% probability after a fixed amount of rolls. The functions for discrete stochastic data generation as well as the accumulation of observed data are provided through the StochasticTools class delivered with the SOMC framework.
For the SOMC verification step, there are now multiple checking variants that we can perform to derive statistical information, and the following ones will be applied in our analysis:

**Determine single value probability:** D.One? \hspace{1cm} (5.4)

**Compare interval probabilities:** D.Six > D.One + 0.2? \hspace{1cm} (5.5)

**Test hypothesis:** D.Six <= 0.95? \hspace{1cm} (5.6)

Depending on the verification interval that we set for the experiment, the first Query 5.4 will result in the probability value that the value 1 will occur at least once within that time interval. By setting the interval size to 1 and assuming that no delay is caused by the query itself, the result value should be equivalent to the current probability value we set for the corresponding transition.

Besides this basic determination of a single probability value, we will also perform comparison (Query 5.5) and hypothesis tests (Query 5.6). They give information about how the specific probability compares to either another probability or a fixed value. In case that the probability of getting a 6 grows beyond 95% within the given interval, we will be informed about it. The same holds for the case that the interval probability of 1 exceeds the probability of 6 by more than 20%. In the actual implementation, Query 5.5 does not need to be provided a specific direction of inequality, as it will be false as soon as the two probabilities in question differ by more than the absolute 0.2 value in any direction.

### 5.1.2 The Call Centre

The second showcase model for the SOMC approach represents a Call Centre scenario, built up on the fact that the time distributions of both *call duration* and the *time between*
calls is assumed to be mostly exponentially distributed. Therefore we will use the model for two purposes: One the one hand we want to apply the insights from Chapter 4 regarding Exponential Rates and test the relevant stochastic tool functions, and on the other hand we can analyse the model regarding future extensions that could be implemented utilizing the remaining SOMC elements not covered by now.

The different components of the Uppaal model can be seen in Figure 5.2. The model consists of 4 distinct parts: The Call Centre Main (5.2a) triggers a signal through the “call” synchronization channel at an alterable exponential rate callFreqRate and therefore serves as the incoming call provider. The Call Checker (5.2b) will then get informed about the call and - depending on whether there exist agents in a free state or not - delegates the incoming call to one of the free agents. Otherwise, the call will be rejected. The third component is the Call Agent (5.2c) itself, where we can find the second Exponential Rate value callDurRate. On delegation, the call agent switches to the active “Call” state and remains there until the exponential distributed call time is up. A “callFinished” synchronization channel finally informs the Call Agent Updater (5.2d) about the state of the agent, which then performs the enqueue step to re-add the agent to the waiting queue.

The main focus lies on the two Exponential Rates callFreqRate and callDurRate. Using the features of the new SOMC framework, we want to be able to track the actual call frequency and duration in the real world, and repeatedly use the observations to derive updated distribution parameters which are then applied to the model to keep it synchronized with the system in question. Again, the necessary functions for both exponential data generation and exponential distribution approximation are provided by the StochasticTools class.
Similar to the Dice Game example, we are then able to apply different sorts of queries to the constantly updated model, and the most significant ones are listed in the following:

- **Determine blocking probability**: \texttt{CH.Blocked?} \hspace{1cm} (5.7)
- **Determine queue value bounds**: \texttt{max/min: occupiedAgentCount?} \hspace{1cm} (5.8)
- **Test hypothesis**: \texttt{CH.Blocked \leq 0.5?} \hspace{1cm} (5.9)

The central point of focus lies on the question whether the given number of call agents is sufficient to handle all incoming calls based on the current frequency and time duration values. With the first Query 5.7, we are able to track exactly this probability over time. Additionally, we may want to obtain even more precise data and demand the exact potential maximum and minimum number of free or blocked agents in the near future of each verification step. For this purpose, Query 5.8 provides us with that desired data, which we can then use to perform countermeasures if needed. In case we only want to make sure that the probability of reaching a blocked state lies below a certain threshold, we can use the third Query 5.9. With that, we may assume that as long as the probability of an empty agent queue is smaller that 50%, we can tolerate the small potential probability of denying a call, and otherwise may need to increase the overall number of agents. Again, all of these verification queries are applied to an up-to-date model, and thus only hold for a fixed period of time.
6 Evaluation of Analysis Results

After conducting the aforementioned SOMC experiments, we will evaluate the obtained data in this chapter, starting with the presentation of the experiment results in Section 6.1. We will identify different factors that have an impact on the experiment outcomes throughout the stages of model adaptation and stochastic verification in Section 6.2, and relate both these factors and the underlying Java implementation to the computation times in Section 6.3. In the end, Section 6.4 will deal with a more general evaluation of the SOMC framework capabilities and limitations.

6.1 Experiment Results

In this section, we take a look at the parameter data and verification outcome during the DiceGame and CallCentre experiments. Starting with the DiceGame, Figure 6.1 shows both the probability distributions of the real system and the distribution sensed via the stochastic tools and therefore used by the system model. Due to the probabilistic property of the data, we can make two noticeable observations: In the sections I and III of diagram 6.1b, where the probabilities of the roll results remain constant in the real system (see 6.1a), we recognize dithering around the fixed real distributions. The abrupt jump from one probability distribution to the other additionally creates the section II of diagram 6.1b, where we observe a rather continuous change between probability values instead of the original jump. Several factors influence this behaviour, and they will be explained in-depth in Section 6.2. Figure 6.2 provides the verification results over time using the queries prepared in the previous chapter. In the first diagram (6.2a), the probability developing for the roll result of 1 is shown for different verification interval lengths. The second diagram (6.2b) shows the comparison of interval probabilities as Boolean value, meaning that the probabilities of the two outcomes 1 and 6 differ by a fixed amount or not. The last diagram (6.2c) shows the Boolean results of a hypothesis test, checking if the probability of the result 6 lies below 0.95.

![Figure 6.1: Dice game distribution developing over time](image-url)
The *CallCentre* experiment was conducted in a similar way. Figure 6.3 illustrates how the stochastic parameters - in this case the exponential rates of call frequency and duration - changed in the observed system and the system model. The two diagrams in Figure 6.3a and Figure 6.3b show the same phenomena of parameter dithering during constant phases and continuous developing in actual jump situations. The verification results of the three prepared queries are shown in Figure 6.4. While the first diagram (6.4a) again depicts the absolute probability of the attribute in focus - the probability of reaching a state of a completely occupied agent queue - with varying verification intervals, the second diagram (6.4b) shows the maximum and minimum amount of occupied agents over time. Finally, a hypothesis test is performed in the last diagram (6.4c), which evaluates if the probability of a blocked queue is still smaller than 0.5.

### 6.2 Influencing Factors

As our experiments show, multiple parameters influence the significance of verification data, computation times, model parameter integrity, and the overall simulation sequence.
The Time Interval Length parameter already had a great influence on the original OMC framework, and needs to be carefully considered at this point as well. By setting the interval length, we decide on the temporal scale which is suitable to obtain sufficient up-to-date validity predictions while still maintaining a tolerable prediction deviation caused by the stochastic and therefore non-deterministic model elements. For instance, the DiceGame model is more efficient for predictions on a smaller scale, as the event cycle is rather short and we want to instantly react to exchanged dice, while the CallCentre example takes more time to accumulate representative data, which we want to use to derive queue boundaries and judge on the best amount of agents on the long run.

Within the data acquisition section, the most influential value for our experiments is the Data Buffer Size chosen for each DataSeries provider. We can distinguish between two different cases: If we set the buffer size to a small value, we achieve a “faster” reaction time on changes of the data distribution. In the DiceGame experiment, this will lead to a more synchronized model adaption, but only within certain bounds. Using a value too small for the experiment, every natural stochastic deviation will instantly cause a change of the model parameters, and we will finally observe a parameter oscillation that may lead
to false interpretations. Both phenomena can be observed in Figure 6.5a. In the other case - meaning that we choose a rather high buffer size value - we can circumvent these oscillations, but highly decrease the reaction time in return. We will only slowly adapt the model to abrupt distribution changes, or might even miss the actual distribution developing in detail. Figure 6.5b illustrates the delay of the model parameter adaption, and the result of distribution data, which sums up to an average distribution on the large scale is shown in Figure 6.5c.

Looking deeper into the stochastic aspects of our experiments, we find another set of relevant parameters that influence the model developing. One of them is the **Parameter Difference**, i.e., the maximum difference between two sequential distribution parameter values. Figure 6.6 illustrates the impact on the *CallCentre* example throughout the period of time in which the model adapts from one parameter value to the next one. Even though an abrupt change from a frequency rate of 1 and duration rate of 9 to a frequency rate of 9 and duration rate of 1 takes place in the real world, the model parameters develop from one to another only stepwise, depending on the previously explained buffer size factor. This results in an intermediate situation with both a frequency and duration rate of around 2 within the model, which creates distribution scenarios that did never occur in reality, and might lead to undesired positive or negative peaks and thus to false verification results.

Strongly connected to the parameter difference appears the actual **Distribution Approximation Method** chosen for the data preprocessing step. The current implementation is built up on the assumption that the distribution parameters stay at some discrete values most of the time, and only rarely switch from one parameter configuration to another. This allows us to directly transfer the currently observed data to model parameters without further processing, but temporarily fails during those time periods in which two or more distributions are mixed up in the buffer. Using this insight, we may consider to extend the distribution approximation method in a way that it sticks to a previous distribution as long as the deviation of the current buffer data from its derived distribution lies above a certain value.

As last aspect on the stochastic side, the **Uppaal SMC Parameters** themselves influence the model simulation and verification process, as we further explained in Chapter 3 already. Figure 6.7 shows the probability values and hypothesis testing results during the
DiceGame experiment with different choices for the parameters Probability Uncertainty (6.7a), Probabilistic Deviation (6.7b) and False Positives / Negatives (6.7c). We can see that depending on the precision demanded from the SMC process, we obtain more accurate verification data (with the trade-off of longer verification times), which might eventually be critical in terms of our real-time constraint. But the greater the verification precision can be set, the greater the probability will be to correctly react to changes in the observed system.

Finally, we can observe that both the Query Formulation and the Query Amount have an impact on the performance achieved by the verification step. Figure 6.8 illustrates an example of the former one, occurring during a basic probability comparison step. Viewing the developing of computation time related to the absolute probability difference between the outcomes of a binary event, we can see that the shorter this difference gets, the longer it takes to come to a final conclusion for the query. This problem can be solved by adapting the Ratio Lower and Upper Bound parameters of the SMC verification tool. As another possibility, we could rely on a custom comparison function, which performs the comparison by determining the absolute probabilities of both events. This way, we
are more flexible to choose the absolute or relative comparison condition on our own while relying on fewer SMC parameters. By using an extended composed function (like our Probability Interval Comparison method), we can get a reasonable result in the time of two probability evaluation queries, treating as temporarily undecidable the case of interval overlapping caused by a too coarse-grained simulation resolution, and comparing the events by their absolute probability difference. This adds to the desired real-time constraint of the SOMC framework.

The query formulation and amount will now lead us to the next section, where we are going to gather the data regarding computation times of the involved processes, and afterwards evaluate it regarding our real-time constraint to derive information about the allowed complexity of the query and the model.

6.3 Computation Times

Regarding the computation times of the involved functions and procedures, we need to consider three different parts of the SOMC process: Model Complexity, Query Properties, and the Java Implementation. The first factor includes the complexity of the model components as well as their composition, and it is already covered in [4]. Generally - depending on the actual implementation of the model - a verification query with fixed scope will consume an increasing amount of time to determine a result. The second factor describes the queries themselves, which take up different amounts of time depending on their extent, scope, and number of atomic sub-queries. The third and last factor deals with the actual Java implementation, and covers the impact of utilized 3rd party libraries and our own approximation functions. The last two aspects will be further investigated in the following.

6.3.1 Impact of the Query Properties

The greatest impact in terms of the duration of a query processing step is caused by the Individual Query itself. Figure 6.9 illustrates the computation times of three differ-
ent queries for absolute probability evaluation, probability comparison and hypothesis
testing. They are captured during test executions of the *DiceGame* experiment, where
they are applied every 2 seconds for an scope of 10 time units. At first sight, all of them
already indicate that the execution time is not constant over time, but highly varies
during the simulation. The first diagram (6.9a) shows that the absolute probability eval-
uation requires time between 10ms and 1s, depending on the distribution parameters
at the given time. In the second diagram (6.9b), we observe an even greater interval
of time values, ranging from 67ms to 2s. For this comparison query, we used our own
function built up on two separate probability evaluations, and by comparing the first
two diagrams, we can in fact see that it takes at maximum around twice the time of a
single probability evaluation. The last diagram (6.9c) shows the duration results of a
hypothesis test, and compared to the former queries, it takes only a short time, between
15ms and only sporadically 350ms. But again, the duration is not constant, and can be
higher in different experiments.

It is apparent that this query factor may be critical for reaching a real-time constraint in
some experiments with a high frequent, dynamic behaviour. Nevertheless, by choosing
a reasonable amount and formulation of queries adapted to the individual experiment,
ity may still be possible to achieve real-time verification. An applicable preprocessing
concept for this query and parameter adaption step needs to be subject to further inves-
tigation.

Another factor which we can use to adapt the query execution times to a desired bound is
the **Scope** of the verification step. Figure 6.10 illustrates the duration of the probability
evaluation query during the *DiceGame* experiment for different scope values. We can see
that the maximum execution time increases with a growing verification interval and - in
fact - is not bound in that respect. But as the scope itself is limited in most applications,
its impact on the execution time will be bounded as well. Finding the right balance
between verification duration and result viability is the critical part at this point.

### 6.3.2 Impact of the Java Implementation

Even with a reasonable choice of model complexity, query composition and parameter
values, the real-time factor can still remain unsatisfied by the code implementation itself.

![Figure 6.10: Duration impact of different query scope values](image)
6 Evaluation of Analysis Results

Besides the factors already given by the original OMC framework and covered in [4], the two most frequently used components of the SOMC framework are the functions for data generation and data approximation. The duration of 50 individual executions of both functions are given in Figure 6.11. The data generation routine of the first diagram (6.11a) builds up on the uniform, random data generator provided by the 3rd party library of Apache Commons Math. The graphs indicate that all time measures lie in the dimension of microseconds, ranging from 100µs to 500µs. An even shorter period of time is required for the data approximation method in the second diagram (6.11b). At this point, the duration values lie between 1µs and 15µs.

In both cases, the duration dimensions are low enough to allow multiple (100 – 10000) executions of the functions within a second without affecting the overall performance of real-time simulation and verification, compared to the much greater influence of the queries and model composition.

6.4 SOMC Capabilities and Limitations

When we recall the underlying process that needs to be executed for every SOMC element we explained in Chapter 4 and combine it with our new insights from its application in Chapter 5, we notice that we can break down the SOMC approach - or rather the way we would like it to behave - into three consecutive steps:

1. **Obtain** updated information from the observed system

2. **Derive** statistical information from the model using updated data and simulation runs based on formulated queries

3. **Simulate** the model completely synchronized with the observed system (including both the parameter data and actual simulation trace)

With the functions we provided through the *StochasticTools* class, the accomplishment of the first step was quite straight-forward. Instead of only collecting the way it is performed by the original OMC framework, we take an additional step to interpret the gathered data in order to derive statistical parameters from it. Even though the data acquisition process is manageable in general, we need to understand the probabilistic
relations behind the system to derive usable parameters. In our sample models, the statistical elements were clearly separable, which allowed an easy transfer from the real system to the model. Systems with a more complex probabilistic background may create the need for additional statistical preprocessing of data, and the StochasticTools may be extended correspondingly.

After data acquisition, the following step - which is the core part of the SOMC framework - always consisted in the integration of the derived parameters into the system model as well as the model simulation and verification. A look at the results from the DiceGame and CallCentre examples shows that we are able to get valuable information from the SOMC verification: The developing of the observed data in comparison to the data predicted by the model is shown in Figure 6.12, and there are two aspects we need to consider. On the one hand, Figure 6.12a shows that we are able to correctly reflect the observed value distributions throughout the model simulation process, taking into account the transition regions between two successive distributions (compare Parameter Difference in 6.2). Figure 6.12b though indicates that the individual outcome of each random, atomic experiment will not correspond to the real world observations, which is clear considering the stochastic nature of both the real and the simulated system. For an exactly modelled copy of the real world during the whole experiment, we would therefore need to frequently update the result value counts as well and/or manually switch to the model location that was chosen in the real world directly after the stochastic component was evaluated (which specifically counts for models traversing an extensive and complex non-stochastic part after a stochastic element). In the DiceGame example with activated counts for the roll outcomes, we may consider to update these counts as well during each model adaption step, or alternatively activate the specific result state which corresponds to the real world outcome after each roll.

The third step - a completely synchronized simulation - finally creates critical problems for our SOMC approach due to current technical limitations of Uppaal, the (S)OMC framework itself and an incomplete mathematical foundation that covers all possible stochastic elements together. In Chapter 4, we already mentioned the framework restriction that disallows us to include Exponential Rates as used in the CallCentre example. This happens due to the fact that the model proceeds much slower than the real system,
as each model adaption and reconstruction step resets the internal time state of the Exponential Rate element. But the greatest cause for this asynchronism can be found in the Uppaal implementation itself. It turns out that the internal simulator of Uppaal - in contrast to the verifier - does not support most stochastic components. Figure 6.13 and Figure 6.14 show basic models of branches and exponential rate states, their assumed data distributions, and the data distributions actually created during the Uppaal simulation. We see that all outgoing transitions of branching points are handled as a uniform distribution, and the same can be observed for the temporal distribution of exponentially rated states.

Regarding the mathematical foundation of the model adaption step, future work will have to further investigate how to include different clock rates into the DBM-based transformation system. It is a necessary step to allow the determination of minimum initial paths for model reconstruction within the stochastic domain.

Summing up these problems for the SOMC approach, we are not able to fully simulate the model synchronously to a real system at this point, and need to rely on the discrete adaption and verification cycle for the present. This means that we observe real system
data, preprocess it to derive stochastic parameters, update the model accordingly and verify the given attributes for a fixed time interval, without simulating the model between these steps in real-time, which is possible with the non-stochastic OMC framework. This specific problem of probabilistic simulation still needs to be addressed, and may require an alternative view of the term “online” used in the original OMC approach.
7 Conclusion

Reviewing the extended SOMC framework as a whole, we could see that even though the approach offers several advantages compared to the original OMC framework, many of these advantages are linked to different trade-offs that need to be thoroughly considered before making the decision to practically apply it to a certain system. In general, the application of SOMC principles can be useful when either time-critical bounds prevent the use of symbolic mechanisms provided by the OMC framework, or when the real system itself contains probabilistic components, which cannot be taken into account by a deterministic approach. In both cases, the SOMC methods can provide weaker, probabilistically weighted satisfiability statements instead of completely abandoning the model checking process. Unfortunately, these weakened statements can lead to critical situations in different applications. Especially for safety-critical systems like medical devices, the outcome of this statistical model checking process needs to be considered with caution, as the safety of derived constrains will always lie between none and the assurance level of symbolic model checking. There is always a chance that the probabilistic outcome of a real event completely differs from the predictions during the verification phase, even when high probabilities are predicted. Nevertheless, the SOMC approach provides useful additional verification data and can be utilized to initiate countermeasures as soon as the probability of misbehaviour or malfunction exceed a set bound. Especially when a big data amount of previous events can be provided, the safety level of future predictions increases. But in the end, there can never be a way to predict probabilistic and future events with total certainty, and as long as parts of a system are non-deterministic, the model checking approach needs to be of a statistical nature as well.

In the course of this thesis, we showed that it is generally possible to transfer the statistical components of SMC to an online approach, but up to this point only within certain bounds. To reach the full potential of the framework, several aspects need to be subject to further investigation and implementation: The internal framework routines regarding the representation and processing of model variables only supports integer values at the moment. As both exponential rate parameters and branch probabilities can also take \( \text{Float Point} \) values (and only the latter one could be circumvented as the Uppaal SMC verifier internally converts absolute integer probabilities into relative float point probabilities), the framework needs to be adapted respectively. Additionally, we pointed out that a \textbf{Mathematical Foundation} for the SOMC components needs to be developed as well. Especially a suitable integration for stochastic features (e.g., different clock rates, exponential rates, alterable instance counts, \ldots) into the state space reconstruction process is still missing, and will be necessary to keep the observed system state and the model state synchronized. This synchronization leads to the next point, a solution to enable the \textbf{Simulation of SMC Components}. This aspect cannot be faced solely by the framework, but it also needs corresponding functionality of the UPPAAL simulator itself.

---

43
In terms of application, we could only provide tests regarding probability data and execution times for proof-of-concept models with limited extent. As soon as a mathematical foundation is completed, the SOMC framework needs to be evaluated for practical Case Studies as well. One example could be the Radiosurgery experiment that was already used by Rinast [4] to showcase the capabilities and limitations of the OMC framework. But not only the conceptual aspect will be important at that point. In fact, the central point we need to keep in mind will always be the real-time constraint, whose neglect can finally make any promising concept useless. For that reason, an even more extended Computation Time Analysis needs to be performed, including factors like the thread scheduler impact, the number of concurrent component instances, and any additional data preprocessing routines.

Finally, we could think of Additional Modules to increase the usability of the SOMC framework. As the concept-related restrictions of and assumptions about the model are known in advance (e.g., the necessary reset of all clock variables on every possible path cycle), it might be helpful to provide a tool that analyses a model beforehand and indicates conflicts with these restrictions.

If all these aspects are taken into consideration, the SOMC approach may eventually turn out to be a reasonable replacement or useful assistance for the symbolic and explicit online concept, and enable an even broader range of models to be verified at run-time in the future.
Bibliography


