WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER

Master thesis

Development of an event-based simulator for model checking hybrid Petri nets with random variables

by Carina Pilch

Matriculation number: 414 244
Course of studies: Computer Science (Master of Science)

supervised by:
Prof. Dr. Anne Remke
Prof. Dr. Markus Müller-Olm

University of Münster

Faculty of Mathematics and Computer Science
Group of safety-critical systems

Münster, October 2016

WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER

Masterarbeit

Entwicklung eines ereignisorientierten Simulators zum Model Checking von hybriden Petri-Netzen mit Zufallsvariablen

von Carina Pilch

Matrikelnummer: 414 244
Studiengang: Informatik (Master of Science)

betreut durch:
Prof. Dr. Anne Remke
Prof. Dr. Markus Müller-Olm

Westfälische Wilhelms-Universität Münster

Fachbereich Mathematik und Informatik
Arbeitsgruppe Sicherheitskritische Systeme

Münster, im Oktober 2016

Abstract

Hybrid Petri nets are used for modeling systems with discrete and continuous features. With the help of these models, it is possible to design, observe and analyse this kind of hybrid systems, like for example different networks or critical infrastructures. For the modeling of stochastic variables, so-called general transitions can be used within hybrid Petri nets. Previous works have introduced different approaches to the analysis of models that contain one or two general transitions which can fire (i.e. change the system state) only once.

We extend the formalism for hybrid Petri nets with more than two general transitions firing multiple times. In this thesis, the development of a tool is presented, which is able to simulate and model check these hybrid Petri nets. The tool realizes a discrete-event simulation of such stochastic models. We explain how events affecting the system state are determined and how the general transitions are sampled for single simulation runs. The model checking of properties on the model is realized with techniques of the Statistical Model Checking, like the calculation of confidence intervals for probabilities and hypothesis testing, using the simulation data of several runs. These properties are expressed by the so-called Stochastic Time Logic, which is adopted from previous works and adjusted within this thesis.

Furthermore, the results of the tool are validated against tools from previous works and the feasibility of the approach of the thesis is illustrated within a case study on the charging process of electric vehicle batteries, modeled by a hybrid Petri net with multiple general transitions.

Acknowledgement

First of all, I would like to thank Anne Remke, who has been a great supervisor and caring mentor for me. I thank you for your constant lead, constructive feedback and pleasant support through out my work on this thesis. I am looking forward to work with you in the future.

I would also like to thank Jannik Hüls for his plenty of advises and for sharing his time and the office with me.

Finally, I would like to say thank you to my parents and my sisters for permanently supporting and motivating me in all the years of my studies.

Table of Contents

List of Tables

List of Figures

1 Introduction

Wherever the analysis of real-life systems turns out to be very complex or even impossible, modeling is a proven approach to handle the system by mapping it onto an approximated image. The so-called Petri nets are one formalism to model a wide field of systems. Coming from the area of computer science, they are used in the modeling of information systems and business processes [14], manufacturing processes [16], communication networks [4] and many more fields by now. With the help of simulation, it is possible to examine the behavior of a model and check if it fulfills specified properties. The results of a simulation can then be used for gaining knowledge about or taking decisions on the real system. Within this first chapter, the main motivation for this thesis and the objectives are explained in Section 1.1 and Section 1.2. Section 1.3 gives an overview about related works, forming the fundamentals of this thesis. Finally, Section 1.4 presents the outline of this thesis.

1.1 Motivation

Hybrid stochastic Petri nets are a modeling formalism which is used to model systems with discrete and continuous parts and whose behavior is determined by stochastic variables. Hybrid models allow to model a wide field of different systems. Critical infrastructures are investigated using hybrid Petri nets, like for example a water distribution plant in [19, 21, 24]. These systems are highly safety-critical and modeling allows to explore their behavior without touching the real system. Modeling can also be used to design and plan a system before the implementation. Using stochastic variables, system outages and repair processes can be described and examined, too. So, modeling allows the estimation of the probability for a system to be in a certain state and to fulfill certain properties.

Previous works offer a detailed definition of hybrid Petri nets in [12] and [23, 24] and tools for analyzing models with the restriction to only one or two stochastic variables, modeled by so-called one-shot general transitions in [18, 20]. Transitions are elements of a Petri net that can fire to change the system state. So-called general transitions fire according to stochastic variables. As this limitation restricts the possibility to model any kind of real-life systems, there is still a lack of tools which allow to investigate hybrid Petri nets with multiple random variables. In addition, models of big size and complexity can be challenging for the analysis techniques. Simulation comes as an alternative approach, which provides opportunities to investigate model by replicating its behavior. Hence, information can be collected and concepts can be tested in an experimental environment.

1.2 Research objectives

The main aim of this work is the development of a Java program that is able to simulate and model check hybrid Petri nets containing an unlimited number of general transitions that can fire multiple times. This is supposed to be done by the implementation of a discrete-event simulator, determining and executing the events that can happen within the evolution of a system model. For the stochastic part of the model, a technique for generating samples is required. Based on the simulation runs, it is required to estimate the probabilities for different properties to hold at specified points in time.

This tool is supposed to be integrated into existing tools developed by the group of saftety-critical systems from the University of Münster. Note that in this work, the term simulation is used for statistical simulation and not for the mathematical approach of numerical simulation. The goal is to develop a program within the given framework which simulates the behavior of a given hybrid Petri net model, defined in an XML file (following a particular XML Schema), for a sufficient number of runs and which calculates the probability for system properties expressed in a model checking logic.

1.3 Related work

The recent research focuses on hybrid Petri nets with a general one-shot transitions. In [11], David and Alla introduce hybrid Petri nets, defining their syntax and semantics and presenting different model types. Gribaudo and Remke extend these hybrid Petri nets with so-called one-shot general transitions and separate their deterministic and stochastic behavior in the *Parametric Reachability Analysis* in [23, 24]. The presented pseudo code algorithms give a significant basis for the implementation of the simulation tool of this work. Additionally, in [21] Ghasemieh et. al. introduce a region-based analysis of the state space of hybrid Petri nets with a single one-shot transition (HPnG), using a graphical representation called *Stochastic Time Diagram (STD)*. In [20], Ghasemieh et. al. extend their analysis approach for hybrid Petri nets with multiple general transitions that can fire a finite number of times. Though, it is mentioned that the existing computational geometry libraries are limited to only analyze models with two stochastic firings. Furthermore, the analysis approach does not consider models with probabilistic choices that occur due to firing conflicts between transitions. Hence, an alternative approach on model checking hybrid Petri nets is desirable. Ghasemieh also give approximate techniques for defining the probability space of hybrid Petri nets in [17, Ch.7], which has inspired the approach in this work.

For model checking this kind of HPnGs, Ghasemieh et. al. introduce the *Stochastic Time Logic (STL)* in [19], which allows to express state- and path-based properties. Within this work this logic has been adapted to a branching time logic whose syntax requires a probabilistic operator embracing linear path-based properties.

The model checking of the properties of hybrid Petri nets is done with the methods of *Statistical Model Checking* (SMC). Regarding this, Nimal gives an overview about the $P_{=?}[\Psi]$ problem in [33, Ch.3], where the probability for a property Ψ is estimated. He investigates confidence intervals and on the number of steps needed to reach a desired confidence level. Another issue is the $P_{\bowtie\Theta}[\Psi]$ problem, where it has to be decided if the probability for a property Ψ is greater or lower than a threshold Θ. This problem is solved by hypothesis tests. The basis for these tests is given by Younes in [40, Ch.2]. The *Sequential Probability Ratio Test (SPRT)* is one of the possible approaches. It is also considered in [31] and [33, Ch.5], but both works reference the work of Younes, too. In [13], de Boer et. al. compare different hypothesis tests. The SPRT has been chosen for this work because it is sequential and uses the optimal number of samples for a defined confidence. In addition, all requirements for the use of this technique are given and, according to de Boer et. al., it is the most considered method in previous works.

Younes also gives a definition of discrete-event systems in his work in [40, Ch.2]. In [26, Ch.18], Haverkort presents the concepts of discrete-event simulation. The inversion method, which is used to generate random variables within simulation, is studied by Cheng in [5, Ch.5] and by Devroye in [15, Ch.2]. Concerning Petri nets, Ciardo et. al. have presented a method for the simulation of fluid stochastic Petri nets (FSPNs) and investigated efficient simulation algorithms [7]. In [38], Riley et. al. present a simulation method as well as an adaptive time stepping algorithm for stochastic hybrid systems.

Within this work, a case study is performed on the charging process of electric vehicles. The hybrid Petri net model used for this study and the settings for it are taken from [29], which compares different charging strategies for electric vehicles. The results are compared to existing tools from the same field of research, like the *Fluid Survival Tool* [35] by Postema et. al.

1.4 Outline

Following this introduction, Chapter 2 recalls the fundamental background for this work and gives an overview about the required knowledge for the topics of this thesis. The chapter focuses on the definition of hybrid Petri nets with general transitions and the Stochastic Time Logic and gives an overview about Statistical Model Checking and discrete-event simulation. Chapter 3 presents the single steps of the tool development from a technical view, including preparatory work, the implementation and quality evaluation. Subsequently, the Chapter 4 explains how properties are verified within the software and how the SMC related techniques for the estimation of probabilities are implemented in detail. Chapter 5 then presents the case study, in which the charging process for electric vehicles is simulated by the tool. Finally in Chapter 6, the thesis ends with a closing discussion and a conclusion followed by an outlook for future works.

2 Fundamentals

This chapter presents the fundamental background on Petri nets, on model checking and on discrete-event simulation. It includes a basic explanation of Petri nets and formal definition of hybrid Petri nets with general transitions in Section 2.1. Section 2.2 presents the syntax and semantics of the Stochastic Time Logic (STL) used in this work. Furthermore, Section 2.3 explains the basic approach of discrete-event simulation. Finally, Section 2.4 recalls the relevant approaches of Statistical Model Checking.

2.1 Petri nets

Petri nets are a mathematical modeling formalism developed in the 1960s and named after their German inventor Carl Adam Petri. They are *"a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic"*[32, p.541]. This section gives an overview about the basic Petri net principles and recalls the specific kind of hybrid Petri nets, which are relevant for this work.

2.1.1 Petri net basics

The following definition of Petri nets is based on [12, pp. 1–10] and [32].

Definition 2.1.1 (Petri net) *A Petri net is a directed bipartite graph* $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, W)$ *with an initial state called marking* $\mathbf{M}_0 : \mathcal{P} \rightarrow \mathbb{N}$. *It consists of two kinds of nodes: a finite number of places, the set* $\mathcal{P}$, *and a finite number of transitions, the set* $\mathcal{T}$. *Each place is marked with a non-negative integer number of tokens. The graph's edges are directed arcs from the set* $\mathcal{A}$ *that connect either a place to a transition or a transition to a place. Arcs are associated with positive weights defined in* $W : \mathcal{A} \rightarrow \mathbb{N}$.

Transitions can change the marking of the Petri net when they fire. Only enabled transitions can fire but do not need to fire immediately, but after a deterministic or stochastic time, depending on the type of transition. When an arc from a place $P_i \in \mathcal{P}$ *to a transition* $T_j \in \mathcal{T}$ *has weight* p, T_j *only gets enabled if* P_i *contains at least* p *tokens. When firing* T_j, p *tokens are withdrawn from* P_i. *When an arc from* T_j *to a place* $P_k \in \mathcal{P}$ *has weight* q, q *tokens will be added to* P_k *at the same firing. Transitions without input place are called source transitions and without output place sink transitions.*

So, the firings of the transitions represent the system's events changing the state of the model. There is an apparently infinite field of application areas for Petri nets, such as performance evaluation, communication protocols, distributed systems, industrial control systems, fault-tolerant systems and many more. The basic behavioral properties to study with Petri nets are reachabilty, boundedness, liveness, coverability and persistence. Their major weakness lies in their complexity: Petri net models of moderate size can tend to become too large for analysis and hence often need to be restricted or modified [32].

2.1.2 Hybrid Petri nets with general transitions

While the basic definition of Petri nets is generally restricted to discrete markings, it is also possible to represent continuous system properties using continuous Petri nets. Instead of integer markings, their places hold floating-point number markings and their transitions have a continuous *flow*. A combined model with discrete and continuous features is called *hybrid*. So, a hybrid Petri net can be informally seen as two-parted: the discrete places and non-continuous transitions build the discrete part of the model and the continuous places and transitions form the continuous one. These parts are connected by test or inhibitor arcs, so that both parts can influence the behavior of the other [11]. (Note that these arcs can also exists within only one of these parts.) The following model definition for hybrid Petri nets mainly follows the definition for hybrid Petri nets with general one-shot transitions in [24, 23]. The main difference lies in the multiple-shots property for the general transitions of the Petri nets defined here.

Formal definition of hybrid Petri nets

In the following Definition 2.1.2 for the hypbrid Petri net formalism is given:

Definition 2.1.2 (Hybrid Petri net with general transitions) *A hybrid Petri net with general transitions is a tuple* $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$.

The finite set $\mathcal{P} = \mathcal{P}^d \cup \mathcal{P}^c$ *consists of discrete and continuous places where any discrete place* $P_i \in \mathcal{P}^d = \{\mathcal{P}_1^d, ..., \mathcal{P}_{n_d}^d\}$ *can hold* $m_i \in \mathbb{N}_0$ *tokens and any continuous place* $P_j \in \mathcal{P}^c = \{\mathcal{P}_1^c, ..., \mathcal{P}_{n_c}^c\}$ *has a fluid level* $x_j \in \mathbb{R}_0^+$. *The initial number of tokens and fluid level amounts are defined in the initial marking* $\mathbf{M}_0 = (\mathbf{m}_0, \mathbf{x}_0)$ *with* $\mathbf{m}_0 = \{m_1, ..., m_{n_d}\}$ *and* $\mathbf{x}_0 = \{x_1, ..., x_{n_c}\}$.

$\mathcal{T} = \mathcal{T}^I \cup \mathcal{T}^D \cup \mathcal{T}^G \cup \mathcal{T}^C$ *is the finite set of the so-called immediate, deterministic, general and continuous transitions. An immediate transition* $T_i \in \mathcal{T}^I = \{\mathcal{T}_1^I, ..., \mathcal{T}_{n_I}^I\}$ *fires as soon as it has the status enabled. A deterministic transition* $T_d \in \mathcal{T}^D = \{\mathcal{T}_1^D, ..., \mathcal{T}_{n_D}^D\}$ *fires after a fixed, specified time after it got enabled. The firing time of a general transition* $T_g \in \mathcal{T}^G = \{\mathcal{T}_1^G, ..., \mathcal{T}_{nG}^G\}$ *follows a specified continuous probability distribution function. Any continuous transition* $T_c \in \mathcal{T}^C = \{\mathcal{T}_1^C, ..., \mathcal{T}_{nC}^C\}$ *does not* *fire, but has a continuous inflow and outflow fluid amount. Let* $\mathcal{T}^S = \mathcal{T} \backslash \mathcal{T}^C$ *be the set of non-continuous transitions.*

The finite set of arcs $\mathcal{A} = \mathcal{A}^d \cup \mathcal{A}^f \cup \mathcal{A}^t \cup \mathcal{A}^h$ *holds discrete, continuous, test and inhibitor arcs. Any discrete arc* $A_k \in \mathcal{A}^d \subseteq ((\mathcal{P}^d \times \mathcal{T}^S) \cup (\mathcal{T}^S \times \mathcal{P}^d))$ *can be an input or an output arc for a discrete place and connects it with a non-continuous transition. In the same way, a continuous arc* $A_l \in \mathcal{A}^f \subseteq ((\mathcal{P}^c \times \mathcal{T}^C) \cup (\mathcal{T}^C \times \mathcal{P}^c))$ *connects a continuous place with a continuous transition. A test arc* $A_q \in \mathcal{A}^t \subseteq (\mathcal{P} \times \mathcal{T})$ *connects a place to any kind of transition and the marking of this place has to fulfill a specific condition, defined by the arc's weight, so that the transition gets enabled. In the case of an inhibitor arc* $A_r \in \mathcal{A}^h \subseteq (\mathcal{P} \times \mathcal{T})$*, the transition only gets enabled if the marking of the place does* ***not*** *fulfill the condition. Test and inhibitor arcs are both also called guard arcs. At most one arc for each type can exist between every pair of place and transition.*

The tuple $\Phi = (\Phi_b^{\mathcal{P}}, \Phi_w^{\mathcal{T}}, \Phi_p^{\mathcal{T}}, \Phi_d^{\mathcal{T}}, \Phi_c^{\mathcal{T}}, \Phi_g^{\mathcal{T}}, \Phi_w^{\mathcal{A}}, \Phi_s^{\mathcal{A}}, \Phi_p^{\mathcal{A}})$ *holds nine parameter functions defining the behavior of the Petri net. For every continuous place there is an upper boundary defined in* $\Phi_b^{\mathcal{P}} : \mathcal{P}^c \to \mathbb{R}^+ \cup \infty$ *for the maximum amount of fluid that the place can hold. A place with an upper boundary of* ∞ *can hold any amount.*

Any transition has an assigned weight in $\Phi_w^{\mathcal{T}} : \mathcal{T}^S \to \mathbb{R}^+$ *and a priority* $\Phi_p^{\mathcal{T}} : \mathcal{T}^S \to \mathbb{N}_0$ *solving conflicts between transitions. The function* $\Phi_d^{\mathcal{T}} : \mathcal{T}^D \to \mathbb{R}^+$ *assigns the constant deterministic firing time to the deterministic transitions and* $\Phi_c^{\mathcal{T}} : \mathcal{T}^C \to \mathbb{R}^+$ *defines a constant flow rate for every continuous transition. The cumulative continuous probability distribution functions of the general transitions are defined in* $\Phi_g^{\mathcal{T}} : \mathcal{T}^G \to (f : \mathbb{R}^+ \to [0,1])$.

The function $\Phi_w^{\mathcal{A}} : \mathcal{A} \to \mathbb{R}_0^+$ *assigns weights to all arcs. For discrete input arcs it defines the produced, for discrete output arcs the required number of tokens for firing the connected transition. For guard arcs the weight defines the required number of tokens for fulfilling their condition and for continuous arcs the actual rate is the product of the weight and the transition flow rate. In addition, the continuous arcs have shares defined in* $\Phi_s^{\mathcal{A}} : \mathcal{A}^f \to \mathbb{R}^+$ *and priorities assigned by* $\Phi_p^{\mathcal{A}} : \mathcal{A}^f \to \mathbb{N}_0$ *for sharing limited fluid.*

Note that the lower boundary for continuous places is always zero and hence, there is no parameter function for the lower bounds included in the tuple Φ. The weight and priority for transitions are needed for resolving conflicts of firing between multiple transitions that are about to fire at the same point in time as explained later in this section. Note that, in contrast to [24], the weight and priority are also needed for general transitions as multiple general transitions are allowed that can also result in a conflict. The share and priority for continuous arcs are needed when a continuous place reaches one of its boundaries, so that the fluid input or output has to be reduced to prevent overflow respectively underflow. In detail, the available fluid amount is first distributed to all fluid arcs with the highest priority. Arcs with the second highest priority then have to share the remaining amount. This distribution is continued for the priorities in descending order, until the amount is completely allocated. When the remaining amount is not sufficient for the current priority, it is distributed to the arcs with this priority according to their defined share and fluid rate of the connected transition.

The places are commonly represented by circles ad the transitions by bars. A possible graphical representation of the places, transitions and arcs is shown in Figure 2.1.

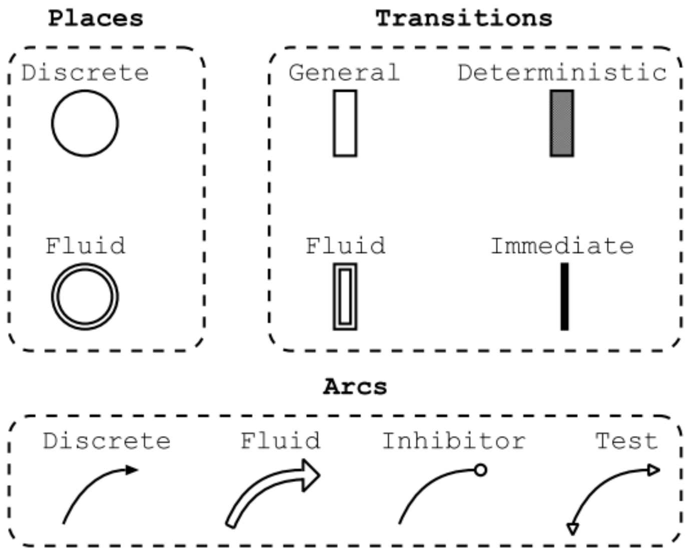


Figure 2.1: Primitives for the hybrid Petri net formalism with general one-shot transitions. Source: [23, p.85]

States of a hybrid Petri net

Referring to [20], the state of a hybrid Petri net with multiple general transitions can be defined as in the following Definition 2.1.3:

Definition 2.1.3 (State of a hybrid Petri net) *The state of a hybrid Petri net with general transitions* $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$ *is denoted by a vector* $\Gamma = (\mathbf{m}, \mathbf{x}, \mathbf{c}, \mathbf{d}, \mathbf{g}, \mathbf{e})$, *where the vector* $\mathbf{m} = (m_1, ..., m_{|\mathcal{P}^d|})$ *with* $\forall i$ *with* $1 < i < |\mathcal{P}^d| : m_i \in \mathbb{N}_0$ *contains the discrete markings (tokens) and* $\mathbf{x} = (x_1, ..., x_{|\mathcal{P}^c|})$ *with* $\forall i$ *with* $1 < i < |\mathcal{P}^c| : x_i \in \mathbb{R}_0^+$ *describes the continuous marking (fluid levels). For each deterministic transition the time since it has been enabled (called clock) is included in* $\mathbf{c} = (c_1, ..., c_{|\mathcal{T}^D|})$ *with* $\forall i$ *with* $1 < i < |\mathcal{T}^D| : c_i \in \mathbb{R}_0^+$. *Clocks do not evolve when the transition is disabled* *but their values are preserved until it is enabled again. The clocks are only reset at firing. The vector* $\mathbf{d} = (d_1, ..., d_{|\mathcal{P}^c|})$ *with* $\forall i$ *with* $1 < i < |\mathcal{P}^c|$: $d_i \in \mathbb{R}$ *holds the drift of each continuous place, which is defined as the change of its fluid level per time unit. The state also includes the enabling status (depending on the marking and set according to the corresponding arc weights) of the transitions in the vector* $\mathbf{e} = (e_1, ..., e_{|\mathcal{T}|})$ *with* $\forall i$ *with* $1 < i < |\mathcal{T}| : e_i \in \{0, 1\}$. *The vector* $\mathbf{g} = (g_1, ..., g_{|\mathcal{T}^G|})$ *with* $\forall i$ *with* $1 < i < |\mathcal{T}^G| : g_i \in \mathbb{R}_0^+$ *holds the enabling time for the general transitions, which is set back to zero after firing.*

Note that the drift and the enabling status are both determined uniquely by the vectors $\mathbf{m}$ and $\mathbf{x}$ and by $\Phi_w^{\mathcal{A}}$, but included in the given definition for simplification. While Gribaudo and Remke in [23, 24] and Ghasemieh et al. in [19] restrict their definitions to hybrid Petri nets with only a single one-shot general transition, the models used within this work are allowed to include multiple general transitions, which can fire more than once. For this reason the vector g has been added.

Definition 2.1.4 (Initial state of a hybrid Petri net) *Let* $S = \{\Gamma = (\mathbf{m}, \mathbf{x}, \mathbf{c}, \mathbf{d}, \mathbf{g}, \mathbf{e})\}$ *be the set of all states that can hold in a hybrid Petri net with general transitions* $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$. *The initial state of the model is defined as* $\Gamma_0 \in S$ *with* $\Gamma_0 = (\mathbf{m}_0, \mathbf{x}_0, \mathbf{0}, \mathbf{d}_0, \mathbf{0}, \mathbf{e}_0) = (\mathbf{M}_0, \mathbf{0}, \mathbf{d}_0, \mathbf{0}, \mathbf{e}_0)$.

Behavior of general transitions

A general transition can be considered as a deterministic transition with firing times that are sampled from their distributions. As in [20], a general transition with the probability density $g_k(s)$ fires with probability $\int_\tau^{\tau+\Delta\tau} g_k(s)ds$ in the interval $[\tau, \tau + \Delta\tau]$. Note that at the firing of a general transition, a new firing time is always reset to zero while $g_k(s)$ stays the same.

In case a general transition is disabled and enabled, there are three policies which are possible [28]:

1. *Resume:* when disabled, the enabling time will be preserved and resumed with the same firing time when the transition is enabled again.

2. *Repeat identical:* when the transition is disabled and enabled again, the firing time will remain the same but the enabling time will be reset to zero when the transition is enabled again.

3. *Repeat different:* the general transition will fire with a different firing time according to the same density function when it is enabled again.

Conflict resolution

Only one transition can fire at a time. If multiple transitions of the same kind (immediate or deterministic or general) are supposed to fire at the exact same point in time, the one with the highest priority is fired. In case multiple transitions have the same priority, all of them are candidates for the firing.

In a conflict of immediate transitions, let $C^I(s)$ be the set of transitions with the same maximum priority in a state s. For an immediate transition $T_i^I \in C^I(s)$ the probability to fire according to [24] is:

$$p(T_i^I) = \frac{\Phi_w^{\mathcal{T}}(T_i^I)}{\displaystyle\sum_{\forall T_j^I \in C^I(s)} \Phi_w^{\mathcal{T}}(T_j^I)} \tag{2.1}$$

The formula is equivalent for deterministic or general transitions for a set $C^D(s,t)$ respectively $C^G(s,t)$ holding all deterministic respectively general transitions that fire at time t and have the maximum priority.

Events

The given Definition 2.1.5 for an event in the evolution of a hybrid Petri net is based on [20, 20], but extended to the purpose of this work by defining how the state of a system changes and which states lie in between two events.

Definition 2.1.5 (Event) *An event $\Upsilon(\Gamma_i, \Gamma_{i+1}) = (\Delta\tau_i, \varepsilon_i)$ with $\varepsilon_i \in \mathcal{P}^c \cup \mathcal{T}^S \cup \mathcal{A}^t \cup \mathcal{A}^h$ denotes the change of state of the hybrid Petri net with general transitions $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$ from a state $\Gamma_i = (\mathbf{m}_i, \mathbf{x}_i, \mathbf{c}_i, \mathbf{d}_i, \mathbf{g}_i, \mathbf{e}_i)$ to a state $\Gamma_{i+1} = (\mathbf{m}_{i+1}, \mathbf{x}_{i+1}, \mathbf{c}_{i+1}, \mathbf{d}_{i+1}, \mathbf{g}_{i+1}, \mathbf{e}_{i+1})$ such that $\Delta\tau_i \in \mathbb{R}_0^+$ is the time after which the event occurs starting from Γ_i and one of the following conditions is fulfilled:*

1. $\mathbf{m}_i \neq \mathbf{m_{i+1}} \wedge \varepsilon_i \in \mathcal{T}^S$ *(transition firing)*
2. $\mathbf{d}_i \neq \mathbf{d_{i+1}} \wedge \varepsilon_i \in \mathcal{P}^c$ *(boundary reached)*
3. $\mathbf{e}_i \neq \mathbf{e_{i+1}} \wedge \varepsilon_i \in \mathcal{A}^t \cup \mathcal{A}^h$ *(guard arc condition fulfilled)*

ε_i denotes the model element that causes the event.

So the first condition describes the firing of a non-continuous transition changing the discrete marking of the model (vector $\mathbf{m}$). The second condition is fulfilled if the upper or lower boundary of a continuous place is reached, which results in an adaption of the drift, hence vector $\mathbf{d}$. The third condition describes the event when the fluid level of a continuous place reaches the weight of a connected guard arc, so that the arc's condition is fulfilled. This changes the enabling status of the connected transition, hence vector **e**. Note that the change of the fluid levels and of the clock values and enabling times are part of the continuous evolution of the system but do not describe events. So due to its continuous part, there is an infinite amount of states in a hybrid Petri net.

Definition 2.1.6 (Set of events) *Let $\mathcal{E}(\Gamma_i) = \{\Upsilon(\Gamma_i, \Gamma_{i+1}) = (\Delta\tau_i, \varepsilon_i)\}$ be the set of events that can occur directly from a state Γ_i, having $\mathcal{E}(\Gamma_i) = \mathcal{E}^{\mathcal{T}}(\Gamma_i) \cup \mathcal{E}^{\mathcal{P}}(\Gamma_i) \cup \mathcal{E}^{\mathcal{A}}(\Gamma_i)$. The set $\mathcal{E}^{\mathcal{T}}(\Gamma_i) = \mathcal{E}^{\mathcal{T}^D}(\Gamma_i) \cup \mathcal{E}^{\mathcal{T}^G}(\Gamma_i) \cup \mathcal{E}^{\mathcal{T}^I}(\Gamma_i)$ is the set of events corresponding to the firing of deterministic, general or immediate transitions. $\mathcal{E}^{\mathcal{P}}(\Gamma_i)$ is the set of events corresponding to a continuous place reaching a boundary. The set $\mathcal{E}^{\mathcal{A}}(\Gamma_i) = \mathcal{E}^{\mathcal{A}^G}(\Gamma_i) \cup \mathcal{E}^{\mathcal{A}^D}(\Gamma_i) \cup \mathcal{E}^{\mathcal{A}^D}(\Gamma_i) \cup \mathcal{E}^{\mathcal{A}^C}(\Gamma_i)$ is the set of guard arcs events partitioned by the type of transitions that is controlled by the arc.*

Table 2.1 shows the possible events that can occur in a hybrid petri net at a state Γ including the order in which they are considered. Note that the last case is adjusted for hybrid Petri nets with multiple general transitions.

Order	**Event type**	**Effects**
I	Firing of immediate transition: $\Upsilon_k^{\mathcal{T}^I} = (\Delta\tau, T_k^I) \in \mathcal{E}^{\mathcal{T}^I}(\Gamma)$	1. Solve immediate conflict 2. Fire immediate transition T_k^I 3. Perform rate adaption
II	Guard arc for immediate transition: $\Upsilon_k^{\mathcal{A}^I} = (\Delta\tau, A_k^I) \in \mathcal{E}^{\mathcal{A}^I}(\Gamma)$	1. Advance continuous marking of $\Delta\tau$
III	Firing of deterministic transition: $\Upsilon_k^{\mathcal{T}^D} = (\Delta\tau, T_k^D) \in \mathcal{E}^{\mathcal{T}^D}(\Gamma)$	1. Advance continuous marking of $\Delta\tau$ 2. Solve deterministic conflict 3. Fire deterministic transition T_k^D 4. Perform rate adaption
IV	Boundary of continuous place: $\Upsilon_k^{\mathcal{P}^c} = (\Delta\tau, P_k^c) \in \mathcal{E}^{\mathcal{P}^c}(\Gamma)$ Guard arc for continuous transition: $\Upsilon_k^{\mathcal{A}^C} = (\Delta\tau, A_k^C) \in \mathcal{E}^{\mathcal{A}^C}(\Gamma)$	1. Advance continuous marking of $\Delta\tau$ 2. Perform rate adaption
V	Guard arc for deterministic transition: $\Upsilon_k^{\mathcal{A}^D} = (\Delta\tau, A_k^D) \in \mathcal{E}^{\mathcal{A}^D}(\Gamma)$ Guard arc for general transition: $\Upsilon_k^{\mathcal{A}^G} = (\Delta\tau, A_k^G) \in \mathcal{E}^{\mathcal{A}^G}(\Gamma)$	1. Advance continuous marking of $\Delta\tau$
VI	Firing of general transition: $\Upsilon_k^{\mathcal{T}^G} = (\Delta\tau, T_k^G) \in \mathcal{E}^{\mathcal{T}^G}(\Gamma)$	1. Advance continuous marking of $\Delta\tau$ 2. Solve general conflict 3. Fire general transition T_k^G 4. Perform rate adaption

Table 2.1: Event execution order and events. Source: own representation based on [24, Table 1, p.17]

The set of events occurring at the minimum time is defined as follows:

Definition 2.1.7 (Set of events with minimum remaining time) *Let* $\mathcal{E}^{min}(\Gamma_i) = \{\Upsilon_j(\Gamma_i, \Gamma_j) = (\Delta\tau_j, \varepsilon_j) \in \mathcal{E}(\Gamma_i) \mid \nexists \Upsilon_k(\Gamma_i, \Gamma_k) = (\Delta\tau_k, \varepsilon_k) \in \mathcal{E}(\Gamma_i) : \Delta\tau_k < \Delta\tau_j\}$ *denote the set of events with the minimum remaining time that can occur when in state* Γ_i*.*

There is an infinite number of states that hold during the evolution of the continuous state variables between two events. This set is defined in given by:

Definition 2.1.8 (Set of states between two events) *Let the state* $\Gamma_{i+1} \in S$ *be a state rising from an event* $\Upsilon(\Gamma_i, \Gamma_{i+1}) = (\Delta\tau_i, \varepsilon_i) \in \mathcal{E}^{min}(\Gamma_i)$*. The set of possible states that lie in between* Γ_i *and* Γ_{i+1} *is denoted by* $S(\Gamma_i, \Gamma_{i+1}) = \{\Gamma_j = (\mathbf{m}_j, \mathbf{x}_j, \mathbf{c}_j, \mathbf{d}_j, \mathbf{g}_j, \mathbf{e}_j) \in S \mid \mathbf{m}_j = \mathbf{m}_i \wedge \mathbf{d}_j = \mathbf{d}_i \wedge \mathbf{e}_j = \mathbf{e}_i \wedge \exists \Delta\tau_j : 0 \leq \Delta\tau_j < \Delta\tau_i \wedge \mathbf{x}_j = \mathbf{x}_i + \mathbf{d}_i \cdot \Delta\tau_j \wedge \mathbf{c}_j = \mathbf{c}_i + \Delta\tau_j \wedge \mathbf{g}_j = \mathbf{g}_i + \Delta\tau_j\}$*. Let this* $\Delta\tau_j$ *be denoted by* $\Delta\tau(\Gamma_j)$*.*

In [20], Ghasemieh et al. define the state of a system in a similar way (including the discrete and continuous marking). They partition the state space into regions that lie in the Stochastic Time Diagram, so that different regions are separated by events. Here, we have defined the set of states, which is the equivalent to a region. The number of set of states is finite for a finite time bound, as long as Zeno behavior [27] caused by cycles of immediate transitions is excluded by the model.

Paths

With the definition of states, it is possible to define a sequence of states called path. This is done in Definition 2.1.9, based on a similar approach in [2].

Definition 2.1.9 (Path) *A path is defined as a finite or an infinite sequence* $\Gamma_0, t_0, \Gamma_1, t_1, \Gamma_2, t_2, ...$ *written as*

$$\sigma = \Gamma_0 \xrightarrow{t_0} \Gamma_1 \xrightarrow{t_1} \Gamma_2 \xrightarrow{t_2} ...,$$

such that for every $i \in \mathbb{N} : \exists \Upsilon(\Gamma_i, \Gamma_{i+1}) = (\Delta\tau_i, \varepsilon_i) \in \mathcal{E}^{min}(\Gamma_i)$ *with* $\Delta\tau_i = t_i$ *the time spent in states of the set* $S(\Gamma_i, \Gamma_{i+1})$ *before the event happens.*

Definition 2.1.10 defines the state that holds in a path at a specific point in time. In addition, Definition 2.1.11 defines the set of paths running out from Γ_0 and the subset of this set with paths of a specified time length.

Definition 2.1.10 (State in a path) *Let $s(\sigma, t)$ denote the state that holds at a specified point in time within the evolution of a single path in a hybrid Petri net. It is defined as:*

$$s(\sigma, t) = \Gamma_k = (\mathbf{m}_k, \mathbf{x}_k, \mathbf{c}_k, \mathbf{d}_k, \mathbf{g}_k, \mathbf{e}_k) \in S(\Gamma_i, \Gamma_{i+1}),$$
$$\textit{with } i = min\{j \in \mathbb{N} : t \leq \sum_{0 \leq l \leq j} t_l\} \textit{ and } \Delta\tau(\Gamma_k) = t - \sum_{0 \leq l \leq i-1} t_l.$$

Definition 2.1.11 (Set of paths starting from a state) *Let the set of paths starting from the initial state be defined as $Paths(\Gamma_0) = \{\sigma = \Gamma_0 \xrightarrow{t_0} \Gamma_1 \xrightarrow{t_1} \Gamma_2 \xrightarrow{t_2} ...\}$ such that $\Gamma_i \in S$ is the state of the system after the i-th event $\Upsilon(\Gamma_{i-1}, \Gamma_i)$.*

Let $Paths(\Gamma_0, t)$ be the set of all finite paths up to time t, so $Paths(\Gamma_0, t) = \{\sigma = \Gamma_0 \xrightarrow{t_0} \Gamma_1 \xrightarrow{t_1} ... \xrightarrow{t_{k-1}} \Gamma_k \xrightarrow{t_k} \in Paths(\Gamma_0) \mid \sum_{i=0}^{k-1} t_i \leq t < \sum_{i=0}^{k} t_i\}$.

2.2 Stochastic Time Logic (STL)

Model checking is an *"automatic technique for verifying finite state concurrent systems"* [8, p.1]. It has shown its plenty advantages over traditional methods in practice where reliability of hardware and software is needed. The main tasks of model checking involve modeling, specification and verification. Specification means stating the properties on which the model is verified to satisfy. A common formalism for this is temporal logic which can describe the behavior of the systems and the ordering of events over time [8, pp.1–4].

In [19], Ghasemieh et al. introduce a so-called Stochastic Time Logic (STL) that allows the specification of properties for hybrid Petri nets. The syntax and semantics stated here are inspired by [19, 39], but are adjusted to the object of this work. A probabilistic operator is added which allows to define a probability bound for specified properties. The syntax for STL is given in Section 2.2.1 and the semantics in Section 2.2.2. Furthermore, the probability space for the evolution of a hybrid Petri net is defined in 2.2.3.

2.2.1 STL syntax

The following Definition 2.2.1 presents the syntax for the Stochastic Time Logic:

Definition 2.2.1 (STL syntax) *An STL formula for a hybrid Petri net* $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$ *is defined as follows:*

$$\varphi ::= P_{\bowtie p}(\Psi),$$

with $\bowtie \in \{<, > \leq, \geq\}$, $p \in [0,1]$ *and*

$$\Psi ::= tt \mid AP \mid \neg\Psi \mid \Psi \wedge \Psi \mid \Psi U^{[t_1,t_2]}\Psi,$$

having $t_1, t_2 \in \mathbb{R}_0^+$ *with* $t_1 \leq t_2$ *and*

$$AP ::= x_P \sim a \mid m_P \sim b \mid c_T \sim c \mid d_P \sim d \mid g_T \sim e \mid e_T \ ,$$

with $\sim \in \{=, <, > \leq, \geq\}$, $a, c, e \in \mathbb{R}_0^+$ *and* $b \in \mathbb{N}$ *and* $d \in \mathbb{R}$.

Note that, though the given STL syntax does not include the *OR* case, it can be derived according to De Morgan's law by $(\Psi_1 \vee \Psi_2) := \neg(\neg\Psi_1 \wedge \neg\Psi_2)$. The syntax includes so-called atomic properties (*AP*). As in [19], the atomic properties include the discrete and continuous marking of places as well as the clock values of deterministic transitions and the drift of continuous places. We extend the set of APs by enabling times of general transitions and enabling status of any kind of transition. In the syntax of STL, we have:

- $x_P \sim a$, comparing the fluid level x_P of a continuous place P to a value a.
- $m_P \sim b$, comparing the marking m_P of a discrete place P to a value b.
- $c_T \sim c$, comparing the clock value c_T of a deterministic transition T to a value c.
- $d_P \sim d$, comparing the drift d_P of a continuous place P to a value d.
- $g_T \sim e$, comparing the enabling time g_T of a general transition T to a value e.
- e_T, checking the enabling status of a transition.

2.2.2 STL semantics

Semantics for STL are defined via a satisfaction relation for an initial state of a hybrid Petri net, see Definition 2.2.2. Note that the relation $\sigma \models^t \Psi$ is a short denotation for the common satisfaction relation for paths $\sigma \models tt\ U^{[t,t]}\Psi$.

Definition 2.2.2 (STL semantics) *Let $\Gamma_0 = (\mathbf{m}_0, \mathbf{x}_0, \mathbf{0}, \mathbf{d}_0, \mathbf{0}, \mathbf{e}_0)$ be the initial state of a hybrid Petri net $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$. The satisfaction relation $\models^t$ for STL is then defined as follows:*

$$\Gamma_0 \models^t P_{\bowtie p}(\Psi) \quad \textit{iff } Pr\{\sigma \in Paths(\Gamma_0) \mid \sigma \models^t \Psi\} \bowtie p,$$

with $t \in \mathbb{R}_0^+$, $\bowtie \in \{<, > \leq, \geq\}$ and $p \in [0, 1]$.

For a given path $\sigma \in Paths(\Gamma_0)$ the satisfaction relation $\models^t$ is defined as:

$$\begin{array}{ll}
\sigma \models^t tt & \forall t, \\
\sigma \models^t x_P \sim a & \textit{iff } \Gamma_i.x_P \sim a \textit{ for } \Gamma_i = s(\sigma, t), \\
\sigma \models^t m_P \sim b & \textit{iff } \Gamma_i.m_P \sim b \textit{ for } \Gamma_i = s(\sigma, t), \\
\sigma \models^t c_T \sim c & \textit{iff } \Gamma_i.c_T \sim c \textit{ for } \Gamma_i = s(\sigma, t), \\
\sigma \models^t d_P \sim d & \textit{iff } \Gamma_i.d_P \sim d \textit{ for } \Gamma_i = s(\sigma, t), \\
\sigma \models^t g_T \sim e & \textit{iff } \Gamma_i.g_T \sim e \textit{ for } \Gamma_i = s(\sigma, t), \\
\sigma \models^t e_T & \textit{iff } \Gamma_i.e_T = 1 \textit{ for } \Gamma_i = s(\sigma, t), \\
\sigma \models^t \neg\Psi & \textit{iff } \sigma \not\models^t \Psi, \\
\sigma \models^t \Psi_1 \wedge \Psi_2 & \textit{iff } \sigma \models^t \Psi_1 \wedge \sigma \models^t \Psi_2, \\
\sigma \models^t \Psi_1 U^{[t_1, t_2]} \Psi_2 & \textit{iff } \exists \tau \in [t + t_1, t + t_2] : \sigma \models^\tau \Psi_2 \wedge (\forall \tau' \in [t, \tau) : \sigma \models^{\tau'} \Psi_1),
\end{array}$$

with $\sim \in \{=, <, > \leq, \geq\}$, $t, a, c, e, t_1, t_2 \in \mathbb{R}_0^+$ with $t_1 \leq t_2$ and $b \in \mathbb{N}$ and $d \in \mathbb{R}$.

STL semantics for nested until formulas

Contrary to [19], nested until formulas are allowed here. Though the semantics for nested until formulas can be derived from Definition 2.2.2, in the following Definition 2.2.3 the two cases of one inner and one outer until formula are explicitly stated.

Definition 2.2.3 (nested STL until semantics) *Let the state $\Gamma_0 = (\mathbf{m}_0, \mathbf{x}_0, \mathbf{0}, \mathbf{d}_0, \mathbf{0}, \mathbf{e}_0)$ be the initial state of a hybrid Petri net $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$. For a given $\sigma \in Paths(\Gamma_0)$ we define the satisfaction relation $\models^t$ for nested until formulas as follows:*

$$\begin{aligned}
&\sigma \models^t (\Psi_1 U^{[t_1, t_2]} \Psi_2) U^{[t_3, t_4]} \Psi_3 \\
&\quad \textit{iff } \exists \tau \in [t + t_3, t + t_4] : \sigma \models^\tau \Psi_3 \wedge (\forall \tau' \in [t, \tau) : (\exists \upsilon \in [\tau' + t_1, \tau' + t_2] : \\
&\quad \sigma \models^\upsilon \Psi_2 \wedge (\forall \upsilon' \in [\tau', \upsilon) : \sigma \models^{\upsilon'} \Psi_1))) \textit{ for } t_1, t_2, t_3, t_4 \in \mathcal{R}_0^+ \textit{ with } t_1 \leq t_2 \\
&\quad \textit{and } t_3 \leq t_4.
\end{aligned}$$

$\sigma \models^{t} \Psi_1 U^{[t_1,t_2]} (\Psi_2 U^{[t_3,t_4]} \Psi_3)$
iff $\exists \tau \in [t+t_1, t+t_2] : (\exists \upsilon \in [\tau + t_3, \tau + t_4] : \sigma \models^{\upsilon} \Psi_3 \wedge (\forall \upsilon' \in [\tau, \upsilon) : \sigma \models^{\upsilon'} \Psi_2)) \wedge (\forall \tau' \in [t, \tau) : \sigma \models^{\tau'} \Psi_1)$ *for* $t_1, t_2, t_3, t_4 \in \mathcal{R}_0^+$ *with* $t_1 < t_2$ *and* $t_3 < t_4$.

Figure 2.2 and 2.3 give an illustration of the time line for these two nested until cases. For the $(\Psi_1 U^{[t_1,t_2]} \Psi_2) U^{[t_3,t_4]} \Psi_3$ case Figure 2.2 shows that there must be a point in time τ within the outer until interval at which Ψ_3 is fulfilled and at all $\tau' < \tau$ the inner until property must be fulfilled as well to satisfy the whole property. There might be points in time for τ' for which the inner until interval lies (i) before τ (as in the figure) or (ii) after it or (iii) so that τ lies within this interval.

Figure 2.3 shows two different cases for $\Psi_1 U^{[t_1,t_2]} (\Psi_2 U^{[t_3,t_4]} \Psi_3)$. There must be a point in time τ at which the inner until formula is fulfilled and for all τ' before τ Ψ_1 has to be fulfilled. The inner until interval can lie (i) behind or (ii) within (as in the figure) or (iii) just partly within the outer until interval.

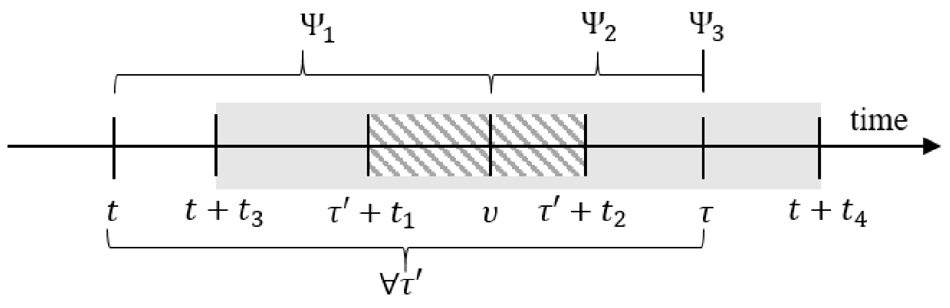


Figure 2.2: Time line for the nested until property (case 1). Source: own representation

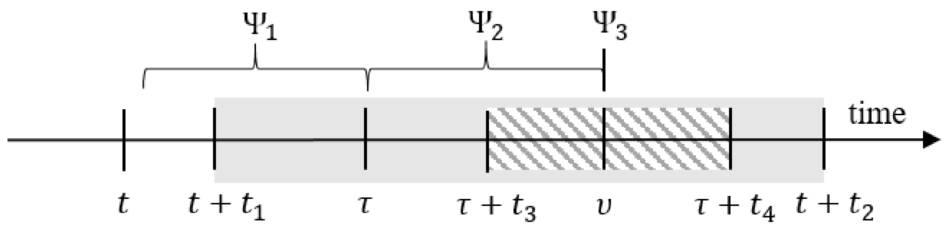


Figure 2.3: Time line for the nested until property (case 2). Source: own representation

2.2.3 Definition of the probability space

What is still left open so far is the question how to determine the probability $\Pr\{\sigma \in \mathit{Paths}(\Gamma_0) \mid \sigma{\models}^t\Psi\}$. The following approach is inspired by the definition of stochastic semantics in [10] and also by the techniques of Ghasemieh in [17, pp.97–113].

Let $\sigma_k = \Gamma_0 \xrightarrow{t_0} \Gamma_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-1}} \Gamma_k$ denote a finite path in $\mathit{Paths}(\Gamma_0)$ with $k+1$ states. Providing the basic elements of a σ-algebra, it is possible to inductively define the following measure for probability:

$$\mathit{Prob}(\sigma_k) = p(\Gamma_{k-1}, \Gamma_k) \cdot \mathit{Prob}(\sigma_{k-1}), \tag{2.2}$$

with $\sigma_{k-1} = \Gamma_0 \xrightarrow{t_0} \Gamma_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{k-2}} \Gamma_{k-1}$. Hence, we yield:

$$\Pr\{\sigma \in \mathit{Paths}(\Gamma_0) \mid \sigma{\models}^t\Psi\} = \sum_{\sigma_k \in \mathit{Paths}(\Gamma_0, t)} \mathbf{1}_\Psi(\sigma_k, t) \cdot \mathit{Prob}(\sigma_k), \tag{2.3}$$

with

$$\mathbf{1}_\Psi(\sigma_k, t) = \begin{cases} 1 & \text{if } \sigma_k{\models}^t\Psi, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

The different cases for calculating $p(\Gamma_{k-1}, \Gamma_k)$ are given in the following:

Definition 2.2.4 (Probability to move between states) *Let $\Gamma_{k-1} = (\mathbf{m}_{k-1}, \mathbf{x}_{k-1}, \mathbf{c}_{k-1}, \mathbf{d}_{k-1}, \mathbf{e}_{k-1}, \mathbf{g}_0)$ be the current state in a hybrid Petri net $(\mathcal{P}, \mathcal{T}, \mathcal{A}, \mathbf{M}_0, \Phi)$.*

Let $\mathcal{T}^G_{enabled}(\Gamma_{k-1})$ be the set of enabled general transitions in Γ_{k-1}. Let $C^I(\Gamma_{k-1}) \subset \mathcal{T}^I$, $C^D(\Gamma_{k-1}, t_{k-1}) \subset \mathcal{T}^D$ and $C^G(\Gamma_{k-1}, t_{k-1}) \subset \mathcal{T}^G$ be the conflict sets for transitions enabled in Γ_{k-1} to fire at the same time. Let also be $\mathcal{A}^I \subset \mathcal{A}^t \cup \mathcal{A}^h$ respectively $\mathcal{A}^C \subset \mathcal{A}^t \cup \mathcal{A}^h$ respectively $\mathcal{A}^D \subset \mathcal{A}^t \cup \mathcal{A}^h$ respectively $\mathcal{A}^G \subset \mathcal{A}^t \cup \mathcal{A}^h$ be the set of guard arcs connected to immediate, continuous, deterministic or general transitions.

The probability $p(\Gamma_{k-1}, \Gamma_k)$ to move from a state Γ_{k-1} to a state Γ_k with one event is then given by:

$$
p(\Gamma_{k-1},\Gamma_k) =
\begin{cases}
\dfrac{\Phi_w^{\mathcal{T}}(\varepsilon_{k-1})}{\sum_{\forall\alpha\in C^I(\Gamma_{k-1})}\Phi_w^{\mathcal{T}}(\alpha)}, \\
\qquad \textit{if } \exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (0,\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{T}^I. \\[2ex]
\dfrac{\Phi_w^{\mathcal{T}}(\varepsilon_{k-1})}{\sum_{\forall\alpha\in C^D(\Gamma_{k-1},t_{k-1})}\Phi_w^{\mathcal{T}}(\alpha)} \cdot \prod_{\beta\in\mathcal{T}^G_{enabled}(\Gamma_{k-1})} \int_{\tau\geq(t_{k-1}+\mathbf{g}_{k-1}.\beta)} \Phi_g^{\mathcal{T}}(\beta)(\tau)d\tau, \\
\qquad \textit{if } \exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{T}^D \\
\qquad \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_l) = (t_{l-1},\varepsilon_{l-1}) \in \mathcal{E}(\Gamma_{k-1}) \setminus \mathcal{E}^G(\Gamma_{k-1}) \textit{ with } t_{l-1} < t_{k-1} \\
\qquad \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I\cup\mathcal{A}^I \wedge t_{j-1}=t_{k-1}). \\[2ex]
\prod_{\beta\in\mathcal{T}^G_{enabled}(\Gamma_{k-1})} \int_{\tau\geq(t_{k-1}+\mathbf{g}_{k-1}.\beta)} \Phi_g^{\mathcal{T}}(\beta)(\tau)d\tau, \\
\qquad \textit{if } ((\exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{A}^I \\
\qquad\quad \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I \wedge t_{j-1}=t_{k-1})) \\
\qquad \vee (\exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{P}^c \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) \\
\qquad\quad = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I\cup\mathcal{T}^D\cup\mathcal{A}^I \wedge t_{j-1}=t_{k-1})) \\
\qquad \vee (\exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{A}^C \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) \\
\qquad\quad = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I\cup\mathcal{T}^D\cup\mathcal{P}^c\cup\mathcal{A}^I \wedge t_{j-1}=t_{k-1})) \\
\qquad \vee (\exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{A}^D \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) \\
\qquad\quad = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I\cup\mathcal{T}^D\cup\mathcal{P}^c\cup\mathcal{A}^I\cup\mathcal{A}^C \wedge t_{j-1}=t_{k-1})) \\
\qquad \vee (\exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{A}^G \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) \\
\qquad\quad = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I\cup\mathcal{T}^D\cup\mathcal{P}^c\cup\mathcal{A}\setminus\mathcal{A}^G \wedge t_{j-1}=t_{k-1}))) \\
\qquad \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_l) = (t_{l-1},\varepsilon_{l-1}) \in \mathcal{E}(\Gamma_{k-1}) \setminus \mathcal{E}^G(\Gamma_{k-1}) \textit{ with } t_{l-1} < t_{k-1} \\
\qquad . \\[2ex]
\int_{t_{k-1}\geq\mathbf{g}_{k-1}.\varepsilon_{k-1}} \left(\Phi_g^{\mathcal{T}}(\varepsilon_{k-1})(t_{k-1}) \cdot \dfrac{\Phi_w^{\mathcal{T}}(\varepsilon_{k-1})}{\sum_{\forall\alpha\in C^G(\Gamma_{k-1},t_{k-1})}\Phi_w^{\mathcal{T}}(\alpha)} \right. \\
\qquad\qquad\qquad \left. \cdot \prod_{\beta\in\mathcal{T}^G_{enabled}(\Gamma_{k-1})} \int_{\tau>(t_{k-1}+\mathbf{g}_{k-1}.\beta)} \Phi_g^{\mathcal{T}}(\beta)(\tau)d\tau \right) dt_{k-1}, \\
\qquad \textit{if } \exists\Upsilon(\Gamma_{k-1},\Gamma_k) = (t_{k-1},\varepsilon_{k-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } \varepsilon_{k-1}\in\mathcal{T}^G \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_j) \\
\qquad = (t_{j-1},\varepsilon_{j-1}) \in \mathcal{E}(\Gamma_{k-1}) \textit{ with } (\varepsilon_{j-1}\in\mathcal{T}^I\cup\mathcal{T}^D\cup\mathcal{P}^c\cup\mathcal{A} \wedge t_{j-1}=t_{k-1}) \\
\qquad \wedge \nexists\Upsilon(\Gamma_{k-1},\Gamma_l) = (t_{l-1},\varepsilon_{l-1}) \in \mathcal{E}(\Gamma_{k-1}) \setminus \mathcal{E}^G(\Gamma_{k-1}) \textit{ with } t_{l-1} < t_{k-1}. \\[2ex]
0 \qquad \textit{otherwise.}
\end{cases}
\tag{2.5}
$$

The previous equation of Definition 2.2.4 follows the cases presented in Table 2.1. The first case is related to the firing of an immediate transition which is the event of the highest order (Order I in the table). This event occurs immediately. The probability for one immediate transition to fire is obtained by solving the conflict resolution for all enabled immediate transition.

The second case is related to the firing of a deterministic transition (Order III), composed of the conflict resolution for deterministic transitions for a given time t_{k-1} times the probability that none of the enabled general transitions fires before t_{k-1}. This event only occurs if there is no other event happening before and no immediate firing at the same time (which is the case if $t_{k-1} = 0$).

The third case covers the place boundary and guard arc events (Order II, IV and V), occuring if there are no other events happens before and no immediate or deterministic transition fires at the same time (only guard arcs of immediate transitions are of higher order than deterministic transitions). The probability for these events correlates to the probability that no enabled general transition fires before.

The fourth case is the firing of the general transition (Order VI). The probability is a integration of the probability densities of possible firing times that are multiplied by the probability that no other general transition fires before times the probability drawn from the conflict resolution. This event occurs if no other event occurs before or at the same time. In case that none of the above four cases has all conditions fulfilled, the probability to reach Γ_k from Γ_{k-1} is zero.

2.3 Discrete-event simulation

Simulation is an approach used to investigate a system model by replicating its behavior. It allows to gain insight into the system, testing model concepts and collect information. Simulation allows experimentation in short time and is in many cases easier to execute than analytic approaches [6, Ch.1.4, 1.5]. This section gives an overview about discrete-event systems in 2.3.1 and event-based simulation in 2.3.2. The concept presented in these sections mainly follows [26, pp.412–418]. In 2.3.3, a technique for the generation of random variables from probability distribution functions is explained.

2.3.1 Discrete-event systems

A stochastic system or process evolves over time with a predictable behavior. At any time it holds a specific state and the system's behavior is determined by probabilities. Mathematically, a stochastic process can be defined as a family of random variables [40, p.33].

Systems whose states only changes at discrete points in time are called discrete-event systems and these changes are caused by *events*. The events happen in discrete time jumps (which can also be of length zero) and in between these jumps the system does not change the discrete part of its state. So the order and timing of the events contribute to the behavior of the system. In [40, p.33] Younes describes a discrete-event system as *"any stochastic process that can be thought of as occupying a single state for a duration of time before an event causes an instantaneous state transition to occur"*.

Contrary, in continuous event systems there are events that change the state continuously and usually model some sort of fluid-like components. It is also possible to combine models which have discrete and continuous parts [6, Ch. 4.2.1].

2.3.2 Event-based simulation

There are two kinds of techniques for simulating discrete-event systems: time-based and event-based simulation. In opposition to time-based simulation, which works with time steps of a constant length, event-based simulation considers the occurrence of the next events at any point in time. These steps then have an optimal length, what makes the simulation more efficient. The Figure 2.4 by Haverkort illustrates the main action steps for event-based simulation.

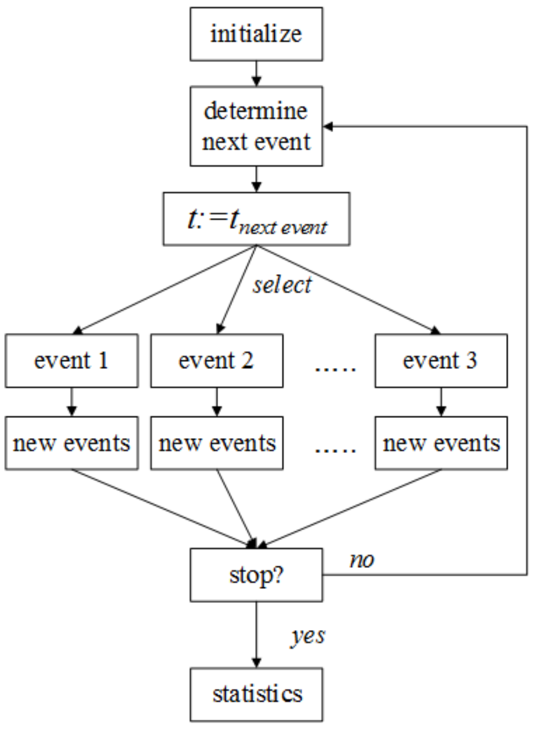


Figure 2.4: Diagram of the actions to be taken in an event-based simulation. Source: own representation based on [26, Figure 18.4, p.416]

Every occurring event can cause further new events to happen in the future, so after the execution of an event, the new events have to be determined.

Event-based simulation is only possible for systems for which the points in time at which the events happen can be computed in advance. The points in time of the events can be deterministic or gathered by sampling from a probability function. So for the latter, the system need to have a stochastic behavior defined by distributions that are computable in practice.

Haverkort defines the term *simulation time* as the time parameter in a simulation program corresponding to the time of the real system. Note that it is different to the run time of the program. He proposes to handle the events to occur in future time steps in an ordered list so that the list head always contains the next event and its occurrence time. After initializing this list, a control loop is started always considering the next event. In an event-oriented approach each type of event has its own procedure defined that is called then changing the system state. In case that this event causes new future events, their occurrence times are calculated and for each new event an entry is inserted into the list. Afterwards, the current event is removed from the list and the simulation time is incremented to the time of the event for the next loop run, until the defined simulation end is reached [26, p.415–418].

2.3.3 Random variable generation by inversion

For the simulation of stochastic processes it might be required to generate random variables from different probability distributions. A common technique for this is the inversion method. The given presentation here follows [5, pp.139–142] and [15, pp.27-28].

It is assumed that the random variable generation for a uniform distribution is available. The inversion method allows to generate a random variable from a known non-uniform cumulative distribution function (CDF) $F(x) = \Pr\{X \leq x\}$ for $x \in \mathcal{R}$ where X is the continuously distributed random variable of $F(x)$. It is possible to define an inverse function of $F(x)$ as:

$$F^{-1}(u) = inf\{x : F(x) = u, 0 < u < 1\}. \tag{2.6}$$

So for any uniform distributed random variable U, $F^{-1}(U)$ has distribution

function F because it is:

$$\begin{aligned} \Pr\{F^{-1}(U) \leq x\} &= \Pr\{inf\{y : F(y) = U\} \leq x\} \qquad (2.7) \\ &= \Pr\{U \leq F(x)\} \\ &= \Pr\{X \leq x\} \\ &= F(x). \end{aligned}$$

As a result, it is then sufficient to generate a uniform random variable and then insert it into the inverse function of the CDF if the inverse is explicitly known.

2.4 Statistical Model Checking

Statistical Model Checking (SMC) is used for the automated verification of stochastic systems. As stated in [10, 31, 37], the core idea is simulating the system for a finite number of runs, i.e. a random set of paths, and then monitoring these runs and using statistical methods to evaluate the satisfaction of properties. These sample runs, following the distribution settings of the system, can provide estimates of the probability measurements. These methods are far less memory and time intensive than classical techniques and the only requirement is that the system is executable. While numerical model checking requires the analysis of the state space, which can gain a huge complexity, SMC does not have to deal with this state space explosion problem. In addition, Statistical Model Checking is rather simple to understand and implement. Nevertheless, disadvantages lie in the fact that only probabilistic guarantees can be given and that choosing an appropriate sample size is challenging. As the convergence is slow, the sample size can grow large for high accuracy. Furthermore, the property satisfaction can only be investigated on a purely probabilistic system without non-determinism that has a fixed initial state.

For the following, let X_i be a Bernoulli random variable with the parameter p, which is the probability for $X_i = 1$, so that $1 - p$ is the probability for $X_i = 0$, as defined in [40, p.17]. When simulating, each variable is associated with one simulation run, so that $X_i = 1$ if the run satisfies the considered property and $X_i = 0$ if not [31].

2.4.1 The $P_{=?}[\Psi]$ problem

The statement $P_{=?}[\Psi]$ represents the problem to compute the probability that Ψ holds in the model. This statement is given in PRISM notation, according to the work of Nimal [33, pp.22–25], which the following approach refers to. Instead of an estimated probability, which can lie close to the real probability or far from it, confidence intervals are used here. Given a specified width $2w$ (with $w \in (0, 1]$ as the half width) and a desired level of confidence $\alpha \in [0, 1]$, an interval can be estimated, so that in $(100 \cdot (1 - \alpha))\%$ of the times the real probability lies within this interval.

Let $\bar{x}$ be the arithmetic mean of the realizations $x_1, ..., x_n$ with $x_i \in X_i$ for $1 \leq i \leq n$ of the random variable for n runs. As the standard variance is unknown, it needs to be estimated with:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2. \tag{2.8}$$

Taking $t_{n-1,1-\alpha/2}$ from the Student distribution with $n-1$ degrees of freedom, we get the confidence interval for the level α:

$$\mathrm{CI} = \left[\bar{x} - t_{n-1,1-\alpha/2}\sqrt{\frac{s^2}{n}}; \bar{x} + t_{n-1,1-\alpha/2}\sqrt{\frac{s^2}{n}}\right]. \tag{2.9}$$

The simulation then can be iterated until $|\mathrm{CI}| \leq |\,[\bar{x} \pm w]\,| = 2w$. For the number of simulation runs this results in:

$$n \geq \frac{t^2_{n-1,1-\alpha/2}s^2}{w^2}. \tag{2.10}$$

Nimal also shows in [33, pp.22–25], that for a total number of n runs with r runs that have $X_i = 1$ fulfilled, it is possible to compute s^2 and $\bar{x}$ after every iteration without storing them using $\bar{x} = \frac{r}{n}$ and $s^2 = \frac{r(n-r)}{n(n-1)}$.

2.4.2 The $P_{\bowtie\Theta}[\Psi]$ problem

$P_{\bowtie\Theta}[\Psi]$ (as well given in PRISM notation) represents the problem that the probability that Ψ holds is $\bowtie \Theta$ with $0 \leq \Theta \leq 1$ and $\bowtie \in \{<, \leq, >, \geq\}$. This problem is known as the hypothesis testing problem which can be solved by a lot of different approaches from literature. One of these is the sequential probability ratio test (SPRT) by Wald. This test is illustrated in detail in [40, pp.18, 24–25] and also in [33, pp.66–67] and [31].

Within this approach, we assume first that a hypothesis $H_0 : p \geq p_0$ is tested against an alternative hypothesis $H_1 : p < p_1$ for $p_0 = \Theta + \delta$ and $p_1 = \Theta - \delta$ with $\delta > 0$ as the half-width of the desired *indifference region* (p_1, p_0). Within this region neither H_0 nor H_1 holds. It is required that the probability, that H_1 is accepted when H_0 holds, is $\leq \alpha$, where α is the desired so-called *type-1-error*. Respectively, the probability of accepting H_0, in case that H_1 holds, has to be $\leq \beta$, with β as *type-2-error*. If the real errors are exactly equal to α respectively β, the test has ideal performance.

After each run, the SPRT computes the likelihood ratio for so far n observations:

$$\frac{p_{1n}}{p_{0n}} = \prod_{i=1}^{n} \frac{\Pr[X_i = x_i | p = p_1]}{\Pr[X_i = x_i | p = p_0]} = \frac{p_1^{d_n}(1-p_1)^{n-d_n}}{p_0^{d_n}(1-p_0)^{n-d_n}}, \tag{2.11}$$

with $d_n = \sum_{i=1}^{n} x_i$ for the n observations $x_1, ..., x_n$ with $x_i \in X_i$.

The result is then compared to the values $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$ and the action to take next (finish or continue with another run) is determined as follows:

- If $\frac{p_{1n}}{p_{0n}} < B$, H_0 is accepted,
- if $\frac{p_{1n}}{p_{0n}} > A$, H_1 is accepted,
- otherwise, another simulation run with $x_{n+1} \in X_i$ is performed.

For testing the other direction, $H_0 : p \leq p_0$ can be simply tested against $H_1 : p > p_1$ by swapping the values for p_1 and p_0, so that $p_0 = \Theta - \delta$ and $p_1 = \Theta + \delta$. Note that the cases $P_{<\Theta}[\Psi]$ and $P_{>\Theta}[\Psi]$ can be evaluated by model checking $P_{>=\Theta}[\Psi]$ respectively $P_{<=\Theta}[\Psi]$ and inverting the result.

3 Development of the simulator

The main target of this work is the development of a tool that is able to simulate the evolution of a given hybrid Petri net model. This is done based on discrete-event simulation, which was presented in Section 2.3. The tool simulates the behavior of the model for a specified number of simulation runs and uses the results to model check a property expressed by an STL formula (see Section 2.2).

This chapter presents the software engineering process from a technical perspective. At first, Section 3.1 explains the preparatory steps that have been done before the implementation, like the requirement definition and the software design. Section 3.2 explains the most important software features and the approaches applied to realize them. Note that the detailed description of the features concerning Statistical Model Checking are presented in the next chapter. Finally, in Section 3.3 an overview about the quality related aspects regarding the tool is given. These sections of this chapter follow the different phases of the software engineering process, as presented in [30, p.25]. Only the phase of testing is presented later in Chapter 5.

3.1 Preparatory work

Before starting the development of software, it is essential to define the main requirements for the product, to determine the technical framework and to start with an appropriate software design. The results of these preparatory steps are presented in this section: the tool requirements in 3.1.1, the programming environment in 3.1.2 and the software design in 3.1.3.

3.1.1 Requirements

An appropriate definition of the requirements is fundamental for gaining the desired software product. It is important to figure out what functionality and requirements are needed and what boundary conditions need to be considered [30, p.51]. With a look on use cases, functional requirements can be derived [30, p.71]. In addition, one should not neglect the non-functional requirements, like quality, technical or contractual needs [30, pp.77–80].

The main aim of the software to develop is the simulation of hybrid Petri net models and the automated verification of defined property formulas. The foundation for the development is a given Java library called *libhpng* that is able to read in hybrid Petri net models with multiple general transitions from an XML file into Java objects. Given an initial state (which is also defined within the file), the simulator is then required to use these Java objects to simulate the behavior of the model for a specified period of time. The results of several runs of this simulation then have to be plotted and using these results, the software must be capable of model checking a property for a specified point in time that is entered by the user. The software then is supposed to be able to either (i) calculate the confidence interval for the probability that the property is satisfied or (ii) to check if this probability is lower or greater than a threshold. Additionally, a function is required to plot the fluid level of all continuous places.

Required functionality

A list of all functional requirements on the simulation tool is attached in Appendix A. As specified in RFC 2119[1], the criteria are subdivided into MUST criteria, which are absolutely mandatory for the tool, SHOULD criteria, which are required to get an adequate software and MAY criteria, which are desirable, but not mandatory. In addition, WON'T criteria are listed, that are not implemented within this work but desired to be part of future works. The main requirements for the tool can be summarized as:

- the discrete-event simulation on a model including distribution sampling for general transitions by setting seeds randomly for a generation of random variables for specified probability density functions
- the model checking of Stochastic Time Logic formulas (see 2.2) based on the simulation results including the formula parsing as well as the calculation of confidence intervals and hypothesis testing
- the integration into the existing *libhpng* library

Product use

The software development is part of the research of the group of safety-critical systems lead by Prof. Dr. Anne Remke. The group is part of the institute of Computer Science at the University of Münster. The simulator is supposed to extend the existing tools of the group and to be used for teaching purposes and for further research on safety-critical systems. Within this environment, the software is also available on GitHub[2] for public use as open-source.

[1]`https://www.ietf.org/rfc/rfc2119.txt`
[2]`https://github.com/jannikhuels/libhpng`

Quality requirements

A guideline for quality requirements is given in the norm ISO/IEC 25000[3] (former ISO/IEC 9126). Classical quality attributes to consider are correctness, safety, reliability, robustness, usability, efficiency and others [30, pp.78–79]. A definition on these terms is given in Section 3.3.

The main quality requirement for the tool is that it delivers correct, accurate and reliable results. The run time has to be adequate to the size and complexity of the input model and to the number of runs. The tool is required to be designed in a user-friendly way. It should be easily operated and robust to errors and incorrect user input. To verify the correctness of the results, an extensive testing and a comparison to existing tools is included in the development process.

3.1.2 Environment

The simulation tool is integrated into the existing Java library *libhpng*. This library is based on a former C++ tool which analyses the same kind of hybrid Petri net models. The change from C++ to Java has been done due to the fact that Java is less platform dependent and there are many additional libraries provided. Like in case of this development libraries for the distribution sampling and the formula parsing can be used easily. In addition, Java is simple, object-oriented and secure.

The *libhpng* library provides the reading of hybrid Petri net models from an XML file into a Java object structure, including exception handling and a shell integration. The previous functions for the analysis of models contained within the C++ tool are not integrated in the Java library yet, but it is planned to do so in the future.

The development for this tool has been done with Java SE 7 OpenJDK using the environment Eclipse 3.8.1 on a 64-Bit Linux system (Ubuntu 14.04 LTS).

3.1.3 Design

The aim of software designing is to transfer the requirements into a draft of needed packages, classes and functions, as well as taking decisions on the system architecture [30, pp.87]. The implementation of the tool is divided into packages to add structure and bundle classes. The library *libhpng* has provided five packages and for the simulator three additional packages are added, so that in total there are eight packages, as shown in the packages diagram in Figure 3.1.

[3]`http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=64764`

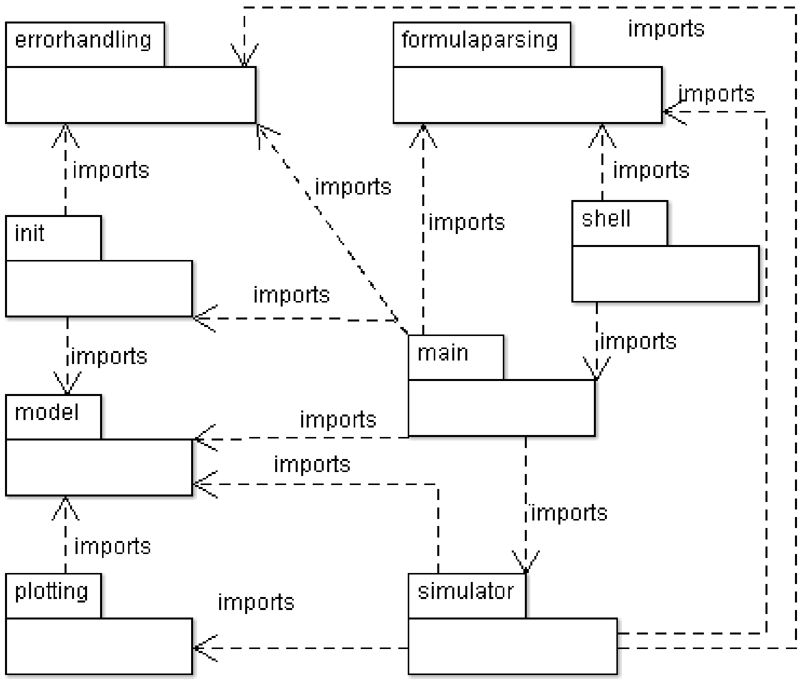


Figure 3.1: Structure of all packages. Source: own representation

The existing library

The package *Main* includes the main class of the test program. The package *shell* contains four classes needed for a command shell. For this purpose the open source library *Spring Shell* [4] is used. Both packages are not designed to be parts of the library but needed for test purposes.

The package *model* contains all the classes for the hybrid Petri net model representation. Figure 3.2 shows a rough UML class diagram for this package. (Note that only the most relevant classes are included in this diagram.)

The main model is an instance of the class *HPnGModel* which contains lists of places, transitions and arcs. In addition to the model definition in Section 2.1.2 based on [24], the class *DynamicContinuousTransition* is included in the package. These transitions behave exactly like continuous transitions but their fluid rates are not fixed but dependent on other (standard) continuous transitions (multiplied by a coefficient). They are useful for an easier implementation of the model file and a clear view, as one transition with flexible rates can be handled easier than multiple transitions with fixed rates.

The package *init* contains the interfaces for reading in the XML model files. The class *XMLReader* provides a function to read the XML file into the object

[4]`http://projects.spring.io/spring-shell/`

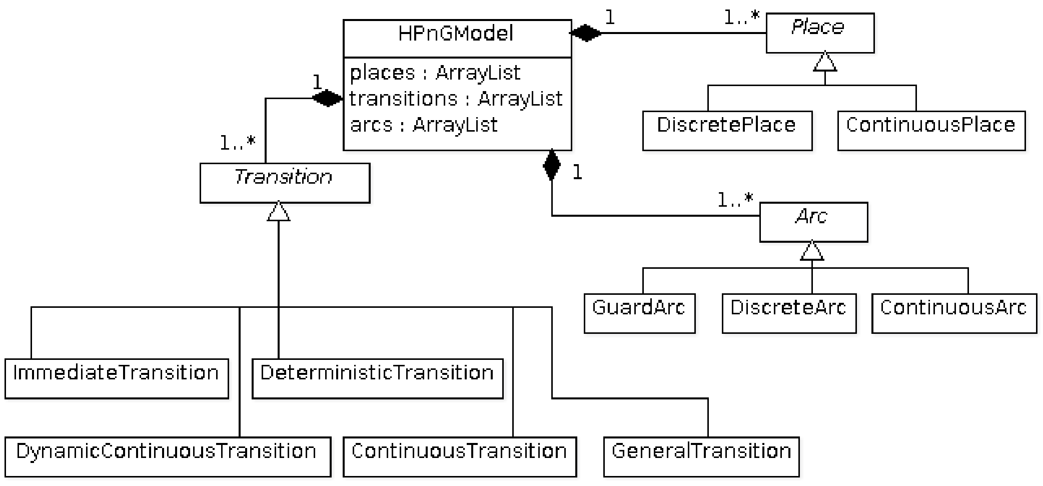


Figure 3.2: Structure of the package 'model'. Source: own representation

structure of the *model* package by using the XML library *JAXB*[5]. The class *ModelReader* provides a function which calls the XML reading function and initializes the model by setting connections between places, transitions and arcs, sorting the place, transition and arc lists and setting the intial state. The fifth package of the existing library is the *errorhandling* package. It contains exception definitions for invalid input models.

The packages and classes for the simulator

The code that implements the simulator is subdivided into the packages *simulation*, *plotting* and *formulaparsing*. The packages *errorhandling* and *Main* are also extended by classes or functions.

The package *simulation* contains a class *Simulator* with the purpose of determining and executing the next event. Another class in this package is *SimulationHandler* for setting the simulation parameters and running the different simulation approaches, like plotting only or model cheking $P_{=?}[\Psi]$ or $P_{\bowtie\Theta}[\Psi]$ formulas. There is also a class *SimulationEvent* for defining an event object containing the event type, the occurence time and relevent references. The package also contains the classes *SampleGenerator* for sampling general transitions, *PropertyChecker* for model checking the properties expressed in STL, *ConfidenceIntervalCalculator* for the calculation of the confidence intervals and *SequentialProbabilityRatioTester* for the hypothesis testing. In addition, the sampling for all the different implemented distributions is done by the class *DistributionSetting*. Figure 3.3 shows an UML class diagram of the package.

The package *plotting* provides classes for recording the simulation data.

[5]`https://jaxb.java.net/`

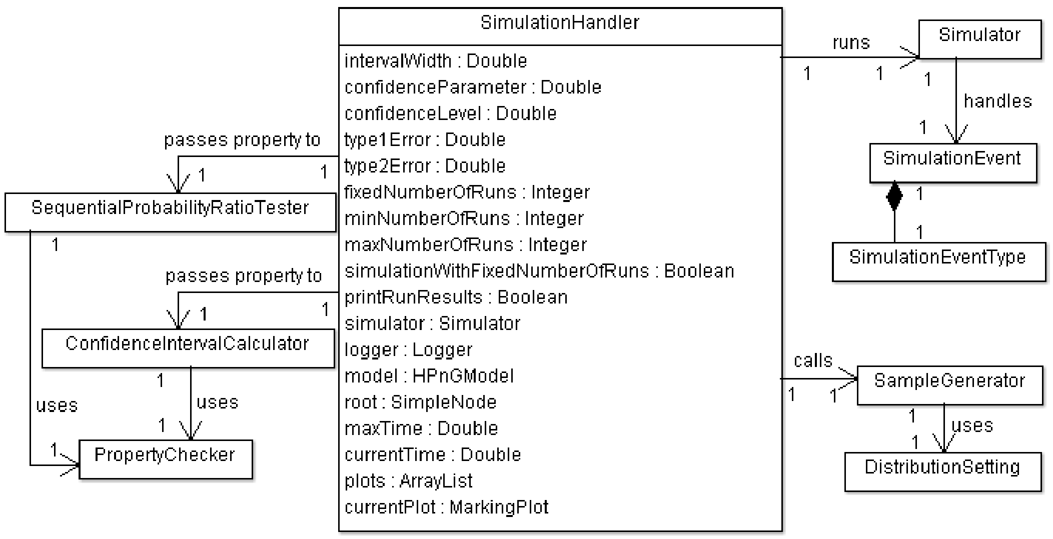


Figure 3.3: Structure of the package 'simulation'. Source: own representation

There are the classes *PlacePlot* and *TransitionPlot* (derived from the abstract class *Plot*) holding a list of entries for a place or a transition. These entries are instances of different classes derived from the abstract class *PlotEntry*, containing the simulation time for the given entry and the plotted information. For example, there is a derived class *ContinuousPlaceEntry* whose instances contain the fluid level and drift of a continuous place for a given simulation time. For a complete recording, the class *MarkingPlot* contains lists with all transition and place plots. It provides different functions for storing, reading and plotting data. The package also contains the class *ContinuousPlacePlotter* providing a function for plotting the fluid levels of all continuous places with confidence intervals in a graph. Within the simulation, every point in time, at which an event happens, is plotted. For the graphical representation the library *JFreeChart*[6] is used within the class *XYLineGraph*.

The package *formulaparsing* provides the possibility to parse user input of a specified syntax into a parse tree object using the Java compiler *JavaCC*[7] and the preprocessor *JJTree*[8]. The library automatically compiles a structure of Java classes that are integrated into the simulation tool.

3.2 Implementation

After finishing the design process, the simulation software can be implemented in Java. This section gives an overview about the solutions for the main

[6]`http://www.jfree.org/jfreechart/`
[7]`https://javacc.java.net/`
[8]`https://javacc.java.net/doc/JJTree.html`

features of the software (excluding the details about Statistical Model Checking which are presented in the next chapter). At first, a general introduction is given into the basic construction of the tool in 3.2.1. Then, 3.2.2, it is explained how the general transitions are sampled and in 3.2.3 the formula parser is demonstrated. Furthermore, this section includes a presentation of the plotting the fluid levels of the continuous places in 3.2.4, of the exception handling in 3.2.5 and of the user interface in 3.2.6.

3.2.1 Basic construction

As already mentioned in the previous section, there is a class *SimulationHandler* which implements the core for the simulation. The simulation handler contains all the parameters for the simulation, the model and the specified property. It provides functions for plotting places and for model checking properties for different parameter settings. Each of these functions creates an instance of the class *Simulator* and an instance of the class *SampleGenerator*. Then , inspired by the approach in [26, p.415–418], the simulation itself follows in form of a loop. Within this loop, the model is reset to its initial state and the general transitions are sampled (using the sample generator). After initializing the plot for the simulation, the following instruction is called:

```
while (currentTime <= maxTime)
  currentTime = simulator.getAndCompleteNextEvent(currentTime)
```

whereas *currentTime* is initialized with zero before and *maxTime* is the maximum time to simulate. This instruction loops over the time to simulate. The function *getAndCompleteNextEvent()* of the simulator class determines the next discrete event that will happen in the current simulation run, seen from the current point in time. Then it updates the state of the model by completing the event and returns the new point in time. This is repeated until the maximum time is reached.

The simulator function is quite complex. This is why a rough pseudo code is given in Appendix B. The ranking of the events to consider is according to [24], as in Table 2.1 of the previous Chapter. TWithin the function an event object is created in line 1 and then a loop first iterates over all transitions in lines 2–15. If the event was not set to an immediate transition (as this event has Order I and always happens immediately), it also loops over guard arcs (lines 18–32) and continuous places (lines 33–52). If an earlier event is found within one of the loops, it is saved in the event object (*event.addEvent()*). If an event is found that happens at the same time as the previous found event, the new event is added to the object instead of replacing the previous one (*event.saveEvent()*). In line 53, the continuous marking is updated first (i.e. the fluid level of all continuous places, the clock value for all deterministic transitions and the enabling time for all general transitions). Then, the function *completeEvent()* in line 54, takes an action depending on the event type. For transition events the transition is fired and the marking of affected discrete places is updated. For guard arc events the condition is checked and their status is updated accordingly and for continuous places boundary events the boundaries are checked and their status is updated, too. Independent on the event type, the model status is updated at the end of the function *completeEvent()*, which includes the update of the enabling status for all transitions and of the fluid rates for all continuous places.

While for guard arcs and continuous places, all related arcs respectively places can be updated at once, only one single transition can fire at a time as an event. As mentioned in 2.1.2, if multiple transitions are supposed to fire at the same time and if they have the same priority, they are in conflict. In the simulation the conflict resolution is solved using the weights of transitions. This is what the function *conflictResolutionByTransitionWeight()* is for. The probability for each transition (of the highest priority) to be selected is derived from their weight devided by the sum of weights of all conflict transitions. Using these probabilities and a uniform random variable generator, it is determined which transition is selected for firing.

3.2.2 Sampling general transitions

At the beginning of each simulation run and after every firing, each general transition has to be sampled, i.e. according to its distribution function, a deterministic firing time has to be determined. For this purpose the library *SSJ*[9] of the Département d'Informatique et de Recherche Opérationnelle of the Université de Montréal is used. It provides a number of random variable generators for a lot of different distributions. A part of these distributions is included in the simulator for sampling the general transitions. An overview about the selected distributions is given in Appendix C. The library creates random variables from these distributions by using the inversion method explained in 2.3.3, inspired by [5, 139–142].

The class *SampleGenerator* provides functions to initialize the random stream by setting seeds randomly and to sample all general transitions. For each general transition a generator instance is needed. This is why the class *DistributionSetting* provides static functions for each kind of distribution returning the generator. Within these functions the correct parameter values defined in the model are assigned to the generator instance taken from the *SSJ* library. When sampling, the generator can then be called to get a new random firing time for the transition.

When a general transition gets disabled and later enabled again, there are three possible ways to handle this: *resume*, *repeat identically* and *repeat different*, as described in 2.1.2. This policy is defined within the model file for each single general transition. Only for *repeat different*, the transition will be sampled again while enabling. For *repeat identically*, the enabling time will be

[9] `http://simul.iro.umontreal.ca/ssj/indexe.html`

reset to zero without changing the sampled firing time. For *resume*, no action is taken except changing the enabling status, as the previous enabling time was preserved and can be used again.

3.2.3 Formula parsing

For model checking properties, the tool needs to read in defined STL formulas. Therefore, a formula parser is needed supporting the STL syntax defined in Section 2.2. Note that the disjunction (*OR* case) is also included in the tool. As mentioned before, the compiler and preprocessor libraries *JavaCC* and *JJTree* are used to parse the user input into a tree structure. When the input function is called, the input is read from the console and, in case of a correct formula, a root object is returned. In case of a incorrect formula, an error occurs. To give an understanding of the formula syntax, an explanation for formula syntax is given in the following.

The input has to begin with a point in time for which the property should be checked (the t from $\sigma \models^t \Psi$). This time specification has to be a decimal value and a colon has to follow. After the time specification, there has to be one of the following expressions:

1. **'P=?'** for the calculation of the mean value of all simulations and the corresponding confidence interval (see 4.2.1).
2. **'P$\sim$ x'** where '$\sim$' has to be replaced by '$<$','$\leq$','$>$' or '$\geq$' and x has to be replaced by a decimal value. This expression is used for the sequential probability ratio test (see 4.2.2).

The expression is followed by a formula surrounded by brackets '(' and ')' that can be nested arbitrarily. Allowed types of formulas are listed in the following, where Ψ_1 and Ψ_2 are formulas again and t_1, t_2 the time bounds for the Until formula (as decimal values).

1. True: **'tt'**
2. Simple atomic property (see below)
3. Negation: **'!$\boldsymbol{\Psi_1}$'**
4. Conjunction: **'AND($\boldsymbol{\Psi_1}, \boldsymbol{\Psi_2}$)'**
5. Disjunction: **'OR($\boldsymbol{\Psi_1}, \boldsymbol{\Psi_2}$)'**
6. Until formula: **'U[$\mathbf{t_1}, \mathbf{t_2}$]($\boldsymbol{\Psi_1}, \boldsymbol{\Psi_2}$)'**

The supported atomic properties are defined as follows, where 'id' be the place ID, transition ID or guard arc ID depending on the kind of property (surrounded by single quotation marks) and '$\sim$' has to be replaced by '$<$','$\leq$','$>$' or '$\geq$'. The value x has to be a decimal value and y has to be an integer value.

1. The fluid level of a continuous place compared to a value: **’fluidlevel(’id’)∼ x’**
2. The drift of a continuous place compared to a value: **’drift(’id’)∼ x’**
3. Check if the upper boundary of a continuous place is reached: **’uboundary(’id’)’**
4. Check if the lower boundary of a continuous place is reached: **’lboundary(’id’)’**
5. The number of tokens in a discrete place compared to a value: **’tokens(’id’)∼ y’**
6. Enabled status of any kind of transitions: **’enabled(’id’)’**
7. The clock value of a deterministic transition compared to a value: **’clock(’id’)∼x’**
8. The enabling time of a general transition compared to a value: **’enablingtime(’id’)∼y’**
9. Check if the condition of a guard arc is fulfilled: **’condition(’id’)’**

Examples for such formulas are:

- **0.0:P>= 0.9(AND(drift(’cp1’)=0.0, fluidlevel(’cp1’)>1.0))**
  for the STL formula: $\Gamma_0 \models^0 (P_{\geq 0.9}(d_{cp1} = 0 \wedge x_{cp1} > 1))$ for a continuous place $P_{cp1} \in \mathcal{P}^c$
- **5.0:P<0.5(U[0.0,2.0](tokens(’dp1’)<3, enabled(’it1’)))**
  for the STL formula: $Gamma_0 \models^5 (P_{<0.5}(m_{dp1} < 3 \; U^{[0,2]} \; e_{it1}))$ for a discrete place $P_{dp1} \in \mathcal{P}^d$ and an immediate transition $T_{it1} \in \mathcal{T}^I$

The formula creates an instance of the class *SimpleNode* which is a node in a tree structure holding an id, a parent node, a list of children nodes and a value. Figure 3.4 shows an example for a parse tree for the formula: $s_0 \models^2 (P_{=?}(x_{cp1} < 6 \; U^{[0,5]} \; (x_{cp2} = 5 \wedge e_{td1})))$.

3.2.4 Plotting the fluid level of continuous places

Next to the model checking of properties, the simulation can also be used to only plot the development of the fluid levels of the continuous places in a model. This is done for a given simulation time and fixed number of runs by calculating the mean value and a confidence level for each single point in time at which an events happens. As the drift (i.e. the change of the fluid level)

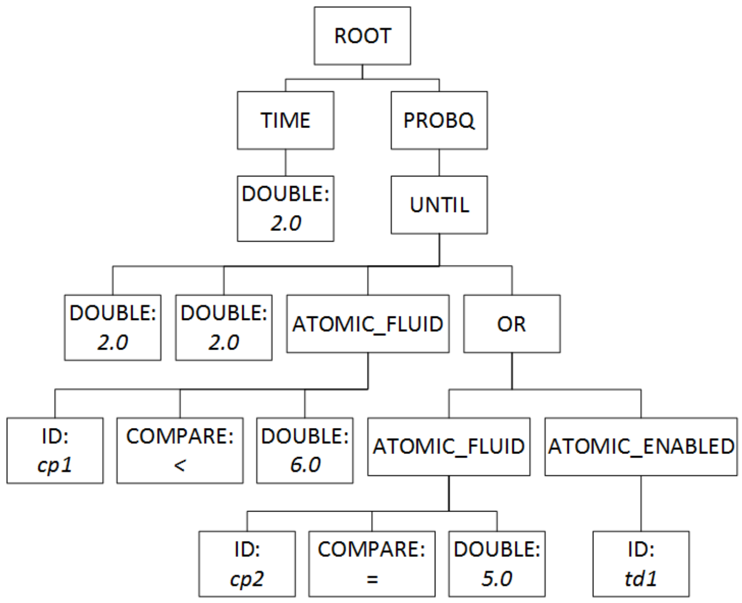


Figure 3.4: Example parse tree. Source: own representation

is constant between the discrete events), the mean can be simply interpolated between the events. This approach has been developed within this work, as no suitable solution has been found in previous works. Whether this technique is more efficient than choosing fixed time steps, may be dependent on the number and distance of the events in proportion to an appropriate fixed step size. An investigation on these methods is not done within this thesis, but desirable for future works. Within this output, the confidence interval borders are interpolated, too, even though this might not reasonable, but for now we could only implement this kind of graphical solution. The plotting structure of the simulator has already been roughly presented in Section 3.1.3. An example for a plot is given in Figure 3.5, which shows the mean value and confidence intervals for the fluid levels of the two continuous places for the model *example2.xml* from Appendix E for 100 runs with a confidence level of 0.95.

Within the function *plotContinuousPlaces()* of the class *ContinuousPlaces-Plotter* an instance of the class *XYLineGraph* (derived from the *JFreeChart* library) is created for the output. For every continuous place three series are added to this graph: one for the mean and one for each confidence interval border. According to the approach for the $P_{=?}[\Psi]$ problem presented in Section 2.4.1, the confidence interval for a single point in time is calculated by calculating the arithmetic mean and then calculating the estimation s^2 for the standard variance (by summing up the squared difference between the fluid level of the current run and the mean value and dividing this sum by the number of runs minus one). To calculate the interval borders, the t value from the

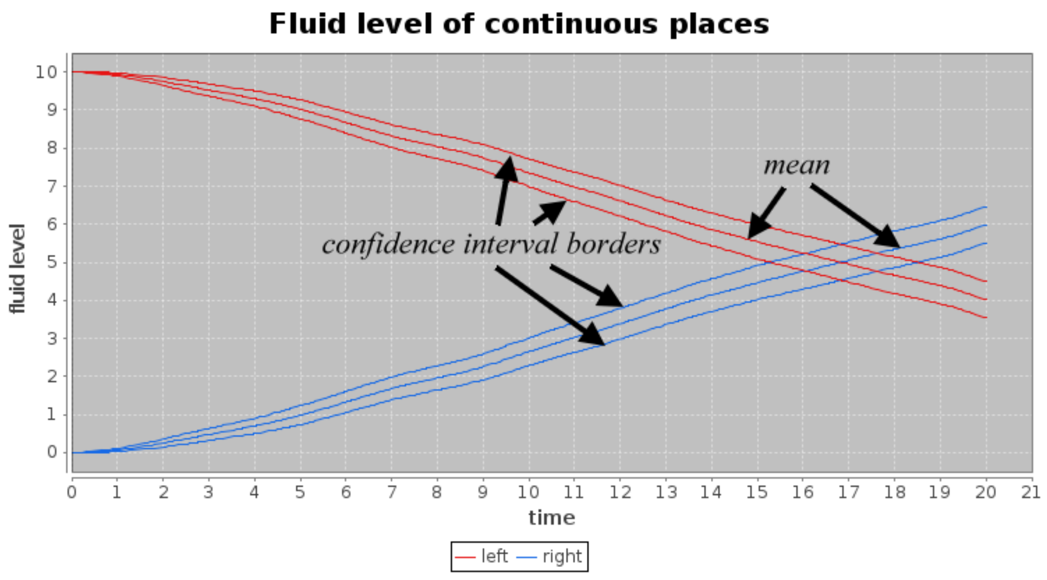


Figure 3.5: Plot of the fluid level of the places *left* and *right* for *example2.xml*. Three lines of one color illustrate the mean value of the fluid level and the confidence interval borders above and below. Source: tool output, adapted with annotations.

Student distribution for given degrees of freedom (number of runs minus one) and a given confidence level is needed. This can be easily taken from the *SSJ* library again.

So the function loops over all points in time with events. Note that for every single simulation run these points can be different (especially regarding the firing of the general transitions). The loop has to go over points in time with events in all runs. This is why (within the outer loop) a first inner loop is calculating the minimum point in time of the next event over all runs and then a second inner loop over all runs is needed to calculate the mean and a third loop to calculate confidence interval borders.

3.2.5 Exception handling

Within the execution of the simulator, a number of errors can occur. To handle these errors the tool holds some exception classes in the package *errorhandling*. These exceptions are:

- ***InvalidDistributionParameterException***: this execution occurs when a general transition is sampled but the parameters for the distribution defined in the model file are incorrect. Reasons for this can be missing parameters or incorrect values for parameters. The exception is thrown by the distribution setting function and catched by the simulation handler which then throws a *ModelNotReadableException*.

- ***InvalidModelConnectionException***: this execution can occur while reading in a model file. After the parsing of the XML code, there are references to the connected place and transition set for every arc. In case there is no valid place ID or no valid transition ID found for an arc, the exception will be thrown within the model reader class and turned into a *ModelNotReadableException*.

- ***InvalidSimulationParameterException***: this execution occurs when changing any of the simulation parameters to an invalid value. It is thrown by the simulation handler and returned to the main function.

- ***ModelNotReadableException***: this execution occurs in case a model file is not existing, contains invalid XML code or in case it is thrown due to the occurence of an other exception (invalid distribution parameter, invalid model connection or XML not valid). The exception is returned to the main function.

- ***InvalidPropertyException***: this execution occurs when the property formula was parsed successfully but holds a logical error. For example this could be a missing node in the parsing tree, an invalid ID for a place, transition or arc or invalid bounds for an until formula. It is thrown by the property checker instance and returned to the main function.

- ***XmlNotValidException***: this execution occurs when the XML model does not fulfill the requirements of the XML schema definition. The validation function then throws this exception, which is caught by the model reader and turned into a *ModelNotReadableException*.

Furthermore, it is possible that the used library *JJTree* throws a *ParseException* or an *IOException* and returns it to the main function.

3.2.6 User interface

The user of the simulator can run and control the tool by using the console. For the shell commands the library *Spring Shell* is used, as mentioned before. It provides a number of basic commands (especially the *help* command showing all available commands). These commands can be extended by further commands. For the simulator the additional commands added are:

1. **'read --p *string*'**: reads in an HPnG model specified by the String parameter **--p**, which needs to contain the path of the xml model file.

2. **'parse'**: parses an STL formula and prints its tree structure. No parameter is required and the formula has to be entered after prompted.

3. **'check'**: model checks the property expressed by an STL formula by running the simulation on the model. The formula has to be entered after prompted. This command is only available after a model has been read in (by the **read** command).

4. **'plot --t *double*'**: plots the fluid levels of all continuous places by running the simulation on the HPnG model. The Double parameter **--t** is required for the maximum time of the simulation. This command is only available after a model has been read in successfully (by the **read** command).

5. **'change logfile --n *path*'**: changes the path of the log file using the String parameter **--n**, which needs to contain the new path for the log file.

6.
   - **'change halfintervalwidth --n *double*'**
   - **'change halfwidthindifferenceregion --n *double*'**
   - **'change confidencelevel --n *double*'**
   - **'change type1error --n *double*'**
   - **'change type2error --n *double*'**
   - **'change fixedruns --n *integer*'**
   - **'change minruns --n *integer*'**
   - **'change maxruns --n *integer*'**

   change the different simulation parameters using the parameter **--n**, which needs to contain the new parameters value. Different requirements for the parameters are given by the console.

7. **'set fixedruns'**: sets the simulation to use the fixed number of runs according to the fixed numer of runs parameter.

8. **'set optimalruns'**: sets the simulation to use the optimal number of runs according to the confidence and error parameters.

9. **'printresults on'**: enables that the results of the single simulation runs are printed to the console.

10. **'printresults off'**: disables that the results of the single simulation runs are printed to the console.

11. **'printparameters'**: prints all simulation parameters and settings to the console.

12. **'storeparameters'**: stores all simulation parameters and settings to the configuration file.

13. **'loadparameters'**: loads all simulation parameters and settings from the configuration file.

Spring Shell automatically stores all called commands of one tool execution in a log file named *shellLogFile.log.* In addition to this file, the tool also creates a second log file named *logFile.log* containing information on the main steps of the simulation process, a summary of the results and any information on exceptions. This is why the output on the console is reduced to the most important information but the details are stored in the log file.

3.3 Quality evaluation

Though there is no standard definition of the term quality, it is generally considered that a quality evaluation is an essential part of software development, as done within the approach of test-driven development. For evaluating software quality, different quality features can be investigated. This section examines the simulation tool with regards to four of these features: functionality in 3.3.1, reliability in 3.3.2, usability in 3.3.3 and efficiency in 3.3.4.

3.3.1 Correctness

Correctness is given if the tool is totally conformable to the specified requirements, which can be checked by verifying test cases [30, pp.78]. For investigating the correctness of the given tool by unit testing, there are seven test cases defined in Appendix D. Note that these test cases focus on the technical feasibility of the scenarios but not on the evaluation of the simulation results. It is only tested if (logically reasonable) results are given but these results are not validated. For the validation of the software, a case study has been executed on the simulator. This is described in Chapter 5.

The test cases have all been executed on the simulation software. As a final result, it can be said that the tool behaved as expected. All executed functions terminated without abortion and gave the required results. The example models were read successfully and properties expressed by STL formulas were parsed and model checked with expected results. The plotting of the fluid levels of the continuous places also gave a reasonable graphical output. For intended errors, like incorrect input or model files, appropriate error messages were shown and correct information was written into the log file. In addition, parameters could be changed, stored and loaded without faults.

3.3.2 Reliability

The term reliability describes the ability of a system to perform permanently under specified conditions while fulfilling all requirements. Reliable systems have a low mean time to failure [30, pp.78]. In the given tool this is assured by the detailed exception handling which catches errors occurring during execution. As a result, critical errors do not lead to an abortion of the tool. During the test phase, the running software has never freezed and fulfilled all functional requirements. In addition, a high data reliability is given as the model files are only read but never changed, so that no data loss is expected. The only files changed are the log files and the configuration file. Latter can be easily restored with default values.

3.3.3 Usability

A software is called usable if it can be easily understood and used by any user [30, pp.79]. For the simulator this usability is limited as a lot of specified background knowledge (especially about Petri nets and model checking) is required. From a technical perspective, the simulator is implemented as a rather simple console application with only a low number of possible commands to execute. The implemented shell provides a proper user guidance, especially with its help function that gives an explanation on the commands. Furthermore, the shell informs the user directly about any kind of incorrect input. In addition, default parameters are preset which reduces the complexity of the commands. With regard to attractiveness, it can be said that there is no appealing interface given, due to simplicity, and that this feature can still be improved in future works.

3.3.4 Efficiency

A system is called efficient if it can preserve a specified proficiency level while not using an oversized amount of resources, i.e. run time and memory [30, pp.78]. It is not trivial to determine the theoretical time complexity of the given simulation algorithm as the run time is totally dependent on the input model file (i.e. its number of places, transitions and arcs and how they are connected) and on the number of events that can occur within the specified simulation time. In addition, the setting of the simulation parameters has a huge influence on the number of simulation runs and it is obvious that the number of runs directly influences the run time as well.

From a practical view, the software was tested on an Intel Celeron N2820 system (2.13GHz/2.39GHz with 4GB DDR3L RAM) running with Ubuntu 14.04 LTS 64-Bit. Using the given example files (see Appendix E), all executed commands of the test cases returned results within a few seconds. The most time-consuming simulation was done within Test Case 3 and took about ten seconds. Within this test case, a confidence interval was calculated and more runs are generally needed for this method compared to the hypothesis testing (but note that this is depending on the set parameters). The plotting of the graph for the fluid levels of the continuous places is highly dependent on the number of simulation runs and simulation time as the plotted data must be processed. This is why the number of runs is reduced in the Test Case 5, so that the graph can be plotted within seconds. A major step to improve the run time of the simulation and to resolve its dependency on the number of runs, is to implement multi-threading. Future works are desired to invest on how to execute simulation runs in parallel and to make the tool more efficient.

The amount of storage used by the tool is mainly dependent on the number of places and transitions as well as on the number of events, as the most intensive memory is needed for storing the system state data after every event.

4 Statistical Model Checking

The purpose of the tool developed within the context of this work is not only the simulation itself but also the automated verification of statistical properties. Therefore, methods for Statistical Model Checking (see Section 2.4) have been implemented for model checking properties expressed in Stochastic Time Logic (see Section 2.2). This chapter gives an overview about how the tool examines the fulfillment of STL properties and how it estimates related probabilities. Hence, it makes clear how the fundamentals from Chapter 2 have been realized. Section 4.1 points out how it is determined if any kind of property holds within a single simulation run. So, this section points out how the model checking itself has been implemented. Section 4.2 explains how confidence intervals for the probability, that a property holds, are estimated for a set of simulation runs and how hypothesis testing is realized to compare this probability to a given threshold.

4.1 Model checking

In the Section 2.2, the Stochastic Time Logic has been presented for the specification of properties to be model checked by the tool. In Section 3.2.3, it has been explained how STL formulas are parsed into a tree structure by the tool. Recall that for the relation $\Gamma_0 \models^t P_{\bowtie p}(\Psi)$, the tree contains a root node with two or three children nodes: (i) the point in time to check, which is the t in the relation, and (ii) the value for p needed for the hypothesis test as well as (iii) a node containing the property itself, where it is separated between $P_{<p}(\Psi), P_{\leq p}(\Psi), P_{>p}(\Psi)$ or $P_{\geq p}(\Psi)$ for hypothesis testing and $P_{=?}(\Psi)$ for calculating confidence intervals (recall Section 2.4.1). This third child node then contains a child node holding information about Ψ, which can be an atomic property or a negated or combined property (see 4.1.2 and 4.1.3). Recall that Γ_0 is the initial state defined in the model and, hence, not specified in the STL formula.

The tool contains a class *PropertyChecker* to model check a specified property against the simulation results. When creating a new instance of this class, a model instance and a property node instance (returned from the formula parser) are required. So, for the property checker, the root of the STL formula parse tree is passed. Within the construction, the point in time t is extracted from the tree structure. Then a function *checkProperty()* can be called to model check the property expressed by the given STL formula (without its probabilistic operator) for the current simulation run, so it can either be fulfilled or not fulfilled for this single run. The function distinguished between the handling of atomic properties, exposed in 4.1.1, and of non-atomic properties. Note that all properties are defined for paths but most of them depend on state characteristics, except the until formula. This is why these properties are examined separately in this section: state-based properties in 4.1.2 and the until property in 4.1.3. In the following, only the inner property Ψ is considered. This property is model checked for a single path, i.e. one simulation run. How the probability operator surrounding Ψ is handled by the tool, is explained in Section 4.2.

4.1.1 Atomic properties

When model checking an atomic property, the ID of the referenced place, transition or guard arc must be considered at first: For properties comparing the fluid level or drift $\sigma \models^t x_P \sim a$ respectively $\sigma \models^t d_P \sim d$, the ID P must reference a continuous place and for comparing the marking $\sigma \models^t m_P \sim b$, P must reference a discrete place. For $\sigma \models^t c_T \sim c$ or $\sigma \models^t g_T \sim e$, which is the comparison of the clock value or enabling time, the P has to reference a deterministic respectively general transition and for the enabling status $\sigma \models^t e_T$ any kind of transition. So, the ID is extracted from the corresponding node and compared to the model elements. Note that, within the implementation, it is also possible to check guard arc conditions and boundary reaching, derived from the fluid level property. This is why, in case of a guard arc, the ID of the arc will be replaced by the ID of the connected place as the fluid level of this place will be checked against the arc's condition.

If the place or transition of the ID is found in the model, the atomic property can be easily model checked by extracting the plot entry for the specified time as well as the comparison operator $\sim \in \{=, <, >, \leq, \geq\}$ and comparison value a, b, c, d or e. For the enabling status property, no comparison is needed as the property is fulfilled exactly when the transition is enabled. For the upper boundary, lower boundary and guard arc properties, the comparison value is equal to the capacity of the place respectively zero respectively the arc's weight. For the other properties, the value and operator need to be included in the formula tree structure.

The plot entry then provides the particular value of the place or transition (i.e. the fluid level, enabling status, number of tokens etc.) at time $t_e \leq t$, where t_e is the time of the last event that happened before (or at) the time t. It is necessary to distinguish between continuous and discrete properties as continuous values change between events. For the fluid level property as well as for upper and lower boundary and guard arc properties, the amount of fluid at time t has to be calculated by taking the fluid level at time t_e plus the drift times $(t - t_e)$. This is possible as the drift is constant between events. Similarly, for clock and enabling time properties, the difference $t - t_e$ has to be added to the value at time t_e as a clock value evolves with a drift of one per time unit. The particular calculated value is compared to the comparison value by the function *compareValues()*, which returns *true* if the comparison operation is fulfilled, or *false*, otherwise.

4.1.2 State-based properties

Next to the atomic properties, STL properties can be negated (NOT) or combined with *AND* (conjunction) and *OR* (disjunction), see 2.2. In addition, there is the property $\sigma \models^t tt(true)$, for which the return value is always *true*. For a negation property, like $\sigma \models^t \neg\Psi$, it is sufficient to check Ψ and then invert the result (so return *false* if Ψ is fulfilled and *true*, otherwise).

Since at this point property formulas are considered, which are only dependent on a single point in time, it is possible to resolve a conjunction $\sigma \models^t \Psi_1 \wedge \Psi_2$ by simply evaluating Ψ_1 and Ψ_2 for this point independently and return *true* exactly when both are fulfilled (logical *AND*). Similarly, for a disjunction $\sigma \models^t \Psi_1 \vee \Psi_2$, *true* is returned exactly when at least one of both properties is fulfilled (logical *OR*). Note that the evaluation of conjunctions and disjunctions is different when considering intervals as explained later in this section.

4.1.3 The path-based until property

Evaluating the until property is far more complex due to considering its time interval. The challenge rising from the interval is that it has to be determined if a property holds starting from any time point, that lies within an interval, until the end of this interval.

The basic approach to model check an until property

Let $\Psi_1 U^{[t_1,t_2]} \Psi_2$ denote the until property that needs to be model checked by the tool. Algorithm 1 shows the main idea of the approach in a pseudo code. The return value of the function *checkUntilProperty()* is the point in time from when Ψ_2 is fulfilled and -1 if the until property does not hold.

The function *findTForProperty(tLeft, tRight, psi)*, called within Algorithm 1 multiple times, returns the first point in time within the event-free interval $(tLeft, tRight]$ at which a property $psi = \Psi$ holds or -1 if Ψ does not hold within the whole interval. Similar to this function, the function *findTForInvalidProperty(tLeft, tRight, psi)* returns the first point in time within the event-free interval $(tLeft, tRight]$ at which Ψ does not hold or -1 if Ψ holds for the whole interval.

At first, the time interval borders t_1 and t_2 and the properties have to be extracted from the node tree (lines 1–4). It is reasonable to check at first if Ψ_1 holds at time t, because in case it does not, the until formula cannot be

Algorithm 1 checkUntilProperty(t, *property*, *plot*)

```
1: $t1$ = property.getLeftTimeBound()
2: $t2$ = property.getRightTimeBound()
3: $psi1$ = property.getFirstSubProperty()
4: $psi2$ = property.getSecondSubProperty()
5: **if** (checkProperty($psi1$, $t$)= $false$) **then**
6:    **return** $-1$;
7: **end if**
8: $currentEventTime$ = plot.getNextEventTime($t$)
9: $previousEventTime = t$
10: **while** ($currentEventTime < t1$) **do**
11:    **if** (findTForInvalidProperty($previousEventTime$, $currentEventTime$,
       $psi1$) >= 0) **then**
12:       **return** $-1$
13:    **end if**
14:    $previousEventTime = currentEventTime$
15:    $currentEventTime$ = plot.getNextEventTime($currentEventTime$)
16: **end while**
17: **if** (plot.eventAtTime($t1$)=$true$ **and** checkProperty($psi2$, $t1$)= $true$) **then**
18:    **return** $t1$;
19: **end if**
20: $currentEventTime$ = plot.getNextEventTime($t1$)
21: $previousEventTime = t1$
22: **while** ($currentEventTime <= t2$) **do**
23:    $tPsi2$ = findTForProperty($previousEventTime$, $currentEventTime$,
       $psi2$)
24:    **if** ($tPsi2 >= 0$) **then**
25:       **if** (findTForInvalidProperty($previousEventTime$, $tPsi2$, $psi1$) >= 0)
          **then**
26:          **return** $-1$
27:       **else**
28:          **return** $tPsi2$
29:       **end if**
30:    **else**
31:       **if** (findTForInvalidProperty($previousEventTime$, $currentEventTime$,
          $psi1$) >= 0) **then**
32:          **return** $-1$
33:       **end if**
34:    **end if**
35:    $previousEventTime = currentEventTime$
36:    $currentEventTime$ = plot.getNextEventTime($currentEventTime$)
37: **end while**
38: **return** $-1$
```

fulfilled at all (lines 5–7). Otherwise, for the interval $[t, t_1)$ it is checked, too, if Ψ_1 is fulfilled (lines 8–16). Using a *while*-loop, it is iterated over the single events that happen within the interval borders and it is determined if Ψ_1 is fulfilled between every two neighbored events, forming a sub interval. How this sub interval is handled is explained in detail later. As soon as Ψ_1 is invalid, the whole until formula cannot be fulfilled and -1 is returned.

If Ψ_1 holds within the whole interval $[t, t_1)$, the interval $[t_1, t_2]$ can be investigated in an equivalent way. If there is an event at t_1, it needs to be checked if Ψ_2 already holds at this point, so that t_1 can be returned (lines 17–19). Otherwise, another *while*-loop iterates over the events until Ψ_2 is fulfilled (lines 22–37): Between two events and their sub interval, it can be checked if Ψ_1 holds until either Ψ_2 holds or the sub interval ends. If a point in time fulfilling Ψ_2 was found (lines 24–29), a decision can be made, depending on the fulfillment of Ψ_1. If Ψ_2 was not found (lines 30–34), it is checked if Ψ_1 holds for the whole interval. Otherwise, the next sub interval will be considered. If t_2 is reached before Ψ_2 has been fulfilled, the until formula does not hold.

Investigating on a property within an interval

The open question is how to determine if and when Ψ_1 or Ψ_2 of the until formula hold in an event-free interval. For a more efficient result, four cases are distinguished depending on whether the sub properties $\Psi 1$ and $\Psi 2$ are continuous or discrete properties.

The discrete properties cover the properties related to the number of tokens, the enabling status and the drift as well as any combined properties that do not contain any continuous atomic properties. Discrete properties are easy to model check, since their fulfillment does not change between events. So when checking if a discrete property holds within a whole sub interval, in which no event is happening (except at the borders), it only has to be checked if it holds at the borders.

For the continuous properties it can be distinguished between atomic and combined properties. Continuous atomic properties are the properties related to the fluid level, the clock and the enabling time as well as the upper and lower boundary and guard arc properties which based on the fluid level. When searching for a point in time within an interval $[t_{left}, t_{right}]$, from which one of these properties holds, it is important to consider that their drifts are constant within the interval, as it is assumed that no event happens before t_{right}. Algorithm 2 gives a rough illustration of how the approach is implemented.

It returns the time from which a continuous atomic property is fulfilled or -1 if it is not fulfilled at all. After extracting the values (fluid level, clock value etc.) in lines 1 and 2, it is checked if the property is fulfilled at t_{left}, see lines 3–5. If this is not the case, the point in time fulfilling the property is calculated using the drift (lines 6–8). Recall that the drift for the clock and enabling time is always one time unit. The time delta is $\Delta t = (v_P - v_{left})/drift$, where v_P is

Algorithm 2 findTForContinuousAtomicProperty(*tLeft*, *tRight*, *property*, *plot*)

```
 1: $vLeft = plot$.getValueAtTime($tLeft$)
 2: $vRight = plot$.getValueAtTime($tRight$)
 3: if (checkProperty($property, vLeft$) $= true$) then
 4:    return $tLeft$
 5: end if
 6: $drift = plot$.getDriftAtTime($tLeft$)
 7: if ($drift! = 0$) then
 8:    $delta = (property.value - vLeft)/drift$
 9:    if ($delta > 0$ and $(tLeft + delta) < tRight$) then
10:       return $(tLeft + delta)$
11:    end if
12: else if (checkProperty($property, vRight$) $= true$) then
13:    return $tRight$
14: else
15:    return $-1$
16: end if
```

the comparison value of the property and v_{left} is the value holding t_{left} (line 8). If $\Delta t > 0$ and $t_{left} + \Delta t < t_{right}$, the atomic property is fulfilled from $t_{left} + \Delta t$ (lines 9–10). If $t_{left} + \Delta t \geq t_{right}$, it is checked in lines 12–15 if the value v_{right} at t_{right} fulfills the property.

The approach to find a point in time within an interval $[t_{left}, t_{right}]$, from which a combined property is fulfilled, is dependent on the kind of property. For a conjunction $\Psi_1 \wedge \Psi_2$, a pseudo code is given in Algorithm 3.

Algorithm 3 findTForAndProperty(*tLeft*, *tRight*, *property*, *plot*)

```
 1: $psi1 = property$.getFirstSubProperty()
 2: $psi2 = property$.getSecondSubProperty()
 3: $tPsi1$ = findTForProperty($tLeft$, $tRight$, $psi1$)
 4: if ($tPsi1 = -1$) then
 5:    return $-1$
 6: end if
 7: $tNotPsi1$ = findTForInvalidProperty($tPsi1$, $tRight$, $psi1$)
 8: if ($tNotPsi1 = -1$) then
 9:    return $findTForProperty$(tPsi1,tRight,psi2)
10: else
11:    return $findTForProperty$(tPsi1,tNotPsi1,psi2)
12: end if
```

First the sub properties Ψ_1 and Ψ_2 are read from the property (lines 1–2).

Then the approach is finding the point in time t_{Ψ_1} first, from which Ψ_1 is fulfilled (line 3). If this point cannot be found, the conjunction does not hold (lines 4–6). Otherwise, in line 7, it is searched for a point $t_{\neg\Psi_1}$ within the interval $[t_{\Psi_1}, t_{right}]$, where Ψ_1 does not hold anymore. If no $t_{\neg\Psi_1}$ was found, it will be set to t_{right}. So then, the conjunction holds if a point in time can be found within $[t_{\Psi_1}, t_{\neg\Psi_1}]$ at which Ψ_2 is fulfilled because at this point both sub properties hold (lines 8–12).

Algorithm 4 shows a pseudo code for the handling of a disjunction $\Psi_1 \vee \Psi_2$. After reading the sub properties (lines 1–2), it is independently searched for a point where Ψ_1 respectively Ψ_2 holds (lines 3–4). If both points are found, the minimum of them is returned (lines 5–6). If only one point is found, it can be returned. Otherwise, the disjunction does not hold (lines 7–9). Note that for finding a point in time, from when a property does not hold, it is required to swap the conjunction and disjunction approaches, as De Morgan's law holds.

Algorithm 4 findTForOrProperty($tLeft$, $tRight$, $property$, $plot$)

```
1: $psi1 = property$.getFirstSubProperty()
2: $psi2 = property$.getSecondSubProperty()
3: $tPsi1 =$ findTForProperty($tLeft$, $tRight$, $psi1$)
4: $tPsi2 =$ findTForProperty($tLeft$, $tRight$, $psi2$)
5: if ($tPsi1 > -1$ or $tPsi2 > -1$) then
6:   return min($tPsi1, tPsi2$)
7: else
8:   return max($tPsi1, tPsi2$)
9: end if
```

For a negation property $\neg\Psi$, a point in time needs to be found within the specified interval, from when Ψ does not hold. At this point, $\neg\Psi$ is fulfilled.

Nested until properties

For the nested until case (as in 2.2.2), it is required to determine if an (inner) until property $\Psi_1 U^{[t'_1,t'_2]} \Psi_2$ holds from a point in time within a sub interval $[t_{left}, t_{right}]$. Note that it might be the case that Ψ_2 holds after t_{right} but the whole (inner) until is fulfilled starting from a point within $[t_{left}, t_{right}]$. In this case it is required to determine if Ψ_1 holds until the interval ends, with regards to the time bounds. As the until operator is path-based, this approach is different to the ones before. Algorithm 5 gives an overview about how the investigation on a sub interval is done. First, the time bounds and inner properties are read (lines 1–4). Then in line 5, the point in time t_{Ψ_2}, when Ψ_2 holds at first, it determined. If t_{Ψ_2} is found, it is sufficient to check if Ψ_1 is fulfilled within in the required time bounds before or not (lines 6–12). If t_{Ψ_2} is not found within the current sub interval, the until formula is called recursively from t_{right} in line 14 and 15. The recursive call returns a value for t_{Ψ_2} (which is then $\geq t_{right}$) and assures that Ψ_1 holds within $[t_{right}, t_{\Psi_2})$. So if Ψ_1 is also valid from any point t_{Ψ_1} within the interval $[t_l, t_{right})$ with $t_l = max\{t_{left}; t_{\Psi_2} - t'_1\}$ up to t_{right}, the nested until holds from t_{Ψ_1} (lines 17–19). If Ψ_1 does not hold (lines 20–22) or if the recursive call does not return a value for t_{Ψ_2} (lines 23–25), the until property does not hold.

Algorithm 5 findTForUntilProperty($tLeft$, $tRight$, $property$, $plot$)

```
 1: $t1$ = $property$.getLeftTimeBound()
 2: $t2$ = $property$.getRightTimeBound()
 3: $psi1$ = $property$.getFirstSubProperty()
 4: $psi2$ = $property$.getSecondSubProperty()
 5: $tPsi2$ = findTForProperty($tLeft + t1$, $tRight$, $psi2$)
 6: if ($tPsi2 >= 0$) then
 7:   $tPsi1$ = findTForProperty($min(tLeft, tPsi2 - t2)$, $tPsi2 - t1$, $psi1$)
 8:   if ($tPsi1 >= 0$ and findTForInvalidProperty($tPsi1$, $tPsi2$, $psi1$) $>= 0$)
      then
 9:     return $tPsi1$
10:   else
11:     return $notFulfilled$
12:   end if
13: else
14:   $property$.setLeftBorder(0)
15:   $tPsi2$ = checkUntilProperty($property$, $tRight$)
16:   if ($tPsi2 >= 0$) then
17:     $tPsi1$ = findTForProperty($max(tLeft, tPsi2 - t1)$, $tRight$, $psi1$)
18:     if ($tPsi1 >= 0$ and findTForInvalidProperty($tPsi1$, $tRight$, $psi1$)
        $>= 0$) then
19:       return $tPsi1$
20:     else
21:       return $notFulfilled$
22:     end if
23:   else
24:     return $notFulfilled$
25:   end if
26: end if
```

A challenge on working with time bounds arises with the distinction between open and closed intervals. While the until property itself is defined with a closed interval, within some of the functions to determine a point in time, it is necessary to exclude the interval border. This is done by adding or subtracting the minimum value that the type *Double* can hold in Java. Nevertheless, it has to be mentioned that there is always theoretically an infinite amount of points in time in between this value and the border, in which a property can begin or stop to hold. This cannot be implemented with a 100% correctness, as the representation of floating points is limited in every programming language and computer system. In addition, there will always be infinitesimal rounding errors.

4.2 Probability estimation

The model checking of the simulator allows to check if a property holds in a single simulation run or not. For Statistical Model Checking it is required to use statistical methods to evaluate the satisfaction of such properties. For this reason, the simulator is able to calculate confidence intervals for probabilities as described in 4.2.1 and do hypothesis testing. The latter is done by using the method of the Sequential Probability Ratio Test as pointed out in 4.2.2. Both methods correspond to the approaches presented before in 2.4.

4.2.1 The $P_{=?}[\Psi]$ problem

As mentioned in 2.4.1, it is possible to calculate a confidence interval for the probability of a property Ψ to hold. For the calculation of a specified confidence interval, it is necessary to calculate the mean $\bar{x}$ and to estimate the standard variance s^2. Recall that the interval borders are given by $\bar{x} \pm t_{n-1,1-\alpha/2}\sqrt{\frac{s^2}{n}}$ for n number of runs and $t_{n-1,1-\alpha/2}$ taken from the Student distribution.

For this method, the class *ConfidenceIntervalCalculator* was implemented in the tool, for which the model and property node and the different simulation parameters are required. An instance of this class holds a counter for the total number of runs n and of the number of runs r fulfilling the considered property. The class provides different functions that can be called by the simulation handler.

The first function to call is the function *calculateSSquareForProperty()*, which implements the main part of the approach: the calculation of the mean $\bar{x}$ and estimated standard variance s^2. The function is called for each single simulation run and requires the plot of the current run. A pseudo code for this function is given in Algorithm 6. For the first run, the counters n and r are initialized with zero (lines 1–4). Using the plot, an instance of the *Property-Checker* class is called to check if the property holds within this run (line 5). If it does, the counter r is incremented in line 6. Independently, the number of runs n is always incremented (line 8). The mean value is then updated to r/n in line 9. With this mean m, the estimated variance s^2 is calculated as $(r \cdot (n-r))/(n \cdot (n-1))$ as in lines 10–14.

After the calculation of the estimated standard variance, the value $t_{n-1,1-\alpha/2}$ can be drawn from the Student distribution, using the *SSJ* library. Note that the result is depending on the specified confidence level. Afterwards, it is verified if the number of runs is sufficient to achieve the required half interval width. If so, the width of the new interval is not greater than the required

Algorithm 6 calculateSSquareForProperty(*propertyChecker*, *root*, *model*, *n*, *r*, *currentRun*, *plot*, *mean*, *ssquare*)

```
1: if ($currentRun = 1$) then
2:     $n = 0$
3:     $r = 0$
4: end if
5: if ($propertyChecker$.checkProperty($root, model, plot$) $= true$) then
6:     $r = r + 1$
7: end if
8: $n = n + 1$
9: $mean = r/n$
10: if ($n = 1$) then
11:     $ssquare = 0$
12: else
13:     $ssquare = (r * (n - r))/(n * (n - 1))$
14: end if
```

half width w, so that the simulation can be terminated. Note that the defined minimum number of runs is considered, too. This condition is fulfilled if the number of runs is greater than $(t^2_{n-1,1-\alpha/2} \cdot s^2)/(w^2)$. Using all the determined values, the borders of the confidence interval can be finally calculated.

4.2.2 The $P_{\bowtie\Theta}[\Psi]$ problem

In 2.4.2, the method of the Sequential Probability Ratio Test was presented. For the cases $P_{\geq\Theta}[\Psi]$ and $P_{<\Theta}[\Psi]$, the null hypothesis $H_0 : p \geq p_0$ is tested against the alternative hypothesis $H_1 : p \leq p_1$ with $p_1 < p_0$, where for the latter case, the result is inverted. Oppositely, for the cases $P_{\leq\Theta}[\Psi]$ and $P_{>\Theta}[\Psi]$, hypotheses are $H_0 : p \leq p_0$ against $H_1 : p > p_1$ having $p_0 < p_1$, where the result is inverted for the greater case, too. The likelihood ratio is calculated by $\frac{p_1^r(1-p_1)^{n-r}}{p_0^r(1-p_0)^{n-r}}$ respectively $\frac{p_0^r(1-p_0)^{n-r}}{p_1^r(1-p_1)^{n-r}}$, if for n runs the property Ψ holds r times.

For the hypothesis testing, the class *SequentialProbabilityRatioTester* provides functions to run the SPRT on the simulation plots. The construction requires a model object, the property, and the different parameters for the test. The four cases $P_{\geq\Theta}[\Psi]$, $P_{<\Theta}[\Psi]$, $P_{\leq\Theta}[\Psi]$ and $P_{>\Theta}[\Psi]$, are separated by two options: the value *notEqual* has to be set to *true* for receiving inverted results and *nullHypothesisLowerEqual* for the cases having $H_0 : p \leq p_0$. Algorithm 7 shows the prepatory steps done (within the construction of the class) before the ratio test. In line 1, the boundary Θ is read from the property. Then the values A and B are calculated in lines 2–3, using the defined type-1-error α and type-2-error β so that $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$. Using the half width of the indifference region δ, the values for p_0 and p_1 are set, while separating between the two different cases for the null hypothesis (lines 4–10).

Algorithm 7 prepareRatioTest(*propertyChecker*, *root*, *model*, *nullHypothesisLowerEqual*, *alpha*, *beta*, *delta* *A*, *B*, *p*1, *p*0

```
1: $theta = (propertyChecker)$.checkProperty$(root, model)$
2: $A = (1 - beta)/alpha$
3: $B = beta/(1 - alpha)$
4: if $(nullHypothesisLowerEqual)$ then
5:     $p0 = \max(0, theta - delta)$
6:     $p1 = \min(1, theta + delta)$
7: else
8:     $p0 = \min(1, theta + delta)$
9:     $p1 = \max(0, theta - delta)$
10: end if
```

The pseudo code in Algorithm 8 illustrates how the SPRT is implemented in the tool. The SPRT is mainly done in a function called *doRatioTest()* which is called for each simulation run and requires the plot of the run and the parameters for the SPRT. The return value of the function indicates if a result has been achieved yet. Otherwise, more simulation runs are required to stay below the specified error bounds. The attribute *propertyFulfilled* indicates if the null hypothesis is fulfilled (but the result is inverted forthe cases named before). For the first run, r and n are initialized with zero in lines 1–4. If the property is fulfilled within the current run, the counter r is incremented (lines 5–7). n is also incremented for every simulation run in line 8. If the minimum number of runs is achieved, the likelihood ratio is calculated (line 10–11). The results is compared to the values of A and B and it is determined if no more simulation runs are needed and, if so, if the property is fulfilled or not (lines 12–18). If needed, the result is inverted in lines 19–21.

Algorithm 8 doRatioTest(*propertyChecker*, *root*, *model*, *n*, *r*, *p*1, *p*0, *A*, *B*, *notEqual*, *currentRun*, *minRuns*, *plot*, *propertyFulfilled*)

```
1: if ($currentRun = 1$) then
2:     $n = 0$
3:     $r = 0$
4: end if
5: if ($propertyChecker$).checkProperty($root, model, plot$) $= true$ then
6:     $r = r + 1$
7: end if
8: $n = n + 1$
9: $resultsAchieved = false$
10: if ($n >= minRuns$) then
11:     $ratio =$pow($p1, r$)$*$pow($1 - p1, n - r$)/(pow($p0, r$)$*$pow($1 - p0, n - r$))
12:     if ($ratio < B$) then
13:         $resultsAchieved = true$
14:         $propertyFulfilled = true$
15:     else if ($ratio > A$) then
16:         $resultsAchieved = true$
17:         $propertyFulfilled = false$
18:     end if
19:     if (notEqual) then
20:         $propertyFulfilled = !propertyFulfilled$
21:     end if
22: end if
23: return resultAchieved
```

5 Case Study

Electric vehicles have aroused more and more interest and popularity in the recent years due to their variety of benefits. Compared to conventional vehicles, they bring lower operating costs and less dependence on oil availability. In addition, plug-in electric vehicles (PEVs) provide possibilities for smart charging techniques that improve the off-peak use of renewable energies [34]. On the other hand, they consume big amounts of electrical energy which makes it necessary to manage undesirable peaks in the electrical consumption. When charging PEVs, the operator of the distribution system has an interest in reducing power losses and avoiding overloads. Furthermore, the owners of PEVs want to charge their vehicle batteries mostly overnight when charging at home, which can create a challenging peak for the operator. In [9], the impact of the charging behavior on the distribution grid in terms of power losses and voltage deviations is investigated.

This chapter presents the details and results of a case study that is done on the topic of the PEV charging process. First, Section 5.1 presents how it is validated if the tool returns correct and reliable results, compared to existing tools. Then, in Section 5.2, the case model is extended to investigate the feasibility of the analysis of models with more than two general multiple-shots transitions. Finally, Section 5.3 gives an evaluation of the whole case study.

5.1 Validation study

In [29], Hüls has presented a hybrid Petri net model (with a single one-shot general transition) to model the charging process for a public charging station for PEVs. In this scenario, when the vehicle owner arrives at this station, he has to choose whether the vehicle should be charged immediately or if he allows coordinated charging, which means that the charging process can be delayed to reduce the stress on the grid while maintaining that the battery is charged in time. The station itself then estimates his recurrence time. The actual recurrence time is modeled by a random variable.

The model is based on a *Tesla Model S* PEV with a battery capacity of 90kWh. The charging rates change with intervals of 10 minutes length and there is a maximum rate of 120kWh that is used within the first 20 minutes and then reduced gradually. This model is explained in detail in Section 5.1.1. The validation part of the study considers four different scenarios with parameters that are used for the random variables and different properties to be model checked. These scenarios are presented in Section 5.1.2 and the obtained results in Section 5.1.3.

5.1.1 The charging process model

Using the hybrid Petri net model from [29], the charging process for plug-in electric vehicles is the topic of the validation part of this case study. The tool is used to simulate the behavior of the Petri net. The main part of the model is shown in Figure 5.1. The model itself contains a dynamic continuous transition *load*, whose rate is dependent on the rates and enabling statuses of the continuous transitions *p1_rate*, *p2_rate*, ..., *p9_rate*, which are shown in Figure 5.2.

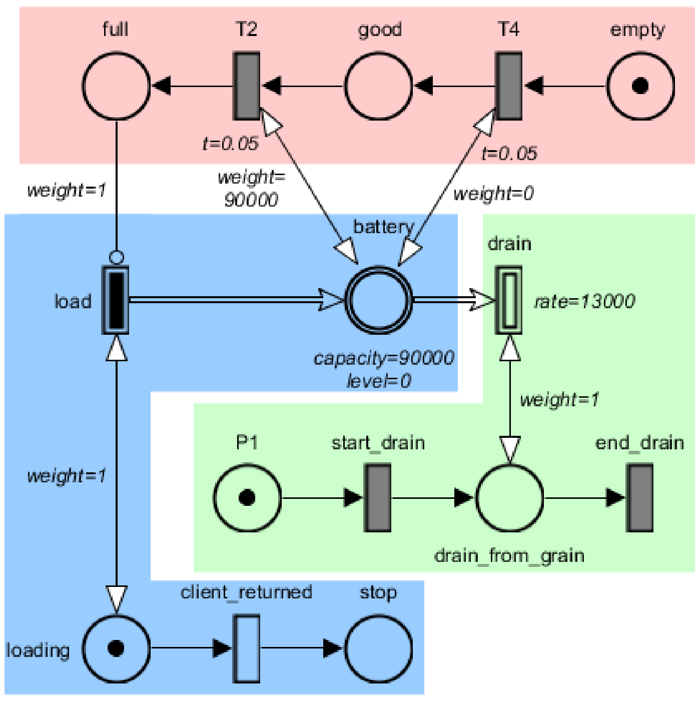


Figure 5.1: Hybrid Petri net model for the charging process of an EV - Part 1. Source: own representation based on [29, Figure 5: HPnG model of the Charging Station, p.3]

The charging of the PEV is possible as long as the customer has not returned to the vehicle, which is determined by a random variable modeled by the general transition *client_returned*. So, as long as the place *loading* contains a token, the dynamic continuous transition *load* is enabled until the place *battery* reaches its upper boundary (see blue part of Figure 5.1). Depending on the fluid level of this place, the transitions *T2* and *T4* might fire and change the status of the charging process presented by the places *empty*, *good* and *full* (red part of Figure 5.1). When the place *full* contains a token, the transition

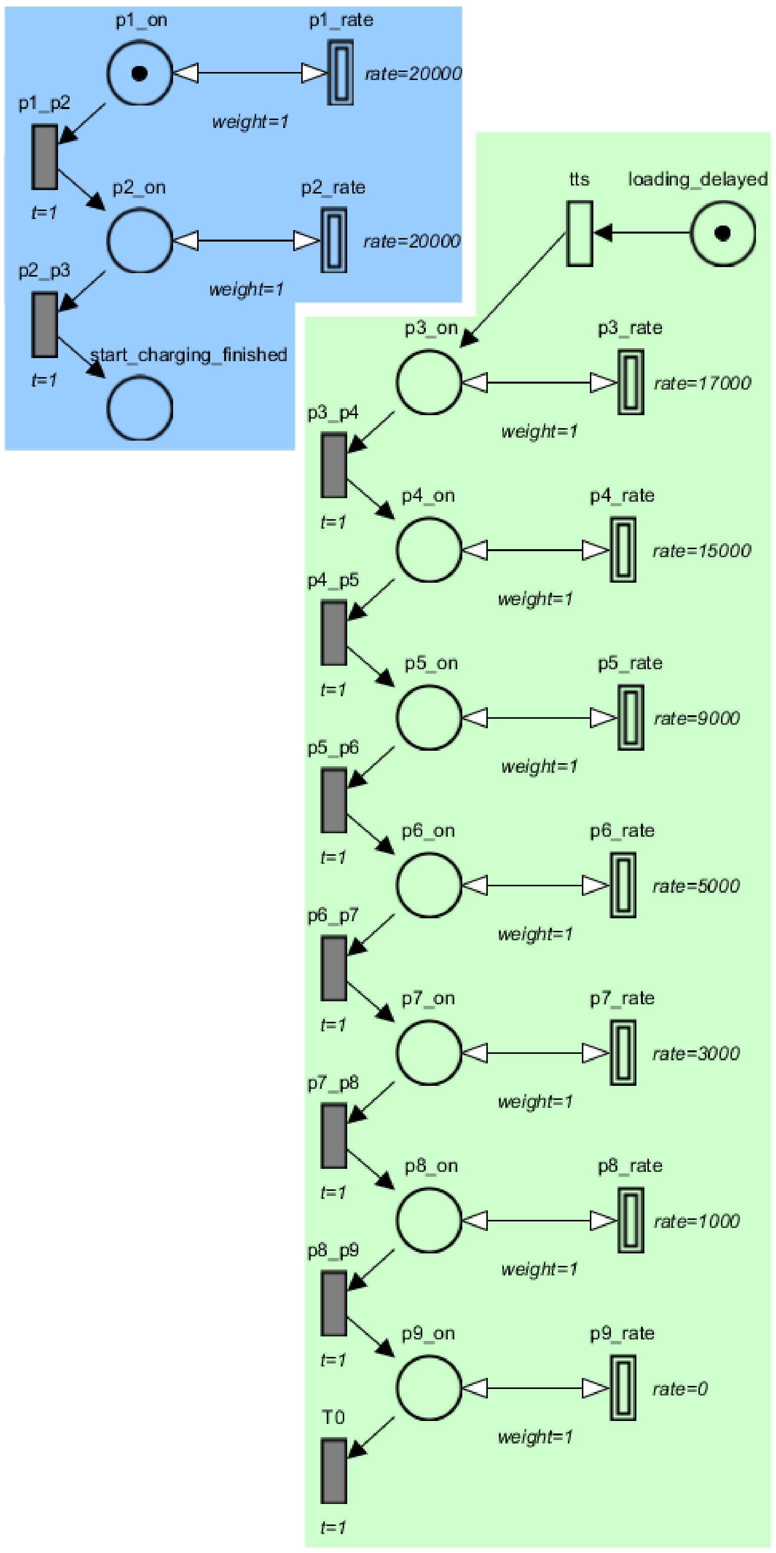


Figure 5.2: Hybrid Petri net model for the charging process of an EV - Part 2. Source: own representation

load is disabled. In addition, there can be a drain from the battery, which is modeled by the continuous transition *drain*. If the place *P1* contains a token within the initial state, as in the figure, the transition *start_drain* can fire and activate the drain (green part of Figure 5.1).

The transition *load* is a dynamic continuous transition and depends on all continuous transitions shown in Figure 5.2. Within the first time unit, *load* has the rate of the transitions *p1_rate*. Then the deterministic transition *p1_p2* fires, which indirectly disables *p1_rate* and enables *p2_rate* (see blue part of part of Figure 5.2). The rest of the charging process is determined by the transitions *p3_rate* to *p9_rate* which are connected in the same way as *p1_rate* and *p2_rate*, so that they got enabled in sequene and exactly one of them is enabled at a time. The general transition *tts* (abbreviation for 'time to start') has to fire at first to start the loading (with *p3_rate*, see green part of Figure 5.2). Note that in [29], only one general transition is used for the client recurrence but within this validation part of the study the time to start the charging process is also determined by a random variable. This is why a second general transition *tts* is included, which is deterministic in the model in [29]. When finally *p9_rate* is achieved, the charging is finished and the rate is set to zero. The rates are also set according to [29].

5.1.2 Case scenarios

For the validation of the results, different parameters need to be defined and the density functions for the general transitions need to be set. Multiple density functions are investigated for the general transitions to have a look on different scenarios and to compare the results to other tools investigating similar properties. For all scenarios, the confidence level is set to 0.99, so that in 99% of the times the real probability lies within the calculated intervals. The interval width is set to $2 \cdot 0.005 = 0.01$.

The first case scenario does not consider drain, so the number of tokens of the place *P1* is set to zero. This means that the battery is fully charged within six time units (i.e. 1 hour) after the transition *tts* has fired. It is assumed that the customer returns after about 9 hours (i.e. a working day). According to this, the recurrence is modeled by a normal distribution, starting with a mean of 54 (i.e. 9 hours) and a standard deviation of 6 (i.e. 1 hour). Within this scenario, the mean and standard deviation are slightly changed to investigate their influence on the results. The property for this study is to model check if the battery is reaching a fluid level of 90% of the capacity within 24 hours before the charging process terminates, assuming that the car is charged only once a day. The corresponding STL formula for the first scenario to check at the initial state at $t = 0$ is:

$$(m_{loading} = 1) \; U^{[0,144]} \; (x_{battery} >= 81000) \tag{5.1}$$

Note that 144 time units are equivalent to 24 hours and 81000Wh is 90% of the battery capacity. The following table shows the distribution settings used for the general transitions in this scenario.

Scenario	**Drain**	***tts***	***client_returned***
1a)	no drain	normal distribution $\mu = 40$, $\sigma = 6$	normal distribution $\mu = 54$, $\sigma = 6$
1b)	no drain	normal distribution $\mu = 40$, $\sigma = 6$	normal distribution $\mu = 52$, $\sigma = 6$
1c)	no drain	normal distribution $\mu = 40$, $\sigma = 12$	normal distribution $\mu = 52$, $\sigma = 12$
1d)	no drain	normal distribution $\mu = 40$, $\sigma = 12$	normal distribution $\mu = 54$, $\sigma = 12$

Table 5.1: Validation study parameters Scenario 1. Source: own representation.

The second scenario is similar to the first one, with the difference that the drain is enabled, i.e. place *P1* does contain a token, so that the transition *start_drain* can fire and indirectly enable the transition *drain* for one time unit (until it is disabled by transition *end_drain*). This means that the battery loses an amount of fluid of 13kWh. Hence, it can only achieve a maximum level of 77kWh. The until formula is adjusted with $x_{battery} >= 75000$ for two of three cases in this scenario. The second scenario is defined as follows:

Scenario	**Drain**	***tts***	***client_returned***	**Battery >=**
2a)	drain rate 13000	normal distribution $\mu = 40$, $\sigma = 6$	normal distribution $\mu = 54$, $\sigma = 6$	81000
2b)	drain rate 13000	normal distribution $\mu = 40$, $\sigma = 6$	normal distribution $\mu = 52$, $\sigma = 6$	75000
2c)	drain rate 13000	normal distribution $\mu = 40$, $\sigma = 12$	normal distribution $\mu = 52$, $\sigma = 12$	75000

Table 5.2: Validation study parameters Scenario 2. Source: own representation.

The third and fourth scenario are defined to compare the results of the simulation tool to the Fluid Survival Tool[1] (FST). The Fluid Survival Tool is able to model check hybrid Petri nets with one general transition by creating the Stochastic Timed Diagram (STD), which is introduced in [19, 21]. A description on the FST is given by Postema et. al. in [35]. There is also a functionality

[1] `https://code.google.com/archive/p/fluid-survival-tool/`

for hybrid Petri nets with multiple general transitions that computes the probability that the fluid level of a continuous place stays below a specified value over time. This probability is determined by discrete-event simulation, too, where all general transitions except one are sampled before each run and handled like deterministic transitions while for the remaining general transition an STD diagram is created. Within this study, the transition *client_returned* is the general transition that is handled differently. Unfortunately, it is not possible to choose the other general transition *tts*, as the FST throws an exception for this setting which still needs to be corrected.

The number of runs used within the FST is set to 7500 for Scenario 3 and to 5000 for Scenario 4 and the confidence interval is again set to 99%. Both scenarios do not consider any drain and the general transitions are set to exponential or uniform distributions because the FST only support these two kinds of distributions. Even though the exponential distribution does not comply with a real-life behavior, it is included to validate if both tools give similar results. As the FST is able to plot the probability that the battery level is below a value of 81kWh over time, the simulator is supposed to model check the same property ($x_{battery} <= 81000$) at different points in time. Table 5.3 and Table 5.4 show the settings for these scenarios.

Scenario	**Drain**	***tts***	***client_returned***	**Time**
3a)	no drain	uniform distribution $a = 46, b = 50$	uniform distribution $a = 52, b = 56$	$t = 0$
3b)	no drain	uniform distribution $a = 46, b = 50$	uniform distribution $a = 52, b = 56$	$t = 36$
3c)	no drain	uniform distribution $a = 46, b = 50$	uniform distribution $a = 52, b = 56$	$t = 54$
3d)	no drain	uniform distribution $a = 46, b = 50$	uniform distribution $a = 52, b = 56$	$t = 66$
3e)	no drain	uniform distribution $a = 46, b = 50$	uniform distribution $a = 52, b = 56$	$t = 72$

Table 5.3: Validation study parameters Scenario 3. Source: own representation.

In addition to the scenarios, the fluid level of the battery is plotted for 1000 runs using the different distribution settings of the scenarios 1a, 1c and 3a.

All the given results are also compared to another tool implemented by Adrian Godde at the University of Münster, presented in [22]. This tool calculates the Stochastic Timed Diagram [19, 21] for hybrid Petri nets with two general transitions by operations on Nef polyhedra and, hence, is able to compute probabilities for until properties.

Scenario	**Drain**	***tts***	***client_returned***	**Time**
4a)	no drain	uniform distribution $a = 0$, $b = 1$	exponential distribution $\lambda = 1$	$t = 0$
4b)	no drain	uniform distribution $a = 0$, $b = 1$	exponential distribution $\lambda = 1$	$t = 12$
4c)	no drain	uniform distribution $a = 0$, $b = 1$	exponential distribution $\lambda = 1$	$t = 36$
4d)	no drain	uniform distribution $a = 0$, $b = 1$	exponential distribution $\lambda = 1$	$t = 72$

Table 5.4: Validation study parameters Scenario 4. Source: own representation.

5.1.3 Results

The calculated mean values and confidence interval borders are given in Table 5.5 (each rounded to four decimal places). The table shows the results of the tool for all scenario cases, compared to the Nef polyhedra tool by Adrian Godde. The second and third column show the mean value and confidence interval for the probability of the defined until property. The fourth and fifth column show the number of runs and time needed by the simulation for the calculation of the values. The sixth column shows the results from the Nef polyhedra tool and the seventh column shows its minimum deviation from the borders of the confidence interval of the simulation, if the value lies outside the interval, otherwise zero. The last column shows the time needed by the tool, but it has to be said that the tools were executed on similar but still different systems.

In the first and second scenario, the probability of the battery reaching the defined fluid level is dependent on the distribution function and parameters of the general transitions. In the first scenario, it is higher for lower standard deviations and for a greater distance between the mean values of both general transitions. The first case of the second scenario (case 2a) shows that, due to the drift, a battery level of 81kWh cannot be reached at all but the level 75kWh can still be reached with a probability depending on the distribution parameters. For the uniform distributions in the third scenario, different points in time were considered to model check the property if the battery fluid level is $\leq$ 81kWh. For $t = 0$ and $t = 36$, this probability is one, since the level cannot be reached within this time period. For $t = 54$ and higher, the probability is then lower than 0.04. The fourth scenario is similar but with an exponential distribution for the client recurrence. For $t = 0$, the probability of staying below 81kWh is one, too. For $t = 36$ and higher, it stays above 0.96. Note that for the cases of having a probability of zero or one, a minimum number of one hundred runs was executed.

Furthermore, the results of the simulation are compared to the ones given by the Fluid Survival Tool for Scenario 3 and 4. The results for mean prob-

Scenario	Mean	CI	Runs	Time	Nef poly-hedra tool	Deviation	Time
1a)	0.9015	$[0.8965, 0.9065]$	23573	31122 ms	0.9026	0.0	2412 ms
1b)	0.8557	$[0.8507, 0.8607]$	32774	64100 ms	0.8556	0.0	2470 ms
1c)	0.7020	$[0.6970, 0.7070]$	55525	84179 ms	0.7016	0.0	2460 ms
1d)	0.7401	$[0.7351, 0.7451]$	51059	81049 ms	0.7411	0.0	2460 ms
2a)	0.0	$[0.0, 0.0]$	100 (min)	1523 ms	0.0	0.0	948 ms
2b)	0.8654	$[0.8604, 0.8704]$	30927	41256 ms	0.8643	0.0	1621 ms
2c)	0.6662	$[0.6612, 0.6712]$	59020	143403 ms	0.6667	0.0	1627 ms
3a)	1.0	$[1.0, 1.0]$	100 (min)	224 ms	1.0	0.0	140 ms
3b)	1.0	$[1.0, 1.0]$	100 (min)	388 ms	1.0105	+0.0105	2336 ms
3c)	0.02887	$[0.02387, 0.03387]$	7447	10588 ms	0.0313	0.0	2826 ms
3d)	0.03338	$[0.02838, 0.03838]$	8569	7594 ms	0.0322	0.0	2350 ms
3e)	0.02866	$[0.02366, 0.03366]$	7396	10777 ms	0.0323	0.0	2308 ms
4a)	1.0	$[1.0, 1.0]$	100	263 ms	1.0	0.0	139 ms
4b)	0.9676	$[0.9626, 0.9726]$	8330	13580 ms	0.9685	0.0	2671 ms
4c)	0.9686	$[0.9636, 0.9736]$	8083	10742 ms	0.9685	0.0	3008 ms
4d)	0.9672	$[0.9622, 0.9722]$	8435	11957 ms	0.9685	0.0	2371 ms

Table 5.5: Validation study results compared to the results of the Nef polyhedra tool by Adrian Godde. Source: own representation.

abilities and confidence intervals obtained from the FST are shown in Table 5.6. The last column shows the minimum deviation of the FST mean value from the confidence interval borders of the simulation results, if it lies outside the interval, otherwise zero.

Scenario	**Mean**	**CI**	**FST Mean**	**FST CI**	**Deviation**
3a)	1.0	$[1.0, 1.0]$	1.0	$[1.0, 1.0]$	0.0
3b)	1.0	$[1.0, 1.0]$	1.0	$[1.0, 1.0]$	0.0
3c)	0.02887	$[0.0239, 0.0339]$	0.0333	$[0.028, 0.0387]$	0.0
3d)	0.03338	$[0.0284, 0.0384]$	0.0368	$[0.0312, 0.0424]$	0.0
3e)	0.02866	$[0.0237, 0.0337]$	0.0395	$[0.0337, 0.0453]$	+0.0058
4a)	1.0	$[1.0, 1.0]$	1.0	$[1.0, 1.0]$	0.0
4b)	0.9676	$[0.9626, 0.9726]$	0.9754	$[0.9698, 0.9810]$	+0.0028
4c)	0.9686	$[0.9636, 0.9736]$	0.9748	$[0.9691, 0.9805]$	+0.0012
4d)	0.9672	$[0.9622, 0.9722]$	0.9756	$[0.97, 0.9812]$	+0.0034

Table 5.6: Validation study results compared to FST results. Source: own representation.

Figure 5.3 shows how the results of the Nef polyhedra tool lie relatively to the mean probability obtained from the simulation tool for Scenario 1 and 2. Similarly, Figure 5.4 shows how the results of the FST and the Nef polyhedra tool in relation to the simulation mean for Scenario 3 and 4.

For all scenarios, the results mostly meet the ones of the Nef polyhedra tool, i.e. all the results of this tool lie within the confidence interval borders and close to the mean, except for Scenario 3b. But the probability cannot be greater than 1.0 and in this case, the whole state space is included, so that the probability must be equal to 1.0. The incorrect result of the Nef polyhedra tool is caused by a numerical error, what he also confirms. In most of the cases, the run time of the Nef polyhedra tool is less than the one of the simulation tool, except Scenario 3b. The comparatively high run time for the simulation might be due to the execution on different systems, but also depends on the chosen simulation parameters influencing the number of runs.

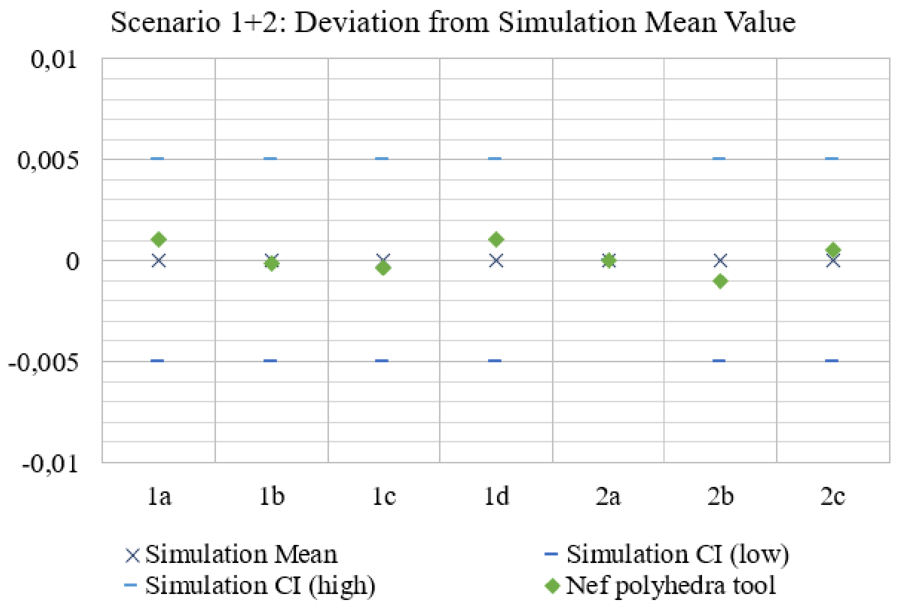


Figure 5.3: Deviations of the results from the mean value of the simulation for Scenarios 1 and 2. Source: own representation.

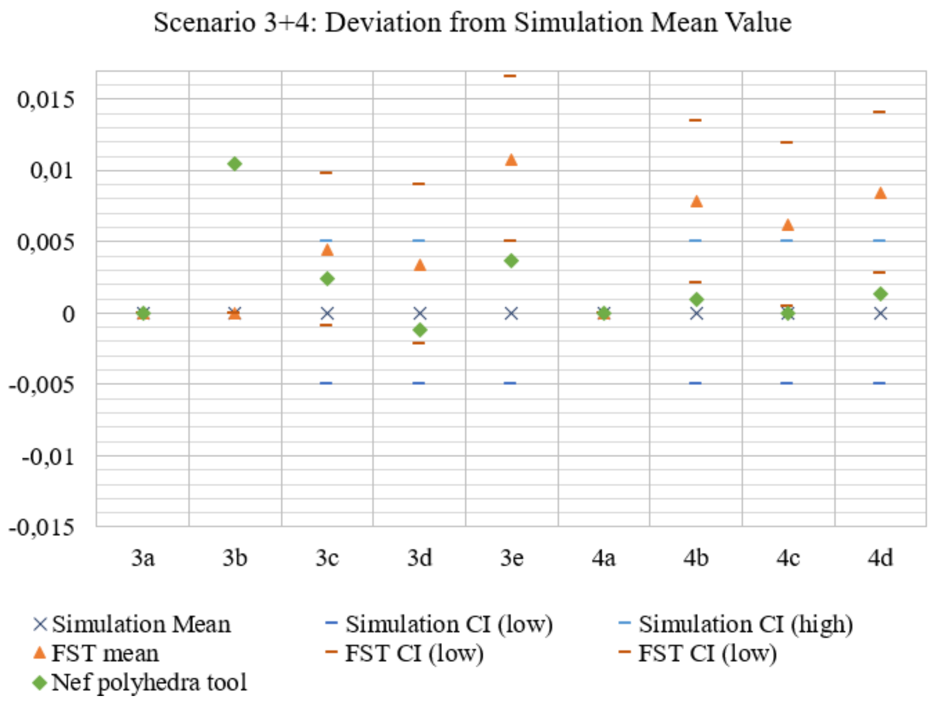


Figure 5.4: Deviations of the results from the mean value of the simulation for Scenarios 3 and 4. Source: own representation.

For Scenario 3a, 3b and 4a the FST also returns a probability of 1.0, so it does not differ from the simulation results. For Scenario 3c and 3d, the mean probability obtained from the FST lies inside the confidence interval of the simulation and the simulation mean also lies within the FST confidence interval. Only in Scenario 3e, the FST interval lies above the simulation interval. As the single scenario cases only differ in the point of time at which the property is model checked, we suppose that the difference between the results of both tools increases with the simulation time. Though the confidence intervals of the tools overlap for Scenario 4b, 4c, 4d, the mean values do not lie in the respective other confidence interval. Although it is unclear why the FST returns higher values, the results of the simulation tool and of the Nef polyhedra tool lie close to each other. Hence, we can assume that these results approximately match the real probability for the considered property. The run time of the FST was about 65-70 seconds, whereas it is possible to obtain results for multiple points in time in one execution (i.e. multiple cases of one scenario). Nevertheless, the FST still needs significantly more time compared to the number of simulation runs.

Figures 5.5, 5.6 and 5.7 show the fluid level for the battery for the scenarios 1a, 2c and 3a running for 1000 runs. Note that these figures are created by the tool itself. Hence, they show interpolated confidence interval borders as implemented within the tool (see Section 3.2.4). As this might not be reasonable to interpolate the confidence interval borders, the Figures created with Microsoft Excel in Section 5.2.3 do not show interpolated interval borders.

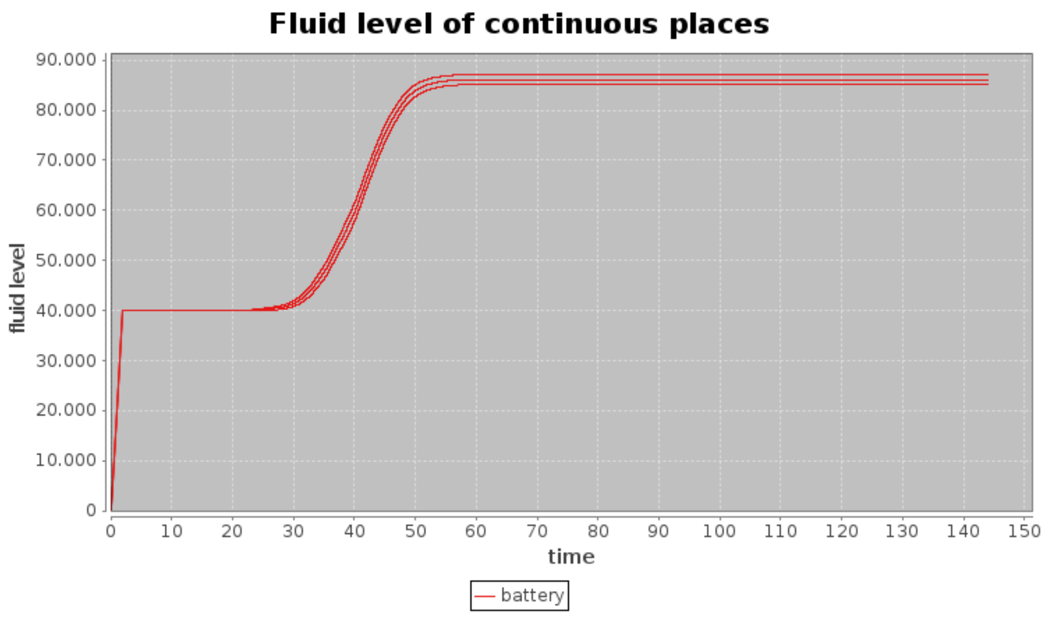


Figure 5.5: Plot of the place *battery* for Scenario 1a. Source: tool output.

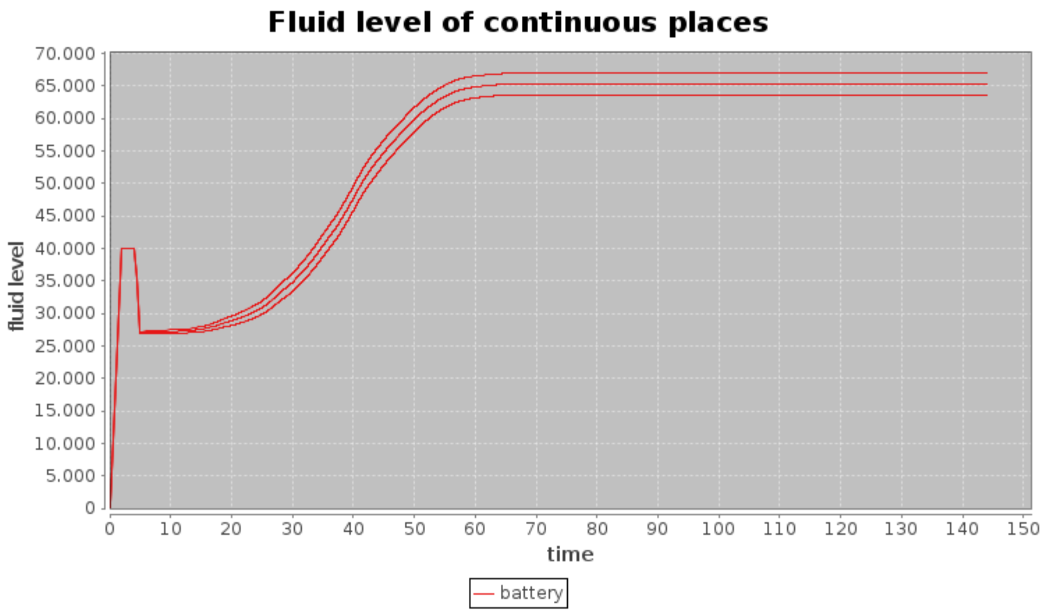


Figure 5.6: Plot of the place *battery* for Scenario 2c. Source: tool output.

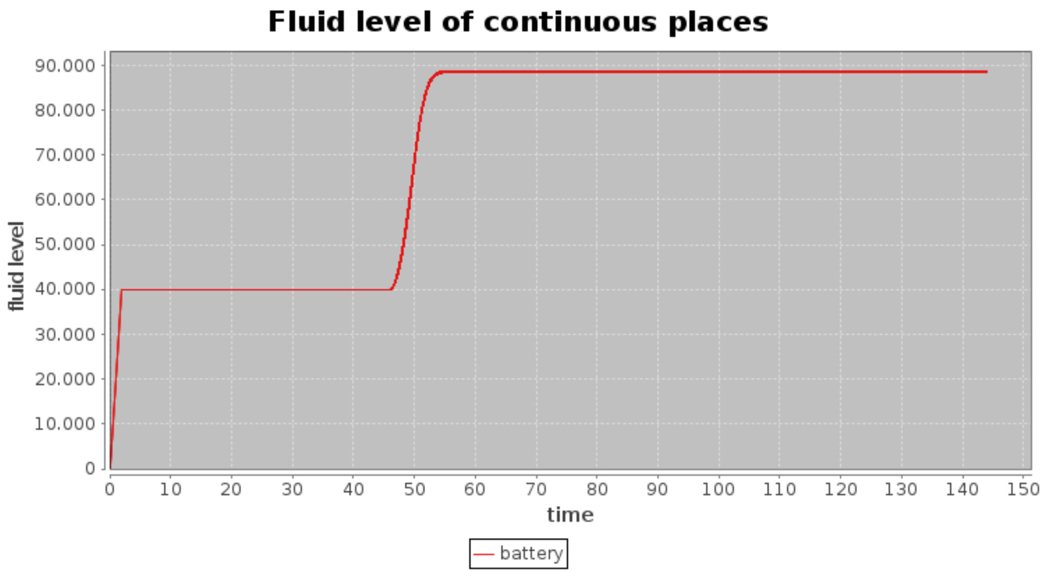


Figure 5.7: Plot of the place *battery* for Scenario 3a. Source: tool output.

All plots clearly show the pre-charging phase up to 40kWh but differ in the following rise. In Scenario 1a, the fluid level reaches a mean value of 86089Wh (which is about $95,65\%$ of the capacity) after about $t = 61$, i.e. approximately 10 hours. The boundary of 81kWh is reached after approximately 8 hours. In Scenario 2c, the fluid level rises up to 40kWh but then the drain starts decreasing the level by 13kWh. Due to the drain and higher standard deviation, only a mean value of 65304Wh (approximately $72,56\%$) is reached after about $t = 74$ (approximately 12 hours). The figure shows that the higher standard deviation also results in a higher width of the confidence interval (for the same confidence level of 99%). The mean value for Scenario 3a with the uniform distributions lies at 88566Wh (approximately $98,41\%$), which is the highest result. This value is reached already after $t = 56$, which is a little above 9 hours. The rise of the level is much steeper than in the other plots, as the range of possible values for the client recurrence is smaller than in the other scenarios. The border of 81kWh is reached just before at approximately $t = 51$, i.e. 8.5 hours.

5.2 Feasibility study

In addition to the validation of the results obtained from the tool, the case study is extended to check if the analysis of complex models with more than two general transitions is feasible. Therefore, the given hybrid Petri net model is extended to a model that allows charging the battery several times. The model randomly chooses (realised by a conflict between multiple immediate transitions) between three behavior patterns for the client recurrence (realised by three general transitions with different density functions). The hybrid Petri net is presented in Section 5.2.1, followed by an overview over the chosen parameters and distributions in Section 5.2.2. Finally, the results are presented in Section 5.2.3.

5.2.1 The extended model

The model used before serves as a basis for the extended feasibility study. The extended hybrid Petri net is shown in Figure 5.8. Both parts of the previous model are included, but the components modeling the drain are excluded here. Note that Figure 5.8 shows a yellow box for the structure from place *p5_on* and *p8_on* as a placeholder for the same structure as in Figure 5.2 (green part). The transitions *reset_p1_on*, *set_empty*, *set_empty2*, *new_charging* and *change_battery* have been added to set the charging process back to the initial state (except for the place *counter*), as if the EV connected to the charging station is replaced by another EV with an empty battery. In addition, the transitions *client_returned* and *T0* have been changed to immediate transitions, controlled by different guard arcs.

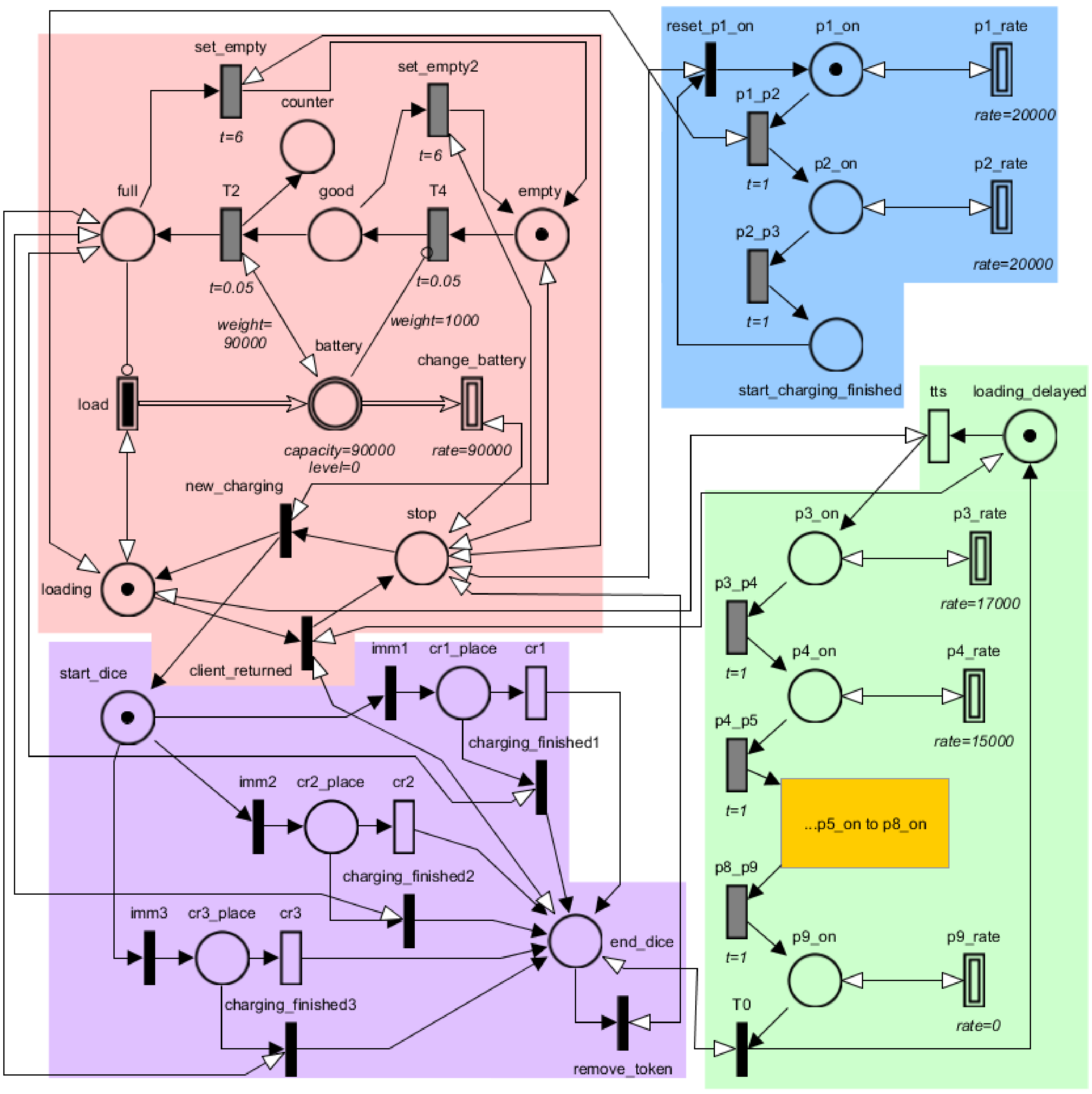


Figure 5.8: Extended hybrid Petri net model for multiple charging cycles of an EV. Source: own representation

The main extension of the model appears in the purple part of Figure 5.8, implemented by the places *start_dice*, *end_dice* and the places, transitions and arcs between them. Within this part, a conflict between transitions is enforced between the three immediate transitions *imm1*, *imm2* and *imm3*, having the same priorities. When the conflict of these transitions is solved, only one of the places *cr1_place*, *cr2_place* and *cr3_place* obtains a token, enabling the connected general transition (*cr1* respectively *cr2* respectively *cr3*). Either this transition fires first or the battery reaches is upper boundary, where the latter indirectly forces the immediate transition *charging_finished1* respectively *charging_finished2* respectively *charging_finished3* to fire. In both cases, the place *end_dice* obtains a token, so that the transition *client_returned* fires (as in the basic model). The place *battery* is flushed by the continuous transition *change_battery*, modeling the replacement of the full battery by an empty one. The hybrid Petri net then nearly holds its initial state, except for the place *counter* which obtains a token whenever the charging progress is completed (when transition *T2* fires). This counter allows to define properties on the number of completed charges.

5.2.2 Case scenarios

Within the feasibility study, two different properties are model checked. First, the mean probability, that the marking of the place *counter* is greater than or equal to a given value, is investigated. In addition, a confidence interval is calculated with a confidence level of 0.99 and an interval width of $2 \cdot 0.05 = 0.1$. (For a few cases with simulation time of 144, the confidence level has been adjusted to 0.98 due to performance reasons.) The property to model check at $t = 0$ is a measure for the satisfaction of the charging processes, which we call *productivity*. In STL this property is expressed as:

$$productivity = tt\ U^{[0,t']}(m_{counter} = z), \tag{5.2}$$

with $z \in \{1, 2, 4, 6\}$ and $t' \in \{18, 36, 54, 72, 108, 144\}$.

For this first productivity property, three different scenarios are defined, which correspond to different standard deviations for two of the general transitions implementing different client behaviors. The main setting is defined as follows:

tts	***cr1***	***cr2***	***cr3***
folded normal $\mu = 2$, $\sigma = 6$	normal $\mu = 8$, $\sigma = \sigma'$	normal $\mu = 14$, $\sigma = \sigma'$	uniform $a = 6$, $b = 16$

Table 5.7: Feasibility study setting for productivity. Source: own representation.

The charging can start (transition *tts*) right after the pre-charging phase (which was 40kWh within the first two time units) according to a folded normal distribution with a standard deviation of $\sigma = 6$. The three distributions modeling the client recurrence behavior are chosen so that the client is likely to return between one and less than three hours. The firings of first two transitions are normal-distributed and different standard deviations are investigated, so that $\sigma' \in \{1, 6, 12\}$. The third transition holds a uniform distribution.

The second property is to model check the probability if the battery is charged up to a level of 90% (i.e. 81kWh), 95% (i.e. 85.5kWh) or 100% (i.e. 90kWh) when the first client returns. We call this property *reliability* and it is expressed by an until property on the fluid level of the place *battery* and the general transitions modeling the client recurrence. The confidence level and interval width are set as above. The property to model check at $t = 0$ is expressed as:

$$reliability = (m_{end\_dice} = 0)\ U^{[0,t']}(x_{battery} >= b), \tag{5.3}$$

with $b \in \{81000, 85500, 90000\}$ and $t' \in \{12, 18, 36, 54, 72, 108\}$.

The place *end_dice* is positioned behind the general transitions modeling the client recurrence, so if one of these transitions fires, the place will obtain a token. We have defined four different scenarios for this property. This time, we keep fixed standard deviation for the two general transitions, but different means, modeling different recurrence times. This setting is defined as follows:

tts	***cr1***	***cr2***	***cr3***
folded normal $\mu = 2$, $\sigma = 6$	normal $\mu = \mu'$, $\sigma = 1'$	normal $\mu = \mu'$, $\sigma = 6$	uniform $a = 6$, $b = 16$

Table 5.8: Feasibility study setting for reliability. Source: own representation.

Again, the firings of first two transitions are normal-distributed, but now different means are investigated, so that $\mu' \in \{3, 6, 10, 14\}$. The third transition holds a uniform distribution.

5.2.3 Results

The results obtained for the property *productivity* are illustrated in Figure 5.9 for $z = 1$, Figure 5.10 for $z = 2$, Figure 5.11 for $z = 4$ and Figure 5.12 for $z = 6$. Remind that z was the value to which the number of tokens of the place *counter*, i.e. the cycles with a fully charged battery, is compared. The complete data is presented in Appendix F (each rounded to four decimal places).

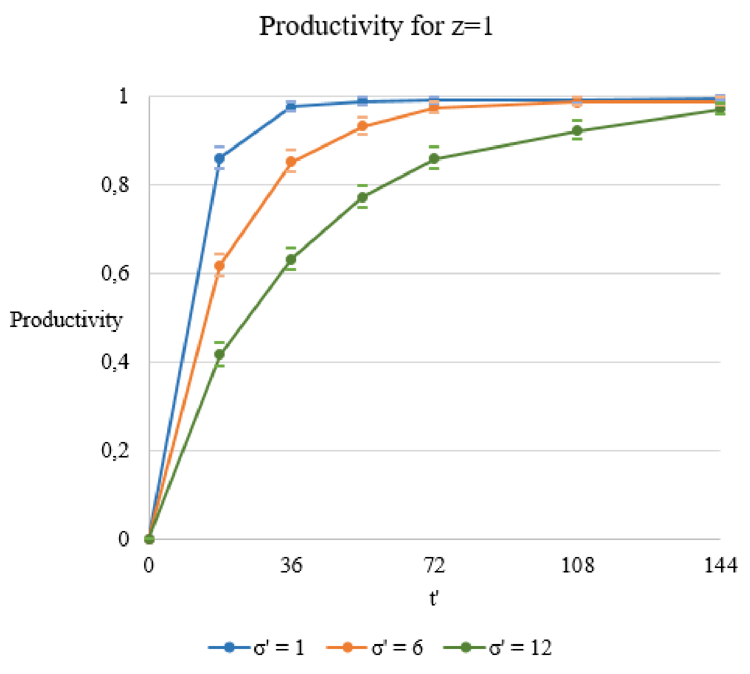


Figure 5.9: Feasibility study results for productivity with $z = 1$. Source: own representation

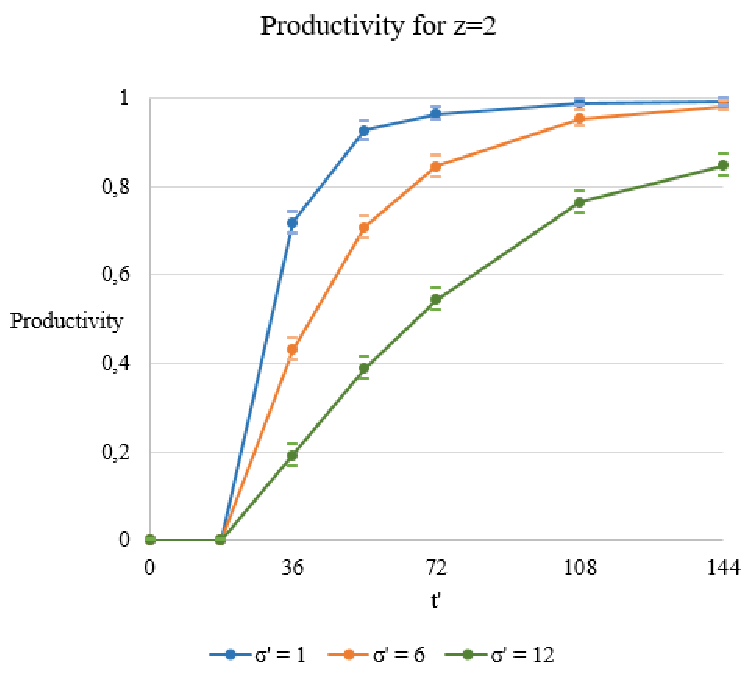


Figure 5.10: Feasibility study results for productivity with $z = 2$. Source: own representation

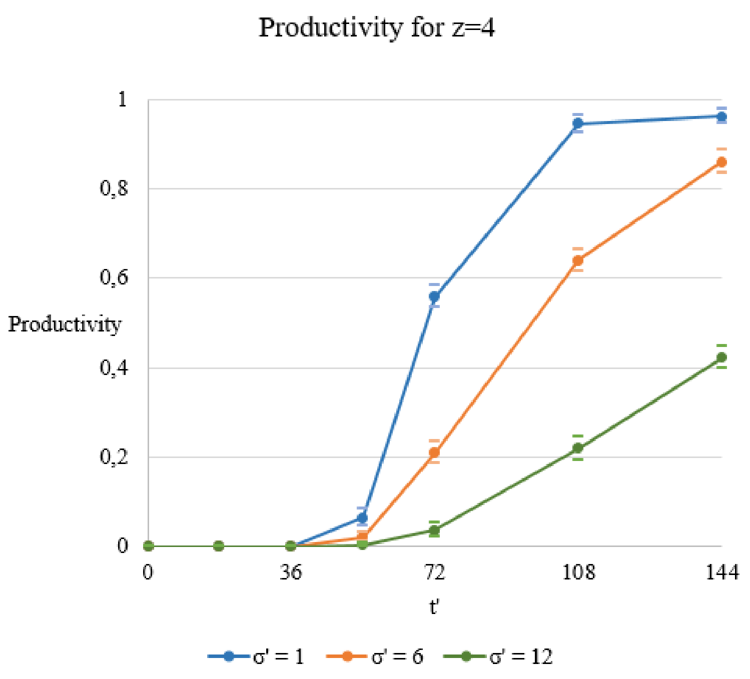


Figure 5.11: Feasibility study results for productivity with $z = 4$. Source: own representation

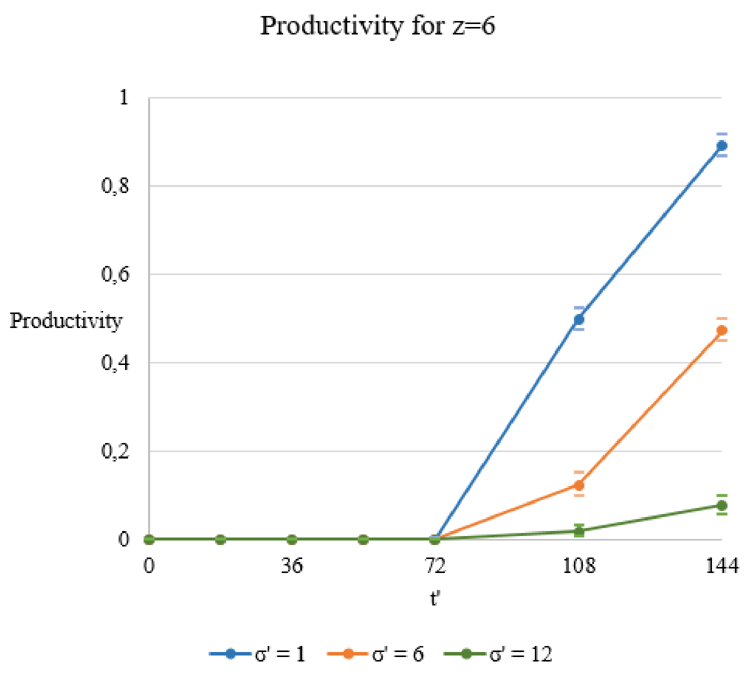


Figure 5.12: Feasibility study results for productivity with $z = 6$. Source: own representation

The figures show the mean probabilities as dots with a linear interpolation between the considered points in time. For these points, the calculated confidence intervals are also shows as bars above and below the dots. For all standard deviations there is a high mean probability greater than 0.86 to have at least one battery fully charged within the the first 12 hours ($t = 72$), as Figure 5.9 shows. After 24 hours ($t = 144$), this probability is greater than 0.97. From all the figures it becomes clear that the lower the standard deviation is, the higher is the mean probability to achieve the particular number of completed charging cycles and the earlier the probability increases. Especially, Figure 5.12 illustrates the big difference between the standard deviations, where the mean probability to have six batteries fully charged within a day is 0.8926 for $\sigma' = 1$, but only 0.078 for $\sigma' = 12$. Hence, even though the mean values of the normal distributions modeling the client recurrence are the same, the standard deviation has a great impact on the productivity of the charging station, as an early recurrence leads to an abortion of the charging cycle. It can be concluded that a high productivity can be only achieved if the clients return not before 90 minutes after the corresponding charging cycle has started. The last two figures also shows clearly that it is not possible at all to fully charge four batteries within the first 6 hours and six batteries within the first 12 hours, independent of the standard deviation. To sum up, we can expect with a high mean probability greater than 0.84 to have at least two batteries completely charged within a day (see Figure 5.10). For lower standard deviations, we can achieve two completely charged batteries within half a day with a mean probability also greater than 0.84 (Figure 5.10) and four completed charging cycles within a day with a mean probability greater than 0.86 (see Figure 5.11).

For the reliability, the results obtained for the related property are illustrated in Figure 5.13 for $b = 81000$, Figure 5.14 for $b = 85500$ and Figure 5.15 for $b = 90000$. Remind that b was the value to which the fluid level of the place *battery* is compared. The complete data is presented in Appendix G (each rounded to four decimal places).

These figures illustrate the property *reliability*, model checking if the first charging cycle achieves a specific battery fluid level indicated by b. Obviously, the later the client returns (modeled by a high mean μ'), the higher is the mean probability to reach a high battery level. With a mean probability greater than 0.87, we can expect a battery level of at least 90%, i.e. 81kWh ($b = 81000$, see Figure 5.13), and with a probability greater than 0.85 a battery level of 95%, i.e. 85.5kWh ($b = 85500$, see Figure 5.14). The mean probability to have the first battery completely charged up to 90kWh ($b = 90000$) is greater than 0.83. As only the first cycle is considered, the mean probability is approximately reached after 9 hours ($t = 54$) and it only slightly fluctuates afterwards. So it can be summarized that a battery level of more than 90% for the first battery cycle is likely reached if the client returns after approximately 2 hours and 20 minutes ($\mu' = 14$) and with a reliability of 96.4% the battery is then completely charged (see Figure 5.15).

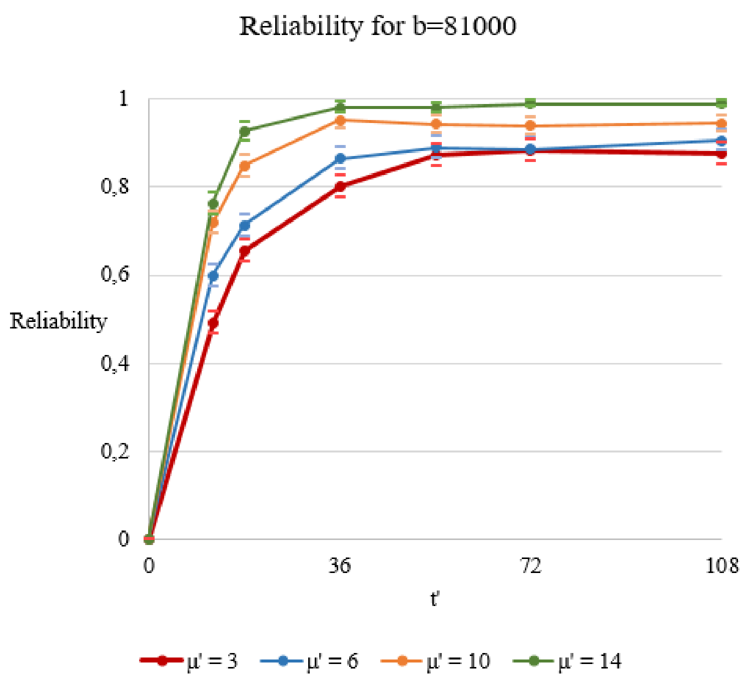


Figure 5.13: Feasibility study results for reliability with $b = 81000$. Source: own representation

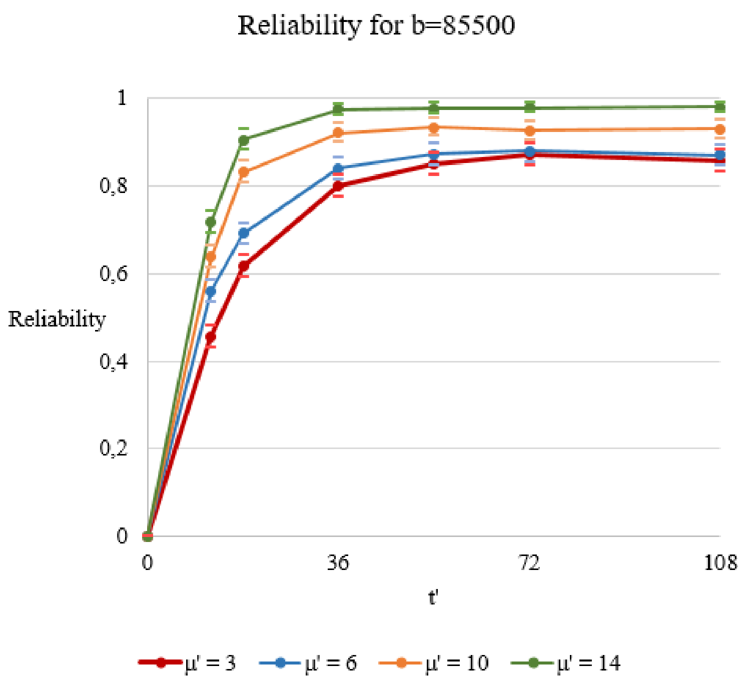


Figure 5.14: Feasibility study results for reliability with $b = 85500$. Source: own representation

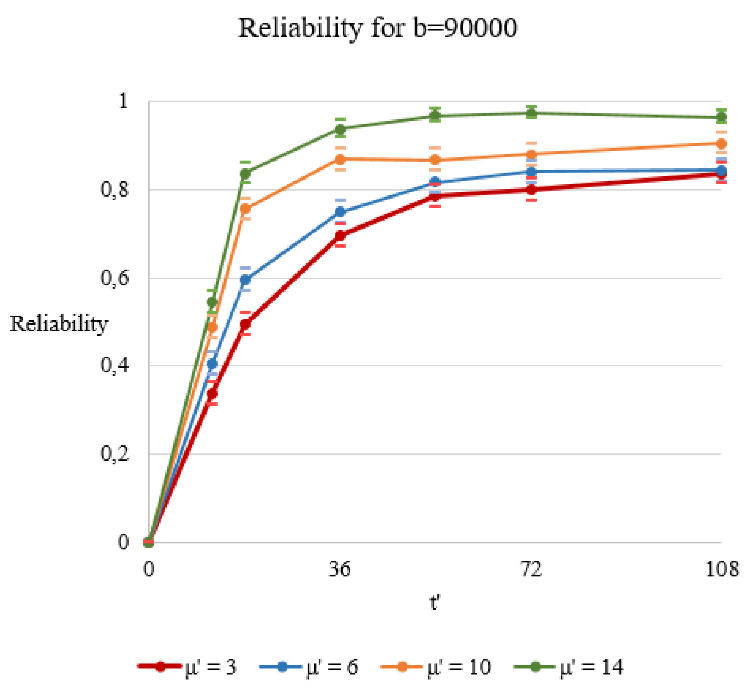


Figure 5.15: Feasibility study results for reliability with $b = 90000$. Source: own representation

5.3 Evaluation

The main conclusion from the validation part of this case study is that the simulation tool delivers results that generally match the results of the other tools. This has been the main purpose of the validation part of this study. Hence, it can be deduced that the simulator works properly and gives accurate results. In addition, the tool delivers accurate results in particular cases where the other tools are error-prone.

Table 5.5 and the tables in Appendix F and Appendix G show the number of simulation runs needed to obtain a confidence interval fulfilling the chosen parameters (confidence level and interval width). They show that there is a wide variety between 100 and 59020 runs, depending on the model, parameters, scenarios and required minimum number of runs. The number of runs and complexity of the model has a high impact on the run time, but within the case study all results could be calculated on an Intel Celeron N2820 system (as in 3.3.4) within seconds or a few minutes (less than 2.5 min). Most time-intensive was the creation of the plot outputs for the validation purposes. It becomes clear that the tool might have performance issues for model files that are much more complex than the case study models when considering a long simulation time and small confidence intervals with a high confidence level.

With regard to the charging process itself, the model check on the reliability property in Section 5.2.3 shows that, with normally distributed random variables with the named parameters, it is likely that a battery level of 90% is reached if the client does not return before 2.5 hours. The earlier the charging starts, the higher is the mean probability that the battery is above 90% when the client returns. But to reduce the stress on the grid, it might be desirable to charge just-in-time. In [29], Hüls arrives at the conclusion that just-in-time charging is less robust but provides higher flexibility compared to immediate charging. It is said that just-in-time charging suffers from high standard deviations. This is verified by the results of model checking the probability property in in Section 5.2.3. To improve the robustness while maintaining flexibility, Hüls introduces the considerate charging that extends just-in-time by adding a safety margin. This is equivalent to decreasing the mean of the transition *tts*, which results in a higher probability to reach a specific level. Scenario 2 of the validation study in Section 5.1.3 illustrates the influence of a drain. This reduces the battery level and hence has to be considered in real-life application.

The feasibility study points out that the tool is capable of handling a complex model with more than two general transitions that fire multiple times. The extended model contains a part that randomly chooses between three different client behaviors modeled by three conflicting immediate transitions. Furthermore, by adding the *counter* place, the charging processes for multiple batteries have been modeled. Hence, we came to the conclusion that is is likely to have at least two batteries completely charged with the named parameters.

It has to be said that the model is just a rough image of the real charging process. There might be more factors influencing the battery behavior in the real system and the results given in this study provides an orientation, but may differ from values that would be measured in reality. Even the extended model still holds abstractions and simplifications. One is that the battery is replaced by simply flushing the place with a rate of 90 kWh, assuming that the next battery is always completely empty. For future research, it is desirable to have a more accurate modeling of the discharging progress of batteries that correlates better with the real processes. Furthermore, the given model only counts the number of complete charging cycles. It would be also of interest to investigate how many started charging cycles reach different battery levels, compared to the number of total charges. It becomes clear that the given model is one further important step to model a realistic battery charging process. Besides, simulation never gives 100% accurate results, as the number of simulation runs is limited. Nevertheless, the study gives an insight about the possibilities that the tool offers and for the purpose of this thesis, the models have been appropriate to validate the results of the simulation tool and to investigate the feasibility of analyzing hybrid Petri nets with more than two multiple-shots transitions.

6 Conclusion, Discussion and Outlook

Within this thesis, an event-based simulator has been developed that is able to simulate hybrid Petri net models with random variables and to use the simulation results to model check properties on the models. Within this last chapter, a conclusion on the work is given in Section 6.2. Section 6.1 then presents a discussion on this thesis and finally, in Section 6.3 finishes with a future outlook.

6.1 Conclusion

Within the six chapters of this thesis, it was described and explained how an event-based simulator for hybrid Petri nets with random variables has been developed. The first chapter introduced into the topic and pointed out the thesis objectives. In the second chapter, the research fundamentals were recalled as a basis for the development and provided definitions and equations that have been transferred into the implementation. Among others, a logic was introduced to express properties on hybrid Petri nets and techniques of the Statistical Model Checking were recalled. The third chapter explained the implementation in detail, pointing out the tool requirements, the design steps, the realization of most important tool features and an evaluation of the tool's quality. A focus on the Statistical Model Checking techniques was given in the fourth chapter. Following in the fifth chapter, a case study was presented investigating on the charging process of electric vehicles and illustrating the correctness of the simulator. This current chapter closes this thesis with a discussion and future outlook.

As a result of this work, a tool for the simulation and model checking of hybrid Petri nets was implemented. The tool was integrated into the existing *libhpng* library of the safety-critical systems group at the university of Münster. The simulator is able to handle a variety of properties, expressed by the Stochastic Time Logic which is parsed by the tool. Within this thesis, approaches have been developed on how to determine events that can occur in the evolution of a hybrid Petri net, on how general transitions can be sampled according to distribution functions, on how state-based and until properties can be model checked upon the current system status and on how probabilities can be estimated using statistical methods. Furthermore, the user of the tool is able to set all major parameters for the simulation, for example the confidence level and the minimum number of simulation runs. A graphical output for the continuous places is given, too. The developed tool fulfills the requirements and delivers adequate results.

6.2 Discussion

The developed tool provides the opportunity to analyze models with an unlimited number of general transitions that can fire multiple times. Hence, the restriction to one or two one-shot general transitions from previous works has been overcome. The development process of the tool was based on the previous works on hybrid Petri nets and proven methods of the Statistical Model Checking and discrete-event simulation. This recent research offered concepts that have facilitated the development.

The hybrid Petri net formalism used within this thesis is mainly based on the definition by Gribaudo and Remke in [24]. This work provides a complete formalism as well as a detailed description on the evolution of hybrid Petri net models, which facilitated the implementation of the simulation enormously. Only the definition of the general transitions had to be adjusted to allow multiple transitions that can fire infinitely often. The Stochastic Time Logic by Ghasemieh et. al. in [19] has been fundamental for the expression of properties, but the logic had to be adapted to investigate probabilities for linear time properties. The previous works on hybrid Petri net analysis [19, 20, 21] lack a definition of the exact probability space for the model to take a single simulation path. For the sake of completeness, this probability has been defined within this thesis.

The techniques of Statistical Model Checking gave a (mathematically sound) basis for the determination of probabilities. The illustrations of Nimal in [33] have provided a complete foundation for the calculation of confidence intervals that can be managed by different parameters. For the hypothesis testing, the previous research offers different approaches. Within this thesis the Sequential Probability Ration Test has been chosen, which was presented in detail by Younes in [40]. This test has proven to match the purpose of the tool well, as it is sequential and can be combined easily with the method of discrete-event simulation. In addition, it does not require more than the optimal number of simulation runs and hence is advantageous for the run time of the tool. Nevertheless, it might have been of scientific interest to compare the results of different approaches for the hypothesis testing, as previous works on the comparison of these methods (like [13]) are available.

For the creation of graphical plots, it has been decided to use the interpolation technique for points in time between events. This has been necessary, as for different simulation runs, the events happen at different points in time. The chosen approach has been the most obvious, but causes performance issues for complex models or long simulation times. Previous works lack a solution for this problem. The existing research on discrete-event simulation (like [26]) has been fundamental for this work, but holds deficiencies in practical details. For the generation of random variables, the inversion method has delivered satisfying outcomes and could be easily implemented by using available libraries.

The results of the simulator have been validated successfully against other tools, like the Fluid Survival Tool in [35] and the Nef polyhedra tool presented in [22]. The fact that most of the results match well shows that (next to the tool of this thesis) the tools from previous works are reliable and accurate. Still, these tools and the tool of this thesis differ in their approaches, their use cases (with regards to general transitions), their performance and accuracy. For the analyzes of hybrid Petri nets, one has to weight up which of the tools fits best to the particular research purpose.

6.3 Outlook

The new simulation tool allows to extend the research on systems that can be modeled by hybrid Petri nets. These can be systems of any kind of application area with discrete and continuous parts and stochastic behavior.

Nevertheless, there are still open issues. The usability of the tool can be increased by creating an appealing user interface. The tool can also be extended by adding more statistical reporting and a graphical output for discrete places, too. The existing graphical output is also desired to be adjusted, so that confidence interval borders are not interpolated. In addition, the tool has still potential for further extensions and optimization, especially with regards to performance. In particular, it has to be investigated how the sequential methods of the tool can be parallelized in a reasonable way. A possible approach for this topic is given by AlTurki and Meseguer in [1].

With regards to the hybrid Petri net, it might be advantageous to extend the formalism by so-called flush-out arcs, which Gribaudo et. al. introduce in [25]. These arcs instantaneously empty a fluid place when a connected transition fires. They could be used in the extended model of the EV battery from Section 5.2. Further improvements for the given EV charging model have been already mentioned in Section 5.3.

Future works should also investigate how rare-event simulation can be integrated in the existing tool. Rare-event simulation deals with events that occur with very small probabilities. One method to handle this problem is importance sampling, where the model and its random variables are modified to enforce the desired cases. A possible approach to combine importance sampling with the Statistical Model Checking (SMC) techniques is given by Barbot et. al. in [3]. Furthermore, it might be interesting to see if it is possible to extend the Stochastic Time Logic for long-time average and steady state properties as well as unbounded until properties and implement these into the existing tool. In [36], Rabih and Pekergin investigate how SMC can be used to model check these kind of properties for Markov chains.

Last but not least, future works can now benefit from the developed tool by using the simulator to investigate (safety-critical and other) systems that can be modeled with hybrid Petri nets.

List of References

[1] M. AlTurki and J. Meseguer. PVESTA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *Proceedings of the 4th International Conference on Algebra and Coalgebra in Computer Science*, CALCO'11, pages 386–392. Springer, 2011.

[2] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate Symbolic Model Checking of Continuous-Time Markov Chains. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 146–161. Springer, 1999.

[3] B. Barbot, S. Haddad, and C. Picaronny. Coupling and Importance Sampling for Statistical Model Checking. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 331–346. Springer, 2012.

[4] J. Billington, M. Diaz, and G. Rozenberg. *Application of Petri Nets to Communication Networks: Advances in Petri Nets.* Springer, 1999.

[5] R. Cheng. Random Variate Generation. In J. Banks, editor, *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, pages 139–172. John Wiley & Sons, 1998.

[6] C. Chung. *Simulation Modeling Handbook: A Practical Approach.* CRC Press, 2004.

[7] G. Ciardo, D. Nicol, and K. Trivedi. Discrete-Event Simulation of Fluid Stochastic Petri Nets. *IEEE Transactions on Software Engineering*, 25(2):207–217, 1999.

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

[9] K. Clement-Nyns, E. Haesen, and J. Driesen. The Impact of Charging Plug-In Hybrid Electric Vehicles on a Residential Distribution Grid. *IEEE Transactions on Power Systems*, 25(1):371–380, 2010.

[10] A. David, K. Larsen, M. Mikucionis, D. B. Poulsen, A. Legay, S. Sedwards, and D. Du. Statistical Model Checking for Stochastic Hybrid Systems. In *Proceedings of the First International Workshop on Hybrid Systems and Biology*, HSB 2012, pages 122–136. ARXIV, 2012.

[11] R. David and H. Alla. On Hybrid Petri Nets. *Discrete Event Dynamic Systems*, 11(1):9–40, 2001.

[12] R. David and H. Alla. *Discrete, Continuous, and Hybrid Petri Nets.* Springer, 2nd edition, 2010.

[13] P.-T. de Boer, D. Reijsbergen, and W. Scheinhardt. Interactive Comparison of Hypothesis Tests for Statistical Model Checking. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 153–157. ICST, 2016.

[14] J. Desel. Validation of Process Models by Construction of Process Nets. In W. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management, Models, Techniques, and Empirical Studies*, pages 110–128. Springer, 2000.

[15] L. Devroye. *Non-Uniform Random Variate Generation.* Springer, 1986.

[16] F. Dicesare, G. Harhalakis, J.-M. Proth, M. Silva-Suarez, and F. Vernadat. *Practice of Petri Nets in Manufacturing.* Springer, 1993.

[17] H. Ghasemieh. *Analysis of Hybrid Petri nets with discrete random events.* PhD thesis, University of Twente, Netherlands, 2016. unpublished.

[18] H. Ghasemieh, A. Remke, and G. C. Boudewijn Haverkort. Approximate Analysis of Hybrid Petri Nets with Probabilistic Timed Transitions. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 127–134. ICST, 2016.

[19] H. Ghasemieh, A. Remke, and B. Haverkort. Survivability Evaluation of Fluid Critical Infrastructures Using Hybrid Petri Nets. In *Proceedings of the 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, PRDC '13, pages 152–161. IEEE Computer Society, 2013.

[20] H. Ghasemieh, A. Remke, and B. Haverkort. Hybrid Petri Nets with Multiple Stochastic Transition Firings. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '14, pages 217–224. ICST, 2014.

[21] H. Ghasemieh, A. Remke, B. Haverkort, and M. Gribaudo. Region-Based Analysis of Hybrid Petri Nets with a Single General One-shot Transition. In *Proceedings of the 10th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'12, pages 139–154. Springer, 2012.

[22] A. Godde. Translating Model Checking of Hybrid Petri Nets into Operations on Nef Polyhedra. Master's thesis, University of Münster, Germany, 2016. Unpublished.

[23] M. Gribaudo and A. Remke. Hybrid Petri Nets with General One-Shot Transitions for Dependability Evaluation of Fluid Critical Infrastructures. In *Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering*, HASE '10, pages 84–93. IEEE Computer Society, 2010.

[24] M. Gribaudo and A. Remke. Hybrid Petri Nets with General One-Shot Transitions. *Performance Evaluation*, 2016. Accepted for publication.

[25] M. Gribaudo, M. Sereno, A. Horváth, and A. Bobbio. Fluid Stochastic Petri Nets Augmented with Flush-out Arcs: Modelling and Analysis. *Discrete Event Dynamic Systems*, 11(1):97–117, 2001.

[26] B. Haverkort. *Performance of Computer Communication Systems: A Model-Based Approach.* John Wiley & Sons, 1998.

[27] M. Heymann, F. Lin, G. Meyer, and S. Resmerita. Analysis of Zeno behaviors in a class of hybrid systems. *IEEE Transactions on Automatic Control*, 50(3):376–383, 2005.

[28] C. Hirel, B. Tuffin, and K. Trivedi. SPNP: Stochastic Petri Nets. Version 6.0. In *Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, TOOLS '00, pages 354–357. Springer, 2000.

[29] J. Hüls and A. Remke. Coordinated charging strategies for plug-in electric vehicles to ensure a robust charging process. *10th EAI International Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '16*, 2016. Accepted for publication.

[30] S. Kleuker. *Grundkurs Software-Engineering mit UML: Der pragmatische Weg zu erfolgreichen Softwareprojekten.* Vieweg+Teubner, 2nd edition, 2011.

[31] A. Legay and B. Delahaye. Statistical Model Checking: An Overview. *CoRR*, abs/1005.1327, 2010.

[32] T. Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, pages 541–580, 1989.

[33] V. Nimal. Statistical Approaches for Probabilistic Model Checking. Master's thesis, Oxford University Computing Laboratory, UK, 2010.

[34] Oak Ridge National Laboratory. Plug-In Hybrid Electric Vehicle Value Proposition Study. Final Report, U.S. Department of Energy, Oak Ridge, TN, USA, July 2010.

[35] B. Postema, A. Remke, B. Haverkort, and H. Ghasemieh. Fluid Survival Tool: A Model Checker for Hybrid Petri Nets. In *Proceedings of the 17th International GI/ITG Conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, MMB & DFT 2014, pages 255–259. Springer, 2014.

[36] D. Rabih and N. Pekergin. Statistical Model Checking Using Perfect Simulation. In *Proceedings of the 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pages 120–134. Springer, 2009.

[37] D. Reijsbergen. *Efficient Simulation Techniques for Stochastic Model Checking*. PhD thesis, University of Twente, Netherlands, 2004.

[38] D. Riley, X. Koutsoukos, and K. Riley. Simulation of Stochastic Hybrid Systems using probabilistic boundary detection and adaptive time stepping. *Simulation Modelling Practice and Theory*, 18(9):1397–1411, 2010.

[39] K. Sen, M. Viswanathan, and G. Agha. Statistical Model Checking of Black-Box Probabilistic Systems. In *Proceedings of the 16th International Conference on Computer Aided Verification*, CAV '04, pages 202–215. Springer, 2004.

[40] H. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon University, PA, USA, 2004.

Appendices

Appendix A

Functional requirements

MUST criteria (mandatory)

1. The tool must be integrated into the existing Java library *libhpng* developed by the group of safety-critical systems at the University of Münster. The XML interface of this library (package *init*) has to be used to read the input for the simulation which is an XML representation of the hybrid Petri nets.

2. The tool must be able to run a discrete-event simulation for a specified period of time on the input model starting from the initial state of the model that is included in the model definition. This simulation includes:
   a) the distribution sampling for multiple multi-shot general transitions by setting seeds randomly for a generation of random variates for specified probability density functions
   b) the determination of the next discrete event, viewed from the current point in time
   c) the execution of an event by changing the state of the model

3. The tool must consider the following events:
   a) firing of immediate, deterministic and general transitions
   b) fulfilling a condition of a test or inhibitor arc
   c) reaching a lower or upper boundary of a continuous place

4. The tool must be able to handle at least the following distributions for the sampling of the general transitions:
   a) uniform distribution
   b) normal distribution
   c) folded normal distribution

5. The tool must be capable of model checking STL formulas as defined in 2.2.1 and 2.2.2. The tool has to be able to parse these formulas. The model checking has to be based on the simulation results. The properties have to be model checked for a specified point in time of the simulation.

- For $P_{=?}[\Psi]$ formulas, the tool has to compute confidence intervals for a specified confidence level and half interval width.
- For $P_{\bowtie\Theta}[\Psi]$ formulas with $0 \leq \Theta \leq 1$ and $\bowtie \in \{<, \leq, >, \geq\}$, the tool has to determine if the probability of the property holds the threshold or not using the sequential probability ratio test method. The test has to consider the specified type-1-error and type-2-error and the half-width of the accepted indifference region.
- For both cases the simulation has to determine the minimum number of required simulation runs for holding the given parameters.

6. The tool must be able to handle at least the following atomic properties:
   a) the fluid level of a continuous place compared to a decimal value
   b) the number of tokens in a discrete place compared to an integer value

SHOULD criteria (required)

1. The tool should display a textual output of the executed events with their occurrence times and the current fluid levels of all continuous places.
2. The tool should be able to handle the following atomic properties:
   a) the drift of a continuous place compared to a decimal value
   b) the enabled status of any kind of transition
   c) the clock value of a deterministic transition compared to a decimal value
   d) the enabling time of a general transition compared to an integer value
3. The tool should be capable to model check nested until formulas.
4. The tool should display a graphical plotting of the mean of the fluid levels of all continuous places with confidence intervals for a specified confidence level and interval width over the simulated period of time.
5. The tool should consider a specified minimum number of runs that should be at least executed before presenting any results.
6. The tool should consider a specified maximum number of runs that should be at most executed to avoid excessive run time.
7. The tool should make it possible for the user to adjust considered simulation parameters in a reasonable way.

8. The tool should have an appropriate error handling implemented. Any kind of user fault or execution exception should be detected, intercepted and a notification should be given to the user.

MAY criteria (desirable)

1. The tool is desired to handle the following properties (handled like atomic properties) derived from the specified atomic fluid level property:
   a) check if the upper boundary of a continuous place is reached
   b) check if the lower boundary of a continuous place is reached
   c) check if the condition of a guard arc is fulfilled
2. The tool is desired to run the simulation for a fixed number of runs. For the $P_{=?}[\Psi]$ problem, the confidence interval width then has to be adjusted to the number of runs. For the $P_{\bowtie\Theta}[\Psi]$ problem, it has to be decided after the runs if a decision can be taken or not.
3. The tool is desired to have an appropriate logging structure saving the main steps of the simulation process and results into an text file.
4. The tool is desired to be integrated into the shell of the *libhpng* test program, so that the simulation functions can be called from the shell.
5. The tool is desired to handle further known distributions for the sampling of the general transitions (as supported by the given Java library).
6. The tool is desired to read simulation parameters from a configuration file and store changed parameters into this file.

WON'T criteria (future goals)

1. The tool will not provide a *Javadoc* documentation as HTML files yet, but it is desirable to implement it in the future.
2. The tool will not be able yet to display a graphical plotting of the mean of the number of tokens of all discrete places with confidence intervals for a specified confidence level and interval width over the simulated period of time.
3. The tool will not run yet as parallel application with multi-threaded simulation runs.
4. The tool will not yet be able to do any statistical reporting on the simulation results (e.g. minimum, maximum and average number of tokens, total arrivals and leavings, arrival rates and leaving rates, average stay time of tokens for discrete places or average fluid of a fluid transition).

Appendix B

Pseudo code algorithm for finding the next event

Algorithm getAndCompleteNextEvent(currentTime)

```
event = new SimulationEvent($maxTime$)
for every enabled transition $t$ do
    if ($t$ is immediate transition and $t.priority > event.priority$) then
        $event$ = saveEvent($t, currentTime$)
    else if ($t$ is immediate transition and $t.priority = event.priority$) then
        $event$.addEvent($t$)
    else if (($t$ is deterministic transition or $t$ is general transition) and no
    immediate transition event found yet) then
        $newTime = currentTime + t.firingTime - t.enablingTime$
        if ($newTime < event.time$ or ($newTime = event.time$ and $t.priority >$
        $event.priority$)) then
            $event$.saveEvent($t, newTime$)
        else if ($newTime = event.time$ and $t.priority = event.priority$)
        then
            $event$.addEvent($t$)
        end if
    end if
end for

if (no immediate transition event found yet) then
    for every guard arc $a$ that is connected to a continuous place $p$ with
    $p.drift <> 0$ do
        $timeDelta = (a.weight - p.fluidLevel)/p.drift$
        if ($timeDelta >= 0$ and condition of $a$ not fulfilled yet) then
            $newTime = currentTime + timeDelta$
            if (newTime¡event.time) then
                $event$.saveEvent($a, newTime$)
            else if (newTime = event.Time) then
                if (current event is guard arc event and transition of $a$ is of higher
                rank as current event) then
                    $event$.saveEvent($a, newTime$)
                else if (current event is guard arc event and transition of $a$ is of
                same rank as current event) then
                    $event$.addEvent($a$)
                end if
            end if
        end if
    end for
    **for** every continuous place $p$ **do**
        **if** ($p.drift < 0$ **and** lower boundary of $p$ not reached yet) **then**
            $timeDelta$ =absolut($p.fluidLevel/p.drift$)
        **else if** ($p.drift > 0$ **and** upper boundary of $p$ not infinity **and** upper
        boundary of $p$ not reached yet) **then**
            $timeDelta = (p.upperBoundary - p.fluidLevel)/p.drift$
        **else**
            next $p$
        **end if**
        $newTime = currentTime + timeDelta$
        **if** ($newTime < event.Time$) **then**
            $event$.saveEvent($a, newTime$)
        **else if** (newTime = eventTime) **then**
            **if** (current event is no event or general transition event or guard arc
            for non-immediate transition event) **then**
                $event$.saveEvent($a, newTime$)
            **else if** current event is place boundary event **then**
                $event$.addEvent(a)
            **end if**
        **end if**
    **end for**
**end if**
advanceMarking($event.time$)
completeEvent()
**return** $event.time$
```

Appendix C

Supported distribution functions for general transitions

The following table gives an overview about the density functions that can be used for the firing behavior of general transitions. The documentation of the used library classes can be found at: `http://umontreal-simul.github.io/ssj/docs/master/namespaceumontreal_1_1ssj_1_1randvar.html`. The third column gives the name of the class for the random variate generator of the library.

Distribution	**Parameters**	**Generator**
Uniform	$\alpha, \beta \in \mathbb{R}$ with $a < b$	UniformGen
Normal	$\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$	NormalGen
Folded normal	$\mu \in \mathbb{R}^{\geq 0}, \sigma \in \mathbb{R}^+$	FoldedNormalGen
Half normal	$\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$	HalfNormalGen
Log normal	$\mu \in \mathbb{R}, \sigma \in \mathbb{R}^+$	LognormalGen
Inverse normal	$\mu, \lambda \in \mathbb{R}^+$	InverseGaussianGen
Beta	$a, b \in \mathbb{R}$ with $a < b$, $\alpha, \beta \in \mathbb{R}^+$	BetaGen
Cauchy	$\alpha \in \mathbb{R}, \beta \in \mathbb{R}^+$	CauchyGen
Chi	$\nu \in \mathbb{N}^+$	ChiGen
Chi square	$n \in \mathbb{N}^+$	ChiSquareGen
Exponential	$\lambda \in \mathbb{R}^+$	ExponentialGen
Fisher's F	$n, m \in \mathbb{N}^+$	FisherFGen
Frechet	$\alpha, \beta \in \mathbb{R}^+, \delta \in \mathbb{R}$	FrechetGen
Gamma	$\alpha, \lambda \in \mathbb{R}^+$	GammaGen
Inverse gamma	$\alpha, \beta \in \mathbb{R}^+$	InverseGammaGen
Gumbel	$\beta \in \mathbb{R} \setminus \{0\}, \delta \in \mathbb{R}$	GumbelGen
Laplace	$\beta \in \mathbb{R}^+, \mu \in \mathbb{R}$	LaplaceGen
Logistic	$\alpha \in \mathbb{R}, \lambda \in \mathbb{R}^+$	LogisticGen
Log logistic	$\alpha, \beta \in \mathbb{R}^+$	LoglogisticGen
Pareto	$\alpha, \beta \in \mathbb{R}^+$	ParetoGen
Rayleigh	$a \in \mathbb{R}, \beta \in \mathbb{R}^+$	RayleighGen
Student	$n \in \mathbb{N}^+$	StudentGen
Weibull	$\alpha, \lambda \in \mathbb{R}^+, \delta \in \mathbb{R}$	WeibullGen

Appendix D

Test cases

The following test cases are defined to verify the main functionality of the tool. The model files are related to *example.xml* and *example2.xml* presented in Appendix E.

1.
    - Start the tool.
    - Enter **read --p examples/example.xml** and press *return* to read in the provided example model.
    - Verify if the model was read in successfully.
    - Terminate the tool.

2.
    - Start the tool.
    - Enter **parse** and press *return*.
    - When prompted, enter the following expression and confirm with pressing *return*: **8.0:P<0.5(fluidlevel('pc1')<10)**
    - Verify if an error output is given as the expression is not accepted by the formula parser.
    - Verify if the log file *logFile.log* contains information about the incorrect formula.
    - Enter **parse** and press *return*.
    - When prompted, enter the following expression and confirm with pressing *return*: **8.0:P<0.5(fluidlevel('pc1')<10.0)**
    - Verify if the formula was parsed successfully and check the output.
    - Terminate the tool.

3.
    - Start the tool.
    - Enter **read --p examples/example2.xml** and press *return* to read in the provided example model.
    - Enter **check** and press *return*.
    - When prompted, enter the following expression and confirm with pressing *return*: **18.0:P=?(tokens('pd3')=3)**
    - Verify if the simulation has run completely and check if a mean value and a confidence interval is given as output.
    - Terminate the tool.

4.
    - Start the tool.

- Enter **read --p examples/example.xml** and press *return* to read in the provided example model.
- Enter **check** and press *return*.
- When prompted, enter the following expression and confirm with pressing *return*:
  5.0:P<=0.9(U[0.0,10.0](tokens('pd1')=1, uboundary('pc1')))
- Verify if the simulation has run completely and check if a result (property is fulfilled or not) is given as output.
- Terminate the tool.

5.
- Start the tool.
- Enter **change minruns --n 50** and press *return*.
- Enter **change fixedruns --n 50** and press *return*.
- Enter **read --p examples/example2.xml** and press *return* to read in the provided example model.
- Enter **plot --t 20.0** and press return.
- Verify if a graphical plot is given for the fluid levels of both continuous places and for the time interval [0,20].
- Terminate the tool.

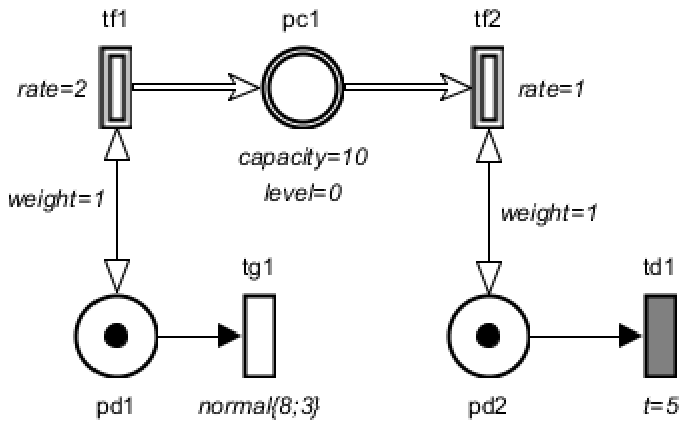
6.
- Start the tool.
- Enter **change halfintervalwidth --n 0.01** and press *return*.
- Enter **change halfwidthindifferenceregion --n 0.01** and press *return*.
- Enter **change confidencelevel --n 2.0** and press *return*.
- Verify if an error output is given as the value is not accepted for the confidence level.
- Verify if the log file *logFile.log* contains information about the incorrect parameter settings.
- Enter **change confidencelevel --n 0.99** and press *return*.
- Enter **change type1error --n 0.1** and press *return*.
- Enter **change type2error --n 0.1** and press *return*.
- Enter **change minruns --n 150** and press *return*.
- Enter **change maxruns --n 50000** and press *return*.
- Enter **change fixedruns --n 5000** and press *return*.
- Call the **storeparameters** command.

- Verify if all parameters in the file *libhpng_parameters.cfg* have been changed to the new values.
- Call the **loadparameters** command.
- Call the **printparameters** command.
- Verify if all parameters are shown as output and have been changed to the new values successfully.
- Terminate the tool.

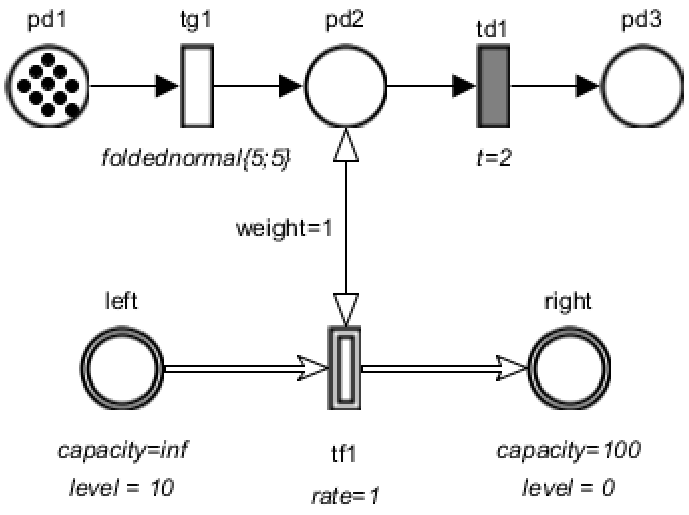
7.
- Start the tool.
- Enter **change logfile --n newlog.log** and press *return*.
- Enter **change fixedruns --n 0** and press *return*.
- Verify if an error output is given as the value not accepted for the number of runs.
- Verify if the log file *logFile.log* contains information about the incorrect parameter settings.
- Enter **change minruns --n 50** and press *return*.
- Enter **change fixedruns --n 50** and press *return*.
- Enter **set fixedruns** and press *return*.
- Enter **printresults on** and press *return*.
- Enter **read --p examples/nofile.xml** and press *return*.
- Verify if an error output is given as the file does not exist.
- Enter **read --p examples/incorrect.xml** and press *return*.
- Verify if an error output is given as the input file does not fulfill the XML schema requirements.
- Verify if the log file *logFile.log* contains information about the incorrect input file.
- Enter **read --p examples/example2.xml** and press *return* to read in the provided example model.
- Enter **plot --t 10.0** and press return.
- Verify if the events are shown as output for every simulation run and if the marking is given after every event.
- Verify if exactly 50 runs have been executed.
- Verify if the log file *newlog.log* was created and contains the simulation log information.
- Terminate the tool.

Appendix E

Example model files



example.xml. Source: own representation based on [24, Figure 6, p.25]



example2.xml. Source: own representation.

Appendix F

Feasibility study results for productivity

Results for $z = 1$ (each rounded to four decimal places):

σ'	t'	**Mean**	**CI**	**Runs**	**Time**
1	18	0.862	$[0.837, 0.887]$	1268	8416 ms
1	36	0.977	$[0.9648, 0.9892]$	1000 (min)	5229 ms
1	54	0.99	$[0.9819, 0.9981]$	1000 (min)	9250 ms
1	72	0.992	$[0.9847, 0.9993]$	1000 (min)	9899 ms
1	108	0.992	$[0.9847, 0.9993]$	1000 (min)	12197 ms
1	144	0.996	$[0.9908, 1.0]$	1000 (min)	22232 ms
6	18	0.619	$[0.594, 0.644]$	2509	8226 ms
6	36	0.8528	$[0.8278, 0.8778]$	1338	7747 ms
6	54	0.933	$[0.9126, 0.9534]$	1000 (min)	6844 ms
6	72	0.976	$[0.9635, 0.9885]$	1000 (min)	6734 ms
6	108	0.989	$[0.9805, 0.9975]$	1000 (min)	13186 ms
6	144	0.988	$[0.9791, 0.9969]$	1000 (min)	17602 ms
12	18	0.4166	$[0.3916, 0.4416]$	2585	11036 ms
12	36	0.6327	$[0.6077, 0.6577]$	2472	13795 ms
12	54	0.774	$[0.749, 0.799]$	1863	20792 ms
12	72	0.8603	$[0.8353, 0.8853]$	1281	13367 ms
12	108	0.924	$[0.9024, 0.9456]$	1000 (min)	17051 ms
12	144	0.972	$[0.9585, 0.9855]$	1000 (min)	21365 ms

Feasibility study results for productivity with $z = 1$. Source: own representation.

Results for $z = 2$ (each rounded to four decimal places):

σ'	t'	**Mean**	**CI**	**Runs**	**Time**
1	18	0.0	$[0.0, 0.0]$	1000 (min)	2678 ms
1	36	0.7185	$[0.6935, 0.7435]$	2153	11455 ms
1	54	0.926	$[0.9046, 0.9474]$	1000 (min)	6285 ms
1	72	0.964	$[0.9488, 0.9792]$	1000 (min)	12439 ms
1	108	0.989	$[0.9805, 0.9975]$	1000 (min)	31033 ms
1	144	0.991	$[0.9833, 0.9987]$	1000 (min)	46718 ms
6	18	0.0	$[0.0, 0.0]$	1000 (min)	1378 ms
6	36	0.4318	$[0.4068, 0.4568]$	2610	32621 ms
6	54	0.7075	$[0.6825, 0.7325]$	2202	39732 ms
6	72	0.8452	$[0.8202, 0.8702]$	1395	13396 ms
6	108	0.953	$[0.9357, 0.9703]$	1000 (min)	16992 ms
6	144	0.981	$[0.9699, 0.9921]$	1000 (min)	19678 ms
12	18	0.0	$[0.0, 0.0]$	1000 (min)	1421 ms
12	36	0.1926	$[0.1676, 0.2176]$	1656	7604 ms
12	54	0.3882	$[0.3632, 0.4132]$	2527	41062 ms
12	72	0.5447	$[0.5197, 0.5697]$	2638	34201 ms
12	108	0.7634	$[0.7384, 0.7884]$	1923	65810 ms
12	144	0.8468	$[0.8218, 0.8718]$	1129	40192 ms

Feasibility study results for productivity with $z = 2$. Source: own representation.

Results for $z = 4$ (each rounded to four decimal places):

σ'	t'	**Mean**	**CI**	**Runs**	**Time**
1	18	0.0	$[0.0, 0.0]$	1000 (min)	6843 ms
1	36	0.0	$[0.0, 0.0]$	1000 (min)	17882 ms
1	54	0.065	$[0.0449, 0.0851]$	1000 (min)	6819 ms
1	72	0.5591	$[0.5341, 0.5841]$	2622	53533 ms
1	108	0.946	$[0.9275, 0.9645]$	1000 (min)	24606 ms
1	144	0.962	$[0.9464, 0.9776]$	1000 (min)	85493 ms
6	18	0.0	$[0.0, 0.0]$	1000 (min)	6376 ms
6	36	0.0	$[0.0, 0.0]$	1000 (min)	2826 ms
6	54	0.021	$[0.0093, 0.0327]$	1000 (min)	30294 ms
6	72	0.2096	$[0.1846, 0.2346]$	1765	32607 ms
6	108	0.6407	$[0.6157, 0.6657]$	2449	47872 ms
6	144	0.8616	$[0.8367, 0.8866]$	1272	74934 ms
12	18	0.0	$[0.0, 0.0]$	1000 (min)	1976 ms
12	36	0.0	$[0.0, 0.0]$	1000 (min)	7127 ms
12	54	0.003	$[0.0.0075]$	1000 (min)	8445 ms
12	72	0.036	$[0.0208, 0.0512]$	1000 (min)	10673 ms
12	108	0.219	$[0.194, 0.244]$	1822	65382 ms
12	144	0.4223	$[0.3973, 0.4473]$	2117	118532 ms

Feasibility study results for productivity with $z = 4$. Source: own representation.

Results for $z = 6$ (each rounded to four decimal places):

σ'	t'	**Mean**	**CI**	**Runs**	**Time**
1	18	0.0	$[0.0, 0.0]$	1000 (min)	2756 ms
1	36	0.0	$[0.0, 0.0]$	1000 (min)	12037 ms
1	54	0.0	$[0.0, 0.0]$	1000 (min)	11187 ms
1	72	0.001	$[0.0.0036]$	1000 (min)	21481 ms
1	108	0.4987	$[0.4737, 0.5237]$	2659	60953 ms
1	144	0.8926	$[0.8676, 0.9176]$	1024	75864 ms
6	18	0.0	$[0.0, 0.0]$	1000 (min)	6573 ms
6	36	0.0	$[0.0, 0.0]$	1000 (min)	5798 ms
6	54	0.0	$[0.0, 0.0]$	1000 (min)	9460 ms
6	72	0.0	$[0.0, 0.0]$	1000 (min)	12530 ms
6	108	0.1246	$[0.0996, 0.1496]$	1164	42069 ms
6	144	0.4734	$[0.4484, 0.4984]$	2163	55198 ms
12	18	0.0	$[0.0, 0.0]$	1000 (min)	1443 ms
12	36	0.0	$[0.0, 0.0]$	1000 (min)	7780 ms
12	54	0.0	$[0.0, 0.0]$	1000 (min)	10203 ms
12	72	0.0	$[0.0, 0.0]$	1000 (min)	11818 ms
12	108	0.019	$[0.0079, 0.0301]$	1000 (min)	38894 ms
12	144	0.078	$[0.0561, 0.0999]$	1000 (min)	82137 ms

Feasibility study results for productivity with $z = 6$. Source: own representation.

Appendix G

Feasibility study results for reliability

Results for $b = 81000$ (each rounded to four decimal places):

μ'	t'	**Mean**	**CI**	**Runs**	**Time**
3	12	0.4913	$[0.4663, 0.5163]$	2658	6311 ms
3	18	0.656	$[0.631, 0.681]$	2401	6333 ms
3	36	0.8022	$[0.7773, 0.8272]$	1689	11991 ms
3	54	0.8731	$[0.8481, 0.8981]$	1182	11284 ms
3	72	0.8825	$[0.8575, 0.9075]$	1106	11631 ms
3	108	0.8749	$[0.8499, 0.8999]$	1167	24044 ms
6	12	0.599	$[0.574, 0.624]$	2556	6090 ms
6	18	0.7136	$[0.6886, 0.7386]$	2175	7105 ms
6	36	0.8651	$[0.8401, 0.8901]$	1245	4149 ms
6	54	0.889	$[0.864, 0.914]$	1054	9639 ms
6	72	0.885	$[0.86, 0.91]$	1087	13534 ms
6	108	0.906	$[0.8822, 0.9298]$	1000 (min)	15103 ms
10	12	0.7185	$[0.6935, 0.7435]$	2153	6019 ms
10	18	0.8479	$[0.8229, 0.8729]$	1374	5363 ms
10	36	0.951	$[0.9334, 0.9686]$	1000 (min)	6253 ms
10	54	0.942	$[0.9229, 0.9611]$	1000 (min)	5691 ms
10	72	0.939	$[0.9195, 0.9585]$	1000 (min)	8232 ms
10	108	0.944	$[0.9252, 0.9628]$	1000 (min)	14663 ms
14	12	0.7637	$[0.7387, 0.7887]$	1921	5336 ms
14	18	0.926	$[0.9046, 0.9474]$	1000 (min)	5247 ms
14	36	0.981	$[0.9699, 0.9921]$	1000 (min)	10946 ms
14	54	0.98	$[0.9686, 0.9914]$	1000 (min)	8531 ms
14	72	0.99	$[0.9819, 0.9981]$	1000 (min)	15254 ms
14	108	0.99	$[0.9819, 0.9981]$	1000 (min)	32966 ms

Feasibility study results for reliability with $b = 81000$. Source: own representation.

Results for $b = 85500$ (each rounded to four decimal places):

μ'	t'	**Mean**	**CI**	**Runs**	**Time**
3	12	0.4562	$[0.4312, 0.4812]$	2639	5782 ms
3	18	0.6176	$[0.5926, 0.6426]$	2513	9282 ms
3	36	0.7993	$[0.7743, 0.8243]$	1709	7301 ms
3	54	0.8492	$[0.8242, 0.8742]$	1366	12032 ms
3	72	0.8708	$[0.8458, 0.8958]$	1200	13377 ms
3	108	0.8569	$[0.8319, 0.8819]$	1307	39047 ms
6	12	0.5601	$[0.5351, 0.5851]$	2621	5765 ms
6	18	0.691	$[0.666, 0.716]$	2272	10206 ms
6	36	0.8401	$[0.8151, 0.8651]$	432	14695 ms
6	54	0.8727	$[0.8477, 0.8977]$	1186	19877 ms
6	72	0.8806	$[0.8556, 0.9056]$	1122	19122 ms
6	108	0.8703	$[0.8453, 0.8953]$	1203	56733 ms
10	12	0.6373	$[0.6123, 0.6622]$	2459	6115 ms
10	18	0.831	$[0.806, 0.856]$	1497	13570 ms
10	36	0.922	$[0.9001, 0.9439]$	1000 (min)	3919 ms
10	54	0.934	$[0.9137, 0.9543]$	1000 (min)	15481 ms
10	72	0.927	$[0.9058, 0.9482]$	1000 (min)	21754 ms
10	108	0.93	$[0.9092, 0.9508]$	1000 (min)	17291 ms
14	12	0.7182	$[0.6932, 0.7432]$	2154	3921 ms
14	18	0.906	$[0.8822, 0.9298]$	1000 (min)	15225 ms
14	36	0.975	$[0.9623, 0.9877]$	1000 (min)	2951 ms
14	54	0.977	$[0.9648, 0.9892]$	1000 (min)	16335 ms
14	72	0.979	$[0.9673, 0.9907]$	1000 (min)	15628 ms
14	108	0.98	$[0.9686, 0.9914]$	1000 (min)	28318 ms

Feasibility study results for reliability with $b = 85500$. Source: own representation.

Results for $b = 90000$ (each rounded to four decimal places):

μ'	t'	**Mean**	**CI**	**Runs**	**Time**
3	12	0.337	$[0.312, 0.362]$	2377	5942 ms
3	18	0.4934	$[0.4684, 0.5184]$	2659	10856 ms
3	36	0.6947	$[0.6697, 0.7197]$	2257	8867 ms
3	54	0.7835	$[0.7585, 0.8085]$	1806	15047 ms
3	72	0.7996	$[0.7747, 0.8246]$	1707	17242 ms
3	108	0.837	$[0.812, 0.862]$	1454	37682 ms
6	12	0.4055	$[0.3805, 0.4305]$	2565	6121 ms
6	18	0.5935	$[0.5685, 0.6185]$	2556	7449 ms
6	36	0.7485	$[0.7235, 0.7735]$	2004	21103 ms
6	54	0.8172	$[0.7922, 0.8422]$	1592	21172 ms
6	72	0.8401	$[0.8151, 0.8651]$	1432	27357 ms
6	108	0.8428	$[0.8178, 0.8678]$	1412	26265 ms
10	12	0.4876	$[0.4626, 0.5126]$	2658	4651 ms
10	18	0.7546	$[0.7296, 0.7796]$	1972	16661 ms
10	36	0.8676	$[0.8426, 0.8926]$	1224	14394 ms
10	54	0.8668	$[0.8418, 0.8918]$	1231	9227 ms
10	72	0.8786	$[0.8536, 0.9036]$	1137	25336 ms
10	108	0.905	$[0.8811, 0.9289]$	1000 (min)	25134 ms
14	12	0.5453	$[0.5203, 0.5703]$	2637	7129 ms
14	18	0.837	$[0.812, 0.862]$	1454	23539 ms
14	36	0.937	$[0.9172, 0.9568]$	1000 (min)	3304 ms
14	54	0.967	$[0.9524, 0.9816]$	1000 (min)	14063 ms
14	72	0.973	$[0.9598, 0.9862]$	1000 (min)	18023 ms
14	108	0.964	$[0.9488, 0.9792]$	1000 (min)	25845 ms

Feasibility study results for reliability with $b = 90000$. Source: own representation.

Plagiatserklärung

Hiermit versichere ich, dass die vorliegende Arbeit zum Thema *Entwicklung eines ereignisorientierten Simulators zum Model Checking von hybriden Petri-Netzen mit Zufallsvariablen* selbstständig verfasst worden ist, dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

Carina Pilch, Münster, 18. Oktober 2016

Ich erkläre mich mit einem Abgleich der Arbeit mit anderen Texten zwecks Auffindung von Übereinstimmungen sowie mit einer zu diesem Zweck vorzunehmenden Speicherung der Arbeit in eine Datenbank einverstanden.

Carina Pilch, Münster, 18. Oktober 2016