WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER

Master Thesis

# Translating Model Checking of Hybrid Petri Nets into Operations on Nef Polyhedra

**by Adrian Godde**

Matriculation number: 383 751

Supervised by:
Prof. Dr. Anne Remke
Prof. Dr. Klaus Hinrichs

Westfälische Wilhelms-Universität Münster

Department of Mathematics and Computer Sciences
Germany

November, 2016

Westfälische Wilhelms-Universität Münster

Masterarbeit

# Übersetzen des Model Checkings Hybrider Petri-Netze in Operationen auf Nef Polyedern

**von Adrian Godde**

Matrikelnummer: 383 751

betreut durch:
Prof. Dr. Anne Remke
Prof. Dr. Klaus Hinrichs

Westfälische Wilhelms-Universität Münster

Fachbereich Mathematik und Informatik
Deutschland

November, 2016

## Abstract

Hybrid Petri nets are used to model systems with discrete and continuous components. By extending them with so-called general one-shot transitions, stochastic variables are added to the system. The resulting hybrid Petri nets with general one-shot transitions hence allow to accurately represent randomised events in safety-critical hybrid infrastructures, thereby enabling a detailed examination of such systems.

In this thesis, a model checking approach for hybrid Petri nets with general one-shot transitions, that is based on boolean-set operations on Nef polyhedra, is introduced. Efficient algorithms for model checking the Stochastic Time Logic, which allows to specify properties for HPnGs, are defined over arbitrary HPnGs. In addition, a case study shows that the proposed method is applicable to real-world systems. The results are validated with the help of other tools, that use approaches different from the one presented here.

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# Chapter 1

# Introduction

Safety-critical systems, like, e.g., nuclear power plants or water treatment facilities, are directly or indirectly present in our everyday life. The functioning of these systems is usually vital for our environment or even for the life of many people. Hence, it is desirable to be able to guarantee that these systems work properly in every situation. A possible technique to verify, that a system meets certain requirements, is the so-called *model checking* approach. An abstraction of the system is defined in some expressive modelling formalism, which allows to describe the system's components and behaviour. If it can be shown, that the abstraction satisfies a property which is specified in a formal logic over the modelling formalism, the original system is assumed to satisfy the property, as well (cf. [7]).

Petri nets are a well-known modelling formalism for the description of systems and processes. They have been introduced by Carl Adam Petri in his dissertation (cf. [51]) for the examination of communication models. Since the concept can be easily adapted and extended to various types of applications, Petri nets are now used in many variations for the modelling of all kinds of systems. The original version of Petri nets can be used to describe discrete processes. However, many real-world systems possess both discrete and continuous properties. To be additionally able to accurately model such continuous qualities, the Petri net formalism has been extended by David and Alla to *hybrid Petri nets* (cf. [18]), which allow to use both discrete and continuous variables in a model. Hybrid Petri nets have in turn be enhanced by Gribaudo and Remke with so-called *general one-shot transitions* [31], which introduce stochastic variables to the formalism. The stochastic variables describe the occurrence of probabilistic events, for example the failure of a component or environmental influences on the modelled system.

*Hybrid Petri nets with general one-shot transitions* (HPnG) are able to represent systems with discrete, continuous and probabilistic features. Model checking of HPnGs can hence be used to verify the reliability of such systems. Model checking methods for HPnGs, that have been proposed before, were either limited to HPnGs with a single general one-shot transition (cf. [27]) or did not include algorithms to verify all types of STL formulae (cf. [29]). Hence, a novel approach for the model checking of hybrid Petri nets with multiple general one-shot transitions, which is based on boolean set-operations on *Nef polyhedra*, is developed in this thesis. In addition, the algorithms are implemented for HPnGs with two general one-shot transitions to be able to test the described concepts.

## 1.1 Related Work

A first approach to model checking of HPnGs, the *parametric reachability analysis*, has been described by Gribaudo and Remke in [31]. The possible evolutions of a system are characterised by *parametric locations*, on which the properties of the HPnG can be tested, efficiently. Ghasemieh et al. later presented a region-based analysis for the state space of an HPnG in [27]. The authors introduced the *Stochastic Time Diagram* (STD), a geometric structure which summarises similar states of the HPnG in so-called *regions*. A state space partitioning approach has been used before, for example for the analysis of timed automata (cf. [3]), probabilistic timed automata (cf. [44]), or *systems having piecewise-constant derivatives* (cf. [5]).

The *Stochastic Time Logic* (STL) for the specification of HPnG properties has been defined in [28] for HPnGs with a single general one-shot transition. The syntax of the STL is similar to logics like MITL (cf. [4]) or the *temporal layer* of STL/PSL (cf. [49]). The model checking techniques, that are presented in [28] for STL formulae over HPnGs with a single general one-shot transition, are the basis for the algorithms developed in this work. A definition of parts of the STL semantics for arbitrary HPnGs has been given in [29], where the algorithms for the creation of STDs have been extended to arbitrary HPnGs, as well. The semantics of the missing *time bounded Until* operator are provided in this thesis.

While the approaches presented so far aim for exact model checking of HPnGs, other research tries to use simulations to generate approximate results. For example, Carina Pilch recently presented a simulator, that samples the stochastic variables in an HPnG, thereby making the evolution of the model deterministic (cf. [52]). Repeating the sampling process several times allows to approximate the overall behaviour of the HPnG, and hence to predict the properties of the system. A similar procedure is used by Postema et al. in the Fluid Survival Tool where all but one stochastic variable is sampled, and the region-based analysis of Ghasemieh et al. is applied to the simplified HPnG (cf. [54]). The simulation approach has been widely used for other model formalisms, as well. For example, the reachability analysis for Fluid Stochastic Petri Nets is not decidable if the model contains more than two continuous variables (cf. [31]). Hence, simulation is the only option to examine more complex FSPNs.

## 1.2 Outline

The thesis begins with an introduction to hybrid Petri nets with general one-shot transitions in Chapter 2. Subsequently, Nef polyhedra and some of their properties are presented in Chapter 3. Chapter 4 discusses Stochastic Time Diagrams, and the algorithms to generate them. In Chapter 5 the Stochastic Time Logic, which is used to specify properties of HPnGs, is defined. Based on these preliminaries, Chapter 6 introduces the new algorithms for the model checking of HPnGs. In the following Chapter 7, a brief overview of the implementation, which accompanies this work, is given. The algorithms and the implementation presented in the previous two chapters are tested in a case study, which is described in Chapter 8. The conclusion in Chapter 9 briefly recaps the discussed topics and completes the thesis.

# Chapter 2

# Hybrid Petri Nets with General One-Shot Transitions

*Hybrid Petri nets with general one-shot transitions* (HPnG) are Petri nets with discrete, continuous, and probabilistic features, which have been first presented in [31]. Only the most important features of HPnG considering the discussions of the following chapters are presented here. A more detailed description of the HPnG formalism can be found in [32].

HPnGs extend the well-known theory of conventional *discrete autonomous Petri nets* (cf. [19]) by introducing new types of places, transitions, and arcs, as well as the corresponding semantics for these additional components. In addition, HPnGs are non-autonomous since their state depends on the observed point in time.

To allow for an extensive analysis of the characteristics of HPnGs, certain restrictions have to be imposed on the formalism. On the one hand, the occurrence of an infinite number of events in a finite time interval during the evolution of an HPnG has to be excluded. This undesired property is called *Zeno-behaviour* (cf. [17]). In this category fall, for example, cycles of immediate transitions, which thus have to be avoided when constructing an HPnG. On the other hand, we only perform *time-bounded reachability analysis*, that is HPnGs are always examined for some fixed time interval $[0, t_{max}]$, to limit the total number of events. If an HPnG is subject to these restrictions, the concepts, which are discussed in the course of this thesis, can be applied to the model.

The chapter is opened by presenting the syntactic elements of the HPnG formalism, which ultimately leads to the definition of HPnGs (Section 2.1). After that, the meaning of the individual components is explained (Section 2.2). Section 2.3 defines the state and the evolution of HPnGs, either of which are essential terms in the following chapters. All parts are accompanied by small sample HPnGs, to highlight the introduced concepts.

## 2.1 Primitives of an HPnG

Just like other Petri nets HPnGs consist of *places*, *transitions* and *arcs*. Combined with the *marking* these primitives allow us to describe a system in a specific state as an HPnG.

### 2.1.1 Places, Transitions, and Arcs

The basic components of an HPnG are divided into several subgroups, to clearly differentiate between discrete, continuous and probabilistic components of the system. The set $\mathcal{P}$ of places consists of the subsets $\mathcal{P}^D$ of *discrete places* and $\mathcal{P}^C$ of *continuous places.*

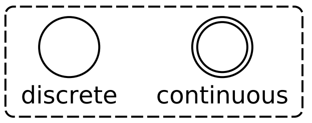


Figure 2.1: Representation of places in an HPnG.

The visualisation of the two types can be found in Figure 2.1. Discrete places are represented by a simple circle, while continuous places are pictured as a double framed circle.

The process of a system is always implied by moving physical or abstract entities (*token*, see below) through the system. For HPnGs, this transport mechanism is expressed by the set $\mathcal{T}$ of transitions. The set consists of four subsets: the *immediate transitions* $\mathcal{T}^I$, the *deterministic transitions* $\mathcal{T}^D$, the eponymous *general one-shot transitions* $\mathcal{T}^G$ and the *continuous transitions* $\mathcal{T}^F$.

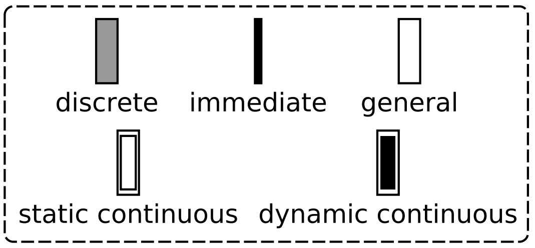


Figure 2.2: Representation of transitions in an HPnG.

Figure 2.2 presents the different types of transitions. Immediate transitions are represented as a thin, vertical, black bar. Deterministic transitions are indicated by a grey filled rectangle, while empty rectangles stand for general transitions. Static and dynamic continuous transitions are both illustrated as double framed rectangles, whereas the inner rectangle is filled for dynamic continuous transitions.

The connection between places and transitions is determined by the directed arcs of the HPnG. In addition to the *discrete arcs* $\mathcal{A}^D$ and the *continuous arcs* $\mathcal{A}^F$, the set $\mathcal{A}$ of arcs besides contains so-called *test arcs* $\mathcal{A}^T$ and *inhibitor arcs* $\mathcal{A}^H$. Discrete arcs only connect discrete places and discrete transitions. The same holds equivalently for continuous arcs. In contrast, test arcs and inhibitor arcs can connect any kind of place with any kind of transition. These arcs can however only point from a place to a transition and not vice versa.

The different types of arcs are shown in Figure 2.3. Arcs connecting discrete places with discrete transitions are represented by simple arrows. Continuous arcs are depicted as shaped arrows. The orientation of the arrows determines the flow direction of the entities. The representation of a test arc is a double arrow. Although there is no graphical distinction between the directions of a test arc, it formally

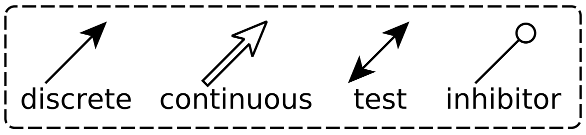


Figure 2.3: Representation of arcs in an HPnG.

always points from a place to a transitions. An inhibitor arc is an edge with a small circle at its end. The circle indicates the direction of the arc, which likewise always points from a place to a transition.

### 2.1.2 Marking

The entities in an HPnG, which have been spoken of vaguely so far, are described by the so-called *marking*. For hybrid Petri nets we differentiate between the discrete marking and the continuous marking. Discrete places contain *tokens*. Depending on the system at hand, a token might for example describe a physical entity like a working piece in a production process or an abstract entity like the functioning of a system component. The continuous marking is represented as non-negative real numbers in the continuous places of the HPnG. In [31], the continuous marking has originally been used to express the amount of fluid in a tank. Hence, we sometimes casually refer to the continuous marking as fluid, although it can of course stand for other continuous properties like the charging level of a battery.

The change of the marking over time represents the behaviour of the system that is modelled by the HPnG. The influence of the places, transitions and arcs on the marking is expressed by a set of functions that are aggregated in the 9-tupel $\Phi = (\Phi_b^{\mathcal{P}}, \Phi_w^{\mathcal{T}}, \Phi_p^{\mathcal{T}}, \Phi_d^{\mathcal{T}}, \Phi_f^{\mathcal{T}}, \Phi_g^{\mathcal{T}}, \Phi_w^{\mathcal{A}}, \Phi_s^{\mathcal{A}}, \Phi_p^{\mathcal{A}})$. The functions assign certain properties to the components of the HPnG, which are explained in the following paragraphs.

Discrete and continuous places can not hold negative amounts of tokens or fluid, which means that they have an implicit lower bound of 0 for their marking. Aside from this implicit bound, continuous places can additionally be tagged with a maximum capacity by the function $\Phi_b^{\mathcal{P}}$. If a place is not provided with an upper bound the marking can in contrast grow infinitely large.

The functions in $\Phi$ with $\mathcal{T}$ in their exponent refer to properties of transitions. $\Phi_w^{\mathcal{T}}$ and $\Phi_p^{\mathcal{T}}$, respectively, assign a weight and a priority, which are both used for the resolution of conflicts, to deterministic and immediate transitions. Both types are also equipped with fixed firing times by the function $\Phi_d^{\mathcal{T}}$. Immediate transitions are a special case of deterministic transitions, for which this time is set to 0. The mapping $\Phi_f^{\mathcal{T}}$ labels continuous transitions with a rate, that stands for the amount of fluid moved per time unit. For a general transition $T_i^G$ the firing time is determined by a random variable $s_i$, which follows an arbitrary continuous distribution $g_i$. The function $\Phi_g^{\mathcal{T}}$ assigns the cumulative distribution function (CDF) of this distribution to the transition, which computes the probability that $T_i^G$ has already fired at a time $\tau$.

All arcs of an HPnG are equipped with a weight by the function $\Phi_w^{\mathcal{A}}$. If the weight of an arc is omitted in the graphical representation it is implicitly assumed as 1. The other two functions $\Phi_s^{\mathcal{A}}$ and $\Phi_p^{\mathcal{A}}$ express so-called *shares* and priorities for fluid arcs.

Now that all syntactic elements of an HPnG have been briefly introduced, they can be summarised in Definition 2.1:

**Definition 2.1.** *A* ***hybrid Petri net with general one-shot transitions*** *(HPnG) is a 6-tuple $P = (\mathcal{P}, \mathcal{T}, \mathcal{A}, m_0, x_0, \Phi)$ with:*

- *the set of places $\mathcal{P} = \mathcal{P}^D \cup \mathcal{P}^C$,*
- *the set of transitions $\mathcal{T} = \mathcal{T}^I \cup \mathcal{T}^D \cup \mathcal{T}^G \cup \mathcal{T}^F$,*
- *the set of arcs $\mathcal{A} = \mathcal{A}^D \cup \mathcal{A}^F \cup \mathcal{A}^H \cup \mathcal{A}^T$,*
- *the initial discrete marking $m_0 \in \mathbb{N}^{|\mathcal{P}^D|}$,*
- *the initial continuous marking $x_0 \in \mathbb{R}^{|\mathcal{P}^C|}$,*
- *the 9-tuple $\Phi$ of functions that assign capacities, weights, etc. to places, transitions, and arcs.*

Note, that the initial marking of both the discrete and continuous places belongs to the definition of an HPnG. This significantly determines the evolution of the model over time as it is described Section 2.3. Hence, if we speak of a specific HPnG $P$, initial markings $m_0$ and $x_0$ are always included.

## 2.2 Semantics of HPnG Primitives

In this section the semantics of the HPnG components is introduced. At first, the basic mechanism of a Petri net, the so-called *firing* of a transition, is explained. During this part we exclude the semantics of test and inhibitor arcs, which is later introduced in the context of *enabling* and *rate adaption* (see Section 2.2.2). Small example HPnGs appear in this sections to illustrate the discussed topics.

### 2.2.1 Firing of Transitions

We say a transition *fires* if it alters the marking of a place. If the corresponding arc points from a place to the transition, the transition consumes tokens or fluid from the *input place.* If, vice versa, the arc is directed from the transition to the place, the transition puts tokens or fluid into the *output place.*

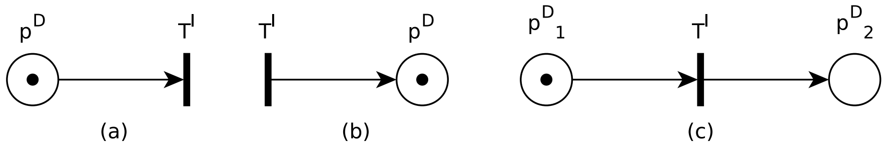


Figure 2.4: Basic patterns of places and transitions in an HPnG.

In Figure 2.4 basic HPnGs are shown using discrete places, discrete arcs, and immediate transitions. The transition in (a) is the *sink transition* of the place. In (b) the transition generates token for the system, and is hence called a *source transition.* If a transition has both an input place and an output place, it shifts tokens or fluid from the input place to the output place (c). The latter represents the most essential mechanism in a Petri net.

Discrete transitions alter the marking of discrete places without a delay as soon as they fire. Deterministic and immediate transitions are annotated with a firing time, that specifies how long the transitions have to be *enabled* (see Section 2.2.2) before they are allowed to fire. Whenever the transition has been re-enabled, it can fire again. General one-shot transitions however are allowed to fire only once to limit the number of random variables of the system.

A *conflict* arises if several deterministic or immediate transitions are able to fire at the same time. To solve this possible non-determinism, the order, in which the simultaneous firings have to be carried out, must be determined by considering the weights and priorities of the transitions. If all transitions in question have different priorities, the firings are simply arranged from highest to lowest priority. However, multiple transitions might have the same priorities, so that the weights of the transitions must be used in addition to compute *relative firing probabilities*. These probabilities do not directly determine the order, in which the conflicting transitions fire, but instead tell us, how likely the firing of each of the transitions is. Since this additional probabilistic component of the model is not compatible with STDs, which are introduced in Chapter 4, all models, that are discussed in the context of this work, are required to resolve all conflicts of deterministic and immediate transitions with priorities only. A more detailed description of conflict resolution can be found in [32].

Contrary to discrete transitions, continuous transitions fire - as the name indicates - continuously. Static continuous transitions do so with a fixed nominal rate, as long as no *rate adaption* (see Section 2.2.2) takes place. The rate of dynamic continuous transitions in contrast depends on the rates of other static continuous transitions. For example, the rate of a dynamic fluid transition can be the sum of two static rates. If the rate of the static fluid transitions changes, the rate of the depending continuous transition changes, too.

The number of tokens or the amount of fluid, that is moved by a transition, is influenced by the weight of the connected arcs. The weight of discrete arcs determines how many tokens are removed from an input place or how many tokens are put into an output place, respectively. The rate of a continuous transition specifies the amount of fluid that is moved per time unit. Thus, the weight of a continuous arc is instead used as a factor of the transition's rate.

Weighting continuous arcs is in particular useful in combination with multiple input and output places for a single transition.

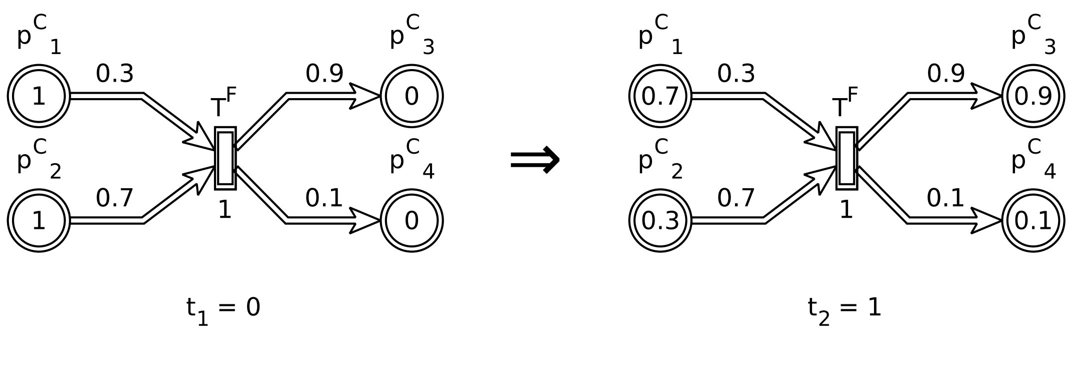


Figure 2.5: Influence of (continuous) arc weights on HPnG behaviour.

An example is depicted in Figure 2.5. The picture shows a simple HPnG consisting of four continuous places and a static continuous transition at times $t_0 = 0$ and $t_1 = 1$. The transition fires with a rate of 1. Since the input arcs are labelled with different weights, the amount of fluid that is consumed per time step however differs for the two input places. The same holds for the two output places, which are filled with respect to their different arc weights. The example might represent a simple relay station, that is on the one hand supplied by two different kinds of power sources (e.g. $p_1^C$: solar energy, 30%; $p_2^C$: coal 70%) and, on the other hand, distributes the combined energy to two destinations (e.g. $p_3^C$: local usage, 10%, $p_4^C$: export, 90%).

### 2.2.2 Enabling and Rate Adaption

In the previous section test arcs and inhibitor arcs have been excluded from the discussion, since they do not transport tokens or fluid through the system, and hence do not directly contribute to the change of the marking. Instead, they can be used to specify the conditions under which a transition is allowed to fire. If a transition is allowed to fire, it is called *enabled*, otherwise it is called *disabled*.

Test arcs and inhibitor arcs begin in a place of arbitrary type and point to an arbitrary type of transition. They are annotated with a weight, that represents a number of tokens for discrete places and an amount of fluid for continuous places. Test arcs require the place to contain at least as much tokens or fluid as determined by the arc's weight in order to enable the connected transition. The opposite holds for inhibitor arcs. Here, the place must not contain more tokens or fluid than specified by the weight.

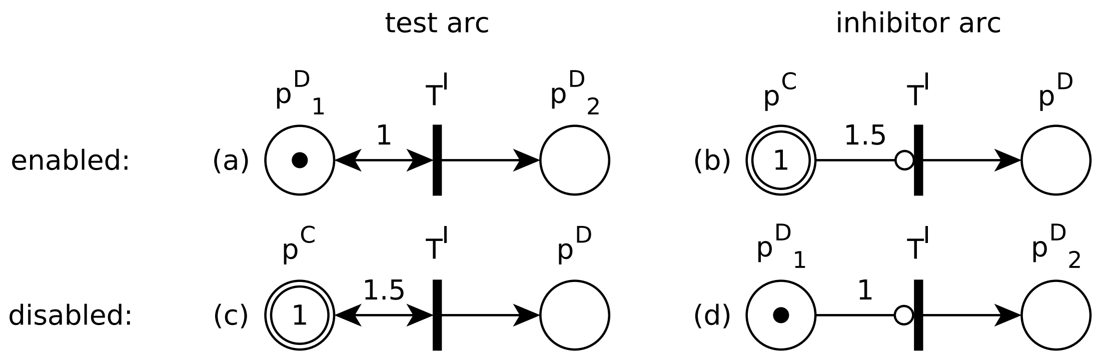


Figure 2.6: Examples for test and inhibitor arcs.

Figure 2.6 depicts four transitions, which are either enabled or disabled by test arcs or inhibitor arcs, respectively. For example, the transition $T^I$ in (a) is enabled, since the test arc demands at least one token in $p_1^D$, which is indeed present in the place. In the corresponding discrete case (d) with an inhibitor arc, $T^I$ is disabled instead, since $p_1^D$ does not contain less than one token. In the examples (b) and (c) the same is valid for test arcs and inhibitor arcs in combination with continuous places. Note, that an arbitrary transition can be controlled via test arcs and inhibitor arcs connected to both discrete places and continuous places.

Aside from test arcs and inhibitor arcs, discrete transitions can be enabled or disabled by the marking of their input places. If a discrete transition requires more tokens than are available in the input place, the transition is disabled, since there are not enough resources present. On the other hand, if a continuous place reaches its lower bound or upper bound, the involved transitions are not disabled. Instead, the so-called *rate adaption* is applied.

This mechanism balances the incoming and outgoing flows of fluid, so that the marking of the concerned place is constant for the time being. The accumulated flows of a place $p^C$ are called the *drift* $d_{p^C}$. The drift is computed from the rates of the connected fluid transitions by adding up the weighted rates of the source transitions and subtracting the weighted rates of the sink transitions. If a continuous place reaches its implicit minimum capacity of 0, more fluid has been removed from the place than has been fed in. So, the drift of the place has been negative and the rates of the sink transitions have to be adjusted to increase the drift to zero. If, in contrast, the drift of the place is positive and the place has been filled, the rates of the source transitions have to be reduced.

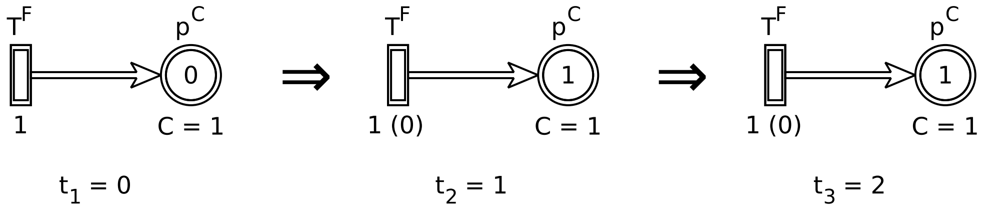


Figure 2.7: An example for rate adaption in continuous places.

Figure 2.7 visualises the effect of rate adaption. The place $p^C$ has a capacity of 1, which is reached with the help of the source transition $T^F$ in one time unit. In the time interval $[0, 1]$ the drift $d_{p^C}$ of $p^C$ is equal to $+1$. As soon as $p^C$ reaches its upper bound, the rate of $T^F$ has to be adapted to reduce $d_{p^C}$ to zero. Thus, the capacity of $p^C$ is not exceeded even after another time unit. Note, that the adapted rate is denoted in brackets next to the transition's nominal rate, which remains unchanged. This notation is not a part of the HPnG formalism, and only serves here as a supporting visualisation.

The detailed rules for the adaption of rates in case of multiple source or sink transitions can be found in [32]. Since these rules depend on the priorities of transitions and we are only concerned with cases, in which the places have at most one continuous source or sink transition, we do not look deeper into this topic here.

## 2.3 Evolution of an HPnG

The *evolution* of an HPnG is characterised by the change of its *state* over a course of time. Some of the parameters like the arc weights, that influence this process, are statically defined in the syntactic elements of the HPnG. However, other characteristics, like the marking or the drift, change dynamically and are thus captured separately in the state.

### 2.3.1 State of an HPnG

The state of an HPnG at a given time basically depends on three parameters: First of all, the initial state of the HPnG, i.e. the marking of the discrete places and continuous places at time $t_0 = 0$ determines the drifts of the continuous places and the enabled and disabled transitions at the beginning of the observation. Accordingly, the internal processes of the HPnG are clearly influenced by the initial state according to the descriptions in the previous sections. Secondly, the firings of general transitions can not be predicted, since their firing times are randomly distributed. If a general transition fires, it alters the discrete marking and can enable or disable transitions, which changes the behaviour of the system, as well. Two possible runs of the same HPnG might thus yield different state evolutions, when the general transitions fire at different times. In addition, the considered point in time obviously influences the state of the HPnG, because, as a non-autonomous Petri net, it depends on time.

While the initial state of an HPnG is already captured in its definition via the discrete marking $m_0$ and the fluid marking $x_0$, either of the other two parameters can be chosen freely, however. This leads us to Definition 2.2 that introduces the state of an HPnG as a quintuple depending on the general transition firing times and the observed time.

**Definition 2.2.** *The* ***state*** *of an HPnG $P$ in the time interval $[0, t_{max}]$ with fixed general transition firing times $\vec{s} \in [0, t_{max}]^{|\mathcal{T}^G|}$ at a time $t \in [0, t_{max}]$ is denoted as $\Gamma(\vec{s}, t) := (m, x, c, d, g)$ with:*

- *the marking $m \in \mathbb{N}^{|\mathcal{P}^D|}$ of the discrete places,*
- *the marking $x \in \mathbb{R}^{|\mathcal{P}^C|}$ of the continuous places,*
- *the clocks $c \in \mathbb{R}^{|\mathcal{T}^D|}$ that record the time for which the deterministic transitions have been enabled,*
- *the drift $d \in \mathbb{R}^{|\mathcal{P}^C|}$ of the continuous places,*
- *a vector $g \in (\mathbb{R}^{\geq 0} \cup \{-1\})^{|\mathcal{T}^G|}$ that marks which general one-shot transition has fired, already.*

According to the definition, the state of an HPnG contains three more parameters aside from the marking of the discrete and continuous places. The vector $c$ carries a clock $c_i$ for each deterministic transition $T_i^D \in \mathcal{T}^D$, that counts the time for which the transition has already been enabled. If the clock reaches the firing time of the transition, it fires and the clock is reset to zero. Similar to these clocks, the vector $g$ stores the enabling time of the general transitions. Since they are allowed to fire only once, the entry $g_i$ of a general transition $T_i^G \in \mathcal{T}^G$ is set to $-1$ after the firing to mark that it is disabled from now on. Furthermore, the drift $d$ for continuous transitions is stored in the state tuple according to [29]. Note, that this entry could be omitted, though, since the drift can be calculated from the marking and the rates of the fluid transitions as has been described in the previous section.

Given an HPnG $P$ with fixed general transition firing times $\vec{s}$ and a state $\Gamma(\vec{s}, t)$, the subsequent evolution of the Petri net is completely determined. Since the markings at time $t$ are captured in the state, the current drifts as well as the enabled transitions are known, which fully characterises the behaviour of the system. Hence, the change in both the discrete marking and the continuous marking plus the future enabling or disabling of transitions can be predicted. If the firing times $\vec{s}$ of the general transitions were not fixed, this prediction would not be possible because the general transitions can potentially fire at any time, and thus change the behaviour of the HPnG at any time. The entire change of the state over time beginning from an initial configuration $\Gamma(\vec{s}, 0)$ up to $\Gamma(\vec{s}, t_{max})$ is therefore identified by the corresponding firing times $\vec{s}$ of the general transitions. This sequence of states is what we call the *evolution* of the HPnG.

The state as presented in Definition 2.2 depends on both the firing times of the general transitions and the observed time. To address instead all possible configurations of the system at a time $t$ at once regardless of the general transition firing times, another term is introduced in the following definition.

**Definition 2.3.** *The set* $\Gamma(t) = \{\Gamma(\vec{s}, t) | \vec{s} \in [0, t_{max}]^{|\mathcal{T}^G|}\}$ *of all states* $\Gamma(\vec{s}, t)$ *at a time* $t$ *is called the* ***system state at a time*** $t$ *of an HPnG* $P$ *observed in the time interval* $[0, t_{max}]$.

A system state at a time $t$ accumulates snapshots of all evolutions at a given time. Some of the contained states might not actually be reachable, since their corresponding general transition firing times occur with a probability of zero.

What we refer to as “the” initial state is actually the system state at time 0, because the starting configuration of the HPnG is independent of the general transition firing times, that is $\Gamma(0) = \{\Gamma(\vec{s}, 0) | \vec{s} \in [0, t_{max}]^{|\mathcal{T}^G|}\}$, where $\Gamma(\vec{s}, 0) := (m_0, x_0, \vec{0}^{|\mathcal{T}^D|}, d_0, \vec{0}^{|\mathcal{T}^G|})$ for all $\vec{s} \in [0, t_{max}]^{|\mathcal{T}^G|}\}$.

### 2.3.2 Events

The most significant changes in the state of an HPnG that occur during its evolution are called *events*. Events mark either the enabling or disabling of any kind of transition or a rate adaption for a continuous transition. We hence distinguish three types of events:

I. a discrete transition fires,

II. a (lower or upper) boundary of a continuous place is reached,

III. a condition of a test arc or inhibitor arc is fulfilled or violated.

Although the continuous marking and the clocks of deterministic transitions are part of the state, the firing of continuous transitions or the progress of the clocks is not among the event types. That is because these state changes do not directly influence the properties of transitions, and thus do not change the behaviour of the HPnG. A type I event, however, alters the marking of a discrete place, and might therefore either disable a discrete transition if its input place no longer contains a sufficient amount of tokens, or trigger a type III event that depends on a discrete marking. Similarly, type II events and other type III events are induced by specific continuous markings, which lead to rate adaption or to the enabling or disabling of transitions.

So, between two events only the markings of the continuous places and the clocks of the enabled transitions change according to their specific drifts. All the other parameters of the state are constant. This implies in particular, that, if we know the state $\Gamma(\vec{s}, t_e)$ at the occurrence $t_e$ of an event $e$ for some fixed general transition firing times $\vec{s}$, the following evolution of the state, that is especially the occurrence of the next event, is predetermined completely.

$\Gamma(\vec{s}, t_e)$ contains the discrete markings, the fluid markings, the drifts, and the clocks of the deterministic and the general transitions at $t_e$. The marking defines which transitions are enabled and which are disabled. Consequently, the clocks of enabled discrete transitions are currently allowed to advance while the clocks of disabled transitions are stopped. The next firing of discrete transitions can be deduced from the difference of the transitions' firing times and the current values of the clocks. Using the fluid marking and the drifts at $t_e$, the times, at which a maximal or minimal capacity in a continuous place is reached, or a test condition or inhibitor condition is met, can be computed, too. If the drifts of fluid places were not captured in the state, they could be computed from the continuous marking and the rates of the enabled transitions. In either case, the next possible event after $e$ can be determined using the information stored in $\Gamma(\vec{s}, t_e)$.

Summarised, the evolution of an HPnG $P$ with fixed firing times $\vec{s}$ of the general transitions in a bounded time interval $[0, t_{max}]$ can be characterised by a finite sequence $\pi = (e_1, \ldots, e_m)$ of events $e_i$ with $\forall 1 \leq i < j \leq m : 0 \leq t_{e_i} \leq t_{e_j} \leq t_{max}$. Such a sequence can be computed completely given the initial state $\Gamma(0)$ of the HPnG.

**Example.** *To illustrate the evolution of an HPnG, we examine a simple example.*

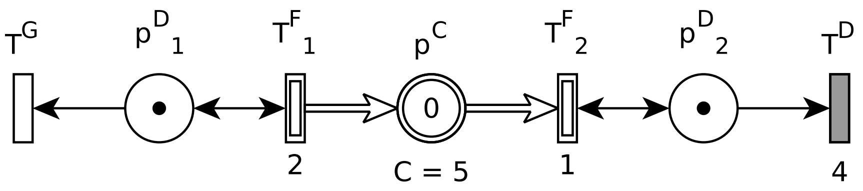


Figure 2.8: A simple HPnG with a single general one-shot transition.

*Figure 2.8 shows an HPnG $P$ with a single general one-shot transition in its initial state. It is a modification of Figure 3 from [27]. The discrete places $p_1^D$ and $p_2^D$ each contain a single token, whereby both the general transition $T^G$ and the deterministic Transition $T^D$ are enabled. The continuous place $p^C$ is empty in the beginning and has an initial drift of $+1$, since both of the static continuous transitions fire according to their nominal rate. We observe the HPnG in the time interval $[0, 5]$.*

*Assume that the general transition fires at time $s_{T^G} = 1$, that is $\vec{s} = (1)$. With the initial drift of $+1$ the fluid place $p^C$ reaches its upper limit of $5$ after $5$ time units, which is clearly after the firing of $T^G$. The deterministic transitions $T^D$ also fires after $T^G$, since its firing time is defined as $4$. The first event $e_1$ to occur is thus the firing of the general transition, which takes place at $t_{e_1} = 1$.*

*$T^G$ removes the token from the place $p_1^D$, thereby violating the condition of the connected test arc and disabling the continuous transition $T_1^F$. This leads to a change in the drift of $p^C$, which is now $-1$. So, after another time step, the continuous place has been emptied again. Because no more fluid can be taken from $p^C$, the rate of the*

*transition $T_2^F$ has to be adapted to* 0. *This rate adaption is a type II event $e_2$ and occurs at $t_{e_2} = 2$.*

*The last possible event $e_3$ occurs at time $t_{e_3} = 4$ which marks the firing of the deterministic transition $T^D$. Just like the general transition $T^G$, it removes a token from the connected place $p_2^D$, and thus disables the transition $T_2^F$ by violating the condition imposed by the test arc. The drift of $p^C$ has already been zero due to rate adaption and, therefore, does not change. Afterwards, no more events occur until the system reaches the time limit at $t = 5$, where we stop our observation. In summary, the evolution of $P$ with the general transition firing times $\vec{s} = (1)$ in the time interval $[0, 5]$ can be described by the sequence $\pi = (e_1, e_2, e_3)$ of events.*

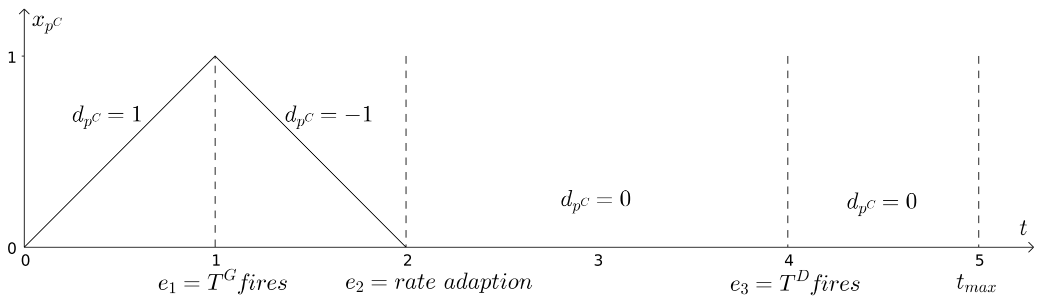


Figure 2.9: Change of the fluid level over time in place $p^C$ from Figure 2.8 for the described scenario.

*The evolution is mainly characterised by the change of the fluid level in $p^C$, which has been plotted against the time in Figure 2.9. By visually relating the events to the fluid level, their influence on the behaviour of the HPnG is emphasised. The drift $d_{p^C}$ of the place is the slope of the shown curve, which changes when an event occurs. For the last event $e_3$ this change is not directly visible in the value of $d_{p^C}$. Instead, it changes why the drift is zero. Before the occurrence of $e_3$ the drift has been zero due to rate adaption, afterwards it is zero since the transition $T_2^F$ has been disabled. This difference is also reflected in the state of the HPnG, where the discrete marking of $p_2^D$ is* 1 *in any state $\Gamma((1), \tau)$ with $\tau \in [0, t_{e_3})$ and* 0 *in any state $\Gamma((1), \tau')$ with $\tau' \in [t_{e_3}, t_{max}]$.*

# Chapter 3

# Theory of Nef Polyhedra

Nef Polyhedra have been introduced in 1978 by Walter Nef in his book *Beiträge zur Theorie der Polyeder mit Anwendungen in der Computergraphik* [48], which, unfortunately, has only been published in German. Thus, we mostly refer to [11] for reference, which gives a concise overview of the topic.

Nef polyhedra are mostly used in the kernels of computer-aided design (CAD) programs for the modelling of basic three-dimensional objects (cf. [57]). The most well-known implementation of Nef polyhedra is part of the *Computational Geometry Algorithms Library* (CGAL, cf. [16]) which has also been used for this work (see Chapter 7). Outside of the field of computer graphics Nef polyhedra are fairly unknown, although they can be defined in arbitrary dimensions and have many desirable properties, as it is described in the following paragraphs.

This chapter is split into two sections: First of all, the required mathematical foundations are briefly introduced. The second part presents possible definitions of Nef polyhedra and their properties. Just like in [11] we do not mention the proofs for these properties, and instead provide more intuitive explanations. The proofs can be found in [48].

## 3.1 Preliminaries

This section presents mathematical concepts which are necessary to understand and define Nef polyhedra. Hyperplanes and half-spaces as well as arrangements of hyperplanes are introduced.

### 3.1.1 Hyperplanes in $\mathbb{R}^n$

Hyperplanes are the generalisation of planes in three-dimensional space. Depending on the type of space they are defined in (e.g. euclidian, affine, projective, etc.), hyperplanes might have different properties. Since we are concerned with Nef polyhedra in $\mathbb{R}^3$, we limit our explanations to the euclidian space $\mathbb{R}^n$.

We use the definition of hyperplanes from [56].

**Definition 3.1.** *Let $v = (v_1, \dots, v_n) \in \mathbb{R}^n \setminus \{0\}^n$ and $k \in \mathbb{R}$.*
*A **hyperplane** $H^0_{v,k} \subseteq \mathbb{R}^n$, or in short $H^0$, is defined as:*

$$\begin{aligned} H^0_{v,k} &= \{x = (x_1, \dots, x_n) : v_1 * x_1 + \dots + v_n * x_n + k = 0\} \\ &= \{x \in \mathbb{R}^n : \langle v, x \rangle + k = 0\} \end{aligned}$$

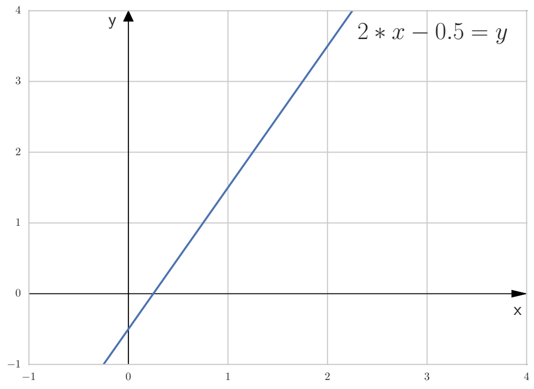
*with the inner product* $\langle \cdot,\cdot \rangle$.
*The vector* $v$ *is perpendicular to* $H^0_{v,k}$ *and is called* ***normal vector of*** $H^0_{v,k}$. *The index* $_{v,k}$ *is usually omitted in the following paragraphs.*

Definition 3.1 shows that a hyperplane is given by the zero-set of a linear equation. The normal vector of a hyperplane determines its orientation in space. That is we say a point is *above* a hyperplane, if it lies on the side of the hyperplane that the normal vector points to. Contrary, we say a point is *below* a hyperplane, if it is located on the opposite side.

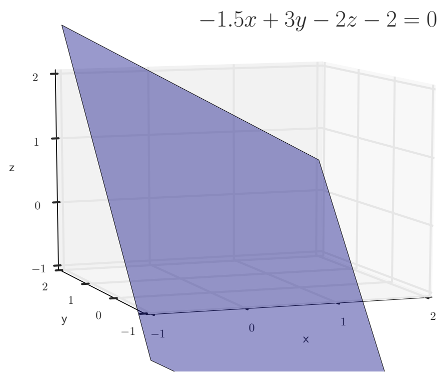
An abbreviated but frequently used representation of hyperplanes only presents the hyperplane equation instead of the complete set notation:

$$H^0 := v_1 * x_1 + \cdots + v_n * x_n + k = 0.$$

For example, in $\mathbb{R}^2$, a hyperplane is a line given by $v_1 * x + v_2 * y + k = 0$, which is equivalent to the more familiar notation $mx + b = y$ with $m = \frac{v_1}{v_2}$ and $b = \frac{k}{v_2}$.



(a) A hyperplane in $\mathbb{R}^2$.



(b) A hyperplane in $\mathbb{R}^3$.

Figure 3.1: Hyperplane examples.

Figure 3.1 shows a plot of the hyperplane $2x - 0.5 = y$ in $\mathbb{R}^2$ (a) and a hyperplane in three dimensions given by $-1.5x + 3y - 2z - 2 = 0$ (b).

Looking at the figure, we can observe that the hyperplanes split the space into two parts, which are called half-spaces. (cf. [14]).

**Definition 3.2.** *Let* $H^0$ *be a hyperplane in* $\mathbb{R}^n$. *The* ***positive, open half-space*** *defined by* $H^0$ *is the set*

$$H^+ = \{x \in \mathbb{R}^n | \langle v, x \rangle + k > 0\}.$$

*The* ***positive, closed half-space*** $H^{(+)}$ *is the union of* $H^0$ *and the positive, open half-space* $H^+$. *Open and closed negative half-spaces are defined correspondingly.*

Positive and negative half-spaces are dual in a sense that the negative, open half-space $H^-$ can be described as the complement of the positive, closed half-space $H^{(+)}$.

### 3.1.2 Arrangement of Hyperplanes in $\mathbb{R}^n$

In Section 3.2 Nef Polyhedra are shown to be formed by the intersection of finitely many half-spaces. A finite set of half-spaces or hyperplanes (which are more or less used as interchangeable terms) defines a so-called arrangement (cf. [21]).

**Definition 3.3.** *An **arrangement of hyperplanes** $\mathcal{A}(\mathbf{H})$ is defined by a finite set of hyperplanes $\mathbf{H} = \{H_0, \dots, H_m\}, m \geq 0$. The arrangement consists of all non-empty intersections $F = \bigcap_{i=0}^{m} H_i^{s_i}$ with $s_i \in \{-, 0, +\}$. The intersections $F \in \mathcal{A}(\mathbf{H})$ are called **faces** of $\mathcal{A}(\mathbf{H})$.*

The concept of arrangements and faces is best described by a simple two-dimensional example.

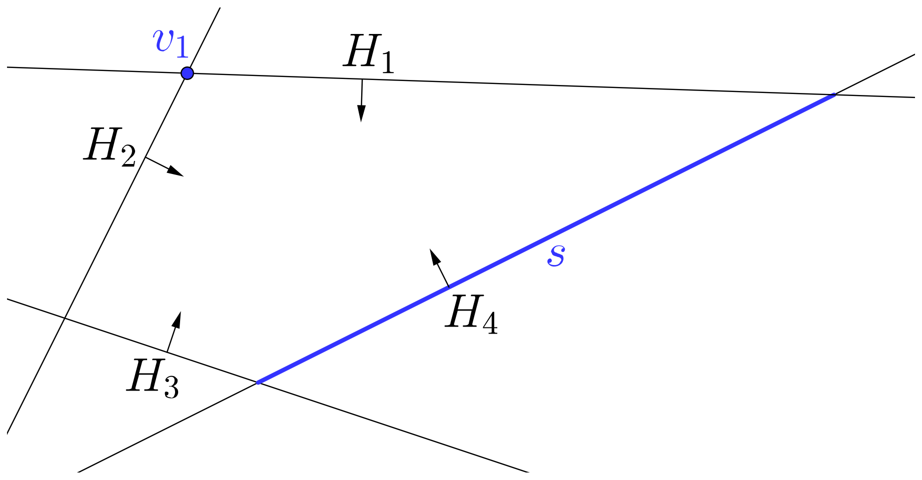


Figure 3.2: An arrangement of hyperplanes in $\mathbb{R}^2$.

Figure 3.2 shows four hyperplanes in $\mathbb{R}^2$, which define the arrangement $\mathcal{A}(\{H_1, \dots, H_4\})$. The arrows are the normal vectors of the hyperplanes and point to their positive half-spaces.

The faces of the arrangement can be characterised by the relative position of their associated points to the hyperplanes $H_1$ to $H_4$. For example, the vertex $v_1$ lies on the hyperplanes $H_1$ and $H_2$ and in the positive half-spaces of the other two hyperplanes. The face of $v$ is thus defined as $F_v = H_1^0 \cap H_2^0 \cap H_3^+ \cap H_4^+$. Another example is the face $F_s$ of the segment $s$, which is defined as the set of points that lie on $H_4$ and above $H_1$, $H_2$, and $H_3$.

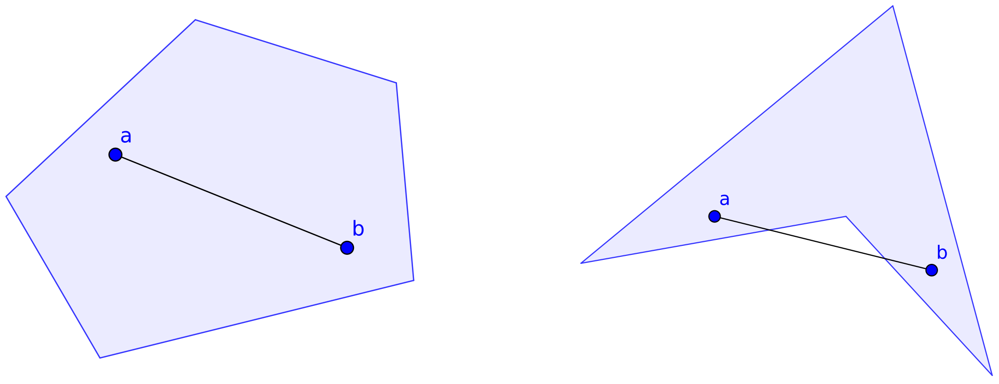
Moreover, the faces of an arrangement can be classified by their dimension ([58]). A face of dimension $d$ in $\mathbb{R}^n$ is generally identified as a $d$-face. Faces of certain dimensions, however, are typically referred to by special names (cf. [21]). 0-faces like $F_v$ from the above example, contain only a single vertex, and are thus simply called *vertices*. 1-faces, i.e. lines and segments, are termed *edges*, while *facet* is the description for $(n-1)$-faces. $n$-faces are objects of the highest dimension and are named *cells*.

### 3.1.3 Convex Sets

Faces of hyperplane arrangements with certain properties have a dual relation to so-called *convex sets*. This relation is used during the generation of Stochastic Time Diagrams which are discussed in Chapter 4.

**Definition 3.4.** *Let $P \subset \mathbb{R}^n$ be a set. $P$ is said to be* ***convex*** *iff for each pair of points $a, b \in P$ the line segment with the endpoints $a$ and $b$ is contained in $P$.*

Definition 3.4 requires, that, if we draw a line between any two points in a convex set, this line has to lie completely inside the set.



(a) A convex set in $\mathbb{R}^2$.

(b) A non-convex set in $\mathbb{R}^2$.

Figure 3.3: Convex set examples.

Figure 3.3 visualises the difference between convex (a) and non-convex sets (b). While in (a) each pair of points can be connected by a line which lies inside the point set, this is not true for the chosen points in (b). Thus, the point set in (b) is not convex.

If a convex set is closed, it can be represented by a possibly infinite number of half-space intersections. This suggests a connection between convex sets and the faces of hyperplane arrangement, which are the result of half-space intersections, as well (see Definition 3.3). The connection is reflected in the definitions of so-called *convex polytopes*. They are either defined as the intersection of half-spaces ($\mathcal{H}$-polytope) or as the *convex hull* of a set of points ($\mathcal{V}$-polytope) (cf. [61]). For a detailed discussion on convex polytopes see for example [33].

For each non-convex set exists a corresponding smallest convex set, which is referred to as the convex hull. This is captured in the definition below which has been adopted from [8].

**Definition 3.5.** *Let $P \subset \mathbb{R}^n$ be a set of points. The* ***convex hull*** *$conv(P)$ is the smallest convex set so that $P \subseteq conv(P)$.*

Since the computation of the convex hull has many applications even outside of the geometric field, it is considered one of the fundamental problems of computational geometry and has thus been studied extensively in the past.

Figure 3.4 shows the non-convex set from Figure 3.3 along with its convex hull. Informally speaking, the convex hull of a point set can be generated by connecting the outmost points of the set with straight lines. In fact, multiple algorithms have been presented to solve the problem of finding the convex hull, both concerned with simplicity and efficiency (cf. [8]).

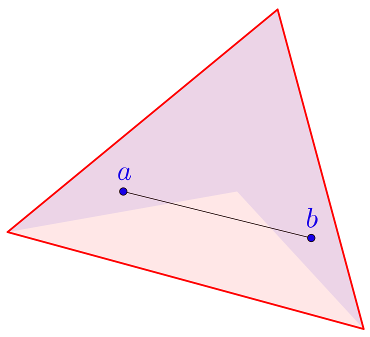


Figure 3.4: Convex hull of a point set in $\mathbb{R}^2$.

## 3.2 Nef Polyhedra

The concepts presented in the previous sections can be used to describe Nef polyhedra. We begin with the most well-known definition as an intersection of half-spaces and the introduction of several properties. Subsequently, an alternative definition using arrangements of hyperplanes is discussed.

**Definition 3.6** ([11, Def.1])**.** *A **Nef polyhedron** in $\mathbb{R}^n$ is a set $P \subset \mathbb{R}^n$ that can be obtained by applying a finite number of set operations cpl (complement) and $\cap$ (intersection) to a finite number of open half-spaces.*

Note that the requirement for open half-spaces is not a restriction, since every closed half-space can be expressed as the complement of an open half-space, as has been mentioned in Section 3.1.1. In addition, Definition 3.6 implies several properties of Nef polyhedra. Only those properties from [11] which are important in the following chapters are listed here.

Given the operations *cpl* and $\cap$ for the formation of Nef polyhedra, other boolean set operations can be easily deduced. The difference $A \setminus B$ of two sets $A$ and $B$ can be expressed as $A \setminus B \equiv A \cap \overline{B}$. Using de Morgan's law, the union $A \cup B$ of two sets $A$ and $B$ can be rewritten, too: $A \cup B \equiv \overline{\overline{A} \cap \overline{B}}$ (cf. [38]). Since Nef polyhedra are defined using these operations, it follows immediately that they are closed with respect to $\cap$, *cpl*, $\setminus$, and $\cup$.

Besides, Nef polyhedra are closed under the formation of their *interior* and their *closure*. The interior is the set of all points inside a set, that do not belong to its boundary. The closure is the complement of the interior. It consists of the Nef polyhedron' boundary plus all points outside of the set. Intersecting a Nef polyhedron with its closure provides the boundary of the Nef polyhedron. Due to the fact, that Nef polyhedra are closed under intersection, the boundary is again a Nef polyhedron.

Note, that even the empty set $\emptyset := H^+ \cap H^-$ and the universe $U := H^+ \cup H^{(-)}$ are Nef polyhedra (for some arbitrary hyperplane $H$).

Figure 3.5 shows a polygon $P$ that is formed by the intersection of three half-spaces $H_1$, $H_2$, and $H_3$. More precisely, it holds that $P := H_1^{(+)} \cup H_2^{(-)} \cup H_3^{(-)}$. Thus, following Definition 3.6, $P$ is a Nef polyhedron.

As we can see in Figure 3.5, $P$ is restricted by the hyperplanes $H_1$, $H_2$ and $H_3$. The Nef polyhedron contains the three vertices in the corners, the three segments of the hyperplanes, that each lie in the half-spaces of the other two hyperplanes, and the inner polygonal area comprised by those segments. Each of the mentioned parts is a face of the arrangement $\mathcal{A}(\{H_1, H_2, H_3\})$. This observation leads to the second definition of Nef polyhedra, which relies on arrangements of hyperplanes.

**Definition 3.7** ([11, Def.4])**.** *A point set $P \subseteq \mathbb{R}^d$ is a Nef polyhedron if there exists a finite family $\mathbf{H}$ of hyperplanes in $\mathbb{R}^d$ such that $P$ is the union of certain faces of the corresponding arrangement $\mathcal{A}(\mathbf{H})$.*

Following Definition 3.7, we can describe $P$ from Figure 3.5 using the faces of the arrangement defined by $H_1$, $H_2$, and $H_3$.

In Figure 3.6 the faces that contribute to $P$ are highlighted. Using the marked faces we can define $P$ as the union $P := F_{v_1} \cup F_{v_2} \cup F_{v_3} \cup F_{s_1} \cup F_{s_2} \cup F_{s_3} \cup F_a$. So, $P$ is formed by three 0-faces or vertices, three 1-faces or edges, and a single polygonal 2-face.

Convex polytopes are Nef polyhedra, as well, since they are defined as the intersection of closed half-spaces, as has been briefly mentioned before. Because convex polytopes can moreover be described as the convex hull of a point set, we can express this special kind of Nef polyhedra as convex hulls, too. The Nef polyhedron $P$ from the example is such a convex polytope. We can thus define $P$ in a third way, for example as the convex hull of the set $\{v_1, v_2, v_3\}$, that is $P := conv(\{v_1, v_2, v_3\})$. Note, that all other subsets of $P$ that contain the vertices $v_i$, could be similarly used to describe $P$ as their convex hull. This approach is mentioned here, since it is used in the creation of STDs (see Chapter 4) to circumvent the computation of hyperplane arrangements.

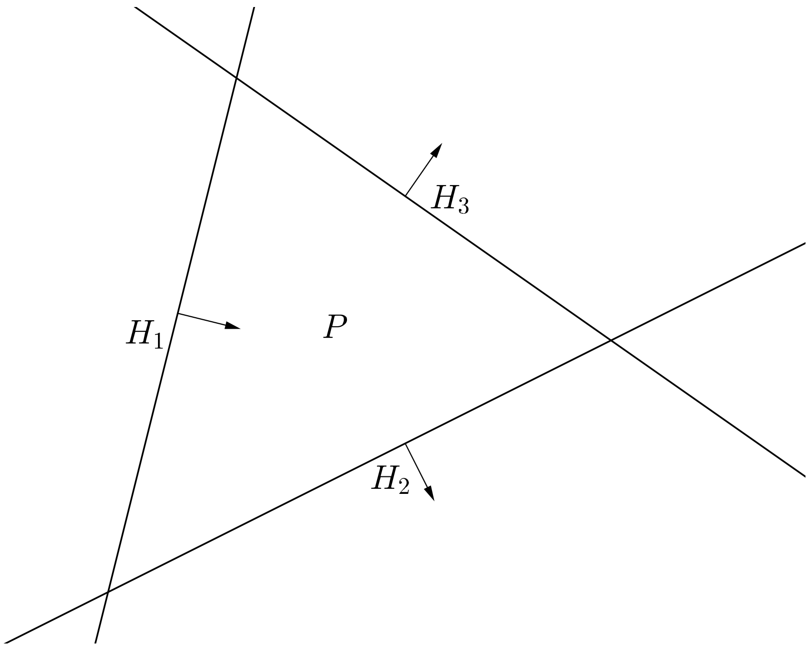


Figure 3.5: A Nef polyhedron formed by the intersections and complements of half-spaces.

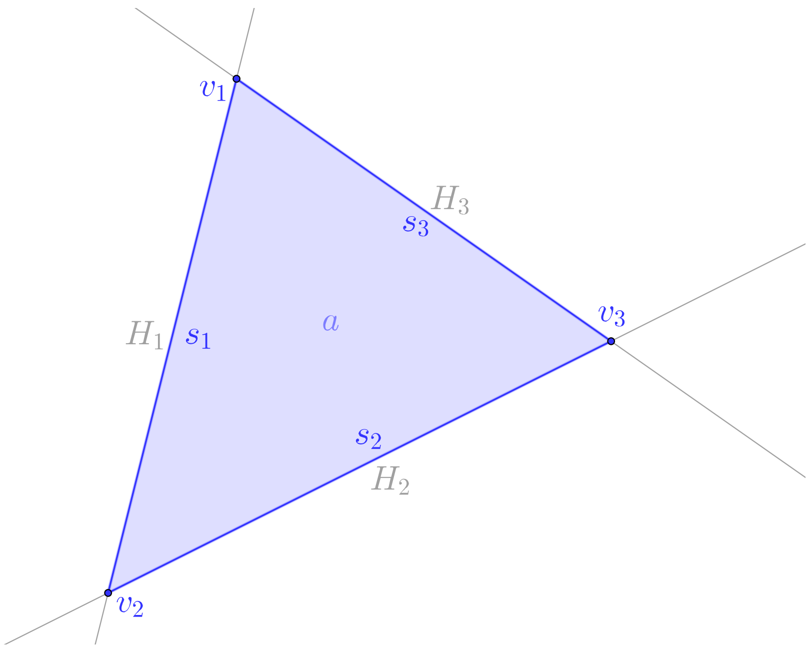


Figure 3.6: A Nef polyhedron formed by the union of faces of an arrangement.

# Chapter 4

# Stochastic Time Diagram

In Chapter 2 the HPnG formalism has been introduced, which allows to model systems with discrete, continuous, and probabilistic components. To be able to examine the overall behaviour of an HPnG, all possible evolutions have to be considered. Since not every single evolution or state can be tested individually, a structure, that aggregates similar states independent of the general transition firing times, is required. The solution to this problem is a graphical representation of the Petri net's state space, which is called *Stochastic Time Diagram* (STD).

STDs have first been introduced in [27] for HPnGs with a single general one-shot transition. In [29] the concept has been extended to an arbitrary number of general transition firings. The actual number of processable general transitions is however limited by the available data structures and algorithms. In particular, a library that provides higher order hyperplane arrangement algorithms or convex hull algorithms and, simultaneously, a suitable polyhedron data type, which can be used in the model checking process, are required. Since Nef polyhedra are for example mostly utilised in Computer Aided Design (CAD) applications, which model real world objects (cf. [57]), there is usually no demand for an implementation of Nef polyhedra in more than three dimensions. Therefore, there only exist implementations for the construction of STDs of HPnGs with one and two general transition firings, of which the latter is used as a basis for this work.

The first part of this chapter introduces the defining components of an STD in Section 4.1. Section 4.2 presents the algorithm to construct an STD. The discussed topics are finally illustrated by a simple two-dimensional example in Section 4.3.

## 4.1 Structure of an STD

In an STD, the possible firing times $s_i$ of the general transition $T_i^G$, $i \in \{1, \dots, n-1\}$, are plotted against the system time $t$ to represent the state space of an HPnG. A point in an STD is hence a tuple $(s_1, \dots, s_{n-1}, t)$ of the $s_i$ and $t$, thereby identifying a single state $\Gamma(\vec{s}, t)$ of the HPnG. According to the parameters of a state, we write $(\vec{s}, t)$ as an abbreviation for such a point.

As has been stated in Chapter 2, the evolution of an HPnG is a series of states in the time interval $[0, t_{max}]$ with fixed general transition firing times $\vec{s}$. Plotting such a series in an STD results in a line segment, that starts in $(\vec{s}, 0)$ and ends in $(\vec{s}, t_{max})$. Figure 4.1 depicts an evolution in a generic three-dimensional STD, which is identified by some arbitrary general transition firing times $s_1'$ and $s_2'$.

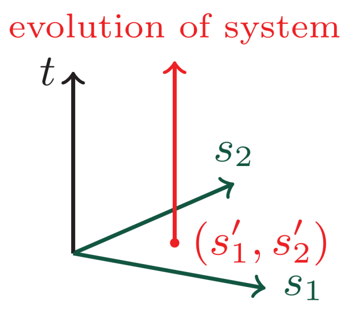


Figure 4.1: STD in $\mathbb{R}^3$ with a highlighted evolution of the corresponding HPnG with two general one-shot transitions (Figure 4 from [29]).

An STD includes all evolutions of an HPnG in a subset $[0, t_{max}]^n$ of $\mathbb{R}^n$, since both the individual general transition firing times $s_i$ and the time $t$ are limited to the observed time interval $[0, t_{max}]$. By recording the times in the STD, at which an event occurs during the evolutions, the subset is partitioned into so-called *regions*. Regions aggregate sets of points $(\vec{s}, t)$, whose associated states $\Gamma(\vec{s}, t)$ only differ in their continuous marking $\Gamma(\vec{s}, t).x$ and their clocks $\Gamma(\vec{s}, t).c$. Accordingly, regions are defined as follows (cf. [29]):

**Definition 4.1.** *A **region** $R$ in an STD is a maximal connected set of points $(\vec{s}, t)$, so that the following holds:*

$$\begin{aligned}\forall(\vec{s}_1, t_1), (\vec{s}_2, t_2) \in R : &\Gamma(\vec{s}_1, t_1).m = \Gamma(\vec{s}_2, t_2).m \wedge \\ &\Gamma(\vec{s}_1, t_1).d = \Gamma(\vec{s}_2, t_2).d \wedge \\ &\Gamma(\vec{s}_1, t_1).g = \Gamma(\vec{s}_2, t_2).g\end{aligned}$$

*The set $\Gamma_R$ of all states $\Gamma(\vec{s}, t)$ with $(\vec{s}, t) \in R$ is denoted as a **system state**.*

The states, which are contained in a region, share the same discrete marking, the same drifts of fluid places and clocks, and the same number of general transitions that have already fired. All these parameters of a state can only be altered by events, as has been explained in Section 2.3.2. Therefore, the borders of a region correlate to the time, at which an event takes places. According to Proposition 1 from [29], this time can be expressed as a linear function, i.e. a hyperplane equation, of $\vec{s}$ and $t$. Hence, the regions are surrounded by hyperplanes, which we call *event hyperplanes*.

**Definition 4.2.** *Let $e$ be an event and $t_e$ the occurrence time of $e$. The hyperplane*

$$H_e := 0 = a_0 + a_1 s_1 + \cdots + a_{n-1} s_{n-1} + a_n t_e$$

*with $a_i \in \mathbb{R}, \forall i \in \{1, \ldots, n\}$ is called the **event hyperplane** of $e$.*

We distinguish three different types of event hyperplanes regarding the form of their equations:

Some events are not affected by the firings of the general transitions, that is they always occur at the same time regardless of the probabilistic influence of the general transitions on the system's behaviour. They are thus called *deterministic* events. The occurrence of a deterministic event only depends on the system time $t$

and not on the general transition firings $\vec{s}$. This means that the normal vector $a$ of $H_e$ for deterministic events is always given by $a = (0, \dots, 0, 1)$, so that $H_e := 0 = a_0 + t \iff H_e := t = -a_0$. Thus, $H_e$ is perpendicular to the $t$-axis.

On the other hand, stochastic events depend on $\vec{s}$, which is reflected in the normal vector of $H_e$, as well. In this case, at least one scalar $a_i$ of a firing time $s_i$ is unequal to zero.

The firing of a general transition $s_i$ is a special type of stochastic event. Like for deterministic events, the hyperplane equation of these events has a specific form. It holds $H_{s_i} := 0 = a_0 + \dots + a_1 s_1 + a_{n-1} s_{n-1} + a_n t$ with $a_i = -1, a_n = 1, \forall j \neq i, n : a_j = 0$, which is equal to $H_{s_i} := t = s_i$. This special type of event hyperplanes partitions the STD into two parts:

The intersection of all negative half-spaces $\bigcap_{i=1}^{n} H_{s_i}^{-}$ in $K$ represents the states of the Petri net, in which no general transition has fired, yet. Thus, the behaviour of the system in this area is deterministic, and only deterministic events occur. The union of the positive half-spaces $\bigcup_{i=1}^{n} H_{s_i}^{+}$ on the other hand, represents the set of states, where at least one general transition has fired. In this area, the system's behaviour depends on the randomly distributed firing times of the general transitions. Both deterministic and stochastic events may occur in this section.

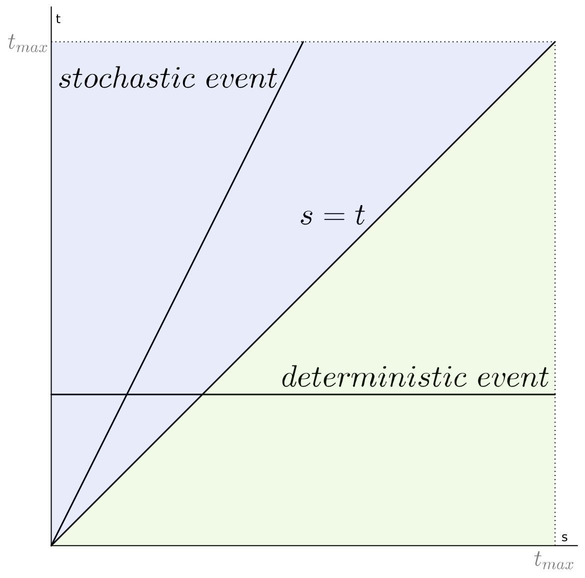


Figure 4.2: Three types of event hyperplanes in a generic STD in $\mathbb{R}^2$ with partitioning in deterministic and stochastic area.

Figure 4.2 illustrates the three types of event hyperplanes for an HPnG with one general transition. The change from the deterministic area to the stochastic area is marked by the hyperplane $s = t$. Points below this hyperplane (green) are associated with states, in which the general transition has not fired. Deterministic events are represented by constant functions similar to the line, which runs parallel to the $s$-axis in the figure. In contrast, the hyperplane corresponding to a stochastic event appears in the stochastic area (blue) of the STD and has a positive or negative slope.

By adding the hyperplanes, that mark all events, which can occur during the evolution of the HPnG to the STD, the regions of the STD are shaped. The process to determine the event hyperplanes and to create the regions is described in detail in Section 4.2 below.

If the set of event hyperplanes, that surround a single region, is viewed as an arrangement, the comprised region corresponds to a union of faces. The states in the interior of the region form an $n$-face of the arrangement, while the parts of the event hyperplanes, that contribute to the region's planar borders, are facets or $(n-1)$-faces. Since the facets represent the occurrence of events, they are called *event facets* following Definition 2 from [29].

**Definition 4.3.** *Let $R$ be a region. An* ***event facet*** *$f_e$ of $R$ is defined by an event hyperplane $H_e$ and a set of boundaries $B = \{b_0 + \sum_{k=1}^{n-1} b_k s_k \leq 0\}$ in $[0, t_{max}]^{n-1}$ with scalars $b_i \in \mathbb{R}$. $B$ is induced by the event hyperplanes of the other facets of $R$, which intersect with $H_e$.*

Faces of lower dimensions, like the vertices and edges of a region, are constituted by the intersections of certain event hyperplanes. By defining regions via arrangements of hyperplanes, we can deduce the following theorem using the relation between arrangements and Nef polyhedra, which has been introduced in Chapter 3:

**Theorem 4.4.** *A region of an STD is a Nef polyhedron.*

*Proof.* A region is a union of the faces of a hyperplane arrangement, as has been described above. With Definition 3.7 from Section 3.2 it follows, that the region is a Nef polyhedron. □

Defining the regions of an STD as Nef polyhedra later guarantees, that boolean set-operations on regions result in Nef polyhedra. This is used in the algorithms presented in Chapter 6.

## 4.2 Creating a Stochastic Time Diagram

An STD forms a partitioning of the set $[0, t_{max}]^n$ into a finite set of regions. The computation of the partitions is described in Algorithm 1 below, which has been introduced in [29]. The algorithm is an extension of the approach for HPnGs with a single general transition first presented in [27].

Compared to [29], the presented algorithm has been slightly altered with respect to a few naming conventions. Apart from that, it is reflected unchanged. An example for the application of the described steps is discussed in Section 4.3.

Starting from an initial event facet $\mathcal{F}_0$ with the initial state $\Gamma$, the next event hyperplanes in $t$-direction are computed (line 1.2). For this purpose, the markings and drifts in $\Gamma$ are analysed, which allows to predict the evolution of the fluid markings and the clocks. Hence, the time, at which the next deterministic transition fires, or the time at which a continuous place reaches its boundary, are computed using the information recorded in the state. If the initial event facet $\mathcal{F}_0$ lies in the deterministic area of the STD, the event hyperplanes marking the firing of a general transition have to be taken into account, as well.

**Algorithm 1** State space partitioning algorithm for the construction of an STD (Algorithm 1 from [29]).

```
Require: $\mathcal{F}_0$, the event facet above which we want to partition the state space, $\Gamma$,
    the current HPnG state, and $\mathcal{R}^{\mathcal{H}}$ as the global set in which all the regions are
    saved.
Ensure: Returns the set of all regions above the given event facet.
1: function PartitionAboveEventFacet($\mathcal{F}_0, \Gamma$)
2:     $E^{\mathcal{H}} \leftarrow$ ComputeNextEvents($\mathcal{F}_0, \Gamma$)
3:     $\mathcal{R} \leftarrow$ CreateRegions($\mathcal{F}_0, E^{\mathcal{H}}$)
4:     $\mathcal{R}^{\mathcal{H}} \leftarrow \mathcal{R}^{\mathcal{H}} \cup \mathcal{R}$
5:     for all $\mathcal{R}_i \in \mathcal{R}^{\mathcal{H}}$ do
6:         for all $f_j \in \mathcal{R}_i$ do
7:             $\Gamma_{new} \leftarrow$ Update($\mathcal{F}_0, f_j, \Gamma$)
8:             PartitionAboveEventFacet($f_j, \Gamma_{new}$)
9: return $\mathcal{R}^{\mathcal{H}}$
```

**Algorithm 2** Computation of regions for a given event facet (Algorithm 2 from [29]).

```
Require: $\mathcal{F}$, the event facet, $E^{\mathcal{H}}$, set of of potential next event hyperplanes.
Ensure: Creates and returns the set of regions directly above the given event facet
    $\mathcal{F}$.
  function CreateRegions($\mathcal{F}, E^{\mathcal{H}}$)
      $\mathcal{F}_{sub} \leftarrow$ CreateSubFacets($\mathcal{F}, E^{\mathcal{H}}$)
      $\mathcal{R} \leftarrow \emptyset$
      for all $f_i \in \mathcal{F}_{sub}$ do
          $\mathcal{R} \leftarrow \mathcal{R} \cup$ FormRegion($f_i, E^{\mathcal{H}}$)
      return $\mathcal{R}$
```

The resulting hyperplanes in $\mathcal{E}^{\mathcal{H}}$ intersect with $\mathcal{F}_0$, and with each other, thereby forming regions adjacent to $\mathcal{F}_0$ (line 1.3). The function CreateRegions, which describes this task, is defined in Algorithm 2.

To create new regions, it uses the event hyperplanes in $E^{\mathcal{H}}$, which are closest to the facet $\mathcal{F}$. The hyperplanes are intersected with $\mathcal{F}$, so that it is partitioned into a set of sub-facets. Above each of the sub-facets, a region is formed. This last step requires an arrangement of hyperplanes or alternatively a convex hull computation.

The computed regions are added to the global set of regions (line 1.4). For each of their event facets, the system state is updated, i.e. the effect of the corresponding event on the system's state is determined (line 1.7), and the partitioning is repeated for the facet with the updated system state $\Gamma_{new}$ (line 1.8). When the initial call of PartitionAboveEventFacet returns, $\mathcal{R}^{\mathcal{H}}$ contains the finite set of regions which constitute the STD.

Each region $R$ has a designated event facet, which is referred to as the *underlying event facet*. It acts as the entry point to the region and is associated with the system state $\Gamma_R$. In fact, the underlying event facets are the facets from Algorithm 1, which are passed to the function PartitionAboveEventFacet.

Given $\Gamma_R$, the marking of each place can be determined for the time period covered by the region. Consequently, instead of recording infinitely many states, only the system state for each region is stored, while the required values can be computed on demand. The discrete marking is recorded directly in the associated state $\Gamma_R$ and does not change inside the region by definition. The continuous marking $x_t$ at a time $t$ inside of region $R$, however, has to be calculated from $\Gamma_R$.

Assume a continuous place $p \in \mathcal{P}^C$ has an initial marking $x_0$ when entering $R$ at time $t_0$. $t_0$ is given as the hyperplane equation of the underlying event facet: $0 = a_0 + a_1 s_1 + \cdots + a_{n-1} s_{n-1} + a_n * t_0$. Then the marking $x_t$ can be computed as $x_t = x_0 + (t - t_0) * d_p^R$, where $d_p^R$ is the drift of place $p$ in $R$. In Chapter 6 this equation is used to compute the hyperplane which marks the time where a continuous place reaches a certain threshold inside a region.

## 4.3 Example

Using the HPnG from Section 2.3.2 with a few changes, the presented algorithm is applied exemplary.

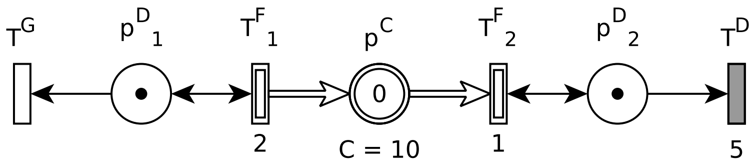


Figure 4.3: A simple HPnG with a single general one-shot transition.

The system is examined for the time interval $[0, 10]$. Beginning from the initial event hyperplane $t = 0$ the next possible events have to be identified. First of all, the general Transition $T^G$ may fire at any time, so that the token from place $p_1^D$ is removed and transition $T_1^F$ is disabled. This event is recorded in Figure 4.4 as the line that splits the two-dimensional space $[0, 10]^2$ diagonally in half.

The second possible event to occur is the firing of the deterministic transition $T^D$ at time $t_D = 5$. Similar to the firing of $T^G$, this event disables transition $T_2^F$ by removing the token from place $p_2^D$. The initial event hyperplane together with the two described events forms the first region $R_1$ of the STD. Compared to the initial state, only the continuous marking of place $p^C$ and the clocks change inside $R_1$.

According to Algorithm 1, the partitioning has to be repeated for all event facets of $R_1$. We thus look at the facet that is part of the hyperplane $s = t$: Before the firing of $T^G$ the drift of $p^C$ has been 1, so that the place has been filled with $s$ units of fluid when the event occurs. When $T^G$ fires, the transition $T_1^F$ is disabled and the drift of $p^C$ is reduced to $-1$. It takes additional $s$ time units to reach the lower boundary of $p^C$ Hence, after a total of $2s$ time units the place has been filled and then emptied, again.

This results in the hyperplane $t = 2s$ shown in Figure 4.5. As before, the firing of the deterministic transition occurs at time $t_D = 5$ and disables $T_2^F$. If both the general and the deterministic transition have fired, the system is in a halting state since all transitions are disabled. The same holds due to rate adaption of $T_2^F$, when $p^C$ has been drained. The upper end of the line $t = 2s$ is thus displayed dashed, to indicate, that the corresponding event can not occur any more, after both the general transition and the deterministic transition have fired.

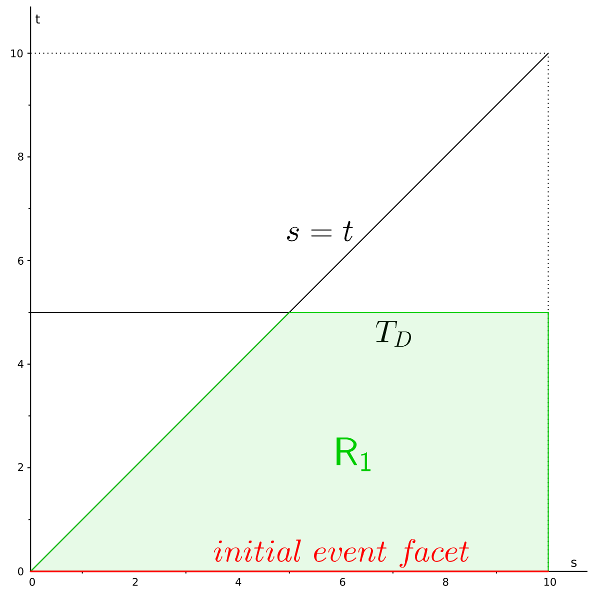


Figure 4.4: STD example: identifying the first set of events.

Depending on the firing time of $T^G$ either reaching the lower bound of $p^C$ or the firing of $T^D$ is the next event. If $p^C$ has been filled with more than 2.5 units of fluid (that is if $T^G$ has fired later than $t = 2.5$), the firing of $T^D$ is the next possible event. Otherwise the marking of $p^C$ reaches zero first at $t = 2s$. In total we receive the second region $R_2$, which is limited by the facet of $R_1$ and the described events.

The same process is recursively repeated for $R_2$ and the subsequently created regions, which results in the STD shown in Figure 4.6. After the algorithm has finished, the STD consists of six regions with six different system states.

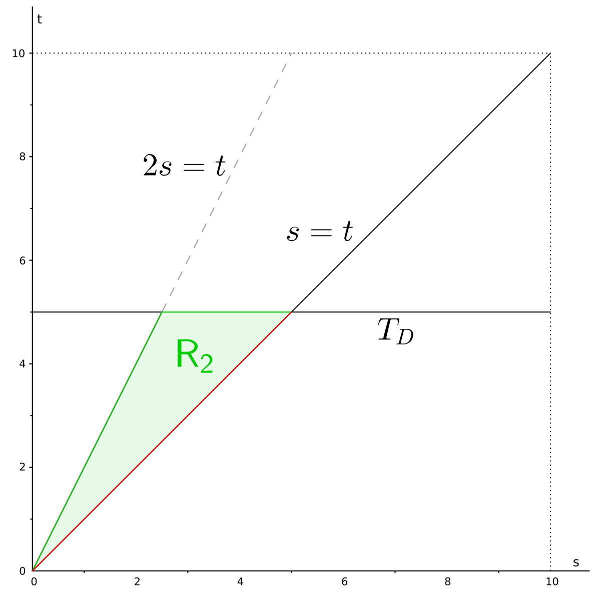


Figure 4.5: STD example: partitioning above the $s$-$t$-plane.

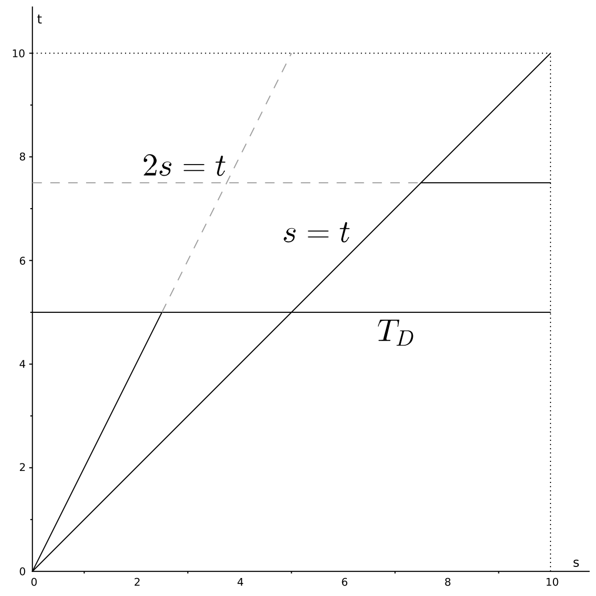


Figure 4.6: The STD for the HPnG example.

# Chapter 5

# Stochastic Time Logic

In this chapter, a logic is introduced, that allows for the specification of HPnG properties. According to other temporal logics like Metric Interval Temporal Logic (MITL, see [4]), the logic for HPnGs is called *Stochastic Time Logic* (STL). The term *Stochastic Time Logic* has been introduced in [28]. It is an extension of the grammar provided in [31], which, in contrast, does not contain an operator for reachability analysis.

An STL formula describes properties which refer to characteristics of an HPnG. As has been described in Chapter 2, a state is determined by the system time and the general transition firing times. It can be easily verified, whether a state satisfies an STL formula. But to allow for the computation of the *satisfaction set* at a given time, all states, that is in particular all combinations of general transition firing times, for which the specification holds, have to be identified.

Since the firing times of the general transitions follow probability distributions, the overall behaviour of an HPnG is probabilistic, too. The STL itself does not take this fact into account. Thus, an additional operator is specified, which determines the transient probability to be in a satisfying state and compares it to a required threshold.

At the beginning of this chapter, the syntax of the STL is presented along with a short introduction to parse trees. Afterwards, the semantics of the formulae and the probability operator are formally defined. This part is crucial for model checking of an HPnG, since it formalises the satisfaction of formulae. In the final section, the HPnG example, which has been studied in previous chapters, is utilised again to demonstrate how STL formulae can be used to specify a property of an HPnG.

## 5.1 Syntax of Stochastic Time Logic

The syntax of a logic specifies the structure of its expressions in the form of a grammar. The syntax of STL is provided in so-called *Extended Backus-Naur Form* (EBNF, see [40]) or a simplification of it, respectively. The sequence of deviations for a word of the grammar can be visualised by a so-called *parse tree.*

### 5.1.1 The STL Syntax

The presented version of the STL grammar follows Definition 2 appearing in [28].

**Definition 5.1.** *Let $m_p$ be the marking of a discrete place, $x_p$ the marking of a continuous place, $\sim \in \{<, \leq, >, \geq\}$, $i \in \mathbb{N}_0$, $a \in \mathbb{R}^{\geq 0}$, and $t_1, t_2 \in [0, t_{max}]$. The grammar of the Stochastic Time Logic is defined as follows:*

$$\phi := true | \underbrace{m_p \sim i | x_p \sim a}_{(a)} | \underbrace{\neg\phi | \phi_1 \wedge \phi_2}_{(b)} | \underbrace{\psi_1 \mathcal{U}^{[t_1,t_2]} \psi_2}_{(c)}$$

$$\psi := true | m_p \sim i | x_p \sim a | \neg\psi | \psi_1 \wedge \psi_2$$

The properties of HPnGs are reflected in the so-called *atomic formulae* (a), which refer to the marking of discrete and continuous places. In contrast to the definition in [28], we allow to compare a discrete marking using the operators $<$, $\leq$, $>$, and $\geq$ instead of $=$. The test for equality can however be implemented using a conjunction of $\leq$ and $\geq$, which is why we do not lose any expressiveness with this approach. *Compound formulae* (b) are formed by negation and conjunction. According to De Morgan's laws (cf. [38]), other logical operators like $\vee$ and $\Rightarrow$ can be constructed using these connectives.

The *time bounded Until* (c), which requires that a property $\psi_1$ holds until another property $\psi_2$ finally holds, is a temporal modality. It is a special formula type, since it is the only one that makes a statement regarding the evolution of the system. It is shown in Section 5.2, that this leads to a more complex definition of the formula's semantics in comparison to the other types. Note, that the grammar explicitly forbids the formation of expressions with nested Until formulae.

The formula *true* is actually just syntactic sugar for the expression $\phi \vee \neg\phi$. It could therefore be omitted in the definition. Nevertheless it is kept to ease the formulation of certain properties like reachability (e.g. $true \mathcal{U} \psi$ stands for "$\psi$ will hold eventually").

### 5.1.2 Parse Trees

Every word of the STL grammar implies a *parse tree*, which depicts the expression's structure. The atomic formulae in an STL word represent the leaves of the parse tree, since they do not contain any sub-formulae. In contrast, the compound formulae and the time bounded Until form the inner nodes of the tree.

The parse tree combined with the method of traversal later determines the order, in which the sub-formulae of an expression are examined during the model checking process. Since the sub-formulae have to be evaluated first before the value of the complete expression can be calculated, a post-order traversal (cf. [43]) is applied to the parse tree.

For example, assume we want to specify a property for a simple web server. The server might not work properly all the time due to technical issues while providing its service, which leads to a decreased throughput. While repairing the server, it still processes requests but with a considerably lower rate. This simple server can be represented by an equally simple HPnG.

Now, we want to guarantee that the utilisation of the server in some time interval $[t_1, t_2]$ is not exceeded (i.e. reaches 100%) as long as it is under repair. This is specified by the following STL formula:

$$\phi := (x_{util} \leq 99.9) \mathcal{U}^{[t_1,t_2]} ((m_{repair} \leq 0) \wedge (\neg(m_{working} \leq 0))).$$

$\phi$ is an expression of the STL grammar with additional brackets to clarify its structure. The sub-formula $\neg(m_{working} \leq 0)$ can and would obviously be replaced in a real-world example by $m_{working} \geq 1$. However, since we are currently only interested in the syntactic structure of $\phi$, this artificial but slightly more complex example is used.

The root of the formula $\phi$ is the time bounded Until, which, all in all, yields the parse tree presented in Figure 5.1.

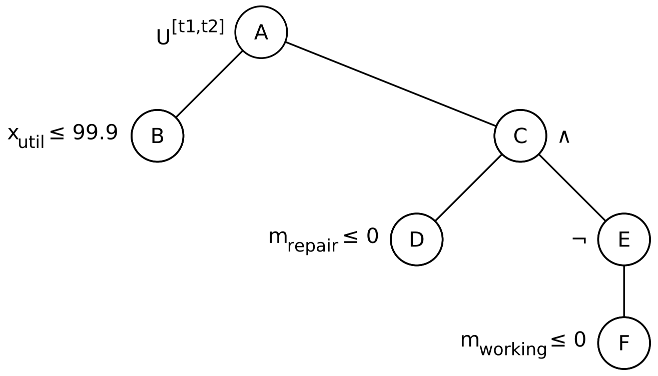


Figure 5.1: The parse tree for the STL formula $\phi := (x_{util} \leq 99.9)\,\mathcal{U}^{[t_1,t_2]}\,((m_{repair} \leq 0) \wedge (\neg(m_{working} \leq 0)))$.

The tree consists of three inner nodes ($A$, $C$, $E$) and three leaves ($B$, $D$, $F$). Traversing the tree in post-order yields the following sequence of nodes: $B$, $D$, $F$, $E$, $C$, $A$. The sequence represents the order, in which the sub-formulae are model checked to determine the set of general transition firing times, which satisfy the entire expression $\phi$. An insight into the implementation of the parse tree traversal is delivered in Chapter 7.

## 5.2 Semantics of Stochastic Time Logic

The semantics of a logic is a mapping, that determines the meaning of the logic's expressions in a certain context. Here, the context is the state of an HPnG. By defining a so-called *satisfaction relation* between a state of an HPnG and the expressions of the STL syntax, the STL is equipped with the means to describe the properties of a hybrid Petri net. To express that an arbitrary context $C$ meets a specification $\phi$, the notation of Definition 5.2 is used (cf. [7]):

**Definition 5.2.** *Given a context $C$ and a specification $\phi$. We write:*

$$C \models \phi$$

*to denote that the specification $\phi$ is* ***satisfied*** *in the context $C$.*

The above definition introduces the general notation for satisfaction. The following paragraphs present satisfaction relations for states, sets of states, and the probability operator.

### 5.2.1 Satisfaction of Formulae on States

The following definition is based on the explanations provided in [29] and [27]. It relates a state $\Gamma(\vec{s}, t)$ of an HPnG to an STL formula.

**Definition 5.3.** *Given a state $\Gamma(\vec{s}, t)$ of an HPnG $P$ at time $t$ with general transition firing times $\vec{s}$ and an STL formula $\phi$. We write:*

$$\Gamma(\vec{s}, t) \models^{\vec{s},t} \phi$$

*to denote that the state $\Gamma(\vec{s}, t)$ $\vec{s}$-$t$-**satisfies** the specification $\phi$. Note, that $\models^{\vec{s},t}$ is a fixed symbol, where $^{\vec{s},t}$ is not related to the parameters $\vec{s}$ and $t$ of the state $\Gamma$.*

The relation assigns a truth value to each pair consisting of a state and a formula. In terms of propositional logic it is thus called a *valuation* of the property $\phi$ in the context $\Gamma(\vec{s}, t)$ (cf. [38]).

While Definition 5.3 introduces the notation to reason about the satisfaction of a state regarding an arbitrary STL formula, specific relations for the satisfaction of atomic, compound, and Until formulae are defined subsequently.

**Definition 5.4** (Satisfaction of atomic formulae)**.** *Let an HPnG with $(n-1)$ general transition firings in the time interval $[0, t_{max}]$ be given.*

- $\forall \vec{s} \in [0, t_{max}]^{n-1}, t \in [0, t_{max}] : \Gamma(\vec{s}, t) \models^{\vec{s},t} true$*,*
- $\Gamma(\vec{s}, t) \models^{\vec{s},t} m_p \sim i \iff \Gamma(\vec{s}, t).m_p \sim i$*, where $p$ is a discrete place,* $\sim \in \{<, \leq, >, \geq\}, i \in \mathbb{N}_0$*,*
- $\Gamma(\vec{s}, t) \models^{\vec{s},t} x_p \sim a \iff \Gamma(\vec{s}, t).m_x \sim a$*, where $p$ is a continuous place,* $\sim \in \{<, \leq, >, \geq\}, a \in \mathbb{R}^{\geq 0}$*.*

The satisfaction of an atomic property can be directly deduced from the given state. While *true* simply holds without a condition, the discrete and continuous formulae check the marking of the state. If the requested condition $\sim$ is met, the formula holds for the state. Otherwise, it does not hold.

**Definition 5.5** (Satisfaction of compound formulae)**.**

- $\Gamma(\vec{s}, t) \models^{\vec{s},t} \neg\phi \iff \neg\Gamma(\vec{s}, t) \models^{\vec{s},t} \phi$*,*
- $\Gamma(\vec{s}, t) \models^{\vec{s},t} \phi_1 \wedge \phi_2 \iff \Gamma(\vec{s}, t) \models^{\vec{s},t} \phi_1 \wedge \Gamma(\vec{s}, t) \models^{\vec{s},t} \phi_2$*.*

Checking compound properties is a two-step process. First the satisfaction of the inner formulae is verified, then the particular operator is applied to the result. The negation operator inverts the truth value of $\Gamma(\vec{s}, t) \models \phi$. A conjunction formula holds if both its sub-formulae hold for the state.

**Definition 5.6** (Satisfaction of time bounded Until)**.** *Let an HPnG in the time interval $[0, t_{max}]$ be given, and let $0 \leq t + t_1 < t + t_2 \leq t_{max}$.*

$$\begin{aligned}\Gamma(\vec{s}, t) \models^{\vec{s},t} \psi_1 \mathcal{U}^{[t_1,t_2]} \psi_2 \iff & \exists \tau \in [t + t_1, t + t_2] : \Gamma(\vec{s}, \tau) \models^{\vec{s},t} \psi_2 \wedge \\ & (\forall \tau' \in [t, \tau] : \Gamma(\vec{s}, \tau') \models^{\vec{s},t} \psi_1)\end{aligned}$$

Investigating the satisfaction of a time bounded Until formula is more complicated than the previous cases, since it requires an examination of the evolution in the time interval $[t, t + t_2]$. The state $\Gamma(\vec{s}, t)$ and its successors have to fulfil three conditions to satisfy an Until formula:

1. The property $\psi_1$ must hold in all states $\Gamma(\vec{s}, \tau')$, where $\tau' \in [t, t+t_1]$. If $t_1 = 0$, this condition is omitted.
2. The property $\psi_1$ must hold in all states $\Gamma(\vec{s}, \tau'')$, where $\tau'' \in [t + t_1, \tau)$.
3. The property $\psi_2$ must hold eventually in some state $\Gamma(\vec{s}, \tau)$, where $\tau \in [t + t_1, t + t_2]$.

So, before the time $\tau$ in the interval $[t + t_1, t + t_2]$, in which $\psi_2$ shall hold, is reached, $\psi_1$ must hold uninterruptedly beginning from $\Gamma(\vec{s}, t)$ through all following states up to $\Gamma(\vec{s}, \tau)$. This is schematically depicted in Figure 5.2.

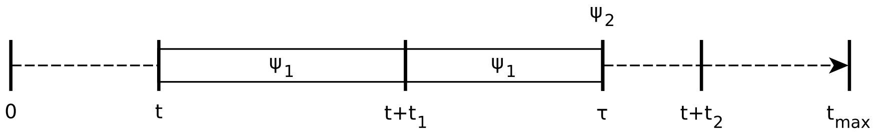


Figure 5.2: Schematic representation of the satisfaction of a time bounded Until formula for a single evolution.

The intervals $[0, t)$ and $(\tau, t_{max}]$ of the system evolution are not relevant for the satisfaction of the Until formula. Hence, it does not matter, whether $\psi_1$ is satisfied before $t$ or whether $\psi_2$ is satisfied after $\tau$.

### 5.2.2 Satisfaction of Formulae on Satisfaction Sets

The firing times of the general transitions follow arbitrary continuous probability distributions. This leads to multiple system evolutions for the initial system state $\Gamma(0)$ at time $t = 0$ as has been emphasised in the previous chapter. By representing the state space of the HPnG as an STD, we were able to condition the stochastic behaviour of the system. The different system evolutions, that are combined in the STD, hence do not all occur with the same probability nor do they fulfil the same properties. The satisfaction of a formula can thus not be verified by the examination of a single system evolution.

Instead, those evolutions, or more specific those sets of general transition firing times, have to be identified, which satisfy a given formula, and afterwards the total probability of the collected general transition firing times has to be inferred by deconditioning their probabilities. This process is captured in the following definitions.

**Definition 5.7.** *Let $P$ be an HPnG with $n$ general transition firings, $\phi$ an STL formula, and $S \subseteq [0, t_{max}]^n$ a set of general transition firing times for $P$. We write:*

$$S \models^t \phi \iff \forall \vec{s} \in S : \Gamma(\vec{s}, t) \models^{\vec{s},t} \phi$$

*to denote that $S$ **$t$-satisfies** $\phi$ at some time $t$. Similar to Definition 5.3, the exponent $^t$ is only part of the operator $\models^t$ and independent of the time $t$.*

Definition 5.7 determines, what it means, that a set of combinations of possible general transition firing times satisfies a formula. The definition is the continuation of Definition 4 presented in [28] for sets of arbitrary dimensions.

**Definition 5.8.** *Let $P$ be an HPnG with $n$ general transition firings. The largest set $S \subseteq [0, t_{max}]^n$, which $t$-satisfies a formula $\phi$ at time $t$, is called the **satisfaction set of $\phi$ at time** $t$ denoted as $Sat(\phi, t)$.*

By deconditioning the firing times in the satisfaction set $Sat(\phi, t)$ with respect to the distributions of the corresponding random variables, the total probability for the satisfaction of the formula $\phi$ at time $t$ can be calculated.

**Definition 5.9.** *Let $P$ be an HPnG with $n$ general transition firings. To denote that a system state $\Gamma(t)$ at time $t$ satisfies a probability operator $\mathbb{P}_{\triangleright p}(\phi)$ with an STL formula $\phi$, a probability $p \in [0, 1]$, and a comparison operator $\triangleright \in \{<, \leq, >, \geq\}$, we write:*

$$\Gamma(t) \models \mathbb{P}_{\triangleright p}(\phi) \iff Prob(\phi, t) \triangleright p$$

*where*

$$Prob(\phi, t) := \int \cdots \int_{Sat(\phi,t)} g_1(s_1) * \cdots * g_n(s_n)\, ds_n \ldots ds_1$$

*is the multiple integral over the domain $Sat(\phi, t)$ with the probability density functions $g_i$ of the general transition firing times.*

The probability operator, that is introduced in Definition 5.9, compares the probability of the firing times in $Sat(\phi, t)$ with a given threshold. A system state $\Gamma(t)$ at time $t$ of an HPnG then satisfies the probability operator, if the threshold is met with respect to the relational operator $\triangleright$.

The formation of the multiple integral over the general transition firing time distributions as a simple product is possible, since the distributions are stochastically independent (cf. [15]) of each other. In the next chapter the calculation of this integral is thoroughly discussed for two general transition firings.

## 5.3 Example

We use an extended version of the example HPnG from Chapter 2 and 4 to demonstrate how STL formulae specify properties of an HPnG.

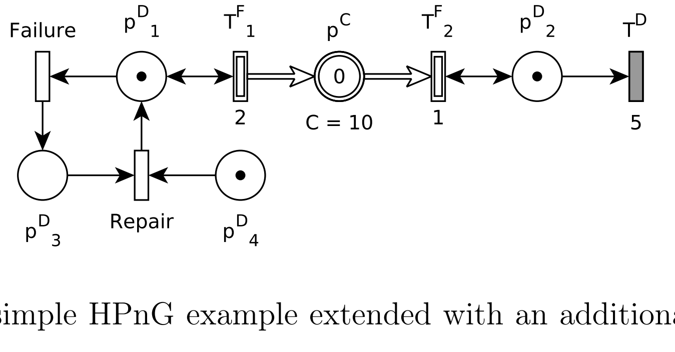


Figure 5.3: The simple HPnG example extended with an additional repair mechanism.

The example HPnG in Figure 5.3 contains four discrete places ($p_1^D$, $p_2^D$, $p_3^D$, $p_4^D$) and one continuous place ($p^C$). The continuous place starts off with a fluid level of zero and can at most contain 10 units. Except for $p_3^D$ all discrete places hold one token in the initial state. The general transition *Repair* is disabled in the initial state, since there is no token in the place $p_3^D$. After the general transition $Failure$ has fired, however, *Repair* is enabled and can re-enable the transition $T_1^F$ consuming both the token in $p_3^D$ and $p_4^D$.

The STL allows us to reason about the marking of the discrete and continuous places of the HPnG. To specify that the place $p^C$ contains at least five units of fluid, we can write: $\phi_1 := x_{p^C} \geq 5$. Another simple atomic formula would be $\phi_2 := m_{p_1^D} \geq 1$. $\phi_2$ postulates that a token is present in $p_1^D$. This condition is fulfilled if either the general transition $Failure$ has not fired, or if *Repair* has fired after it was enabled.

Characteristics like *safety* and *liveness* can be expressed using the time bounded Until. A safety property demands that an undesired state is never reached, whereas a liveness property claims that a desired state finally occurs (cf. [7]).

Now, we specify a safety property for the HPnG from Figure 4.3. If the general transition $Failure$ fires, the drift of $p^C$ becomes negative. Thus, the place might run empty while the repairing is still in progress. We therefore want to guarantee that the amount of fluid in $p^C$ does not drop below 5 as long as $T_1^C$ is disabled. This can be expressed by:

$$\phi_3 := \underbrace{\neg((\phi_2 \wedge m_{p_4^D} \geq 1) \wedge \neg(\phi_1))}_{\psi_1} \, \mathcal{U}^{[t_1,t_2]} \, \underbrace{m_{p_4^D} < 1}_{\psi_2} .$$

The parse tree of $\phi_3$ is shown in Figure 5.4.

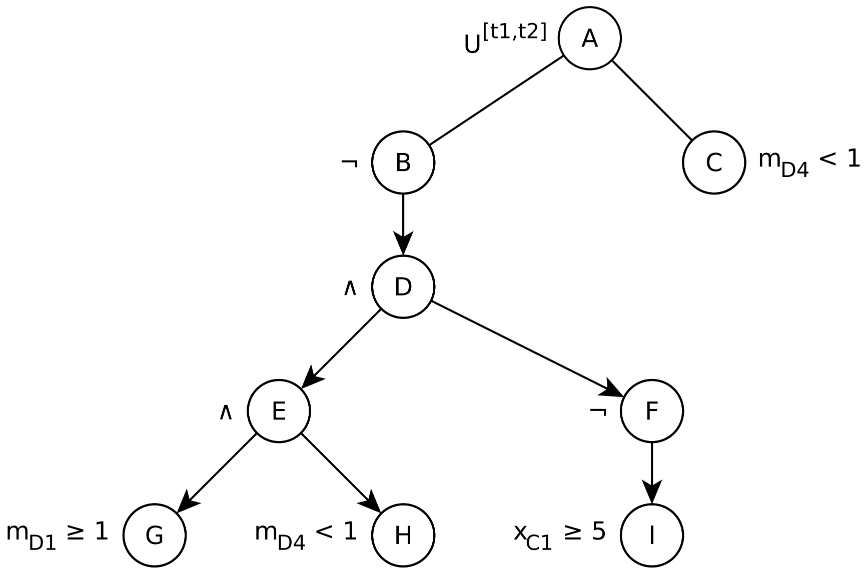


Figure 5.4: Parse tree of the safety property $\phi_3 := \neg((\phi_2 \wedge m_{D_4} \geq 1) \wedge \neg(\phi_1))\,\mathcal{U}^{[t_1,t_2]}\, m_{D_4} < 1$.

The property $m_{p_4^D} \geq 1$ is satisfied, if a token is present in the place $p_4^D$. The system is under repair, when $m_{p_4^D} \geq 1$ holds and the token in place $p_1^D$ has been consumed by the transition $Failure$. During this period, the amount of fluid in $p^C$ shall in addition be greater than or equal to 5, which is specified by the formula $\phi_1$. Or otherwise stated, if the system is under repair, $p^C$ must contain at least 5 units of fluid. This implication is expressed by $\phi_2 \wedge m_{p_4^D} \geq 1 \Rightarrow \phi_1$ which is equivalent to the left hand sub-formula $\psi_1$. Since the implication operator $\Rightarrow$ is not part of the STL syntax, it is described by multiple conjunctions and negations.

As soon as the repairing process is finished, i.e. the transition *Repair* has fired, the token in $p_4^D$ is consumed and the right hand sub-formula $\psi_2$ is satisfied. If it can be shown that the Until formula $\phi_3$ is $\vec{s}$-$t$-satisfied by a state $\Gamma(\vec{s}, 0)$ for the parameters $t_1 = 0$ and $t_2 = t_{max}$, it is guaranteed that $p^C$ never contains less than 5 units of fluid when the system is under repair during the corresponding evolution. The vectors $\vec{s}$ of all those states together form the satisfaction set $Sat(\phi_3, 0)$. How this set is determined, is discussed in the following chapter on model checking of HPnGs.

# Chapter 6

# Model Checking Hybrid Petri Nets with General One-Shot Transitions

In this chapter the structures presented in the preceding sections are combined in order to model check HPnGs. The desired properties of an HPnG are specified as STL formulae, for which the satisfaction set is computed using the regions of the corresponding STD. As before, the introduced concepts are applicable to HPnGs with an arbitrary number of general transitions.

Model checking atomic and compound formulae mostly reduces to intersecting regions with hyperplanes, and can thus be dealt with in short. Investigating time bounded Until formulae however requires a more complex process, which utilises the structure of the STD extensively. The algorithms, that are presented in the following sections, rely on geometric operations on Nef polyhedra. In Chapter 4 an implementation has been mentioned, which creates STDs for HPnGs with two general transitions. It also includes a function to check a continuous property, that, however, does not operate on Nef polyhedra, which is why it differs fundamentally from the approach presented here.

The chapter is organised as follows: A rough overview of the model checking process is given at first in Section 6.1. Afterwards, the model checking of STL formulae on HPnGs is examined in detail. Checking atomic and compound formulae is discussed briefly in Section 6.2, before the process is highlighted separately for the time bounded Until in Section 6.3. The chapter is concluded by a discussion on the evaluation of the probability operator in Section 6.4.

## 6.1 Overview of the Model Checking Process

In general, the model checking process consists of three steps. At first, a model of the system, that shall be examined, is defined. For this model, a property, which describes a desired or undesired state of the system, is expressed in a formal language. At last, the actual checking is executed by the so-called *model checker*, which results in one of two possible outcomes. Either the property is satisfied by the system, or it is not satisfied. In the latter case, a counterexample is usually provided to prove why the property has been declined (cf. [7]).

Some parts of the process specialised for HPnGs have already been introduced in previous chapters. Systems are of course modelled as *HPnGs* (Chapter 2), while their properties are described as *STL formulae* (Chapter 5). The model checker first generates an *STD* (Chapter 4) consisting of regions, which are *Nef polyhedra* (Chapter 3). Up to this step, all involved components have been presented before. The current chapter now discusses the subsequent work of the model checker in detail.

The created STD is a graphical representation of the HPnG's state space. Every point in the STD stands for a vector $\vec{s}$ of general transition firing times and a point $t$ in time, which together uniquely identify a state $\Gamma(\vec{s}, t)$ of the HPnG. Since the satisfaction of STL formulae is defined over these states, the STD can be used to determine the satisfaction set $Sat(\phi, t)$ for some formula $\phi$. We utilise, that each region of the STD comprises states, which are identical except for their fluid marking and the values of their clocks. It is thus possible to compute the satisfaction set of $\phi$ by applying geometric operations to the regions, instead of checking each state individually. The used algorithms are described in detail in Section 6.2 and Section 6.3.

The satisfaction set of a formula $\phi$ contains the general transition firing times, which correspond to the evolutions fulfilling the formula. For two general transitions, $Sat(\phi, t)$ is a set of polygons in $[0, t_{max}]^2 \subset \mathbb{R}^2$. Since the firing times of the general transitions follow arbitrary continuous probability distributions, the different evolutions occur with different probabilities. By integrating the probability density functions over the domain covered by $Sat(\phi, t)$, the probability for the satisfaction of $\phi$ at time $t$ can be calculated. If the probability is high or low enough compared to some threshold, the model checker accepts the formula as satisfied. Otherwise, no single counterexample is provided, but instead the too small or too large satisfaction set serves as a witness for the rejected formula. The threshold for the acceptance can be defined with the help of the probability operator. This last step is discussed in Section 6.4.

The complete process is illustrated in Figure 6.1.

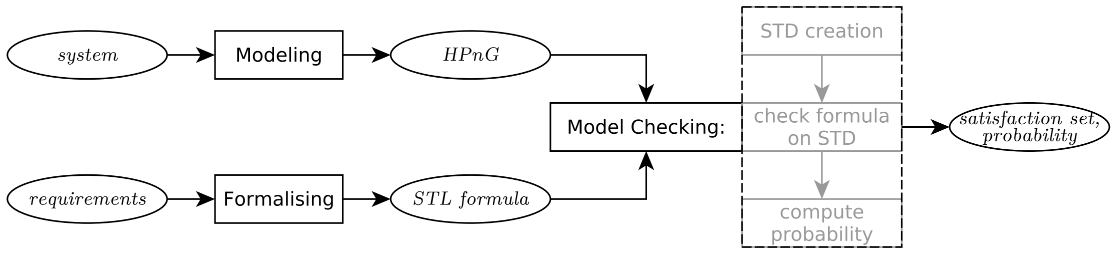


Figure 6.1: The model checking process for hybrid Petri nets with general one-shot transitions.

The oval nodes in the picture represent entities, whereas the rectangles stand for processes. Modelling the system to examine as an HPnG and formalising the properties as STL formulae are tasks, which have to be performed manually. The subsequent model checking step, however, can be executed automatically according to the algorithms, that are introduced in the following sections.

## 6.2 Model Checking Atomic and Compound Formulae

This section is exclusively concerned with the study of atomic and compound STL formulae. The descriptions that are given here do thus not hold in the context of time bounded Until formulae unless stated otherwise.

As has been indicated in Chapter 5, it is sufficient to examine the STD at a time $t = d$ to compute the satisfaction set $Sat(\phi, d)$ for atomic and compound formulae. To identify the relevant regions, all Nef polyhedra of the STD are intersected with the hyperplane $H_d := t = d$. It is then determined which parts of these regions fulfil the formula $\phi$. Depending on the type of $\phi$, this step requires different approaches.

Discrete STL formulae reason about the discrete marking of the Petri net, which does not change inside a region. Such a formula therefore either holds in the complete region or not at all. Continuous properties specify a threshold for a fluid marking. The points of a region, that identify states in which this threshold is reached, form a hyperplane that splits the region in two parts. In one part, the formula is fulfilled, in the other part it is violated. The subsets of a region that satisfy a conjunction are determined by the intersection of the point sets fulfilling both the left-hand formula and the right-hand formula, respectively. For a negation, the difference between the whole region and the set, that satisfies the positive formula, is computed.

Irrespective of the formula type, the set resulting from the analysis of a region is again a Nef polyhedron, which is stated in Theorem 6.1. Keep in mind, that we explicitly exclude Until formulae here, which is why they do not need to be covered in the theorem.

**Theorem 6.1.** *The maximal set $S$ of points $(\vec{s}, t)$ in a region $R$, which $\vec{s}$-$t$-satisfy an atomic or compound STL formula $\phi$, is a Nef polyhedron.*

*Proof.* The theorem is proven by structural induction over the formula $\phi$.

**Base case:** Let $\phi := true$. Every point of a region fulfils *true*. Since every region is a Nef polyhedron (Theorem 4.4), $S$ is a Nef polyhedron.

Let $\phi := m_p \sim i$ be a discrete formula. Either $R$ fulfils $\phi$ completely or not at all. In either case, $S$ is a Nef polyhedron (cf. Section 3.2).

Let $\phi := x_p \sim a$ be a continuous formula. The limit $a$ for $x_p$ in $R$ implies a hyperplane $H_a$. Since every half-space is a Nef polyhedron and Nef Polyhedra are closed under intersection (cf. Section 3.2), $S$ is a Nef Polyhedron as the intersection of $H_a^{(+)}$ and $R$.

**Inductive hypothesis:** Suppose the theorem holds for all proper sub-formulae of $\phi$.

**Inductive step:** Let $\phi := \phi_1 \wedge \phi_2$ be a conjunction. The sets $S_1$ and $S_2$, which consist of the points satisfying $\phi_1$ and $\phi_2$ in $R$, respectively, are Nef polyhedra by the inductive hypothesis. Since Nef polyhedra are closed under intersection, $S = S_1 \cap S_2$ is a Nef polyhedron.

Let $\phi := \neg\phi_1$ be a negation. The set $S_1$ of the points satisfying $\phi_1$ is a Nef polyhedron by the inductive hypothesis. Since Nef polyhedra are closed under set difference, the set $S = R \setminus S_1$ is a Nef polyhedron.

□

**Corollary 6.2.** *The union $\bigcup_R S_R$ of all maximal sets $S_R$ of points $(\vec{s}, t)$, which t-satisfy a formula $\phi$ in the regions $R$ of the STD, is a Nef polyhedron.*

*Proof.* Follows immediately, since Nef polyhedra are closed under $\cup$. □

Theorem 6.1 and Corollary 6.2 later play an important role in the checking of time bounded Until formulae. For the examination of atomic and compound properties Theorem 6.1 is relevant from a practical point of view, since it guarantees that there is no difference between the data structures resulting from the checking of the different formula types. Thus, the next steps can be described irrespective of the sort of formula.

The computed Nef polyhedra now consist of all points $(\vec{s}, t)$, that identify states $\Gamma(\vec{s}, t)$ for which the formula $\phi$ holds. Since we are only interested in the states $\Gamma(\vec{s}, d)$ at time $t = d$ which satisfy $\phi$, the Nef polyhedra are again intersected with the hyperplane $H_d$. By stripping the time component $d$ from the points $(\vec{s}, d)$ of this intersection, we receive the general transition firing times, for which the formula $\phi$ is satisfied at time $t = d$. Projecting all points of the intersection in this manner, results in the satisfaction set $Sat(\phi, d)$.

The procedure to determine the fulfilling parts of the regions in an STD for an atomic or compound STL formula is presented in Algorithm 3.

**Algorithm 3** Generic model checking procedure for atomic and compound STL properties.

```
Require: $\mathcal{R}$, the regions of the STD, $\phi$, the formula to check, $d$, the time to check.
Ensure: Returns the satisfaction set $Sat(\phi, d)$ containing all general transition firing times that fulfil $\phi$ at time $d$.
1: function CheckFormula($\mathcal{R}$, $\phi$, $d$)
2:     for all regions $R \in \mathcal{R}$ do
3:         if $R$ is at time $d$ then
4:             $P \leftarrow P \cup$ Check<type>($R$,$\phi$)
5:     return IntersectionToSubspace($P$, $H_d$)
```

The condition in line 3 excludes the regions which do not intersect with $H_d$. The difference between the checking of the various formula types is reflected in line 4, which defines what parts of a region satisfy a formula. At last, the call in line 5 intersects the Nef polyhedron $P$ with the hyperplane $H_d$ and extracts the general transition firing times from the result to receive the satisfaction set $Sat(\phi, d)$.

The notation Check<type> with angle brackets indicates that the appropriate sub-procedure, which matches the type of $\phi$, has to be called, that is for example CheckDiscrete if $\phi$ is a discrete STL formula. In the following, we look in particular into the algorithms for fluid formulae (CheckFluid) and conjunction formulae (CheckConjunction).

Checking a fluid formula $\phi := x_p \sim z$ is more complex than examining a discrete formula, since its satisfaction might change inside a region. Therefore we have to determine for each region intersecting the time plane $H_d$, which of its parts fulfils the continuous formula. Algorithm 4 defines the approach to this task.

**Algorithm 4** Procedure to identify the subset $P$ of a region $R$ fulfilling a continuous STL property $\phi = x_p \sim z$.

```
Require: $R$, a region, $\phi := x_p \sim z$, the fluid formula.
Ensure: Returns the subset $P$ of $R$ that fulfils $\phi$.
1: function CheckFluid($R$,$\phi$)
2:     $\mathcal{H}_z \leftarrow$ ComputePropertyPlane($R$, $p$, $z$)
3:     if $\sim \in \{>, \geq\}$ then
4:         $\mathcal{H}_z \leftarrow \mathcal{H}_z.opposite()$
5:     $P \leftarrow$ HalfspaceIntersection($R$,$\mathcal{H}_z$)
6:     return $P$
```

It has been described in Chapter 4, that each region $R$ has a designated initial event facet, which is marked as the entry point to the region during the creation of the STD. Each of these facets is associated with a system state $\Gamma_R$, which is used in the function ComputePropertyPlane to compute a hyperplane $\mathcal{H}_z$ marking the threshold $z$ in $R$ (line 2). In $\Gamma_R$ the marking $x_p(t_0)$ and the drift $d_p$ are stored. With these values, the fluid level $x_p(t)$ for place $p$ at time $t$ in $R$ can be calculated as follows:

$$x_p(t) = x_p(t_0) + (t - t_0) * d_p. \tag{6.1}$$

Since the drift is constant within a region, the amount of fluid in $p$ is the sum of the value when entering $R$ ($x_p(t_0)$) and the fluid accumulated during the time already spent in $R$ ($(t - t_0) * d_p$). By replacing $x_p(t)$ with the threshold $z$ in Equation 6.1, we can solve it for the time $t$ at which $z$ is reached.

**Example.** *In the specific case of HPnGs with two general transitions, $t_0$ can be expressed as a hyperplane $t_0 = as_1 + bs_2 + c$ with the general transition firing times $s_1, s_2 \in [0, t_{max}]$ and scalars $a, b, c \in \mathbb{R}_0^+$. $t_0$ is the hyperplane equation of the initial event facet of the examined region. The initial fluid level $x_p(t_0)$ in region $R$ also follows a hyperplane equation $x_p(t_0) = \alpha s_1 + \beta s_2 + \gamma$ depending on the general transition firing times and three scalars $\alpha, \beta, \gamma \in \mathbb{R}_0^+$. By inserting the hyperplanes $t_0$ and $x_p(t_0)$ into Equation 6.1, we receive the hyperplane $\mathcal{H}_z$ given by:*

$$\mathcal{H}_z := 0 = (\alpha - d_p a)s_1 + (\beta - d_p b) * s_2 + d_p t + (\gamma - d_p c - z).$$

Depending on the operator $\sim$, the orientation of $\mathcal{H}_z$ might need to be inverted, so that its normal vector points to the set that fulfils the formula (line 4). The region is then cropped via half-space intersection to the positive half-space defined by $\mathcal{H}_z$ (line 5). The resulting Nef polyhedron $P$ is the subset of $R$, in which holds $x_p \sim z$.

Figure 6.2 shows a region $R$ in $\mathbb{R}^2$ along with the hyperplane $H_z$ that marks the threshold $z$ of some fluid property. The normal vector of $H_z$ points to the part $P$ of $R$, in which the property is satisfied.

A conjunction formula $\phi = \phi_1 \wedge \phi_2$ consists of the connection of two sub-formulae, which can be any type of atomic or compound formula. Contrary to the investigation of atomic formulae that do not depend on the satisfaction of other properties, model checking a conjunction requires to find the parts of a region that fulfil both $\phi_1$ and $\phi_2$. The steps to identify the satisfying subsets of a region are described by Algorithm 5.

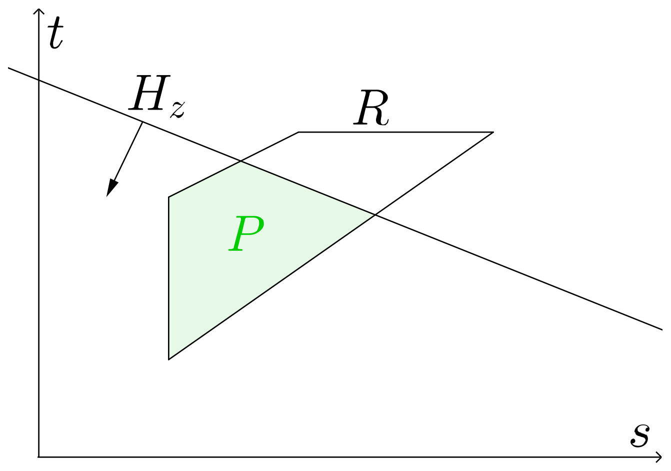


Figure 6.2: Checking a region in $\mathbb{R}^2$ for a fluid property.

**Algorithm 5** Procedure to identify the subset of a region fulfilling an STL conjunction property.

```
Require: $R$, a region, $\phi := \phi_1 \wedge \phi_2$, the fluid formula.
Ensure: Returns the subset $P$ of $R$ that fulfils $\phi$.
1: function CheckConjunction($R$,$\phi$)
2:     $P_1 \leftarrow$ Check<type>($R$,$\phi_1$)
3:     $P_2 \leftarrow$ Check<type>($R$,$\phi_2$)
4:     $P \leftarrow P_1 \cap P_2$
5:     return $P$
```

Looking at the lines 2 and 3 of the algorithm, one can observe that the post-order traversal of the formula's parse tree, which has been mentioned in Chapter 5, is caused at this point. First, the left-hand sub-formula $\phi_1$ is checked and then the right-hand sub-formula $\phi_2$ is checked. If the sub-formulae are compound formulae (conjunction, negation), too, the algorithm is called recursively on their sub-formulae and so on, until an atomic formula (true, discrete, fluid) ends the recursive call.

The result of the recursion are two Nef polyhedra $P_1$ and $P_2$, that represent the points of the region $R$ in which hold $\phi_1$ and $\phi_2$, respectively. Intersecting $P_1$ and $P_2$ yields the subset $P$ of $R$ that satisfies both $\phi_1$ and $\phi_2$, thereby fulfilling $\phi$ completely (line 4). As has been shown in Theorem 6.1, $P$ is a Nef polyhedron, as well.

The result of Algorithm 5 is visualised in Figure 6.3 for some arbitrary region $R$ in $\mathbb{R}^2$. The part $P$ of the displayed region $R$, which is covered by both areas $P_1$ and $P_2$, represents the subset of $R$ that satisfies the conjunction $\phi$.

Checking regions for other formula types, does not differ much from the algorithms presented here. Only the examination of a time bounded Until formula requires a completely different approach, which is discussed in the following section.

## 6.3 Model Checking Time Bounded Until Formulae

In contrast to the other types of STL formulae, the satisfaction of a time bounded Until depends on the evolution of the state in a specific time interval. The calculation of the satisfaction set $Sat(\phi, t)$ for an Until formula $\phi := \phi_1 \, \mathcal{U}^{[t_1,t_2]} \, \phi_2$ is executed in

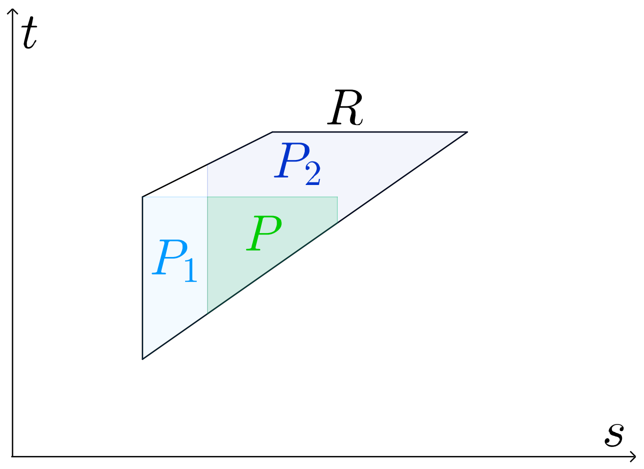


Figure 6.3: Checking a region in $\mathbb{R}^2$ for a conjunction property.

three steps. As has been mentioned in Chapter 5, all state sequences, that fulfil the left-hand sub-formula $\phi_1$ beginning from an initial time $t$ until finally the right-hand sub-formula $\phi_2$ is satisfied inside the interval $[t + t_1, t + t_2]$, have to be identified.

At first, the set $M \subseteq [0, t_{max}]^{n-1}$ of general transition firing times is determined, for which holds:

$$\forall \vec{s} \in M, t' \in [t, t + t_1] : \Gamma(\vec{s}, t') \models^{\vec{s},t} \phi_1.$$

The points $\vec{s}$ in the set fulfil the first condition for the satisfaction of an Until formula described in Section 5.2. The next step is to examine for the interval $[t + t_1, t + t_2]$, for which evolutions the left-hand formulae $\phi_1$ holds until eventually the right-hand formula $\phi_2$ holds. This results in another set $N \subseteq [0, t_{max}]^{n-1}$ of general transition firing times, and corresponds to the conditions 2 and 3 from Section 5.2.

All points $\vec{s}$ that are contained in both sets $M$ and $N$ then fulfil all three conditions and therefore satisfy the Until formula $\phi$. The intersection of $M$ and $N$ forms the satisfaction set $Sat(\phi, t)$, accordingly.

To be able to compute both $M$ and $N$, we first have to determine the subsets of all regions in the respective time interval, in which $\phi_1$ and $\phi_2$ hold. Aside from the lower and upper bound of the considered time interval, the computation of these subsets is identical for $M$ and $N$. Hence, we already introduce the required steps in Algorithm 6 for arbitrary time intervals $[l, u]$, before the further process is discussed separately for $M$ and $N$ in Section 6.3.1 and Section 6.3.2.

In contrast to the computation of satisfaction sets for atomic and compound formulae, we do not only have to consider regions at a fixed time. Instead, all regions inside the respective time interval $[l, u]$ have to be examined.

Thus, for each region $R$ it is checked, whether it lies between the lower bound $l$ and the upper bound $u$ of the interval $[l, u]$ (line 3). For this purpose, the time coordinates $t$ of a region's vertices $(\vec{s}, t)$ are compared to the interval limits. If $l \leq t \leq u$ holds for any vertex of the region, it is at least partially located inside the interval and is hence relevant for the further process.

Subsequently, it is determined which parts of the regions inside the interval $[l, u]$ satisfy the formulae $\phi_1$ and $\phi_2$ (line 4 and line 5). Depending on the formula's type, the call to CHECK<TYPE> might lead to multiple recursive calls, as has been mentioned in the previous section. Details on the implementation of this mechanism

**Algorithm 6** Function that identifies the $s$-$t$-satisfying subsets of the regions $\mathcal{R}$ in the time interval $[l, u]$ for an Until formula $\phi := \phi_1 \,\mathcal{U}^{[t_1,t_2]}\, \phi_2$.

```
Require: $\mathcal{R}$, the regions of the STD, $\phi_1$, the left-hand formula, $\phi_2$, the right-hand
    formula, $l$, the lower time bound, $u$ the upper time bound.
Ensure: Returns two Nef polyhedra $P_1$ and $P_2$ that cover all points of the STD in
    the time interval $[l, u]$ satisfying $\phi_1$ and $\phi_2$, respectively.
1: function CheckInterval($\mathcal{R}$, $\phi_1$, $\phi_2$, $l$, $u$)
2:     for all regions $R \in \mathcal{R}$ do
3:         if $R \in [l, u]$ then
4:             $P_1 \leftarrow P_1 \cup$ Check<type>($\phi_1$, $R$)
5:             $P_2 \leftarrow P_2 \cup$ Check<type>($\phi_2$, $R$)
6:     $P_1 \leftarrow$ LimitToInterval($P_1$, $l$, $u$)
7:     $P_2 \leftarrow$ LimitToInterval($P_2$, $l$, $u$)
8:     return $P_1$, $P_2$
```

are discussed in Chapter 7.

The resulting point sets are collected in $P_1$ and $P_2$. Following Corollary 6.2 both these sets are Nef polyhedra. The corollary can be applied, since $\phi_1$ and $\phi_2$ can only be atomic or compound properties. Nested Until formulae have been explicitly excluded in the syntax of the STL before (see Chapter 5).

$P_1$ and $P_2$ are limited to the interval $[l, u]$ by intersection with another Nef polyhedron $I := H_l^{(+)} \cap H_u^{(+)}$ where $H_l := 0 = t + l$ and $H_u := 0 = -t + u$ (line 5 and line 6). $I$ contains all points of the STD inside the interval $[l, u]$. Points outside of $I$ are not relevant for the further process and can thus be cut from the satisfying sets.

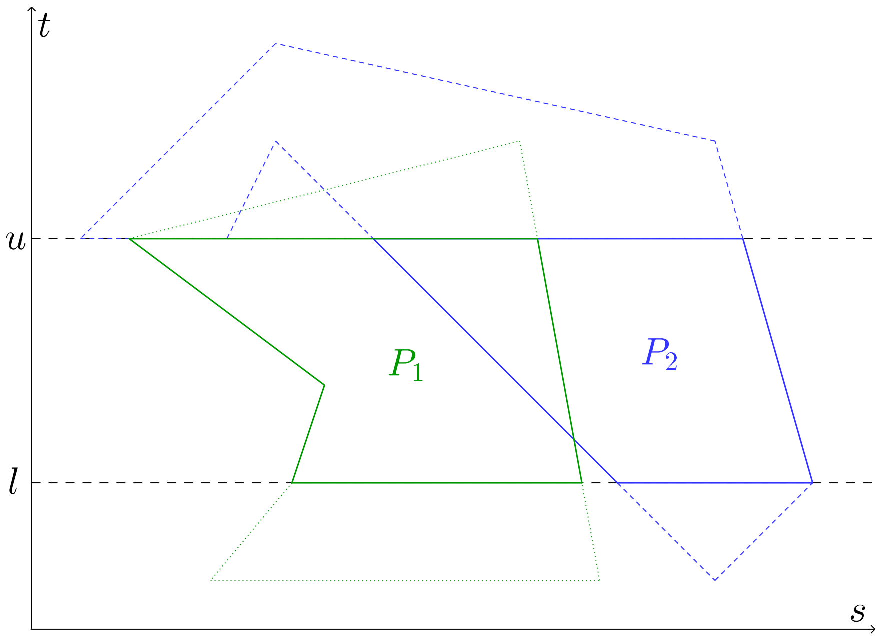


Figure 6.4: Example for the result of the procedure CheckInterval in $\mathbb{R}^2$.

Figure 6.4 presents an exemplary output of the described procedure for a two-dimensional STD. Since some regions, or rather parts of some regions in the example, satisfy both $\phi_1$ and $\phi_2$, the sets $P_1$ and $P_2$ overlap. The diagram is used in the following paragraphs to illustrate the introduced concepts.

The Nef polyhedra $P_1$ and $P_2$ now contain all points inside the interval $[l, u]$ that $\vec{s}$-$t$-satisfy the formulae $\phi_1$ and $\phi_2$, respectively. To gather the satisfaction set $Sat(\phi, t)$ from such Nef polyhedra, algorithms have been developed that are presented in the following sections.

### 6.3.1 Retrieval of Candidate Sets

By examining the first interval $[t, t + t_1]$ we want to reveal the system evolutions for which $\phi_1$ holds permanently inside the interval. These evolutions potentially fulfil the Until formula, which is why we call them *candidates*. The function ComputeCandidateSets which determines the candidate evolutions is described in Algorithm 7.

**Algorithm 7** The algorithm that determines the $t$-satisfying sets for $\phi_1$ in the interval $[t, t + t_1]$.

```
Require: $\mathcal{R}$, the regions of the STD, $\phi_1$, the left hand formula, $\phi_2$, the right hand
    formula, $t$, the checking time, $t_1$, the lower time bound of the Until formula.
Ensure: Returns the set of general transition firing times which $t$-satisfies $\phi_1$ in the
    time interval $[t, t + t_1]$.
1: function ComputeCandidateSets($\mathcal{R}$, $\phi_1$, $\phi_2$, $t$, $t_1$)
2:     $P_1, P_2 \leftarrow$ CheckInterval($\mathcal{R}$, $\phi_1$, $\phi_2$, $t$, $t + t_1$)
3:     $P \leftarrow P_1 \setminus P_2$
4:     return IdentifyFulfillingSystemEvolutions($t$,$t + t_1$, $P$)
```

We begin by determining the point sets of the STD in which the sub-formulae of the time bounded Until are satisfied (line 2). The function CheckInterval has been presented previously for arbitrary intervals in Algorithm 6.

To prevent that the Until formula is satisfied to soon, that is before the second interval, we have to exclude those points from the further process which fulfil $\phi_2$ in $[t, t + t_1]$. Thus, $P_1$ is limited to the volumes in which only $\phi_1$ and not $\phi_2$ holds by forming the difference between $P_1$ and $P_2$ (line 3).

Now, a system evolution with the general transition firing times $\vec{s}$ does not satisfy $\phi_1$ in the time interval if and only if there are points $(\vec{s}, t')$ with $t' \in [t, t + t_1]$ that lie outside the Nef polyhedron $P$. If the relevant part of such an evolution is represented as a line segment in the STD with source $(\vec{s}, t)$ and destination $(\vec{s}, t + t_1)$, it lies partially inside and partially outside of $P$. The change between points inside and points outside of $P$ (or vice versa) is marked for the segment by the intersection with a facet of $P$. Conversely, a segment, that fulfils $\phi_1$ at time $t$ and does not pierce through any facet of $P$, represents a system evolution, which constantly satisfies $\phi_1$ in $[t, t + t_1]$.

The idea is graphically presented in Figure 6.5 for a two-dimensional STD. Evolution $(a)$ passes through a facet of $P$ and does hence not fulfil $\phi_1$ in the interval $[t, t + t_1]$. However, the other evolution $(b)$ does not leave the area covered by $P$ inside of the interval. $(b)$ is therefore considered a candidate evolution.

From this observation the function IdentifyFulfillingSystemEvolutions which is presented in Algorithm 8 has been deduced. The procedure determines the

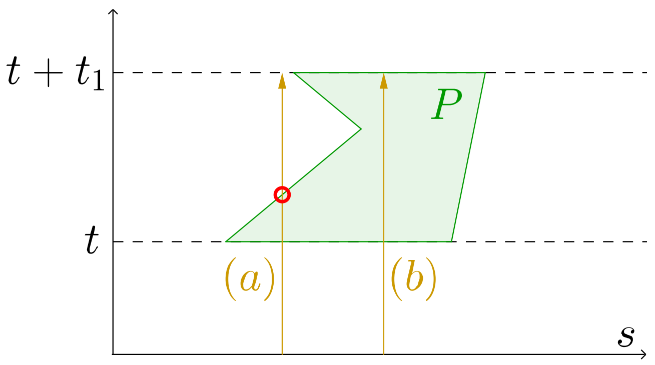


Figure 6.5: An evolution that does not satisfy a formula in the complete interval $[t, t + t_1]$ (a) vs. an evolution that satisfies the formula throughout the interval (b) in an STD in $\mathbb{R}^2$.

general transition firing times of all evolutions that are completely enclosed by $P$. Since the algorithm is reused during the model checking of the second interval, it is defined for arbitrary intervals $[l, u]$.

**Algorithm 8** Procedure to identify the system evolutions that fulfil a formula $\phi$ in a time interval $[l, u]$.

```
Require: $l$, the lower bound for the interval, $u$, the upper bound for the interval,
    $P$, the Nef polyhedron representing the satisfaction set of $\phi$ in $[l, u]$.
Ensure: Returns the set of general transition firing times that identify the fulfilling
    system evolutions.
1: function IdentifyFulfillingSystemEvolutions($l$,$u$,$P$)
2:     $M \leftarrow$ IntersectionToSubspace($P$, $H_l$)
3:     for all facets $f \in P$ do
4:         if $f \neq H_l \wedge f \neq H_u$ then
5:             $M \leftarrow M \setminus$ ProjectToSubspace($f$)
6:     return $M$
```

By intersecting the input polyhedron with the hyperplane $H_l := t = l$ and subsequently projecting the result to $\mathbb{R}^{n-1}$, a set $M$ is generated, that contains all firing times of the general transitions, for which $\phi$ holds at time $l$ (line 2).

In Figure 6.6, $M$ is an interval in $\mathbb{R}$. It results from the intersection of the Nef polyhedron $P$ with the hyperplane $H_l$, which is not explicitly depicted but corresponds to the lower dashed line. In the next step, $M$ is restricted using the facets of $P$ to keep only those firing times, which correspond to evolutions that fulfil $\phi_1$ in the complete interval.

A facet whose hyperplane is equal to the borders $l$ or $u$ of the interval is excluded from this process (line 4). The projection to $\mathbb{R}^{n-1}$ of a facet $f$, that lies inside the interval $(l, u)$, marks the firing times of those evolutions, which would intersect with $f$. Hence, the projection of such a facet is subtracted from the set $M$ (line 5). After having processed all facets of $P$, $M$ is reduced to the firing times, for which the corresponding system evolutions do not intersect with facets of $P$.

Figure 6.7 shows the result of the method applied to our running example. The facets of $P$ between $l$ and $u$ have been used to restrict $M$ to the marked interval.

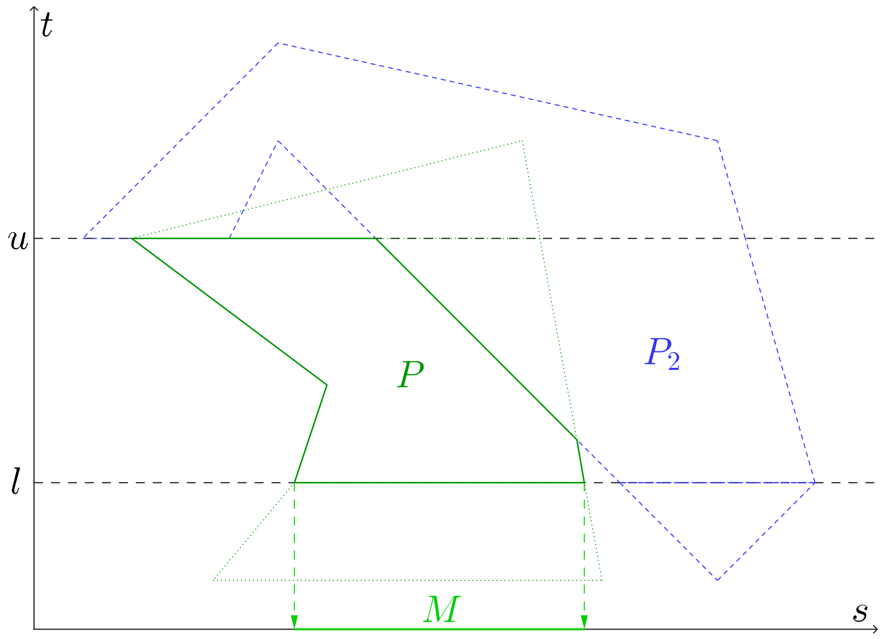


Figure 6.6: Identification of evolutions that satisfy $\phi_1$ at a time $l$ by hyperplane intersection in $\mathbb{R}^2$ (Algorithm 8, line 2).

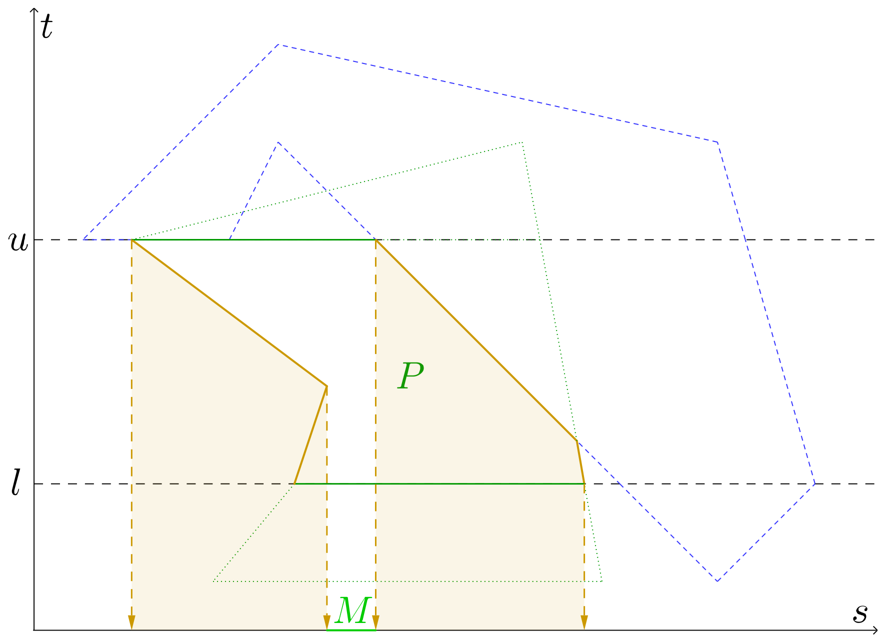


Figure 6.7: Restriction of satisfying evolutions using the facets of $P$ (Algorithm 8, line 5).

Only the remaining elements in $M$ represent evolutions that constantly fulfil $\phi$ in $[l, u]$.

If $M$ is empty after IDENTIFYFULFILLINGSYSTEMEVOLUTIONS returns from the call in COMPUTECANDIDATESETS, there is no system evolution that constantly fulfils $\phi_1$ in the first interval $[t, t + t_1]$. The underlying Petri net can therefore not satisfy the Until formula, so that the process can be interrupted at that point. Otherwise, the second interval is examined following the approach described in the next section.

## 6.3.2 Retrieval of Satisfaction Sets

The challenge for the analysis of the interval $[t + t_1, t + t_2]$ is based on the fact that the observation of a system evolution has to be cancelled as soon as a state is reached, in which $\phi_2$ holds. So, this time we do not look at a fixed interval. Instead, the points in time have to be determined, at which $\phi_2$ is first satisfied.

The procedure for handling the second interval appears in detail in Algorithm 9.

**Algorithm 9** The procedure that determines the $t$-satisfying sets for a time bounded Until formula $\phi$ in the interval $[t + t_1, t + t_2]$.

**Require:** $\mathcal{R}$, the regions of the STD, $\phi_1$, the left hand formula, $\phi_2$, the right hand formula, $t$, the checking time, $t_1$, the lower time bound of the Until formula, $t_2$, the upper time bound of the Until formula.

**Ensure:** Returns the set of general transition firing times which $t$-satisfy the Until formula $\phi_1 \mathcal{U}^{[0,t_2]} \phi_2$ at time $t + t_1$.

```
1: function ComputeSatisfyingSets($\mathcal{R}$, $\phi_1$, $\phi_2$, $t$, $t_1$, $t_2$)
2:     $P_1, P_2 \leftarrow$ CheckInterval($\mathcal{R}$, $\phi_1$, $\phi_2$, $t + t_1$, $t + t_2$)
3:     $P_1 \leftarrow P_1 \setminus P_2.interior()$
4:     $B \leftarrow P_1 \cap P_2.boundary()$
5:     for all facets $f \in B$ do
6:         $f_{projected} \leftarrow$ ProjectToSubspace($f$)
7:         $Temp \leftarrow P_1 \cap \{(f_1, \ldots, f_{n-1}, t') | (f_1, \ldots, f_{n-1}) \in f_{projected}, t' \in [t + t_1, t + t_2]\}$
8:         $N \leftarrow N \cup$ IdentifyFulfillingSystemEvolutions($t + t_1$, $f$, $Temp$)
9:     return $N \cup$ IntersectionToSubspace($P_2$, $H_{t+t_1}$)
```

In a manner similar to Algorithm 7 the first step is to compute the subsets of the regions in the interval $[t + t_1, t + t_2]$, which satisfy the formulae $\phi_1$ and $\phi_2$ using Algorithm 6 (line 2).

The next task is to determine the points, at which an evolution switches from $P_1$ to $P_2$, that is the facets of $P_2$ which intersect with $P_1$ have to be found. Points that are shared between $P_1$ and $P_2$ do not need to be considered, which is why they are excluded from $P_1$ (line 3). By intersecting the limited $P_1$ with the boundary of $P_2$ (line 4), we receive the desired point set. As has been mentioned in Section 3.2, the interior and the border of a Nef polyhedron are again Nef polyhedra, so that the result set $B$ of the operation is a Nef polyhedron, too.

For each point on a facet of $B$ then both $\phi_1$ and $\phi_2$ hold. A system evolution with general transition firing times $\vec{s}$, which crosses such a facet at some time $\tau \in [t+t_1, t+t_2]$, potentially fulfils the Until formula. If, in addition, all points of the evolution in the interval $[t, \tau)$ satisfy the formula $\phi_1$, the firing times $\vec{s}$ belong to the set $N$. For this purpose, we can reuse the function IDENTIFYFULFILLINGSYSTEMEVOLUTIONS (Algorithm 8), which has been introduced for the computation of the set $M$ in the first interval (line 8).

Each facet $f$ of $B$ is visited separately to identify the fulfilling system evolutions. The area $f_{projected}$, that we receive by projecting $f$ to $\mathbb{R}^{n-1}$ (line 6), defines the subset of $P_1$, which has to be processed for the facet. We use $f_{projected}$ as a *basis*, to create a so-called *prism*, which is subsequently intersected with $P_1$ (line 7).

**Definition 6.3** ([53, Def. 13-8]). *A **prism** is a polyhedron, whose faces consist of two parallel and congruent polygons, called bases, and the parallelograms, called lateral faces, formed by connecting pairs of corresponding vertices of the parallel polygons.*

Definition 6.3 describes prisms in $\mathbb{R}^3$, although the concept is applicable to arbitrary dimensions by replacing polyhedra with $n$-polytopes and polygons with $(n-1)$-polytopes in the definition (cf [33]). Since the prism is a Nef polyhedron, the intersection $Temp$ is a Nef polyhedron, too.

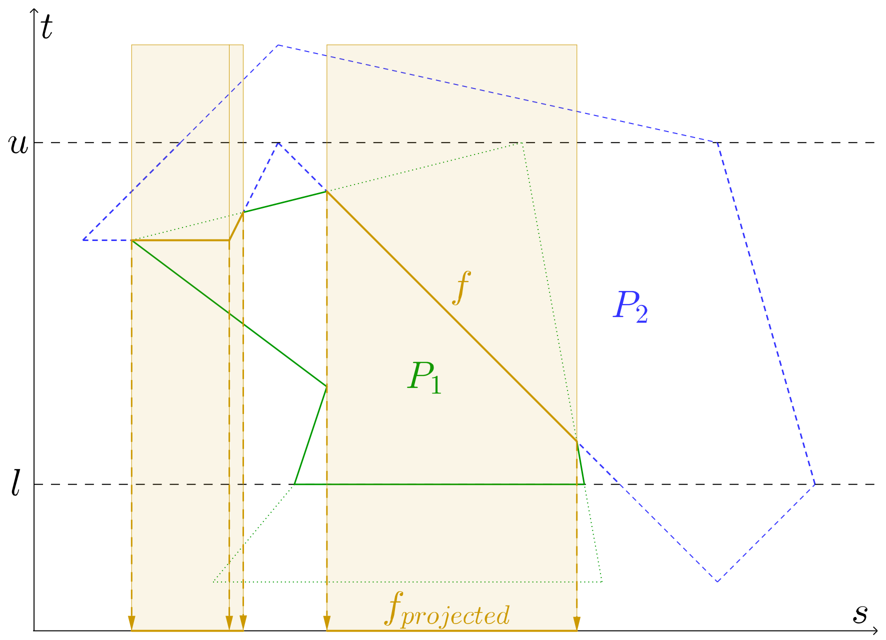


Figure 6.8: Limiting the polyhedron $P_1$ with the prisms generated from the facets of $B$ in $\mathbb{R}^2$ (Algorithm 9, line 6 to 7).

We continue to illustrate the process with our running example in Figure 6.8. It depicts the facets $f$ of $B$ which are projected to $\mathbb{R}^{n-1}$, and the prisms that are generated for each of the projected facets. In $\mathbb{R}^2$ the prisms are simple rectangles.

The intersections $Temp$ of the individual prisms and $P_1$ are passed successively to the function IDENTIFYFULFILLINGSYSTEMEVOLUTIONS. Note, that instead of a time point the facet $f$ is passed as the upper boundary. The hyperplane $\mathcal{H}_u$, which is used as the upper limit inside Algorithm 8, is however just replaced by the hyperplane of the facet. Otherwise, nothing else has to be changed. The function call determines the projected sets in $\mathbb{R}^{n-1}$ from the intersection of $H_{t+t_1}$ and $Temp$. Afterwards they are limited with the projection of the facets of $Temp$, which are located below $f$. The result is a set of general transition firing times, which fulfil the second and third condition for the satisfaction of the Until formula.

After all facets of $B$ have been processed in this manner, $N$ does not yet contain all these general transition firing times, though: So far, the system evolutions which satisfy $\phi_2$ immediately when entering the interval $[t + t_1, t + t_2]$ have been ignored. They can however be easily computed by the intersection of $P_2$ and the hyperplane $H_{t+t_1}$ and subsequent projection of the result to $\mathbb{R}^2$ (line 9). The union of $N$ with the result of this last operation now corresponds to the set of all general transition firing times, which fulfil the second and third condition in the interval $[t+t_1, t+t_2]$.

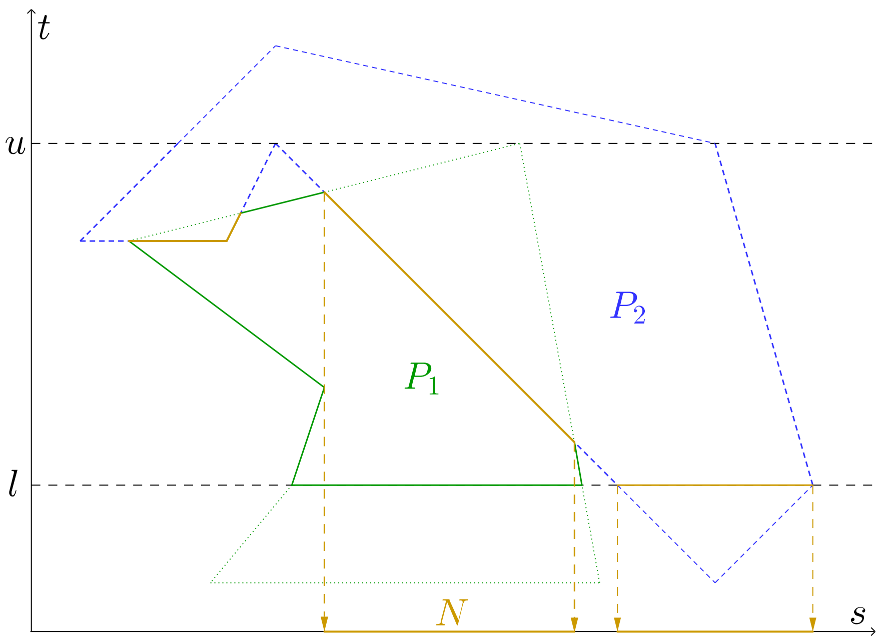


Figure 6.9: Result of the checking process according to Algorithm 9 for the two-dimensional example.

Figure 6.9 shows the final result for the running example. The left two facets of $B$ can not be reached without leaving $P_1$, which is why only the marked interval is part of $N$. In addition, the interval covered by the intersection of $H_l$ with $P_2$ is part of $N$, since these evolutions immediately satisfy $\phi_2$ when entering $[t + t_1, t + t_2]$.

If we intersect the sets $M$ and $N$ which are computed by Algorithm 7 and Algorithm 9, we receive the satisfaction set $Sat(\phi, t)$ for the time bounded Until formula. The evolutions identified by the elements in $M$ fulfil the first condition for the satisfaction of the Until formula, while the points in $N$ fulfil the second and third condition. Hence, it holds that $Sat(\phi, t) = M \cap N$ for an Until formula $\phi$, which is checked at time $t$.

## 6.4 Deconditioning General Transition Firing Times

The formation of the satisfaction set $Sat(\phi, t)$ for an STL formula $\phi$ at time $t$ has been discussed in the previous sections. The set contains all general transitions firing times, for which the system fulfils the formula. Since we want to predict the general behaviour of the HPnG, the total probability for the satisfaction of the formula has to be deduced from the elements in $Sat(\phi, t)$.

During the construction of the STD in Chapter 4 the probabilistic properties of the general transitions have been encoded in the variable vector $\vec{s}$, whereby we could represent the various possible system evolutions in a single $n$-dimensional structure. For this reason, there has been no need to consider the probabilistic nature of HPnGs in the previous discussions. To be able to make a statement concerning characteristics like the reliability of a system, however, the probability for the values in $Sat(\phi, t)$ has to be determined.

In Chapter 5 the probability operator $P_{\triangleright p}(\phi, t)$ has been introduced to compare the probability $Prob(\phi, t)$ for the satisfaction of an STL formula $\phi$ at time $t$ with a threshold $p$. $Prob(\phi, t)$ has been presented for an arbitrary number of general transition firings as the multiple integral over the domain of $Sat(\phi, t)$. Higher-dimensional integration, however, requires highly complex numerical techniques like *quasi-Monte Carlo methods* (cf. [13]), for which an introduction would exceed the scope of this work. Hence, we limit our explanations to the integration of two-dimensional sets, which can in certain cases be reduced to simple one-dimensional integration. In addition, the implementation, which has been created along with this thesis, handles HPnGs with two general transitions. It therefore uses the exact same approach, which is described in the following paragraphs.

In the special case of two general transition firings, the function $Prob(\phi, t)$ from Definition 5.9 of the probability operator has the following form:

$$Prob(\phi, t) := \iint_{Sat(\phi,t)} g_1(s_1) * g_2(s_2)\ ds_2 ds_1,$$

and the satisfaction set $Sat(\phi, t)$ is a set of two-dimensional *polygons with holes* (cf. [24]).

To determine the probability that the formula $\phi$ is satisfied, the partial probabilities for each polygon with holes are computed independently. For a single polygon with holes, the probability of the surrounding polygon is computed first. Then, the probability of the sets covered by each hole are determined and subtracted from the probability of the surrounding polygon. The sum of the probabilities for all polygons with holes in $Sat(\phi, t)$ corresponds to the value of $Prob(\phi, t)$.

The complicated part about the calculation of multiple integrals is the dependence of the inner integrals on the outer integrals. Since this dependence can be easily resolved for the integration of triangular shapes in two dimensions, the arbitrary polygons in the satisfaction set are split into triangles. This is called *triangulation.* By using triangular shapes, we can exploit that the limits of the $y$-coordinates in a triangle with one edge, that is perpendicular to the x-axis, can be represented by linear equations of the $x$-coordinates as shown in Figure 6.10.

The values of $y$ in the triangle are limited by line segments, whereby it holds that: $a_1x + b_1 \leq y \leq a_2x + b_2$. Given two functions $f$ and $g$, the double integral for the triangle reduces to:

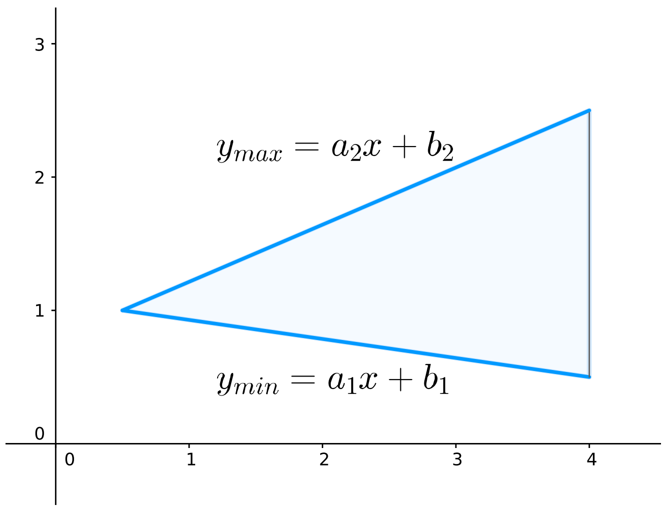


Figure 6.10: A triangular shape, whose $y$-coordinates are limited by line segments depending on $x$.

$$
\begin{aligned}
&\int_{x_{min}}^{x_{max}} \int_{y_{min}}^{y_{max}} f(x) * g(y)\ dydx \\
&= \int_{x_{min}}^{x_{max}} \int_{a_1x+b_1}^{a_2x+b_2} f(x) * g(y)\ dydx \\
&= \int_{x_{min}}^{x_{max}} f(x) * (G(a_2 * x + b_2) - G(a_1 * x + b_1))\ dx.
\end{aligned}
$$

$G$ is the antiderivative of $g$. The simple integral for the interval $[x_{min}, x_{max}]$ can be calculated using well-known numerical integration methods like Gaussian quadrature or Monte Carlo integration (cf. [55]). If the functions $f$ and $g$ are replaced by the probability density functions $g_1$ and $g_2$ of the random variables for the general transition firing times, we obtain the probability of the point set covered by the triangular shape.

Adding up the probabilities calculated with this approach from all triangles of the fragmented polygons in $Sat(\phi, t)$ yields the probability, that the formula $\phi$ is satisfied at time $t$. If the probability matches the threshold $p$ defined by some probability operator $P_{\triangleright p}(\phi, t)$, the operator is satisfied, which means that the HPnG fulfils the property specified by the STL formula $\phi$ with an adequate probability.

# Chapter 7

# Implementation

A C++ implementation of the algorithms, which have been introduced in the previous chapter, has been used to perform the case study, which is presented in the next chapter. Here, we look briefly into the structure of the implementation and emphasise some of its features.

As has been indicated in the previous discussions, the implementation is limited to the processing of HPnGs with two general one-shot transitions. The STD, which is a central part of the model checking, is hence generated in a subset of $\mathbb{R}^3$. The program uses several three-dimensional geometric structures and applies operations to them, which are both implemented by the *Computational Geometry Algorithms Library* (CGAL, cf. [16]). CGAL is a collection of separate C++-libraries, that provide various geometric structures and algorithms in two-, three-, and sometimes higher-dimensional space.

At the beginning of the chapter, a rough overview of the implementation is given (Section 7.1). Afterwards, the implementation of STL formulae and a problem arising from the chosen approach are explained in detail in Section 7.2. A discussion on the components of CGAL, that are used for the implementation, and a brief comparison to other libraries, which offer similar polyhedron data structures and algorithms, concludes the chapter with Section 7.3.

## 7.1 Structure of the Implementation

The program implements the process, which is depicted in Figure 6.1 from the previous chapter. It is controlled via configuration files, that determine the HPnG, the STL formula, and other parameters relevant for the checking routine.

The configuration files are handled with the help of an external library called *libconfig* (cf. [46]). An HPnG is described by an XML document, that defines the places, transitions, and arcs, and their relation to each other. By specifying the path to the XML document in the configuration file, the program can interpret and process the HPnG model, thereby generating a processable C++ object. The EBNF syntax of the STL has been adapted to allow for a specification of STL formulae with plain ASCII symbols. Hence, STL formulae are in contrast provided as simple strings. A parser for the modified syntax reads in the formula, checks it for its syntactic validity, and subsequently creates a C++ object, which represents the STL formula during the following steps.

In addition to these two parameters, the maximal execution time $t_{max}$ of the

HPnG, the time $t$, at which the formula shall be examined, and the distribution functions $g_1$ and $g_2$ for both of the general transitions have to be recorded in the configuration file. To facilitate the double integration as explained in Section 6.4, one of the distribution functions has to be passed as a probability density function, while the other must be provided as a cumulative distribution function. Currently, the implementation supports the following continuous distribution functions: normal distribution, folded normal distribution, log-normal distribution, uniform distribution, exponential distribution, Weibull distribution.

```
# Path to the HPnG model file:
model = "~/models/hpng.xml"
# The maximal observed system time:
max_time = 120.0
# The STL formula to verify:
formula = "U(d(1,1,>=), f(9,1000.0,>=), 20.0, 80.0)"
# The time at which the formula should hold:
time_to_check = 10.0
# The distributions for the general transitions G1 and G2:
distributions =
(
  {
    function = "normalCDF";
    parameters = (50.0, 4.0);
  },
  {
    function = "normalPDF";
    parameters = (40.0, 6.0);
  }
)
```

Listing 7.1: An example configuration file that controls the behaviour of the model checker.

Listing 7.1 shows an example configuration file for the model checker. The individual fields correspond to the parameters described above. In the exemplary case, the HPnG, which is observed in the time interval $[0, 120]$, is specified by an XML document named *hpng.xml.* The formula

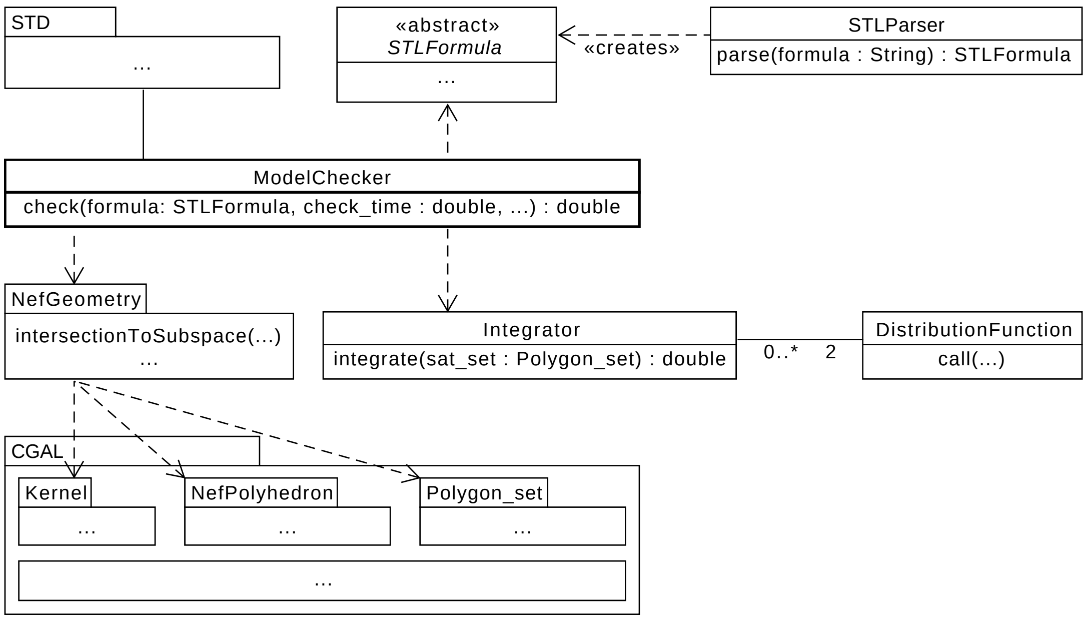
$$\phi := m_{p_1} \geq 1\, \mathcal{U}^{[20,80]}\, x_{p_9} \geq 1000$$

should be model checked at time $t = 10$.

With the adapted syntax, an Until formula is specified by an upper-case U which is followed by four parameters in round brackets. The first two parameters are the left- and right-hand sub-formula, while the third and fourth parameter represent the lower and upper time bounds $t_1$ and $t_2$. Atomic formulae are defined by three arguments in the following order: the identifier of the concerned place, the desired threshold, and the comparison operator. Discrete properties are indicated by a lower-case d, and fluid properties are indicated by a lower-case f. Translating the formula string in the example configuration file to the actual STL syntax from Chapter 5 results hence in the formula $\phi$.

The last entry of the configuration file sets the probability distributions of the two general transitions in the model. Depending on the type of the specified distribution function, the *parameters* field can have different meanings. Here, the first parameter represents the mean of the normal distribution, whereas the second parameter defines its standard deviation.

After having set up the program with the described parameters, the model checking can begin. At first, the STD is generated from the HPnG for the time interval $[0, t_{max}]$. This step is already covered by the implementation from [29], which has been mentioned before. Afterwards the model checker examines the formula $\phi$ at time $t$, by applying the algorithms introduced in Chapter 6 to the STD. Some details concerning the implementation of this process are described in Section 7.2 and Section 7.3. Using the resulting satisfaction set, the probability for the satisfaction of $\phi$ is computed next. As has been emphasised in Section 6.4, the polygons in the satisfaction set are first triangulated for this purpose, and then the probability is computed by integrating via Gaussian quadrature (cf. [55]) over the generated triangles. The numerical integration itself is performed by an external library (see [36]).



STD
...
«abstract»
STLFormula
...
«creates»
STLParser
parse(formula : String) : STLFormula
ModelChecker
check(formula: STLFormula, check_time : double, ...) : double
NefGeometry
intersectionToSubspace(...)
...
Integrator
integrate(sat_set : Polygon_set) : double
0..*
2
DistributionFunction
call(...)
CGAL
Kernel
...
NefPolyhedron
...
Polygon_set
...
...

Figure 7.1: The (simplified) UML class diagram of the model checker implementation.

The most important parts of the implementation are depicted in Figure 7.1 as an UML class diagram (cf. [50]). The central element is the `ModelChecker` class, which provides a single function `check()` for the user, that utilises the other pictured components.

To create an instance of the class, an `STD` object is passed to its constructor. We represent the `STD` class in the figure as a package, to emphasise, that the implementation from [29] is a more complex construct than a simple data structure. It consists of the read-in mechanism for the HPnG model, the STD creation algorithms, and on top of that the STD class. However, since the STD implementation is not part of this work, we do not show all these components in the class diagram, although the `STD` data structure plays an important part in the model checking process.

The `check()` method accepts an `STLFormula` object, the time to check the formula, and a few other parameters to operate on the stored `STD`. The abstract `STLFormula` class is a unified interface for the concrete STL formula types, which are not contained in the figure. A more detailed image and further discussions on the implementation of STL formulae can be found in Section 7.2.

Using the given `STD` and `STLFormula`, model checking is performed according to the algorithms described in Chapter 6. The required geometric operations on the regions, that are used by the `ModelChecker` during these steps, are defined in a separate namespace called `NefGeometry`. Functions in the namespace utilise the data structures and algorithms offered by packages of CGAL, which are explained more extensively in Section 7.3. The three named packages in Figure 7.1 are the ones, that are mainly used by the implementation.

After the satisfaction set of the `STLFormula` object has been determined, the cumulative probability of the covered general transition firing times is computed with the help of the `Integrator` class. The distribution functions of both general transitions are each wrapped in a `DistributionFunction` object, that is passed to the `Integrator`. By calling the function `integrate()` on the satisfaction set, its probability is computed according to the steps explained in Section 6.4.

When the `integrate()` method is finished, the `ModelChecker` returns the calculated probability. The comparison to a threshold, that is the last step in the processing of a probability operator, must be currently performed manually by the user.

## 7.2 Implementation of STL Model Checking

The interface of the `ModelChecker` class offers a single function `check()` to pass an STL formula of arbitrary type to the program. In addition, the C++ objects representing compound formulae and Until formulae need to be able to hold arbitrary formula objects as sub-formulae. For this purpose, all STL formula types derive from a common abstract base class *STLFormula*. While it is convenient to offer a uniform interface to the user, this simultaneously leads to challenges during the checking process, which are explained in the following paragraphs.

Inside the model checker, the actual formula types have to be determined, since the required routines differ for the various STL formulae as has been described extensively in the previous chapter. To understand why this is a problem, we have to look at both the implementation of the STL formula types and the internal mechanism of C++, that defines how a function call is executed. The solution to the conflict lies in a special design pattern which is a modification of the well-known *visitor pattern* (cf. [25]).

### 7.2.1 Static vs. Dynamic Dispatch in C++

Classes, that share certain functions or traits, can be summarised in C++ using an interface, that describes the common properties of the similar classes. Such an interface is called a *base class*, from which similar classes are *derived*.

Figure 7.2 depicts an UML diagram of a base class `Shape` and two derived classes `Square` and `Circle`. The base class has a member function `area()`, which computes the area of the respective geometric shape. Obviously, the area computation for

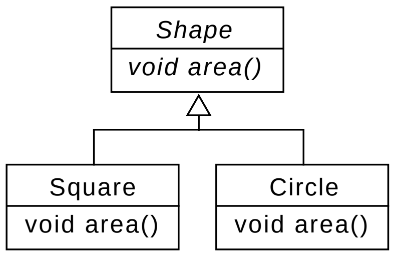


Figure 7.2: A simple inheritance example representing geometric shapes.

circles and squares differ, which is why both derived classes have to implement their own version of the `area()` function.

An instance of a derived class can be addressed via a reference of the base class type. In the above example, we can reference both `Squares` and `Circles` using a `Shape` reference, since both classes are derived from `Shape` and implement the function `area()`. If a method is called on a derived object using the base class reference, the appropriate implementation of the method has to be chosen depending on the type of the referenced object.

The process of selecting the appropriate procedure is called *dispatch* (cf. [47]). The most common approach is the *static dispatch*, which determines the function to call at compile time. If in contrast the required information to select the procedure becomes only available at runtime, the dispatch must take place at runtime, as well, and is hence called *dynamic*.

The member functions of a class are uniquely identified by their *signature*, that is its name and parameter types (cf. [41]). If the compiler finds such an identifying combination in the source code, it can link the calling point to the memory address of the function's body. Assume for example the (simplified) C++-code for the class `Square` from Figure 7.2 in Listing 7.2.

```
 1 class Square {
 2   double a;
 3   public:
 4     void area() {
 5       std::cout << a * a << std::endl;
 6     }
 7 }
 8
 9 int main() {
10   Square s(1);
11   s.area();
12 }
```

Listing 7.2: Example for static dispatch in C++.

The call to the function `area()` in line 11 is a static dispatch. All information required to identify the function, which should be called, are known at compile time. The compiler reads the object `s` of class `Square`, which offers a single function `area()` without parameters. The combination of class `Square`, function name `area()`, and no parameter types identifies the instructions in line 5, which prints 1 in this case.

Hence, the call is completely determined by the information given in the source code.

In contrast, a dynamic dispatch is required if we have to deal with virtual methods and inheritance, similar to the initial shape-example. Declaring a method of a class as `virtual` in C++ allows inheriting classes to override the virtual method with their own implementation. If they do not override the method, the routine of the base class is used, instead. To force a derived class to override a virtual method, it can be declared as *pure virtual* (Listing 7.3, line 3). A pure virtual method can not be implemented in the base class, which therefore automatically becomes an abstract class (cf. [41]). The function `area()` of the `Shape` class in Figure 7.2 should for example be declared as pure virtual, since the area of an unspecified shape can not be computed.

The selection of the correct function is usually managed in C++ with a so-called *virtual method table* (vtable). Each instance of a class, that incorporates virtual methods, is associated with such a vtable. The table contains references to the functions that should be called for the object at runtime.

| `Circle` vtable | | `Square` vtable | |
|---|---|---|---|
| Function name | Memory address | Function name | Memory address |
| `area()` | $0x1234$ | `area()` | $0x9876$ |

Table 7.1: Exemplary virtual method table (vtable) for instances of the `Circle` class and of the `Square` class from Figure 7.2.

Table 7.1 shows exemplary vtables for both a `Circle` and a `Square` object. Only the entries in the columns *Memory address* differ, which identify the memory locations of the respective `area()` method implementations.

At the creation of any derived class object in C++, a vtable similar to the ones presented in Table 7.1 is created, which references the function bodies of the derived class. When the object is then addressed via a base class reference, the vtable is not overridden. Thus, a call to a virtual method on the base class reference still finds a link to the derived class implementation in the vtable. An example, in which a dynamic dispatch is required, is shown in Listing 7.3.

```
 1 class Shape {
 2   public:
 3     virtual void area() = 0;
 4 }
 5 
 6 class Square : public Shape {
 7   double a;
 8   public:
 9     void area() { std::cout << a * a << std::endl; }
10 }
11 
12 class Circle : public Shape {
13   double r;
14   public:
15     void area() { std::cout << π * r * r << std::endl; }
16 }
17
18 void dispatch(Shape& x) {
19   return x.area();
20 }
21
22 int main() {
23   Square s(1);
24   Circle c(1);
25
26   dispatch(s);
27   dispatch(c);
28 }
```

Listing 7.3: Example for dynamic dispatch in C++.

The code snippet defines the three classes from Figure 7.2. The class `Shape` defines the pure virtual function `area()` (line 3) that is implemented by both of the derived classes. One object of either derived type is created (line 23 and line 24) and passed to the function `dispatch()` (line 26 and line 27). The function expects a reference of type `Shape`, which is why the actual type of the passed object is unknown in the scope of `dispatch()`. However, since the vtable of the particular parameter has been created for the proper type, the calls to `area()` inside of `dispatch()` yield the output 1 and 3.14159... in the correct order.

### 7.2.2 Double Dispatch in STL Model Checking

STL formulae are implemented according to the structure described by the UML diagram in Figure 7.3. It is a simplified version of the actual implementation leaving out a few details, which nevertheless suffices to show how STL formulae are represented.

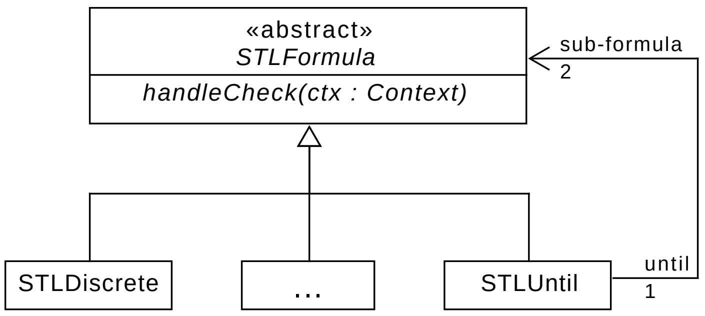


Figure 7.3: A reduced class diagram of the STL formula inheritance hierarchy.

The base class `STLFormula` provides a pure virtual function `handleCheck()`, and is hence an abstract class. The derived classes represent the actual formula types. The circular association of the `STLUntil` class with the base class `STLFormula` emphasises the tree structure of both compound and Until STL formulae.

As has been mentioned before, the model checker accepts an `STLFormula` reference, which represents an STL formula of arbitrary type, via its main function `check()`. During the course of the function, the input `STLFormula` must be passed to the routine `check<type>()`, which has been described specifically for fluid and conjunction formulae in Chapter 6 by Algorithm 4 and Algorithm 5, respectively. The `check<type>()` function determines the parts of the regions, which satisfy the STL formula.

```cpp
void check(STLformula& formula, ...) {
  ...
  check<type>(formula);
  ...
}
```

Listing 7.4: Source of conflict in the implementation of the model checking process.

The crucial part of the `check()` method is highlighted in Listing 7.4. The dots indicate, that the function consists of many more instructions, which, however, are not of interest here. The problem with the shown excerpt is, that the actual type of the formula object is unknown in the scope of the method `check()` at runtime, because it works with a reference of the abstract `STLFormula` class. Therefore, it can neither be decided at compile time nor at runtime, which specific `check<type>` routine has to be called for the formula object.

A first solution, that comes to mind, could be to *overload* the function `check()` to accept the individual `STL<type>` classes. Offering separate implementations for every concrete formula type instead of providing a single interface, that only uses the abstract `STLFormula` reference, solves the problem for atomic formulae, which do not hold sub-formulae.

```cpp
void check(STL<type>& formula, ...) {
  ...
  check<type>(formula);
  ...
}
```

Listing 7.5: Overloading the `check()` routine in the model checker.

In Listing 7.5 the formula to check is passed to the model checker with its concrete type `STL<type>`. Consequently, the appropriate `check<type>()` function can be selected in the scope of `check()`. If, for example, an `STLFluid` reference is passed to the function `check(STLFluid& formula, ...)`, the routine described by Algorithm 4 can be directly called from within the scope of `check()`. However, compound and Until formulae contain references to the abstract base class as it is indicated in Figure 7.3, since they comprise one or two sub-formulae of arbitrary type. So, we must be able to apply the proper `check<type>()` function to the sub-formulae of compound or Until objects as well, which leaves us again confronted with the initial problem. In addition, this approach requires us to implement the routine `check()` for every single formula type, thereby creating unnecessary overhead.

The solution lies in the dispatch mechanism of C++, which has been briefly introduced in the previous section. The pure virtual function `handleCheck()` of the abstract `STLFormula` class must be implemented by the derived concrete STL types. For each formula type, it consists of the same single line, which is shown in Listing 7.6.

```
void handleCheck(Context& ctx) {
  model_checker.checkFormula(*this, ctx);
}
```

Listing 7.6: Implementation of the `handleCheck()` method in concrete STL type classes.

If the method is called on an `STLFormula` reference, a dynamic dispatch takes place and redirects the call via the vtable to the implementation of the derived class. In the scope of the respective `handleCheck()` implementation, the actual type of the formula object (represented by `this`) is known. The identified object is then passed back via the function `checkFormula()` to the model checker, which can now act according to the actual type of the formula. The method `checkFormula()` uses an overloaded interface in the model checker class to keep the implementations of `handleCheck()` in the STL type classes as uniform as possible, although individual `check<type>()` functions could be equally implemented.

```
void check(STLFormula& formula, ...) {
  ...
  formula.handleCheck(ctx);
  ...
}
```

Listing 7.7: Conflict solution via dynamic dispatch in the API function `check()` of the model checker.

In Listing 7.7 the adapted snippet of the `check()` function is represented. Instead of trying to directly pass the formula of unknown type to some checking routine, the dynamic dispatch via `handleCheck()` is used to first identify the formula's type, and then pass the unmasked formula back to the model checker from inside the `handleCheck()` method. The steps are illustrated by a sequence diagram that is depicted in Figure 7.4.

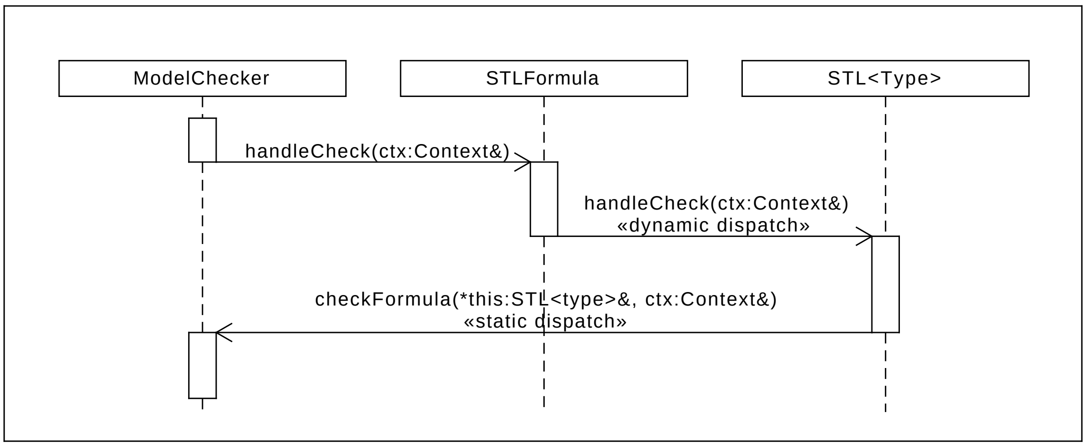

ModelChecker
STLFormula
STL<Type>
handleCheck(ctx:Context&)
handleCheck(ctx:Context&)
«dynamic dispatch»
checkFormula(*this:STL<type>&, ctx:Context&)
«static dispatch»

Figure 7.4: Sequence diagram of the calls which lead to the identification of the STL formula type in the checking process.

The approach adopts a modified version of the visitor pattern, which has been proposed in [25]. In the terminology of the pattern, the model checker is the visitor and the formula is the acceptor. The purpose of this approach is to enable a double dispatch mechanism in a language like C++, which only supports single dispatch. The sequence diagram in Figure 7.4 reveals, how this behaviour can be simulated by a concatenation of a dynamic dispatch and a static dispatch.

Starting from the `check()` function in the `ModelChecker`, the method `handleCheck()` is called on the formula of arbitrary type according to Listing 7.7. The call to the pure virtual function `handleCheck()` of the `STLFormula` class invokes a vtable lookup, that is a dynamic dispatch takes place to determine the appropriate function body of the concrete `STL<type>` object. The implementation of `handleCheck()` as shown in Listing 7.6 then redirects the call back to the method `checkFormula()` of the `ModelChecker`, which is finally able to compute the satisfaction set of the formula.

## 7.3 A Brief Overview of Employed CGAL Packages

The algorithms for finding the satisfaction set of a formula make use of geometric data structures and operations, which are implemented by the *Computational Geometry Algorithms Library* (CGAL, pronounced *seagull*). CGAL comprises various packages ranging from basic components like the definition of number types, to highly specialised three-dimensional algorithms, e.g. for the processing of meshes. We incorporate in particular the packages *2D and 3D Linear Geometry Kernel* ([12]), *2D Regularized Boolean Set-Operations* ([24]), and *3D Boolean Operations on Nef Polyhedra* ([35]) in Version 4.8.1 of the library.

Geometric primitives like points, lines, and planes are defined in the first package. These basic data types are used in several other packages of CGAL to build more complex structures. For example, a polygon consists of vertices and edges, which are defined in the geometry kernel. The package *2D Regularized Boolean Set-Operations* specifies polygons with holes and polygon sets. In addition, the package offers functions to apply boolean set-operations to the named types, that is polygon sets can for example be intersected or joined with each other. We use polygon sets to aggregate satisfying general transition firing times, which are identified during the model checking process.

The central data structures for the implementation of the algorithms, that have been presented in the previous chapter, are Nef polyhedra. Their implementation in three-dimensional space is based on the work of *Granados et al.* ([30]), which has been optimised by *Hachenberger and Kettner* ([34]), and is part of CGAL since 2004. It defines Nef polyhedra using a so-called *boundary representation* (B-rep), which means, that a Nef polyhedron is described by its vertices, edges, and facets. The B-rep implemented by the CGAL package consists of the components, that are depicted in Figure 7.5.

Each vertex is surrounded by a so-called *sphere map*. A sphere map defines the neighbourhood of its vertex, that is edges, faces, and volumes, which are adjacent to the point, by specifying points and edges on the surface of the sphere (cf. [20]). A point on the surface of the sphere (*svertex*) corresponds to the beginning of an edge of the polyhedron. If two *svertices* on the sphere's surface are connected by a *sedge*, the *sedge* describes a facet of the Nef polyhedron, which is enclosed by the edges

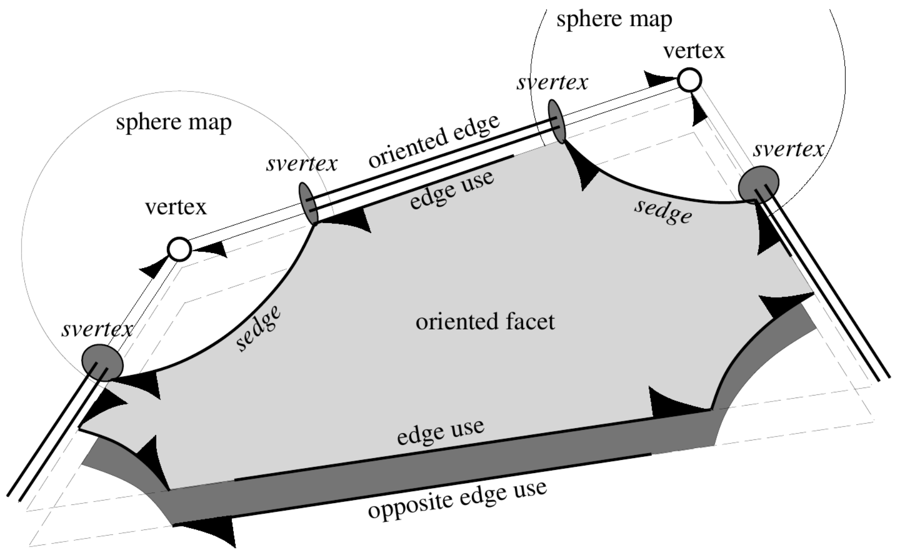


Figure 7.5: Detailed illustration of the components contributing to the B-rep of three-dimensional Nef polyhedra in CGAL (Figure 3 from [34]).

that are indicated by the two svertices. Similarly, volumes of the Nef polyhedron are identified by *sfaces*, which are parts of the sphere's surface surrounded by *sedges*. *sfaces* do not appear in the presented figure.

Actually, the vertices combined with the sphere maps are enough to fully describe and process a Nef polyhedron. The boolean set-operations for example are solely defined on sphere maps. However, since using the sphere maps is rather unintuitive and complicated, the structure is enriched with edges and facets, both of which are oriented, and thus have to be stored twice. The oriented edges can be used by the programmer to easily iterate over the faces of the Nef polyhedron to successively access its defining components.

The described implementation has many advantages compared to other polyhedron implementations. First of all, it is the only freely available implementation, which is exclusively concerned with Nef polyhedra, that we know of. Other comparable implementations handle the subclass of convex polytopes, for example *polymake* (cf. [26]) or the *Parma Polyhedra Library* (cf. [6]). This limitation allows the mentioned libraries to define polytopes of arbitrary dimension, but simultaneously imposes restrictions on the shape of the processable polyhedra, which disallows operations like set difference.

Another advantage of the CGAL package is the use of the *exact geometric computation paradigm* (cf. [60]), which guarantees the precision of all computations. The paradigm replaces the default floating point arithmetic, which is prone to rounding effects and hence too inaccurate in certain cases as has been shown in [34]. The precision (or *robustness*, cf. [59]) comes at a cost, however, which is why many other geometric implementations abandon accurate results in favour of computation speed. *Hachenberger and Kettner* emphasised this trade-off in their own work, where they compared their Nef polyhedron implementation to a commercial 3D modelling system ([34]). The results from the investigation showed, that the commercial system is faster, but can in contrast not handle geometric operations that require highly accurate computations.

Then again, other researchers demonstrated that a robust polyhedron implementation can be as efficient as a non-robust implementation. *Bernstein et al.* ([9]) implemented a different representation for their polyhedra, which allowed them to compute boolean set-operations 16 to 28 times faster than the CGAL package, and only 1.5 to 2 times slower than the commercial 3D computer graphics software *Autodesk Maya*, which relies on fixed precision floating point arithmetic. Similar results have been achieved by *Leconte et al.* ([45]), who outperformed the Nef polyhedron implementation even more by a factor of 90 to 380. Both implementations are not publicly available, though, which prevents us from using these apparently much more efficient approaches.

The third advantage of the Nef polyhedron implementation is, that it is compatible with other packages of CGAL due to the usage of the same geometric primitives, thereby providing access to the wide variety of algorithms, which are part of CGAL. Hence, no additional library is needed, for example, to project three-dimensional facets of a Nef polyhedron to a polygon set in two-dimensional space. Another package of CGAL, which uses the same geometric kernel as the Nef polyhedron implementation and, in addition, offers a similarly designed API, is simply included in our project.

# Chapter 8

# Case Study

This chapter examines two exemplary scenarios, to emphasise the applicability of the algorithms presented in Chapter 6. On the one hand, a modification of Figure 4.3 from Chapter 4 with two general transitions is used to illustrate the functionality of our implementation with a comprehensible, small-scale example. On the other hand, a more sophisticated scenario from [39], which describes the charging cycle of a battery in an electric vehicle, is presented to compare the results from our implementation to simulation-based approaches introduced in [52] and [54]. With the help of these other solutions, it is shown that our algorithms yield correct results in a reasonable amount of time.

At first, the simple example is discussed in Section 8.1. We introduce the STD of the model and subsequently follow the model checking process for several STL formulae. Afterwards, the battery charging scenario is examined in Section 8.2. The more complex model is explained in detail, before the test configurations and test results are presented.

## 8.1 Scenario 1: Reservoir Example

In this section we illustrate the functionality of the developed model checking algorithms with the help of the HPnG, that has been used previously in Chapter 4. The model represents a reservoir with two pumps, each of which transports fluid into or out of the reservoir, respectively. Both pumps may fail during operation, thereby stopping the influx or efflux of liquid.

In the HPnG, which is shown in Figure 8.1, the reservoir is represented by a continuous place $p^C$ with a capacity of 10 units of fluid. The pumps are modelled by fluid transitions $T_1^F$ and $T_2^F$, which fire with a rate of 2 and 1. Compared to Figure 4.3, the deterministic transition has been replaced with a second general transition $T_2^G$, so that the failure of both pumps is probabilistic.

The continuous place $p^C$ has an initial drift of 1, which remains unchanged as long as no general transition fires or the maximum capacity of 10 is not reached. If $T_1^G$ fires first, the fluid source transition $T_1^F$ is disabled and the drift of $p^C$ becomes $-1$. With the reduced drift, the place is drained until it has run empty. In contrast, if $T_2^G$ fires first, the sink transition $T_2^F$ is disabled, which leads to an increased drift of 2. After both discrete places $p_1^D$ and $p_2^D$ have been cleared, the system is in a halting state, since the drift of $p^C$ is zero and no events can occur, any more.

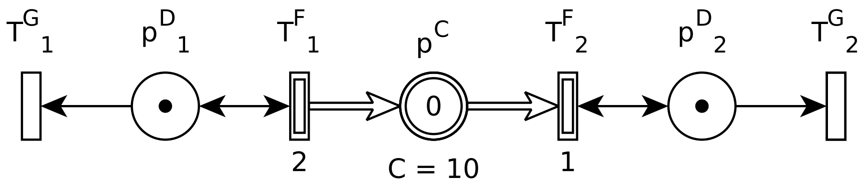


Figure 8.1: The HPnG from Chapter 4 with two general one-shot transitions instead of one.

### 8.1.1 STD

The HPnG is observed for 10 time units. Within this time span, the events, that have been mentioned before, can take place in different orders or not at all depending on the firing times of the general transitions. The corresponding STD, which comprises all possible evolutions, is depicted in Figure 8.2. It has been created using the implementation from [29], while the graphical representation has been generated by *Geomview*, which is a program for displaying and manipulating three-dimensional geometric objects (cf. [23]).

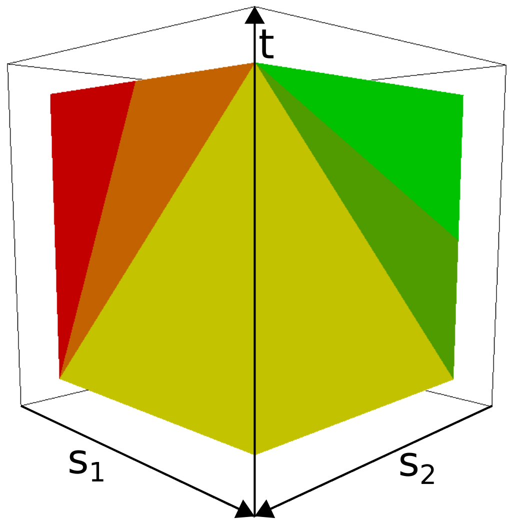
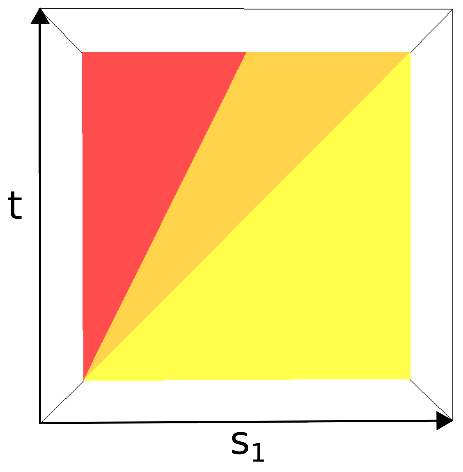


Figure 8.2: STD for the HPnG from Figure 8.1 in the time interval $[0, 10]$ with deterministic area in centre (yellow).

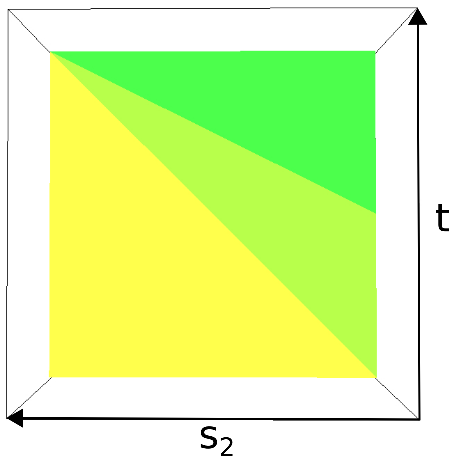
In total, the three-dimensional STD contains nine regions, although only five regions are visible from the point of view chosen in Figure 8.2. The central region, that is depicted in yellow, represents the deterministic area of the STD. Both continuous transitions $T^F_1$ and $T^F_2$ are enabled in this regions, which is why the drift of $p^C$ is 1. Since it takes 10 time units to fill the place $p^C$ with that drift, no events occur in the interval $[0, 10]$ given that no general transition fires. Hence, the deterministic area is not split up, but consists of a single region.

The influence of the general transition firings on the system's behaviour can in this particular case be inferred by looking at the left- and right-hand side of the STD. Separate views on each of these sides are hence displayed in Figure 8.3.

The transition from the deterministic part to the orange region in Figure 8.3a represents the firing of the general transition $T^G_1$. As mentioned before, this event



(a) Side view excluding $s_2$ axis.



(b) Side view excluding $s_1$ axis.

Figure 8.3: Two-dimensional side views on the STD of the HPnG from Figure 8.1.

disables the continuous transition $T_1^F$, and thus sets the drift of $p^C$ to $-1$. The amount of fluid in the place is reduced until the lower capacity bound of 0 is reached or the observed time is exceeded. In the former case, a rate adaption takes place, that is marked by the border of the red region.

Similar observations can be made for the right-hand diagram, in which the $s_1$-axis is not visible. Switching from the deterministic region to the light green region stands for the firing of the general transition $T_2^G$, which leads to an increased drift of 2 in $p^C$. The third region in Figure 8.3b accordingly comprises states, in which $p^C$ has reached its maximum capacity of 10 units of fluid, so that the drift has to be adapted to 0, again.

In the remaining four regions, which neither appear in Figure 8.2 nor in Figure 8.3, both general transitions have fired. Hence all transitions are disabled, and there is no more change in the marking of $p^C$. Although the four regions share the same system state, they are separated in the STD due to the different order, in which the events, that lead to the corresponding system state, took place.

### 8.1.2 Computation of Satisfaction Sets

After having discussed the STD in detail, we define a few properties for our HPnG as STL formulae, and illustrate their evaluation by our model checking tool. We begin with a simple, discrete formula:

$$\phi_1 := m_{p_1^D} \geq 1,$$

which is model checked at some arbitrary time $\tau \in [0, 10]$. The property is satisfied by states, in which the marking of $p_1^D$ is greater than or equal to 1, that is the general transition $T_1^G$ has not fired. This holds in every region below the Hyperplane $H_{s_1} := t = s_1$. $H_{s_1}$ splits the STD in half, and corresponds to the transition from the deterministic region to the orange region, which is visible on the left-hand side of Figure 8.2 or in the middle of Figure 8.3a.

Three regions, which are all shown in Figure 8.3b, are located below $H_{s_1}$. As has been stated in the previous section, the yellow region resides in the deterministic area, that is characterised as the set of states, in which no general transition has fired. Hence, $T_1^G$ in particular has not fired, and $\phi_1$ holds in the region, accordingly. In the states of both other regions, only $T_2^G$ has fired, which is why the formula holds there, as well.

The next step is to determine the points of the three regions, that belong to the time $\tau$. Therefore, the three mentioned regions, that satisfy the formula, are intersected with the hyperplane $H_\tau := t = \tau$.

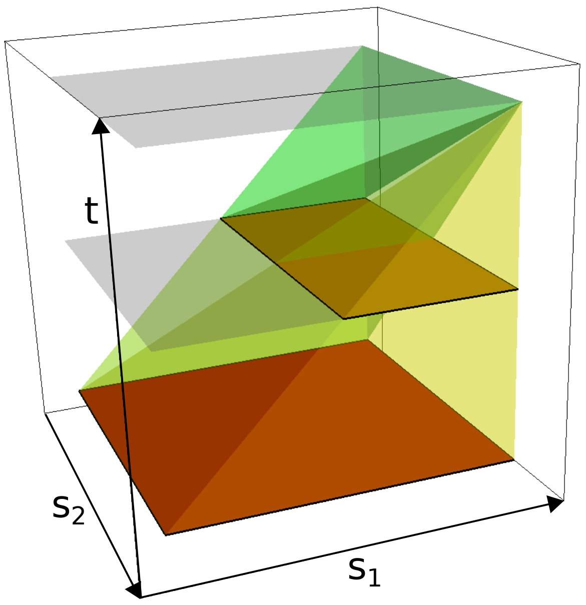


Figure 8.4: The three regions of the STD, that satisfy $\phi_1$, and intersections of hyperplanes at $t = 0$, $t = 5$, and $t = 10$.

In Figure 8.4 we can see three exemplary hyperplanes, that are intersected with the three regions. Compared to Figure 8.2 the STD has been turned counter-clockwise. The red area at the bottom of the STD is the result of the intersection of $H_0$ for the checking time $\tau = 0$ and the yellow region. It contains all possible firing times for $T_1^G$ and $T_2^G$ in the time interval $[0, 10]$. Gradually moving the hyperplane $H_\tau$ up reduces the size of the intersection. For $\tau = 5$ it is only half the size compared to $\tau = 0$ (orange), and for $\tau = 10$ the intersection is not visible, any more.

The resulting point sets are projected to the $s_1$-$s_2$-plane, whereby the satisfaction set $Sat(\phi_1, \tau)$ is formed. For $\tau = 5$ for example, the satisfaction set corresponds to the area $[5, 10] \times [0, 10]$, which is covered by the orange surface in Figure 8.4. Based on $Sat(\phi_1, \tau)$, the probability, that $\phi_1$ is fulfilled at time $\tau$, can be computed via integration over the polygons in the satisfaction set. In case of the red plane from Figure 8.4, this yields a probability of 1, since all possible firing times in the interval $[0, 10]$ for both general transitions are contained in the projection of the plane.

The described process applies nearly equally to all atomic and compound formulae. Only the step to determine the region's subsets, that satisfy a property, has to be adjusted depending on the formula's type. For example, a fluid property naturally requires the model checker to examine the fluid marking of a region instead of its discrete marking. All other steps, however, stay the same.

As has been emphasised in Chapter 6, model checking a time bounded Until differs drastically from the process for atomic and compound formulae. Instead of a point in time, which corresponds to a hyperplane intersected with the STD, a time interval has to be considered. A time bounded Until formula consists of three components: the left-hand sub-formula $\psi_1$, the right-hand sub-formula $\psi_2$, and a time interval $[t_1, t_2]$. Beginning from some time $t$, at which the formula is model checked, the property $\psi_2$ has to hold after at least $t_1$ time units, but not after more than $t_2$ time units. In addition, the left-hand sub-formula $\psi_1$ has to be satisfied permanently from $t$ until $\psi_2$ is eventually fulfilled. These conditions are specified again in the listing below, which has already appeared in Chapter 5.

1. The property $\psi_1$ must hold in all states $\Gamma(\vec{s}, \tau')$, where $\tau' \in [t, t+t_1]$. If $t_1 = 0$, this condition is omitted.
2. The property $\psi_1$ must hold in all states $\Gamma(\vec{s}, \tau'')$, where $\tau'' \in [t + t_1, \tau)$.
3. The property $\psi_2$ must hold eventually in some state $\Gamma(\vec{s}, \tau)$, where $\tau \in [t + t_1, t + t_2]$.

Since the first condition can be omitted if $t_1$ is equal to zero, we first look into this simpler case. For example, we want to determine the probability, that the continuous place $p^C$ at some point contains at least 5 units of fluid during the evolution of the HPnG, which can be expressed using the following STL formula:

$$\phi_2 := \mathit{true}\ U^{[0,10]}\ x_{p^C} \geq 5.0.$$

This property is model checked at time $t = 0$ to include the complete observed time interval $[0, 10]$ in the examination.

The first step of the process is to determine the subsets of the regions in the STD, that satisfy the left- and right-hand sub-formulae. The sub-formula *true* simply holds in every point of the STD, which is why the second condition listed above is fulfilled, automatically. Hence, we only have to test the third and last condition for the right-hand sub-formula $\psi_2 := x_{p^C} \geq 5.0$.

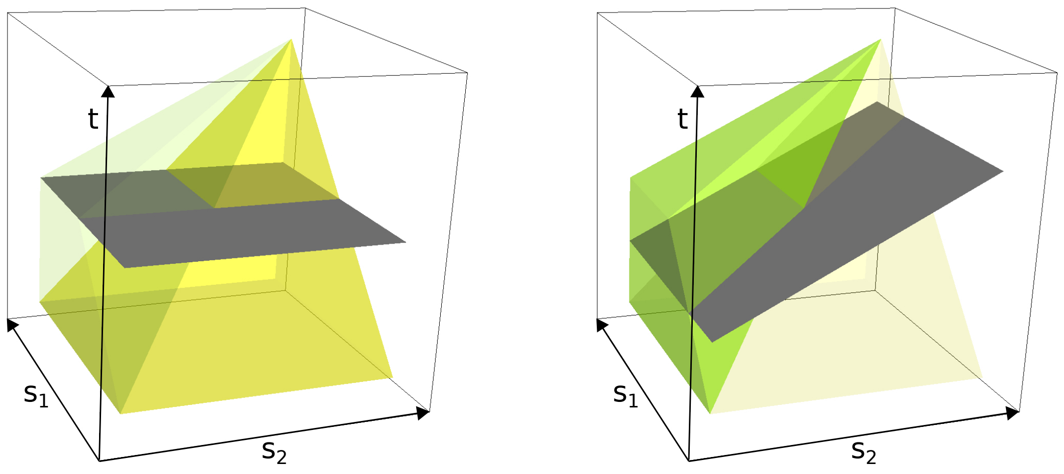
To find the parts of all regions, that satisfy $\psi_2$, a hyperplane is calculated for each region, that indicates the time, at which the threshold is reached. The required information, like the initial fluid level $x_{p^C}(t_0)$ of $p^C$ when entering the region at time $t_0$ and the drift $d_{p^C}$ of $p^C$, are contained in the region's system state $\Gamma_R$, which is associated with the *underlying event facet* (see Chapter 4) of the region during the creation of the STD. The hyperplane for $x_{p^C} \geq 5.0$ is then computed using the following equation:

$$5 = x_{p^C}(t_0) + (t - t_0) * d_{p^C}. \tag{8.1}$$

We execute the computation exemplary for the yellow region in the deterministic area and for the light green region region from Figure 8.3b in the stochastic area.

In the initial state, which is simultaneously the system state of the yellow region, hold $t_0 = 0$, $x_{p^C}(t_0) = 0$, and $d_{p^C} = 1$. That is the region is entered at $t_0 = 0$ with an initial fluid level $x_{p^C}(t_0) = 0$ and a drift $d_{p^C}$ of 1, since both continuous transitions $T_1^F$ and $T_2^F$ are enabled. If we insert the parameters into Equation 8.1, we receive the hyperplane $H_{yellow} := t = 5$. Above this hyperplane, the property $\psi_2$ is satisfied.

The light green region, in contrast, is entered from the yellow region after the general transition $T_2^G$ has fired. The amount of fluid, which has been accumulated in $p^C$ until then, thus depends on the firing time $s_2$ of $T_2^G$. Since the drift $d_{p^C}$ in the yellow region was 1, the place has been filled with $s_2$ units of fluid, when the transition from the yellow to the light green region takes place. So, it holds that $t_0 = s_2$, $x_{p^C}(t_0) = s_2$, and $d_{p^C} = 2$. In this case, the quantity of the drift depends on the disabled sink transition $T_2^F$. Again, if we apply the parameters to Equation 8.1, we receive another hyperplane $H_{green} := t = \frac{s_2}{2} + 2.5$.



(a) Hyperplane $H_{yellow}$ splitting the yellow region in the deterministic area.

(b) Hyperplane $H_{green}$ splitting the light green region in the stochastic area.

Figure 8.5: Division of selected regions by hyperplanes representing the satisfaction of $\psi_2 := x_{p^C} \geq 5.0$.

Both hyperplanes are separately depicted in Figure 8.5a and Figure 8.5 along with the respective regions. The parts above the hyperplanes each comprise states, that satisfy the formula $\psi_2$.

In case of the yellow region, the hyperplane is perpendicular to the $t$-axis. This is characteristic of hyperplanes in the deterministic area. All evolutions in the deterministic part run through the same state changes, so that fluid markings and clocks evolve in the same manner. Hence, the property is fulfilled by all evolutions in that area at the same time.

The hyperplane for the light green region however is inclined, since the continuous marking of $p^C$ there depends on the variable $s_2$ representing the firing time of $T_2^G$. $s_2$ determines the duration, for which the continuous marking of $p^C$ evolves with the drift of the previous system state, and thus also determines the initial fluid level in $p^C$ when entering the region. Therefore, the required threshold of 5 units of fluid is reached at different times depending on the firing of $T_2^G$.

After having identified the subsets satisfying the left- and right-hand sub-formulae, we have to find the evolutions, that reach a state, in which the property $x_{p^C} \geq 5.0$ holds. In the specific case of our formula $\phi_2$, no further restrictions are required, since the left-hand sub-formula *true* holds in the complete STD. Moreover, our analysis includes the full time interval $[0, 10]$, so that we do not even need to consider time boundaries.

Figure 8.6 shows the intersections of the hyperplanes from the previous step with the respective regions and their projection to the $s_1$-$s_2$-plane. In this case,

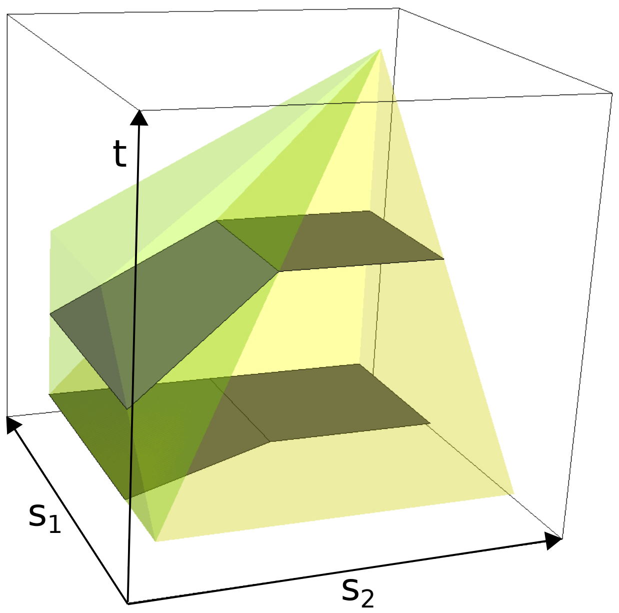


Figure 8.6: Intersections of hyperplanes, that represent the property $x_{p^C} \geq 5.0$, with regions contributing to the satisfaction set $Sat(\phi_2, 0)$.

the projection alone identifies the evolutions, that fulfil the time bounded Until formula. The projections do not need to be restricted with the facets of the polyhedra satisfying *true*, since there are simply no such areas in which *true* does not hold, and hence there are no facets to restrict the projections.

As soon as all intersections have been projected to the $s_1$-$s_2$-plane, the resulting polygons form the satisfaction set $Sat(\phi_2, 0)$. In fact, no new elements are added to the area depicted in Figure 8.6. The other regions either do not satisfy $\psi_2$, or the additional firing times are already contained in the mentioned area.

After having discussed this case of a time bounded Until with a lower bound of $t_1 = 0$, we can examine another property with a lower time bound, that is unequal to 0. For this purpose, we model check the following STL formula at time $t = 2.5$:

$$\phi_3 := true\,\mathcal{U}^{[2.5,7.5]}\, x_{p^C} \geq 5.0.$$

$\phi_3$ is identical to the previously examined formula $\phi_2$, except for the different time bound $[2.5, 7.5]$, which requires $\psi_2 := x_{p^C} \geq 5.0$ to be fulfilled after $t + 2.5$ and before $t + 7.5$ time units. Since the right-hand sub-formula has not changed, the hyperplanes, that have been computed previously, still mark the parts of regions, in which $p^C$ contains more than 5.0 units of fluid.

Now, the main difference in the examination of $\phi_3$ compared to $\phi_2$ is, that the interval $[t, t + t_1]$ has to be handled separately, to check whether the first condition from the list above is fulfilled. At first glance, the condition is again automatically fulfilled in all points of the STD due to the fact, that the left-hand sub-formula is *true*. However, the third condition demands the property $\psi_2$ to be satisfied in the time interval $[t + t_1, t + t_2]$, and not before. Hence, if an evolution reaches a state, in which $\psi_2$ is satisfied at a time in $[t, t + t_1]$, the time bounded Until formula is not satisfied.

For the example with checking time $t = 2.5$ this means, that $\psi_2$ must not be satisfied in the time interval $[2.5, 5]$. Nevertheless, this is the case in the light green region as can be seen already in Figure 8.6. The intersection of the hyperplane $H_{green}$ with the region lies exactly in the concerned interval $[2.5, 5]$.

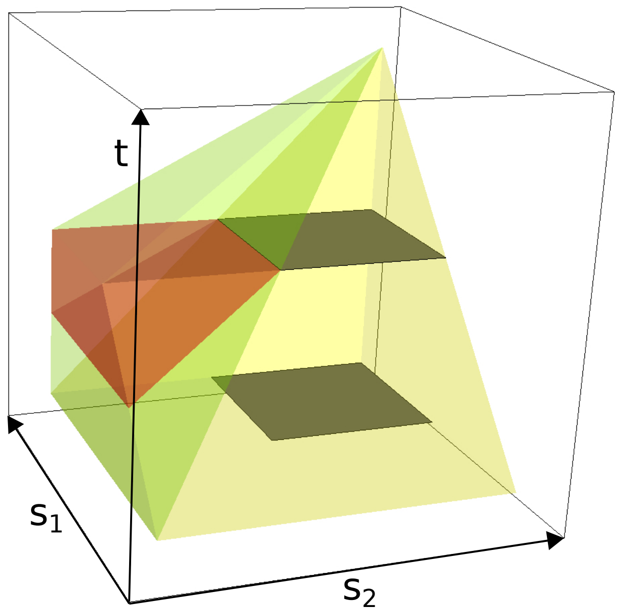


Figure 8.7: Part of the light green region, that satisfies the formula $\phi_3$ too soon, highlighted red.

The subset of the light green region, that is highlighted red in Figure 8.7, contains states, in which the Until formula is satisfied too soon, i.e. before time $t+2.5 = 5$. An evolution with states in that part of the STD hence does not fulfil the Until formula $\phi_3$, so that this time the light green region does not contribute to the satisfaction set $Sat(\phi_3, 2.5)$. The hyperplane $H_{yellow}$ however intersects the yellow region at $t = 5$, which is why all points of the region, for which $\psi_2$ holds, are located in the interval $[5, 10]$. Hence, only the intersection of $H_{yellow}$ and the yellow region, that is projected to the $s_1$-$s_2$-plane as shown in Figure 8.7, constitutes the satisfaction set $Sat(\phi_3, 2.5)$

The probability for the satisfaction of the formula at the chosen point in time is computed from the satisfaction set in a last step. For the first time in the process the distributions for the general transition firing times are needed. The previous steps have been independent of these distributions, since the STD comprises all evolutions without taking the probability of their firing times into account. In this example, we choose a normal distribution $\mathcal{N}$ with mean $\mu = 5$ and standard deviation $\sigma = 0.5$ for both general transitions in our HPnG.

As has been explained in Section 6.4 in Chapter 6, we can in general not directly integrate over the satisfaction set. Instead, a triangulation of the polygons in $Sat(\phi_3, 2.5)$, that is a partition in triangles, has to be performed. Here, the satisfaction set consists of a single square area, which is split in two triangles, as can be seen in Figure 8.8.

The probability for the firing times covered by each of the triangles is computed

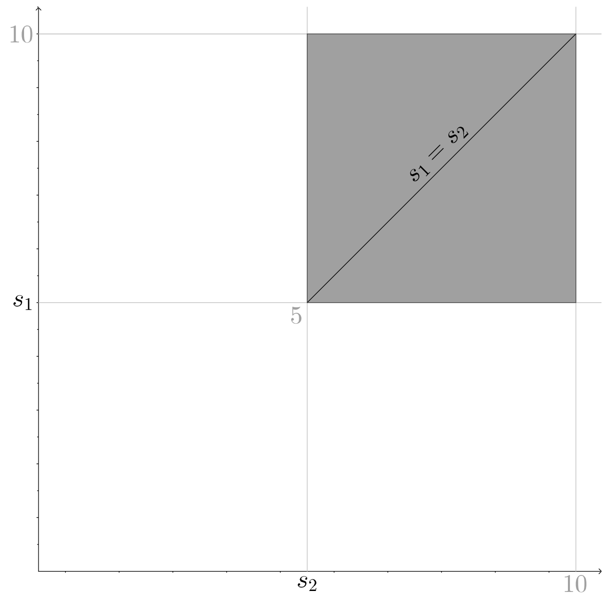


Figure 8.8: Satisfaction set $Sat(\phi_3, 2.5)$ with partitioning line from triangulation step.

separately, and added up afterwards. Since the computation is more or less equal for every triangle, we look exemplary into the integration of the upper triangle, for which the following double integral has to be determined:

$$\iint_{Triangle} g_1(s_1) * g_2(s_2) ds_1 ds_2, \tag{8.2}$$

with $g_1(x) = g_2(x) = \mathcal{N}(5, 0.25)(x)$, where $\mathcal{N}(\mu, \sigma^2)(x) = \frac{1}{\sqrt{2\sigma^2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ is the normal distribution with mean $\mu$ and variance $\sigma^2$ (cf. [22]). The domain $Triangle$ defines the area, for which the integral has to be computed.

In a first step, the double integral is converted to a single integral by specifying the boundaries of the variable $s_1$ from Equation 8.2 with the variable $s_2$. This allows us to transform the inner integral to an expression which depends on $s_2$ instead of $s_1$, thereby reducing the number of variables in the integral to one. $s_1$ is bounded below by the line $s_1 = s_2$, which is plotted in Figure 8.8, while its upper boundary is the constant 10. The interval boundaries for $s_1$ can hence be indicated by $s_2 \le s_1 \le 10$. $s_2$ however takes values between 5 and 10. If the boundaries are applied to Equation 8.2, the integral can be reshaped as follows:

$$\begin{aligned}\int_5^{10} \int_{s_2}^{10} g_1(s_1) * g_2(s_2) ds_1 ds_2 &= \int_5^{10} g_1(s_1)\Big|_{s_1=s_2}^{s_1=10} * g_2(s_2) ds_2 \\ &= \int_5^{10} (G_1(10) - G_1(s_2)) * g_2(s_2) ds_2,\end{aligned}$$

where $G_1(x) = \frac{1}{2} * (1 + erf(\frac{x-\mu}{\sigma\sqrt{2}}))$ is the cumulative density function of $\mathcal{N}(\mu, \sigma^2)$.

The second step is the numerical integration of the transformed integral via Gauß quadrature (cf. [55]), which is not carried out manually, here. Instead, we rely on our model checker, which determines a probability of 0.125 for the firing times in the triangle. The same probability is computed for the second triangle, so that the total probability for $Sat(\phi_3, 2.5)$ amounts to 25%. The formula $\phi_3$ is hence satisfied by a quarter of the possible evolutions at the time $t = 2.5$.

## 8.2 Scenario 2: Charging an Electric Vehicle

Electric mobility is an important technology of the future and subject to recent research (e.g. [2], [1], [10]). In combination with renewable energy sources, so-called *plug-in electric vehicles* (PEV) can help to reduce our dependence on fossil fuels. Since electric vehicles can be both consumers and providers of power, they are able to contribute to the stability of the power grid by absorbing excess energy or providing power in times of a high demand. A grid that is able to detect available electric vehicles and can incorporate them into its load balancing strategy is called a *smart-grid* (cf. [37]). Depending on the current overall production and consumption of power, a smart-grid can charge or discharge electric vehicles to stabilise itself by requesting the needed power from or transferring the excess power to the connected batteries.

To not interfere with the interests of the car owner, that is the availability of the vehicle or the lifetime of the battery, the smart-grid can only access the resources provided by the electric vehicle according to predefined strategies. The user can generally choose between several strategies, which allow different levels of interaction between the battery and the grid. Charging the battery instantaneously to its full capacity, for example, prevents that the grid can extract energy from the vehicle. More sophisticated strategies try to comply with both the requirements of the user and the grid (cf. [42]).

In this section we examine the charging process for the battery of an electric vehicle. The battery is charged according to a just-in-time strategy, which tries to coordinate the charging process with the client's behaviour. By predicting the recurrence time of the car owner, the charging level can be held at a state which is optimal for the battery's lifetime as long as possible. The battery is only charged to its full capacity just before the client arrives to unplug the vehicle from the charger (cf. [39]). We investigate different settings for this strategy with our model checking tool. To prove the validity of our results, the computed values are compared to two other tools, which are capable of processing HPnG with two general one-shot transitions, as well.

Since the model of the charging process is considerably larger and not as intuitive as the HPnG in the previous section, it is at first explained in detail. In the next step, we discuss the different system configurations, which are used to test our program. The results of this step are presented along with the comparison to the outputs of the other tools in the last section. A brief introduction to the other tools is included in this part.

### 8.2.1 The Model

We examine a model of a battery in a *Tesla Model S*[1], that is connected to a charging station. The HPnG is observed for 144 time units, which corresponds to 1440 minutes or 24 hours. Depending on the configuration of the HPnG, the corresponding STD consists of 335 or 360 regions, which is why we do not display it here. Before introducing the configurations that are model checked by our tool, the components of the system, which are depicted in Figure 8.9 and Figure 8.10, are explained in detail.

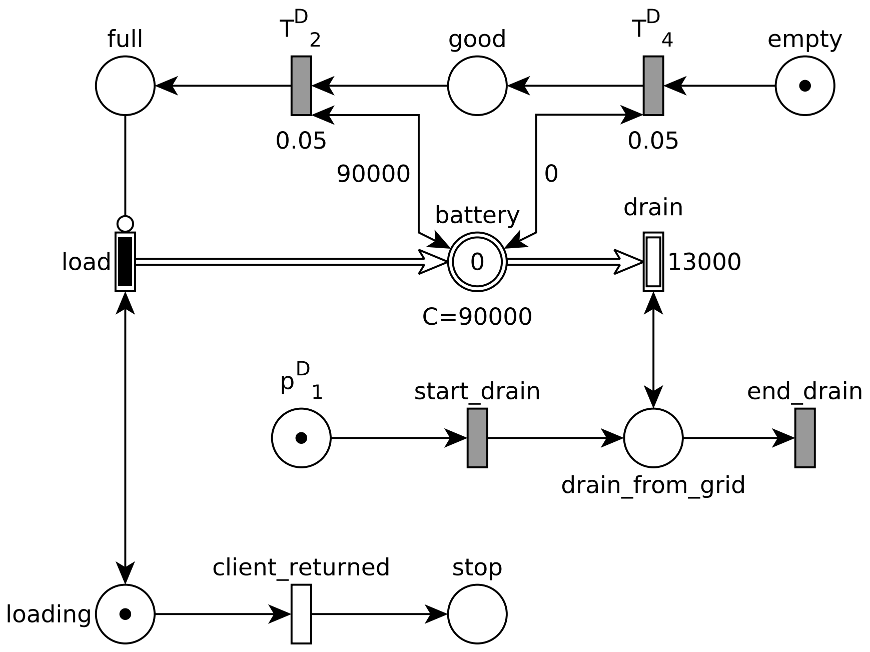


Figure 8.9: Part 1: HPnG for modelling the charging process of a battery in an electric vehicle (based on Figure 5 from [39]).

The central element is the place *battery* with a capacity of 90000 (in Watt-hours) and its dynamic continuous source transition *load*. As long as the place *loading* contains one token, the car is connected to the charger. The duration of the load cycle is determined by the first general transition *client_returned*, which specifies, after how many time units the owner of the car unplugs it from the charger.

Another limiting factor for the charging process is the capacity of the *battery*. The abstract state of the battery is indicated by the three places *empty*, *good*, and *full*. A token is moved by the transitions $T_4^D$ and $T_2^D$ from *empty* over *good* to *full* during the charging process. The delay of 0.05 time units (30 seconds) for the firing of $T_4^D$ and $T_2^D$ models the reaction time of the charging station to begin or respectively end its work. If the *battery* is fully charged, $T_D^2$ fires after 0.05 time units and moves the token to the place *full*, thereby disabling the transition *load* via the inhibitor arc.

The last part of the HPnG represents a consumption of power from the battery to stabilise the grid. The transition *drain* is enabled, after the deterministic transition *start_drain* has fired. We assume that the grid's demand for power is very limited,

[1]`https://www.tesla.com/models`

so that the fluid transition can be disabled again after a short period of time. Hence, the transition *end_drain* fires after 1 time unit of power consumption.

The charging strategy, which is implemented by the described model, fills the battery in two consecutive steps. In a first phase, the capacity is brought up quickly to a level of 40000. This level, on the one hand, improves the lifetime of the battery, and, on the other hand, is sufficient to satisfy small power demands of the grid (cf. [39]). The second phase charges the battery up to its capacity limit, in order that the client can use the maximum range of his electric vehicle. Between the two phases, the charging is stopped and the power level of the battery remains unchanged.

The strategy is modelled via the dynamic continuous transition *load*. As has been stated in Chapter 2, the rate of a dynamic continuous transition depends on the rates of other continuous transitions. In case of the transition *load*, it is defined as the sum of rates of all static fluid transitions, that belong to the second part of the HPnG shown in Figure 8.10.

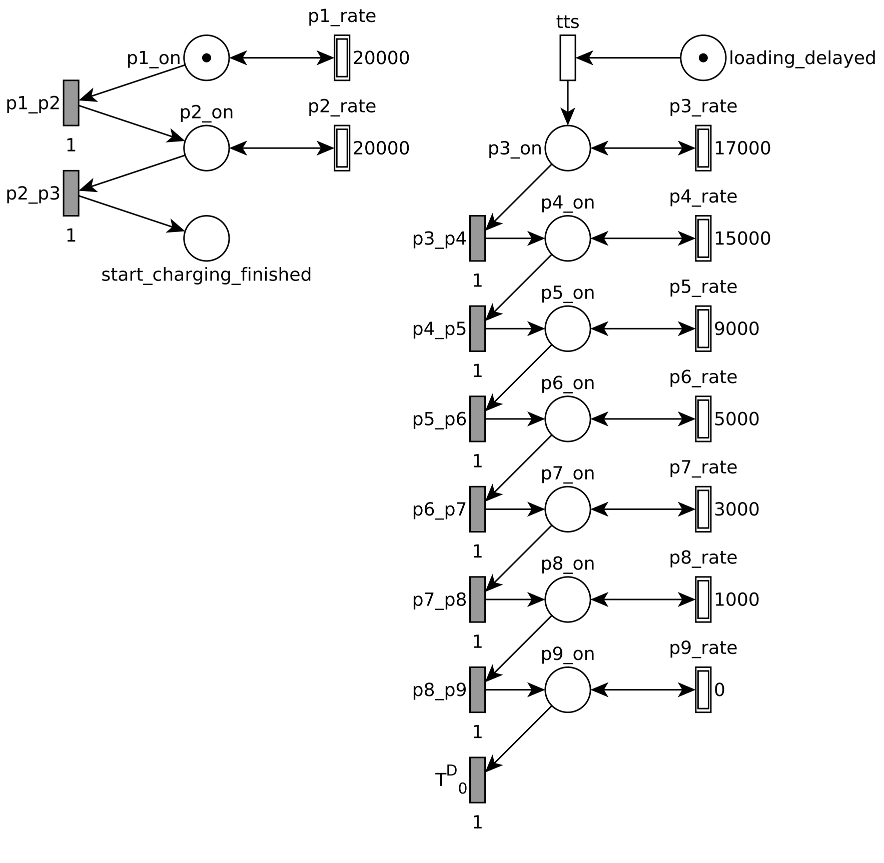


Figure 8.10: Part 2: Control part for the dynamic continuous transition *load* in Figure 8.9 (based on Figure 5 from [39]).

The first phase of the charging process is controlled in the left side of the HPnG. The deterministic transitions move the token through the discrete places, whereby the fluid transitions are enabled or disabled, respectively. Whenever the token is moved, the rate of *load* is adapted to the rate of the enabled fluid transition. If no transition is enabled, the rate of *load* is zero. The token resides in each place for one time unit, only. Hence, *battery* contains 40000 units of fluid, when the token reaches the place *start_charging_finished*.

The second phase begins, when the general transition *tts* fires. Like for the first phase, the place with the token activates one of the static continuous transitions, which is then used as the rate of *load*. Since it takes longer to load the battery when it is nearly full than when it is empty, the rate gradually reduces with each movement of the token. After the battery is fully charged with 90000 Wh, which is equal to the sum of rates of all fluid transitions in Figure 8.10, the rate of *load* is set to zero.

### 8.2.2 Configurations

The model is examined under different configurations and with different STL formulae to show that the approach presented in this work is applicable to real-world scenarios. Strictly speaking, we use two HPnGs for this purpose, since the transition *drain* is disabled in some of the concerned cases. The token, which is present in the place $p_1^D$ in Figure 8.9, is removed from the initial state, so that the deterministic transition *start_drain* can not fire. However, nothing else is changed in the HPnG, which is why the models have not been depicted, separately.

Another parameter, which is altered across the test cases, is the distributions of the general transition firing times. The distributions model both the variation in the recurrence time of the client and its approximation by the charger. Thus, changing the distributions for the recurrence time (transition *client_returned*) or for the delay of the second charging phase (transition *tts*) represents different charging strategies.

A charging strategy is considered reasonable, if the battery is loaded up to at least 90% when the car is disconnected from the charger. This property is specified by the following STL formula:

$$\phi_1 := m_{loading} \geq 1 \, \mathcal{U}^{[0,144]} \, x_{battery} \geq 81000.$$

The place *loading* contains a token as long as the general transition *client_returned* has not fired. Model checking the formula at $t = 0$ hence determines all evolutions, in which the place *battery* is filled with at least 81000 units of fluid before the charging process is interrupted. Note, that this property does not take an optimisation of battery life or other desirable characteristics of the charging process into account.

| case ID | *drain* enabled? | distribution *client_returned* | distribution *tts* | STL formula |
|---|---|---|---|---|
| 1*a*) | ✗ | normal distribution $\mu = 54$, $\sigma = 6$ | normal distribution $\mu = 40$, $\sigma = 6$ | $\phi_1$ |
| 1*b*) | ✗ | normal distribution $\mu = 52$, $\sigma = 6$ | normal distribution $\mu = 40$, $\sigma = 6$ | $\phi_1$ |
| 1*c*) | ✗ | normal distribution $\mu = 52$, $\sigma = 12$ | normal distribution $\mu = 40$, $\sigma = 12$ | $\phi_1$ |
| 1*d*) | ✗ | normal distribution $\mu = 54$, $\sigma = 12$ | normal distribution $\mu = 40$, $\sigma = 12$ | $\phi_1$ |

Table 8.1: Test series 1.

In a first series of tests, the influence of different distribution parameters on the probability for the satisfaction of $\phi_1$ is investigated. Table 8.1 lists the corresponding configurations. Here, the firing times of both general transitions *client_returned* and *tts* follow normal distributions with varying mean and standard deviation. In case 1*a*), the client is expected to unplug the car from the charger after $\mu = 54$ time units (9 hours). The standard deviation $\sigma = 6$ (1 hour) models that this expected value varies in most cases between 8 and 10 hours. In contrast, the standard deviation of the cases 1*c*) and 1*d*) is twice as large, to represent that the client's return time is not as steady. The impact of these parameter changes is reflected in the results, which are discussed in the following section.

The second test series assumes that the grid consumes some of the batteries power during the charging process. Our HPnG however fills the place *battery* with a fixed total amount of 90000 units of fluid, so that the reachable level in *battery* is reduced by the amount of power, which is consumed by the transition *drain*. If *drain* is enabled for one time unit, for example, the reachable level of *battery* is reduced to 77000 units of fluid, whereby the right-hand sub-formula of $\phi_1$ can not be fulfilled, any more. Hence, the STL formula has to be adjusted, accordingly:

$$\phi_2 := m_{loading} \geq 1\, \mathcal{U}^{[0,144]}\, x_{battery} \geq 75000.$$

This formula is model checked using the configurations shown below in Table 8.2.

| case ID | *drain* enabled? | distribution *client_returned* | distribution *tts* | STL formula |
|---|---|---|---|---|
| 2*a*) | ✓ | normal distribution $\mu = 54$, $\sigma = 6$ | normal distribution $\mu = 40$, $\sigma = 6$ | $\phi_1$ |
| 2*b*) | ✓ | normal distribution $\mu = 54$, $\sigma = 6$ | normal distribution $\mu = 40$, $\sigma = 6$ | $\phi_2$ |
| 2*c*) | ✓ | normal distribution $\mu = 54$, $\sigma = 12$ | normal distribution $\mu = 40$, $\sigma = 12$ | $\phi_2$ |

Table 8.2: Test series 2.

Aside from the enabled transition *drain*, the configurations listed in Table 8.2 are identical to 1*a*) and 1*d*). Case 2*a*) is included to verify that $\phi_1$ can not be satisfied, when the grid takes power from the *battery*. This means, that *battery* can not be charged to over 90% with our static strategy.

In the remaining two test series we want to determine the probability that the *battery* is filled with up to 81000 units of fluid at some specific time. Instead of a time bounded Until, a simple atomic formula is used:

$$\phi_3 := x_{battery} \leq 81000.$$

The configurations recorded in Table 8.3 use uniform distributions for the general transition firing times. A continuous uniform distribution $\mathcal{U}(a, b)$ is defined as follows (cf. [22]):

$$\mathcal{U}(a,b)(x) = \begin{cases} \frac{1}{b-a}, & \text{if } a \leq x \leq b \\ 0, & \text{otherwise.} \end{cases}$$

With the parameters shown in Table 8.3, the firing times of *client_returned* and *tts* are hence limited to the interval $[52, 56]$ and $[46, 50]$, respectively. That is, the

| case ID | *drain* enabled? | distribution *client_returned* | distribution *tts* | checking time |
|---|---|---|---|---|
| 3$a$) | ✗ | uniform distribution $a = 52$, $b = 56$ | uniform distribution $a = 46$, $b = 50$ | 0 |
| 3$b$) | ✗ | uniform distribution $a = 52$, $b = 56$ | uniform distribution $a = 46$, $b = 50$ | 36 |
| 3$c$) | ✗ | uniform distribution $a = 52$, $b = 56$ | uniform distribution $a = 46$, $b = 50$ | 54 |
| 3$d$) | ✗ | uniform distribution $a = 52$, $b = 56$ | uniform distribution $a = 46$, $b = 50$ | 66 |
| 3$e$) | ✗ | uniform distribution $a = 52$, $b = 56$ | uniform distribution $a = 46$, $b = 50$ | 72 |

Table 8.3: Test series 3.

| case ID | *drain* enabled? | distribution *client_returned* | distribution *tts* | checking time |
|---|---|---|---|---|
| 4$a$) | ✗ | exponential distribution $\lambda = 1$ | uniform distribution $a = 0$, $b = 1$ | 0 |
| 4$b$) | ✗ | exponential distribution $\lambda = 1$ | uniform distribution $a = 0$, $b = 1$ | 12 |
| 4$c$) | ✗ | exponential distribution $\lambda = 1$ | uniform distribution $a = 0$, $b = 1$ | 36 |
| 4$d$) | ✗ | exponential distribution $\lambda = 1$ | uniform distribution $a = 0$, $b = 1$ | 72 |

Table 8.4: Test series 4.

client is assumed to disconnect the car from the charger after 520 to 560 minutes, and the charger starts its second phase after 460 to 500 minutes.

The configurations for the fourth and last test series are presented in Table 8.4. Again, the distributions or their parameters have been exchanged. Here, we use an exponential distribution, which is defined as follows (cf. [22]):

$$exp(\lambda)(x) = \begin{cases} \lambda * e^{-\lambda x}, & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Compared to the other test series, this last set of configurations does not involve distributions, which correspond to a just-in-time strategy. The configurations are however mainly used to validate, that our model checker properly computes the probability for a satisfaction set. Moreover, one of the tools, whose results are compared to ours, is only capable to handle uniform and exponential distributions and can only process fluid properties. The cases in the last two series are hence limited to these types of distributions and to the atomic formula $\phi_3$.

### 8.2.3 Results

The configurations, which have been introduced in the previous section, are model checked with our tool. In addition, the same tests are performed with two other tools to validate our computations. Both tools rely on different approaches compared to the solution presented in this thesis.

The first tool by Carina Pilch simulates runs of HPnGs with an arbitrary number of general one-shot transitions. For this purpose, it samples the firing times of all general transitions, in order that they can be treated as deterministic transitions. The behaviour of the HPnG hence becomes deterministic and its evolution can be predicted completely. Model checking a formula then reduces to testing the single evolution for the desired property. By repeating this process several times, the program is able to approximate the probability for the satisfaction of the formula. After the requested number of iterations, the tool outputs a mean value with a confidence interval (cf. [52]).

The second program, the *Fluid Survival Tool* (FST) by Björn Postema[2], is a combination of a simulator and a model checker. It can process HPnGs with multiple general one-shot transitions, as well, but is restricted to continuous STL formulae. The program uses the model checking algorithms which have been presented in [27] to determine the probability for the satisfaction of a formula in an HPnG with a single general one-shot transition. If more general transitions are present in the model, the firing times of all but one transition are sampled multiple times similar to the approach of the first tool. Then, the HPnG with the sampled firing times can be handled by the model checker for HPnGs with one general transition. The results are written to a file as mean values without a confidence interval (cf. [54]).

| **case ID** | **Nef polyhedra tool probability** | **Simulation mean** | **Simulation confidence interval** | **FST mean** |
|---|---|---|---|---|
| 1$a$) | 0.9026 | 0.9015 | $[0.8965, 0.9065]$ | — |
| 1$b$) | 0.8556 | 0.8557 | $[0.8507, 0.8607]$ | — |
| 1$c$) | 0.7016 | 0.7020 | $[0.6970, 0.7070]$ | — |
| 1$d$) | 0.7411 | 0.7401 | $[0.7351, 0.7451]$ | — |
| 2$a$) | 0 | 0 | $[0, 0]$ | — |
| 2$b$) | 0.8643 | 0.8654 | $[0.8604, 0.8704]$ | — |
| 2$c$) | 0.6667 | 0.6662 | $[0.6612, 0.6712]$ | — |
| 3$a$) | 1 | 1 | $[1, 1]$ | 1 |
| 3$b$) | 0.9985 | 1 | $[1, 1]$ | 1 |
| 3$c$) | 0.0313 | 0.0289 | $[0.0239, 0.0339]$ | 0.0360 |
| 3$d$) | 0.0317 | 0.0334 | $[0.0284, 0.0384]$ | 0.0330 |
| 3$e$) | 0.0314 | 0.0287 | $[0.0237, 0.0337]$ | 0.0400 |
| 4$a$) | 1 | 1 | $[1, 1]$ | 1 |
| 4$b$) | 0.9685 | 0.9676 | $[0.9626, 0.9726]$ | 0.9770 |
| 4$c$) | 0.9685 | 0.9686 | $[0.9636, 0.9736]$ | 0.9680 |
| 4$d$) | 0.9685 | 0.9672 | $[0.9622, 0.9722]$ | 0.9710 |

Table 8.5: Comparison of results for all test series.

Table 8.5 summarises the output of all three programs rounded to four decimal places. The FST can only handle fluid formulae, which is why it can not be applied to the test series 1 and 2. From the results, the following observations can be

[2]`https://github.com/bjornpostema/fluid-survival-tool`

deduced:

In test series 1, configuration *a*) produces the most satisfying results. Given that the chosen distribution $\mathcal{N}(54, 6^2)$ correctly models the recurrence time of the client (transition *client_returned*), the start time for the second charging phase (transition *tts*) is approximated accurately enough to satisfy the property $\phi_1$ in over 90% of all cases. As one could expect, a higher variation in the client's recurrence time, like in cases 1*c*) and 1*b*), leads to a less accurate prediction of the start time for the second phase, so that the battery is not as often sufficiently charged as in the cases 1*a*) and 1*b*).

The same observation can be made for series 2. The first case shows, that $\phi_1$ can not be satisfied by the model, if the grid discharges the battery. Switching to $\phi_2$ however leads to similar results as in test series 1.

Test series 3 and 4 incorporate an atomic formula instead of a time bounded Until, which demands the battery's charging level to be below 81000. Case 3*c*), for example, implies, that the battery is charged to less than 81000 after 54 time units in only about 3% of the possible evolutions. Conversely, this means that in 97% of all cases the battery is charged to over 81000. So, if the client returns to his car after 9 hours, the strategy will have charged the battery to over 90% of its capacity in almost any case.

Furthermore, the table shows that the results of our model checker correspond to the computed probabilities of both other tools. The 99% confidence intervals generated with the simulation tool by Carina Pilch contain our probabilities in almost any case. The only outlier occurs in case 3*b*) with the (obviously wrong) value 0.9985, where both the simulation tool and the FST yield a probability of 1. The deviation can be traced back to a numerical error in the integration over the satisfaction set. By increasing the number of iterations in the numerical integration, the error can be reduced, but the higher precision can increase the cost of the computation, significantly. Hence, one has to take the trade-off between precision and computation time into account when changing the number of iterations.

Table 8.6 shows the computation times of the Nef polyhedra tool and the simulation tool. The highlighted column contains the times for the default number of iterations, which have been used to generate the probabilities from Table 8.5. The FST has been excluded here, since it computes multiple test cases in a single run and can hence not be compared to the other tools.

In most test cases, the Nef polyhedra tool in its default set-up computes the probabilities faster than the simulation tool. Only in the cases where the probability amounts to 0 or 1 (2*a*), 3*a*), 3*b*), 4*a*)), the simulation tool outperforms our program. Even with the doubled number of iterations the Nef polyhedra tool is often faster or only slightly slower than the simulation tool.

The computation time of our program is composed of three parts: building the STD, computing the satisfaction set, and integrating over the satisfaction set. The former two steps are not affected by the number of iterations, which is why in test series 1, for example, the generation of the STD and the following computation of the satisfaction set take about 3.14 seconds for every configuration. The mere integration thus requires about 0.02 seconds with 1024 iterations, about 0.3 seconds with 4096 iterations, and about 1.1 seconds with 8192 iterations, which implies that doubling the number of iterations approximately quadruples the computation time. The other test cases, in which the probability is not 0 or 1, confirm this observation.

| **case ID** | **Nef polyhedra tool runtime** | | | **Simulation runtime** |
|---|---|---|---|---|
| | 1024 iterations | 4096 iterations | 8192 iterations | |
| 1$a$) | 3.157$s$ | 3.454$s$ | 4.144$s$ | 31.122$s$ |
| 1$b$) | 3.213$s$ | 3.443$s$ | 4.15$s$ | 64.1$s$ |
| 1$c$) | 3.167$s$ | 3.397$s$ | 4.145$s$ | 84.179$s$ |
| 1$d$) | 3.187$s$ | 3.384$s$ | 4.119$s$ | 81, 049$s$ |
| 2$a$) | 1.718$s$ | 1.715$s$ | 1.717$s$ | 1.523$s$ |
| 2$b$) | 2.394$s$ | 2.626$s$ | 3.378$s$ | 41.256$s$ |
| 2$c$) | 2.397$s$ | 2.628$s$ | 3.334$s$ | 143.403$s$ |
| 3$a$) | 0.884$s$ | 0.897$s$ | 0.879$s$ | 0.224$s$ |
| 3$b$) | 3.105$s$ | 6.675$s$ | 15.303$s$ | 0.388$s$ |
| 3$c$) | 3.112$s$ | 6.015$s$ | 15.333$s$ | 10.588$s$ |
| 3$d$) | 3.043$s$ | 6.783$s$ | 16.04$s$ | 7.594$s$ |
| 3$e$) | 3.482$s$ | 6.063$s$ | 15.295$s$ | 10.777$s$ |
| 4$a$) | 0.889$s$ | 0.877$s$ | 0.887$s$ | 0.263$s$ |
| 4$b$) | 3.046$s$ | 6.065$s$ | 15.315$s$ | 13.58$s$ |
| 4$c$) | 3.117$s$ | 6.006$s$ | 15.293$s$ | 10.742$s$ |
| 4$d$) | 3.108$s$ | 6.056$s$ | 15.315$s$ | 11.657$s$ |

Table 8.6: Comparison of computation times for Nef polyhedra tool and simulation tool.

If a probability of 0 or 1 is returned, the satisfaction set is normally empty or covers all general transition firing times, respectively. Hence, the model checker does not need to actually compute the probability, but immediately outputs 0 or 1. This is the reason for the virtually identical computation times in the cases 2$a$), 3$a$), and 4$a$).

For the other configurations, the number of iterations has a serious impact on the overall computation time. Thus, only if the resulting probabilities were to change drastically, the increased costs would be justified. However, as Table 8.7 illustrates, the precision gain is often negligible or even non-existent.

In series 1 and 2, the computed probabilities remain unchanged, so that there is no benefit from an increased number of iterations. The probabilities in series 4, on the other hand, are affected by the different settings. However, the change occurs only after the fourth decimal place, so that the precision gain is relatively low compared to the increase in the computation times. The same holds for the configurations in series 3. In the outlier case 3$b$) it can be observed, that the error reduces as expected with the increasing number of iterations. Raising the iteration count even more, improves the precision, but multiplies the computation time, as well. For example, if 65536 iterations are chosen, we receive a probability of 0.999802 in over 800 seconds. Compared to the actual probability of 1, the error is only 0.000192 instead of 0.00206 for 8192 iterations.

All in all, Table 8.6 and Table 8.7 show, that the selected default precision is a good trade-off between precision and performance. A much higher number of iterations might improve the accuracy of the computed probabilities a lot, but one has to keep the costs for the better results in mind.

| case ID | Nef polyhedra tool probability | | |
|---|---|---|---|
| | 1024 iterations | 4096 iterations | 8192 iterations |
| 1$a$) | 0.902575 | 0.902575 | 0.902575 |
| 1$b$) | 0.855578 | 0.855578 | 0.855578 |
| 1$c$) | 0.701629 | 0.701629 | 0.701629 |
| 1$d$) | 0.741137 | 0.741137 | 0.741137 |
| 2$a$) | 0 | 0 | 0 |
| 2$b$) | 0.864322 | 0.864322 | 0.864322 |
| 2$c$) | 0.666744 | 0.666744 | 0.666744 |
| 3$a$) | 1 | 1 | 1 |
| 3$b$) | 1.0105 | 0.998546 | 1.00206 |
| 3$c$) | 0.03125 | 0.03125 | 0.03125 |
| 3$d$) | 0.0321557 | 0.0316706 | 0.0312642 |
| 3$e$) | 0.0323329 | 0.031387 | 0.0314433 |
| 4$a$) | 1 | 1 | 1 |
| 4$b$) | 0.968357 | 0.968631 | 0.968528 |
| 4$c$) | 0.968537 | 0.968531 | 0.968528 |
| 4$d$) | 0.968537 | 0.968531 | 0.968528 |

Table 8.7: Comparison of probabilities computed by the Nef polyhedra tool with different numbers of iterations for the numerical integration.

# Chapter 9

# Conclusion

In this thesis, a novel approach to the model checking of hybrid Petri nets with general one-shot transitions has been introduced. The described algorithms allow to test safety-critical systems, that can be modelled as HPnGs, for their reliability or several other important properties.

For this purpose, the HPnG formalism, the Stochastic Time Logic, and the representation of an HPnG's state space as an STD have been presented. It has been shown, that an STD consists of a set of Nef polyhedra, which is a special class of polytopes, that are defined by intersections of half-spaces. Based upon the structure of STDs, algorithms have been developed to model check STL properties using geometric operation on Nef polyhedra. We have seen, that the described model checking process strongly differs for atomic and compound formulae and time bounded Until formulae, since the satisfaction of an Until formula depends on a time interval, whereas an atomic or a compound formula is satisfied at a point in time.

After having briefly looked into the implementation of the model checking algorithms for HPnGs with two general one-shot transitions, their applicability has been investigated in a case study consisting of two-scenarios. With the help of a simple HPnG, the first case illustrated how the model checking is carried out for different types of STL formulae. In the second case, a model for the charging process of an electric vehicle has been examined using our implementation. The computed probabilities for the satisfaction of the processed properties have been validated with two other tools, that can model check HPnGs with two general transitions, as well. The results have confirmed, that our approach is applicable to HPnGs that model real-world scenarios with a fairly large state space.

Compared to the other tools, the advantage of our program is that it not only computes the probability for the satisfaction of an STL formula, but also the complete satisfaction set. However, due to the available geometric data structures, the implementation of our approach is currently limited to HPnGs with two general transitions, although, in principle, the algorithms are applicable to arbitrary HPnGs. A future goal is hence to extend our program to be able to handle HPnGs with any number of general transitions. This requires, on the one hand, an implementation of Nef polyhedra in arbitrary dimensions, and, on the other hand, a more sophisticated technique for the integration over the (then equally multi-dimensional) satisfaction set. Especially the former requirement can not be fulfilled at the moment, though, due to the lack of suitable Nef polyhedra implementations.

# Bibliography

[1] Ahmadian, A., Sedghi, M., Aliakbar-Golkar, M., Fowler, M., and Elkamel, A. Two-layer optimization methodology for wind distributed generation planning considering plug-in electric vehicles uncertainty: A flexible active-reactive power approach. *Energy Conversion and Management 124* (2016), 231 – 246.

[2] Akhavan-Rezai, E., Shaaban, M. F., El-Saadany, E. F., and Karray, F. Online intelligent demand management of plug-in electric vehicles in future smart parking lots. *IEEE Systems Journal 10*, 2 (June 2016), 483–494.

[3] Alur, R., and Dill, D. L. A theory of timed automata. *Theor. Comput. Sci. 126*, 2 (Apr. 1994), 183–235.

[4] Alur, R., Feder, T., and Henzinger, T. A. The benefits of relaxing punctuality. *J. ACM 43*, 1 (Jan. 1996), 116–146.

[5] Asarin, E., Maler, O., and Pnueli, A. Reachability analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science 138*, 1 (1995), 35 – 65.

[6] Bagnara, R., Hill, P. M., and Zaffanella, E. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program. 72*, 1-2 (June 2008), 3–21.

[7] Baier, C., and Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[8] Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. *Computational Geometry: Algorithms and Applications*, 3rd ed. ed. Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.

[9] Bernstein, G., and Fussell, D. Fast, exact, linear booleans. In *Proceedings of the Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2009), SGP '09, Eurographics Association, pp. 1269–1278.

[10] Bi, Z., Kan, T., Mi, C. C., Zhang, Y., Zhao, Z., and Keoleian, G. A. A review of wireless power transfer for electric vehicles: Prospects to enhance sustainable mobility. *Applied Energy 179* (2016), 413 – 425.

[11] Bieri, H. *Nef Polyhedra: A Brief Introduction.* Springer Vienna, Vienna, 1995, pp. 43–60.

[12] Brönnimann, H., Fabri, A., Giezeman, G.-J., Hert, S., Hoffmann, M., Kettner, L., Pion, S., and Schirra, S. 2D and 3D linear geometry kernel. In *CGAL User and Reference Manual*, 4.8.1 ed. CGAL Editorial Board, 2016.

[13] Caflisch, R. E. Monte carlo and quasi-monte carlo methods. *Acta Numerica 7* (1998), 1–49.

[14] Calafiore, G., and El Ghaoui, L. *Optimization Models.* Control systems and optimization series. Cambridge University Press, October 2014.

[15] Capasso, V., and Bakstein, D. *An Introduction to Continuous-Time Stochastic Processes: Theory, Models, and Applications to Finance, Biology, and Medicine.* Modeling and Simulation in Science, Engineering and Technology. Springer New York, 2015.

[16] CGAL Project, The. *CGAL User and Reference Manual*, 4.8.1 ed. CGAL Editorial Board, 2016.

[17] Cuijpers, P. J. L., Reniers, M. A., and Engels, A. G. Beyond zeno-behaviour. Tech. rep., ITU-T Recommendation G.723.1, 2001.

[18] David, R., and Alla, H. On hybrid petri nets. *Discrete Event Dynamic Systems 11*, 1 (2001), 9–40.

[19] David, R., and Alla, H. *Discrete, Continuous, and Hybrid Petri Nets*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[20] Dobrindt, K., Mehlhorn, K., and Yvinec, M. *A complete and efficient algorithm for the intersection of a general and a convex polyhedron.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 314–324.

[21] Edelsbrunner, H. *Algorithms in Combinatorial Geometry.* Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[22] Everitt, B. S., and Skrondal, A. *The Cambridge Dictionary of Statistics; 4th ed.* Cambridge University Press, Leiden, 2010.

[23] Fabri, A., and Pion, S. Geomview. In *CGAL User and Reference Manual*, 4.8.1 ed. CGAL Editorial Board, 2016.

[24] Fogel, E., Setter, O., Wein, R., Zucker, G., Zukerman, B., and Halperin, D. 2D regularized boolean set-operations. In *CGAL User and Reference Manual*, 4.8.1 ed. CGAL Editorial Board, 2016.

[25] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[26] Gawrilow, E., and Joswig, M. polymake: a framework for analyzing convex polytopes. In *Polytopes — Combinatorics and Computation*, G. Kalai and G. M. Ziegler, Eds. Birkhäuser, 2000, pp. 43–74.

[27] Ghasemieh, H., Remke, A., Haverkort, B., and Gribaudo, M. *Region-Based Analysis of Hybrid Petri Nets with a Single General One-Shot Transition.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 139–154.

[28] Ghasemieh, H., Remke, A., and Haverkort, B. R. Survivability evaluation of fluid critical infrastructures using hybrid petri nets. In *IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013* (2013), IEEE Computer Society, pp. 152–161.

[29] Ghasemieh, H., Remke, A., and Haverkort, B. R. Hybrid petri nets with multiple stochastic transition firings. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools* (ICST, Brussels, Belgium, Belgium, 2014), VALUETOOLS '14, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 217–224.

[30] Granados, M., Hachenberger, P., Hert, S., Kettner, L., Mehlhorn, K., and Seel, M. *Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, and Implementation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 654–666.

[31] Gribaudo, M., and Remke, A. 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* (Washington, DC, USA, 2010), HASE '10, IEEE Computer Society, pp. 84–93.

[32] Gribaudo, M., and Remke, A. Hybrid petri nets with general one-shot transitions. *Performance Evaluation 105* (2016), 22 – 50.

[33] Grünbaum, B. *Convex Polytopes*, 2nd ed. Graduate Texts in Mathematics. Springer, 2003.

[34] Hachenberger, P., and Kettner, L. Boolean operations on 3d selective nef complexes: Optimized implementation and experiments. In *Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling* (New York, NY, USA, 2005), SPM '05, ACM, pp. 163–174.

[35] Hachenberger, P., and Kettner, L. 3D boolean operations on nef polyhedra. In *CGAL User and Reference Manual*, 4.8.1 ed. CGAL Editorial Board, 2016.

[36] Holoborodko, P. Numerical integration. `http://www.holoborodko.com/pavel/numerical-methods/numerical-integration/`, 2010. Last Access: 2016-09-26.

[37] Hossain, M., Madlool, N., Rahim, N., Selvaraj, J., Pandey, A., and Khan, A. F. Role of smart grid in renewable energy: An overview. *Renewable and Sustainable Energy Reviews 60* (2016), 1168 – 1184.

[38] Hurley, P. *A Concise Introduction to Logic.* Wadsworth, 2012.

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

[40] International Standards Organization, and International Electrotechnical Commission. ISO/IEC 14977:1996(E) First edition – Information technology – Syntactic metalanguage – Extended BNF. Tech. rep., International Standards Organization, 1996.

[41] International Standards Organization, and International Electrotechnical Commission. *ISO/IEC 14882:2011, Programming languages – C++*, first ed. American National Standards Institute (ANSI), Sept. 2011.

[42] Kang, J., Duncan, S. J., and Mavris, D. N. Real-time scheduling techniques for electric vehicle charging in support of frequency regulation. *Procedia Computer Science 16* (2013), 767 – 775.

[43] Knuth, D. E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd Edition*, 3ed. ed. Addison-Wesley Professional, 1997.

[44] Kwiatkowska, M., Norman, G., Segala, R., and Sproston, J. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science 282*, 1 (2002), 101 – 150. Real-Time and Probabilistic Systems.

[45] Leconte, C., Barki, H., and Dupont, F. Exact and Efficient Booleans for Polyhedra. Tech. Rep. RR-LIRIS-2010-018, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon, Oct. 2010.

[46] Lindner, M. libconfig – C/C++ Configuration File Library. `http://www.hyperrealm.com/libconfig/`, 2013. Last Access: 2016-10-20.

[47] Milton, S., and Schmidt, H. W. Dynamic dispatch in object-oriented languages. Tech. rep., CSIRO – Division of Information Technology, 1994.

[48] Nef, W. *Beiträge zur Theorie der Polyeder: mit Anwendungen in der Computergraphik*. Beiträge zur Mathematik, Informatik und Nachrichtentechnik. Lang, 1978.

[49] Nickovic, D., and Maler, O. *AMT: A Property-Based Monitoring Tool for Analog Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 304–319.

[50] Object Management Group (OMG). Unified Modeling Language (UML) Specification, Version 2.5. `http://www.omg.org/spec/UML/2.5/`, 2015. Last Access: 2016-10-20.

[51] Petri, C. A. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[52] Pilch, C. Development of an event-based simulator for model checking hybrid petri nets with random variables. Master's thesis, Wesfälische Wilhelms-Universität Münster, 2016.

[53] Posamentier, A., and Bannister, R. *Geometry, Its Elements and Structure: Second Edition.* Dover Books on Mathematics. Dover Publications, 2014.

[54] Postema, B., Remke, A., Haverkort, B. R., and Ghasemieh, H. Fluid survival tool: A model checker for hybrid petri nets. In *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance* (Cham, Switzerland, March 2014), vol. 8376 of *Lecture notes in computer science*, Springer International Publishing, pp. 255–259.

[55] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3 ed. Cambridge University Press, New York, NY, USA, 2007.

[56] Schöbel, A. *Locating Lines and Hyperplanes: Theory and Algorithms.* Applied Optimization. Springer US, 1999.

[57] Tammik, J. Autocad nef polyhedron implementation, 2007.

[58] Weisstein, E. W. Dimension. From MathWorld—A Wolfram Web Resource. Last access: 2016-09-26.

[59] Yap, C. *Robust geometric computation*, 2 ed. Chapman and Hall/CRC, 2004, pp. 927–952.

[60] Yap, C.-K. Towards exact geometric computation. *Computational Geometry 7*, 1 (1997), 3 – 23.

[61] Ziegler, G. *Lectures on Polytopes.* Graduate Texts in Mathematics. Springer New York, 2012.

## Plagiatserklärung der / des Studierenden

Hiermit versichere ich, dass die vorliegende Arbeit über ______________________________
__________________________ 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 entnommenen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

_______________________________________
(Datum, Unterschrift)

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.

_______________________________________
(Datum, Unterschrift)