University of Twente

Faculty of Electrical Engineering, Mathematics & Computer Science

Formal Methods and Tools & Design and Analysis of Communication Systems

Master Computer Science: Methods and Tools for Verification

Date: 28 August, 2013

Master thesis

Fluid Survival Tool: A model checker for Hybrid Petri nets

by Björn F. Postema, BSc.

Supervisors

Dr. A.K.I. Remke
Prof.dr.ir. B.R.H.M. Haverkort
Dr.ir. R. Langerak
H. Ghasemieh, MSc.

Colossians 2:2-3, NIV
"My goal is that they may be encouraged in heart and united in love, so that they may have the full riches of complete understanding, in order that they may know the mystery of God, namely, Christ, in whom are hidden all the treasures of wisdom and knowledge."

Acknowledgments

I would like to thank my supervisors Anne Remke for her honest constructive feedback, valuable advices and encouraging words, Hamed Ghasemieh for his cooperation, feedback and taking time for my many questions, Boudewijn Haverkort for his helpful comments and sharp remarks and Rom Langerak for his feedback and cooperation. I would like to thank my father and mother for their support throughout my entire study in prayer, compassion and provision. I would like to thank my brothers, sisters and friends for lending a sympathetic ear, encouraging me with their words and prayers. And above all I would like to thank God, my Father, for making all things possible and for showing me love every day, mostly through my great example Jesus Christ, who is giving my being peace, protection, restoration, hope, courage and joy in every situation.

Fluid Survival Tool: A model checker for Hybrid Petri nets

Björn F. Postema

Faculty of Electrical Engineering, Mathematics & Computer Science
Formal Methods and Tools & Design and Analysis of Communication Systems
University of Twente
2013

ABSTRACT

Recently, algorithms for model checking Stochastic Timed Logic (STL) on Hybrid Petri nets with a general one-shot transition (HPNG) have been introduced. Currently, an actual tool is being developed for model checking HPNG models against STL formulas. A graphical user interface (GUI) helps to demonstrate and validate existing algorithms. Additionally, the tool gives insight into model checking by generating a Stochastic Timed Diagram. Moreover, from the output of the model checker 2D and 3D plots can be generated for the transient probability distributions to be in a state that fulfils a certain property. An extendable object-oriented tool design with a GUI has been carried out that uses the Model-View-Controller and Facade patterns, Doxygen for documentation and Qt for GUI development in C++. Furthermore, an approach for the general case of model checking formulas has been developed, which is based on generating and traversing the data structure of an Abstract Syntax Tree. Still, the general case offers more challenges for example with respect to nesting inside until formulas. Moreover, in cooperation with bachelor students from the mathematics department, the implementation and algorithms have been tested and validated with simulations on an elaborated case study of sewage water cleaning. Additionally, the number of continuous variables in this case study has been scaled in this master thesis to show the feasibility of the approach.

Table of Contents

Chapter 1

Introduction

Our lives are fragile, so are systems. Many systems do not function as intended. Most of us are not surprised any more when a system produces errors. We try to keep the systems in a state that satisfies the intended purpose of the system. Since exploring all states of a complex system is exhaustive, model checking provide automated answers to questions about the satisfaction of the intended purpose of a system. To be more precise, model checking checks automatically whether a model of a system meets a given specification.

In this master thesis a model checking tool is introduced to analyse *survivability*, which is the ability of a system to recover to a predefined service level. The tool is introduced with an extendable object-oriented tool design including a graphical user-interface. Recently, [20] and [21] proposed model checking algorithms for the exact analysis of Hybrid Petri nets. Hybrid Petri nets enable modelling of various system and are useful for designing critical infrastructures. The tool has data structures and algorithms following these model checking algorithms. Moreover, a general model checking approach is introduced that glues these algorithms together and adapts them to the tool. Furthermore, in cooperation with bachelor students from the mathematical department the tool has been used on a case study for validation of simulation that provide valuable insight in the feasibility of the tool. In the end, a case study shows that the tool can handle also larger models.

1.1 Motivation

Critical infrastructures [7] like electricity, gas, oil, telecommunication, water, health, transportation, finances and security can only be tested at very high cost. For instance, a complex system of a dam is tested for the critical situation that under certain conditions only one gate in the dam may be opened. The cost to test this system are high, because many settings need to be tested which is exhaustive and trivially recovery costs of the system are very high, especially when a system under test fails. Therefore, modelling of critical systems and the analysis of such a model reduces costs, since these do not influence the system directly.

In literature there are more model checking and analysis tools for Hybrid Petri nets such as Oris [45], GRIF Petri [48] and SimHPN [34]. However, a tool for model checking Hybrid Petri nets with one general one-shot transition (HPNG) models, which allows to model continuous and discrete variables of systems with a single stochastic event, against Stochastic Time Logic (STL) specification, which is a language to formulate analytical questions about HPNG models, does not exist. Such a tool provides an advantage in the field of Petri nets, because the methods by [26] and [20] can be used to analyse systems with an arbitrary number of continuous variables and one stochastic transition. This makes the tool useful in the area of critical infrastructures where repair actions often take a randomly distributed amount of time.

Moreover, recent news [49], [44] and [51] reports that on 20 June 2013 homes and streets in Enschede, the Netherlands, were flooded by sewage water. The tool introduced in this master thesis has successfully been used to analyse HPNG models with STL specification for a water treatment system [36]. This case study is used for validation and analysis of the scalability of the models.

1.2 State of the art

HPNGs allow to model critical infrastructures with fluid flows like water, gas and oil distribution. Since critical infrastructures may fail, survivability is an important quality for these systems. The analysis of survivability properties can be investigated for HPNGs as shown in [26].

In [20], [21] the theory of STL is introduced. However, the theory is too coarse to conclude an algorithm for the general case of model checking. This master thesis contributes to the theory of model checking by combining the until operator as introduced in [21] with the simple STL introduced in [20] by proposing a data structure of the parse tree that is traversed to evaluate a general STL formula.

1.3 Research approach

This research aims at introducing a tool called Fluid Survival Tool (FST) with an extendable software design based on software engineering principles and a region-based analysis algorithm for model checking and analysing HPNG models against STL formulas. Data structures are used that support traversal through STL formulas in a structural way in order to support the evaluation of the STL formulas with the existing algorithms.

1.4 Research questions

Since this research aims at introducing a model checking tool for HPNG models against STL specification the following main research question arises:

How to design, implement and validate an extendable object-oriented tool for model checking HPNGs models against general STL specification with appropriate software engineering principles?

This main question leads to the following sub questions:

- What is the state of the art with respect to literature and implementations and where is the model checker located in literature?
  - What are the syntax and semantics of HPNGs?
  - How are HPNGs analysed, especially with the aid of STL?
  - What is the state of the algorithms and implementations?
  - How is the tool embedded in literature and among similar model checkers?
- What is the software design for model checking tool for HPNGs?
  - What are the functional and non-functional requirements of the design?
  - How is the conceptual design of the tool?
  - How is the detailed design of the tool?
- How to implement the model-checking algorithms for full STL and the probability operator for this tool?
  - What data structures are required to support the algorithms?
  - How to connect the implementation code and the algorithms?
- What is an interface for the input and the output?
  - What are the input representations?
  - What are the output representations?
- What interesting case study shows the feasibility of the model checking tool?
  - How to show the feasibility of the model-checking tool?
  - What is an interesting case study for investigating the feasibility?
  - How does the case study show the feasibility of the model checking tool?

1.5 Thesis outline

The master thesis is split into three main parts: *Background*, *Tool Development* and *Case study & Conclusions*. *Background* summarizes definitions, algorithms, representations and related work in the following chapters:

Chapter 2 explains the basics of Petri nets that are required to understand the full definitions of Hybrid Petri nets with a general one-shot transition elaborated in Chapter 3. Next, the representation of the evolution of the system is described in Chapter 4.

In the *Tool Development* part the development process of the tool is described by elaborating the domain analysis and requirements in Chapter 8. The environment of the tool is described in Chapter 9. A global and detailed design for the architecture of the tool is introduced with several software engineering principles that are explained and applied in Chapter 10. The general case for model checking is explained and applied to the tool in Chapter 11. An manual installation and usage of the graphical user-interfaces is elaborated in Chapter 12.

The part *Case Study & Conclusions* discusses a case study to show the feasibility of the tool with models of a sewage water cleaning facility in Chapter 13. Chapter 14 answers the research questions, summarizes findings and provide suggestions for future work.

Part I

Background

Chapter 2

Petri nets

Petri nets (PNs) allow to model a variety of phenomena. They have been successfully used to model and visualize systems with parallelism, concurrency, synchronization and resource sharing. The models of interest in this research are *Hybrid Petri nets with general one-shot transitions* (HPNGs). In order to comprehend HPNGs, some simpler PNs are explored in this chapter. First, *Discrete PNs* (DPNs) *with immediate and deterministic transitions* are discussed. Since there is a variety of PNs, additional expressiveness of PNs is explained. The reason for this variety is based on the variety of modelled systems.

The definitions as given in this chapter are inspired by [10].

Section 2.1 describes the syntax of DPNs. Section 2.2 and Section 2.3 continues with the behaviour of these discrete DPNs. Section 2.4 reasons about properties of PNs as foundation for Section 2.5 that discusses additional expressiveness for PNs.

2.1 Syntax of Discrete Petri nets

In order to understand DPNs, its syntax and an example of DPNs are discussed in this section. First, the syntax of a DPN is defined as a structure with five elements as follows:

Definition 2.1 (Discrete Petri net)**.** A DPN consists of a 5-tuple $\langle \mathcal{P}^D, \mathcal{T}, \mathcal{A}^D, \boldsymbol{m_0}, \phi_d^{\mathcal{T}} \rangle$. The finite set of discrete places are represented as $\mathcal{P}^D$. The finite set of transitions $\mathcal{T} = \mathcal{T}^I \cup \mathcal{T}^D$ are composed of *immediate transitions* and *deterministic transitions*. The finite set of discrete arcs $\mathcal{A}^D$ are directed, so a source and a target for each arc is defined: $\mathcal{A}^D \subseteq (\mathcal{P}^D \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P}^D)$. Arcs only go from either a place to a transition or from a transition to a place. A discrete place $P \in \mathcal{P}^D$ contains a discrete and finite number of tokens which is also called a marking $m(P) \in \mathbb{N}$. The marking $\boldsymbol{m}$ of such a Petri net is a vector. A convenient way of describing such a marking vector is $\boldsymbol{m} = (m(1), ..., m(k)) = [m(1), ..., m(k)]^T$, where k is equal to the number of places, $\boldsymbol{m}$ contains the number of tokens of each place in the DPN, $\boldsymbol{m_0}$ is the initial marking of the DPN and represents the initial distribution of the DPN and T transposes the matrix $[m(1), ..., m(k)]$. The function $\phi_d^{\mathcal{T}} : \mathcal{T}^D \to \mathbb{R}^+$ assigns a constant deterministic firing time to the discrete transitions. Immediate transitions are special deterministic transitions with a firing time of zero. ■

Since a DPN has a graphical representation, all the elements of a DPN are illustrated in Figure 2.1. A discrete place is depicted as a white circle with a black border. A discrete arc is represented as an arrow, where the tail is connected to the source and the arrowhead to the target. An immediate transition is depicted as a black rectangle. A deterministic transition is represented by a rectangle with a grey rectangle with a black border. A discrete place contains a marking which is depicted by black dots inside the white circle. Transitions determine incoming and outgoing number of tokens during system evolution by associating a discrete place to a immediate or deterministic transition and vice versa with the discrete arc.

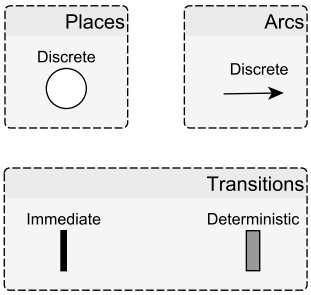

Fig. 2.1: The elements of a discrete PN

In addition to the illustration of the elements of DPNs, Example 2.2 is given to show a DPN of a water pump that is regularly switched off for maintenance.

Example 2.2 (Pump maintenance)**.** A critical system contains a new pump that is switched off two consecutive times after a period of 12 hours for maintenance by a technician. There is a skilled technician who needs 1 hour to maintain the pump, so the technician will turn the pump back on after exactly 1 hour. This skilled technician is hired for one maintenance of the pump. So, when the pump is switched off the second time the technician is not available any more. However, the pump requires maintenance another time.

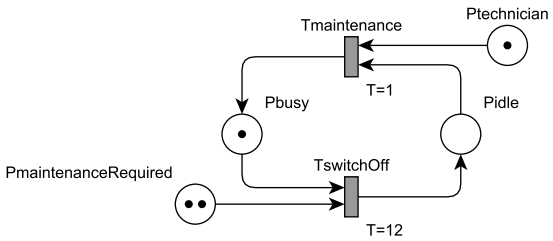

Fig. 2.2: Pump maintenance

Recall the representations of the elements of a DPN, Figure 2.2 shows a DPN for pump maintenance. The places *Ptechnician*, *Pidle*, *Pbusy*, *PmaintenanceRequired* $\in \mathcal{P}^D$ are deterministic places. The transitions *Tmaintenance*, *TswitchOff* $\in \mathcal{T}$ are deterministic transitions. The deterministic transitions display the firing time on the bottom right of the deterministic transition, e.g. $T = 1$. The number of tokens is either represented with dots or represented with a string of an element of the natural numbers. For this example, $m(2) = 0$, $m(1) = m(3) = 1$ and $m(4) = 2$. So, this PN has the marking $\boldsymbol{m} = (1, 0, 1, 2)$.

The labels show additional information, but are not in the definition. In this case the labels display the names for each place (e.g. *Ptechnician*) and each transition (e.g. *Tmaintenance*).

In Figure 2.2 the pump is initially busy, hence, *Pbusy* contains one token. *Pidle* only has a token when the pump is idle, because it requires maintenance. *Ptechnician* limits the model for the requirement that the technician is hired for only one maintenance. In the same way, *PmaintenanceRequired* limits the amount of times the pump is switched off for maintenance. Transition *Tmaintenance* waits 1 hour ($T = 1$) before firing a token, because then the technician turns the pump back on. Transition *TswitchOff* waits 12 hours ($T = 12$) before firing the token. ■

2.2 Events in Discrete Petri nets

Having discussed the syntax of DPNs, the events of DPNs are explained in this section. In more detail, events are caused by transitions and result in a change in the marking of the system.

Immediate transitions are *firable* or *enabled* if each of the input arcs of this transition contains at least one token. So, transitions with no input arcs are always enabled. If a transition fires, then the transition withdraws one token from the source places of input arcs and adds one token to each of the places of the output arcs. Deterministic transitions are firable if each of the input arcs of this transition contain at least one token *and* the deterministic firing time corresponding to that transition is reached.

Example 2.3 (Firing transitions)**.** Consider the DPN of Figure 2.2 with an initial marking $\boldsymbol{m_0} = (1, 0, 1, 2)$. Transition $T1$ is the only transition enabled. If $T1$ fires, then the marking becomes $\boldsymbol{m_1} = (0, 1, 1, 1)$. Next, $T2$ is enabled and when $T2$ is fired the marking becomes $\boldsymbol{m_2} = (1, 0, 0, 1)$. ■

2.3 States of Discrete Petri nets

After firing transitions the marking of the system changes, so this information should be stored. A *state* represents a snapshot of the system at a given time. The snapshot contains information that describe the changes in the system. This information in a DPN is the markings and a clock for the deterministic transition. So, the state consist of two elements as follows:

Definition 2.4 (States of a Discrete Petri net)**.** The state of an DPN is defined by the 2-tuple $\Gamma = \langle \boldsymbol{m}, \boldsymbol{c} \rangle$. $\boldsymbol{m}$ is a vector of markings for the discrete places. Vector $\boldsymbol{c} = (c_1, \ldots, c_{|\mathcal{T}^D|})$ contains a clock $c_i \in \mathbb{R}^+$ for each deterministic transition that represents the time that $Ti \in \mathcal{T}^D$ has been enabled. So, the initial state of the system is $\Gamma_0 = \langle \boldsymbol{m_0}, \boldsymbol{0} \rangle$, where $\boldsymbol{m_0}$ represents the initial marking of the system and $\boldsymbol{0}$ is the zero vector. ■

2.4 Properties of Petri nets

Before considering additional expressiveness for PNs, some PN properties are discussed in this section. These properties lead to extensions and simplifications of PNs in Section 2.5. In general, the PN properties hold for all types of PNs.

A PN is called *autonomous* when the firing instants are either unknown or not indicated. A *non-autonomous* PN is *timed* and/or *synchronized*. For instance, a DPN is non-autonomous, since it is timed.

A *conflict* occurs between two or more transitions when the source of the input arcs of two or more transitions is the same place. If two or more transitions are enabled from a place with one token, the system does not know which transition should be fired.

Example 2.5 (Conflict)**.** The PN of Figure 2.3 has a conflict, since transition $T1$ and $T2$ are both enabled, but the system does not know which transition the PN should be fired.

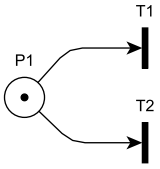

Fig. 2.3: A Petri net with conflict

■

Heuristics, such as prioritizing, can be applied to solve conflicts in PNs. The choices are then based on the priority of transitions. Section 2.5 discusses this in more detail.

Given a PN and an initial marking, the PN is a *live* PN if it has no transitions that cannot be fired. So a live PN will never reach a state, that from that state on a transition can never be fired in a future state. Given a PN and an initial marking, the PN is a *deadlock free* PN if it has no state in which there is no transition that is firable. If such state exists, then in that state a *deadlock* occurs. So the difference between a live and deadlock free PN is that all transitions in live PNs are always fireable, while in a deadlock free PN at least one transition should always be firable.

Example 2.6 (Live and deadlock free)**.** Consider the DPN of Figure 2.2 and the initial marking $\boldsymbol{m_0} = (1, 0, 0, 1)$. The next state is $\boldsymbol{m_1} = (0, 1, 0, 0)$. From this state no transition can be fired, since there is no reparation equipment available. So, in this state a deadlock occurs, because there are no transitions that can be fired.

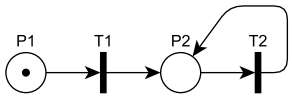

Fig. 2.4: A PN that is not a live PN, but it *is* deadlock free

Consider the example PN as shown in Figure 2.4. The initial marking is $\boldsymbol{m_0} = (1, 0)$. The set of future states originating from $\boldsymbol{m_0}$ is $\{m_i = (0, 1) \mid i > 0\}$. So transition $T1$ cannot be fired after the second state. The PN is deadlock free, since $T2$ can always be fired in all future states. The PN is not live, since in the second state $T1$ cannot be fired in a future state. ■

A PN is *reversible* if a return to the initial marking is always possible.

Example 2.7 (Reversible)**.** Consider the PN of Figure 2.4 and the initial marking $\boldsymbol{m_0} = (1, 0)$. This PN is not reversible to $\boldsymbol{m_0}$. Recall from Example 2.6 that the future states of the system are $\{m_i = (0, 1) \mid i > 0\}$, the system cannot return to the initial marking. Assume a different initial marking for the same PN: $\boldsymbol{m_0} = (0, 1)$, then the initial marking is repeated after every other state. Therefore a return to the initial marking is always possible, hence, the PN is reversible. ■

Definition 2.1 states that the number of tokens of each places are an element of the natural numbers. A *bounded* PN has a finite number of tokens. The maximum number of tokens in the PN is n iff the PN is *n-bounded*. A *safe* PN has at most one token in each place.

Example 2.8 (Bounded and safe)**.** Figure 2.4 shows a safe PN with the initial marking $\boldsymbol{m_0} = (1, 0)$. When the initial marking is $\boldsymbol{m_0} = (2, 0)$, then we say the PN is 2-bounded. ■

2.5 Abbreviations and extensions

The properties of PNs as discussed in Section 2.4 lead to abbreviations and extensions for PNs. Abbreviations are simplified representations of PNs and extensions are PNs with additional functionalities. Abbreviations are useful to simplify the graphical representations, i.e. abbreviations provide syntactic sugar for PNs. Mostly, abbreviations are realized by reducing the number of elements with simple additions. The extensions enable a greater number of applications for PNs.

This section only discusses the abbreviations and extensions that lead to the HPNG model. Therefore, this section discusses the abbreviations that lead to *generalized* PNs and *finite capacity* PNs. Furthermore, this section describes the extensions that lead to *extended* PNs, *priority* PNs and *continuous* PNs.

A *generalized* PN has weights associated to the arcs. Intuitively, generalized PNs allow transitions only to fire batches of tokens with the size of a weight. Arc weights are defined as a function as follows:

Definition 2.9 (Arc weights)**.** The function $\phi_w^{\mathcal{A}}$ assigns a weight to an arc of a PN.
The function $\phi_w^{\mathcal{A}} : \mathcal{A} \to \mathbb{N}_{>0}$ maps arcs to the natural numbers greater than zero. Let Ti be a transition and Pj be a place for some PN. Let a be an arc from Ti to Pj and let b be an arc from Pj to Ti with the weights given by the function: $\phi_w^{\mathcal{A}}(a) = k$ and $\phi_w^{\mathcal{A}}(b) = l$ where $k, l \in \mathbb{N}_{>0}$. Then the transition Ti will only be enabled if place Pi has at least k tokens. When the transition Ti is fired the system subtracts k tokens from Pj and adds l tokens to Pj. ■

The graphical representation of the weights is a string of a strictly positive integer. When no string is displayed for an arc this means the weight is 1. Arc weights simplify the representation of PNs.

Example 2.10 (Arc weights)**.** Figure 2.5 shows a small PN with weights on the arcs with two transitions with a weight of two.

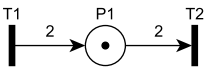

Fig. 2.5: A Petri net with weights

The initial marking is $\boldsymbol{m_0} = (1)$. Transition $T2$ is not enabled, since $P1$ does not contain 2 tokens. Transition $T1$ is enabled, and when $T1$ fires the marking becomes $\boldsymbol{m_1} = (3)$. Now both transitions are enabled. So when $T2$ fires, then the marking becomes $\boldsymbol{m_2} = (1)$. ■

A *finite capacity* PN has capacities associated with the places. Intuitively, finite capacity PNs allows to limit the number of tokens in a place. If there exists a transition with an arc to a place with a capacity, the transition is only fired if it does not result in a number of tokens that is greater than the capacity of that place. A place with a capacity is called bounded.

Definition 2.11 (Place capacities)**.** The function $\phi_c^{\mathcal{P}}$ assigns a capacity to a place of a PN.
The function $\phi_c^{\mathcal{P}} : \mathcal{P} \to \mathbb{N}_{>0}$ maps the places to the natural numbers greater than zero. Let Pi be a place for some PN. The capacity of Pi is given by the function: $\phi_c^{\mathcal{P}}(Pi) = k$ where $k \in \mathbb{N}_{>0}$. Transition Tj and place Pi are connected by an arc with a weight w. Transition Tj will only be fired if the property $m(Pi) + w \leq k$ holds. If no weights are assigned to the places the default weight of 1 is used. ■

The graphical representation of capacities is written as a string of a strictly positive integer next to a place. If no string is displayed for an place this means that the capacity for that place is unbounded. Place capacities simplify the representation of PNs.

Example 2.12 (Place capacities)**.** Figure 2.6 shows a small PN with a capacity of 2 for place $P1$.

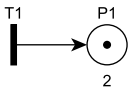

Fig. 2.6: A Petri net with weights

The initial marking is $\boldsymbol{m_0} = (1)$. Transition $T1$ is enabled, since the property $m(P1) + w \leq k \Rightarrow 2 \leq 2$ holds. If $T1$ fires, then $\boldsymbol{m_1} = (2)$. Now $T1$ is not enabled, since the next firing of $T1$ would result in that the property $m(P1) + w \leq k \Rightarrow 3 \leq 2$ does not hold. ■

An *extended* PN has a special type of arc, namely the inhibitor arc. Intuitively, such an arc can only fire when there is no token in the source place. An inhibitor always connects a place to a transition, and thereby puts an extra requirement on the enabling of the transitions.

Definition 2.13 (Inhibitor arcs)**.** Let $\mathcal{A}^D$ be a set of discrete arcs and $\mathcal{A}^I$ be a set of inhibitor arcs. Then $\mathcal{A} = \mathcal{A}^D \cup \mathcal{A}^I$ is the set of all arcs. Let $A \in \mathcal{A}^I$ be an inhibitor arc with a source place Pi and a target transition Tj. The transition Tj can only be fired if place Pi contains no tokens. If transition Tj fires no token is removed from place Pi. ■

In the graphical representation of the inhibitor arc the arrowhead is replaced by a small circle. The small circle points to the target transition. An inhibitor arc is an extension for PNs, because it adds additional functionality which could not be constructed before.

Example 2.14 (Inhibitor arcs)**.** Figure 2.7 shows a PN with an inhibitor arc.

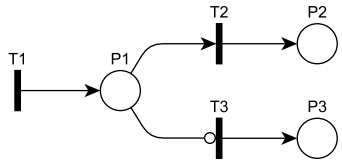

Fig. 2.7: A Petri net with weights

The initial marking is $\boldsymbol{m_0} = (0, 0, 0)$, and transitions $T1$ and $T3$ are enabled, since $T1$ is always firable and $P1$ contains no tokens. If $T3$ fires, then the next marking is $\boldsymbol{m_1} = (0, 0, 1)$, because $P1$ does not contain any token. $T1$ and $T3$ remain enabled. If $T1$ fires after $T3$, then $\boldsymbol{m_2} = (1, 0, 1)$. Only now $T1$ and $T2$ are enabled and $T3$ is no longer enabled, because $P1$ contains a token. ■

A *priority* PN assigns priorities to transitions to resolve conflicts between a number of enabled transitions. A prioritizing heuristic is then used to solve non-deterministic choices between transitions in conflict and hence avoids the exploration of an entire state space. Prioritizing is used when a specific firing order is desired or the firing order of the transitions does not matter, else the system in conflict needs to keep track of all states the system could be in. Eventually, the state space joins into the same state of the system when the firing order of the transitions does not matter.

Definition 2.15 (Priorities)**.** The function $\phi_p^{\mathcal{T}}$ assigns a unique priority to a transition of a PN. The function $\phi_p^{\mathcal{T}} : \mathcal{T} \to \mathbb{N}_0$ maps transitions to the natural numbers greater than or equal to zeros. Let Ti and Tj be a two transitions for some PN. The priorities of Ti and Tj are given by the assignments: $\phi_p^{\mathcal{T}}(Ti) = k_1$ and $\phi_p^{\mathcal{T}}(Tj) = k_2$ where $k_1 > k_2$ and $k_1, k_2 \in \mathbb{N}_0$. In some state both transitions are enabled. Then transition Ti will fire first. *Unique* priorities have the property $\forall P1, P2 \in \mathcal{P}$: $(\phi_p^{\mathcal{T}}(P1) = l_1 \wedge \phi_p^{\mathcal{T}}(P2) = l_2 \wedge l_1 = l_2) \Rightarrow (P1 = P2)$. ■

The marking of a *continuous* PN is a real positive number instead of an integer, furthermore, the continuous PNs are represented with continuous places, continuous transitions and continuous arcs. A state in continuous PNs is defined by a marking as follows:

Definition 2.16 (State of continuous Petri nets)**.** The state of a place is defined by $\boldsymbol{x} = (x(1), ..., x(k))$, where k is equal to the number of places. $x(i) \in \mathbb{R}_0^+$, where $i \in [1, k], i \subseteq \mathbb{N}_{>0}$, is the amount of fluid in a place. $x(i)$ is also denoted as $x(Pi)$. $\boldsymbol{x_0}$ is the initial amount of fluid in all continuous places. ■

Continuous places and continuous transitions have the same graphical representation as the discrete places and deterministic transitions with the exception that the continuous places have a double black border and the continuous transitions have a white rectangle with a double black border. A continuous arc is represented by a white arrow with a black border. Continuous transitions determine inflow and outflow during system evolution by associating a continuous place to a continuous transition and vice versa with the continuous arc. Continuous places, continuous transitions and continuous arcs are an extension for PNs, since these continuous elements provide additional functionality.

Example 2.17 (Continuous Petri nets)**.** Figure 2.8 shows a PN with one continuous place ($P1$) and two continuous transitions ($T1$ and $T2$) which are connected by two continuous arcs that connect $T1$ to $P1$ and $P1$ to $T2$.

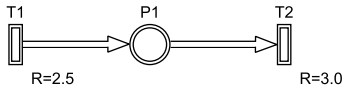

Fig. 2.8: A continuous Petri net

The initial amount of fluid is often not depicted graphically, so assume the initial amount of fluid is $\boldsymbol{x_0} = (10.0)$. The $P1$ has an inflow of 2.5, because of transition $T1$, and a outflow of 3.0, because of transition $T2$. Effectively 0.5 is subtracted each incrementation of the clock by 1, i.e. $\boldsymbol{x_1} = 9.5$, $\boldsymbol{x_2} = 9.0$, $\boldsymbol{x_2} = 8.5$ and so on. ■

Chapter 3

Hybrid Petri nets with a general one-shot transition

Having discussed discrete Petri nets with timed transitions, Hybrid Petri nets are explained in this chapter by combining definitions and adding several features. Hybrid Petri nets are a combination of discrete and continuous Petri nets with several abbreviations and extensions as explained in Section 2.5. Furthermore, a Hybrid Petri net contains a general one-shot transition (HPNG), which is a generally distributed transition that can only fire once during the evolution of the system.

Besides the definitions from Chapter 2, the definitions as given in this chapter are inspired by [20] and [25].

Section 3.1 describes the syntax of HPNGs that is illustrated with an example of a water tower in Section 3.2. Section 3.3 explains how a state of the system is stored.

3.1 Syntax of HPNGs

Recall that HPNGs are used to model critical infrastructures with fluid flows and that HPNGs are a combination of discrete and continuous Petri nets with several additions. HPNGs allow to model an arbitrary number of continuous places that are connected to continuous transitions via continuous arcs, and discrete places that are connected to deterministic, immediate and generally distributed transitions via discrete and inhibitor arcs and allow to control the continuous transitions via test arcs. A *test arc* allows a transition only to be enabled if there is at least one token in a discrete place. So, HPNGs allow to model critical infrastructures with a continuous part for the fluid flows and a discrete part for controlling the flows.

An HPNG combines the generalized, finite capacity, extended and priority PN according to Section 2.5 into one definition. An HPNG is an non-autonomous PN, since the PN is timed. So all types of transitions have a description of the moment of firing with respect to time. A hybrid PN has a continuous part and a discrete part. The definition shows how these two are combined. The general distributed transitions $\mathcal{T}^G$ are general one-shot transitions, because of the constraint that they can fire only once.

Next, the graphical representation for the elements of HPNGs is introduced:

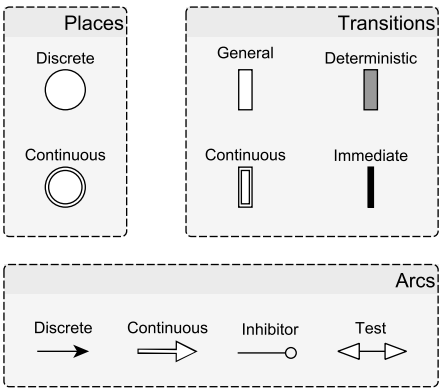

Fig. 3.1: The elements of an HPNG

Figure 3.1 shows the graphical representations of places, transitions and arcs of HPNGs. Recall the elements from Figure 2.1 for DPNs and the elements from Example 2.17 for continuous PNs, some new elements have been added. An inhibitor arc makes a transition only firable when there is no token in an associated place. An inhibitor arc is similar to the discrete arc except for the arrowhead is replaced by a small white circle with a black border. A test arc allows a transition only to be firable when there is at least one token in an associated discrete place. A test arc is similar to the discrete arc, except that the arrowhead and the tail are both white filled arrowheads.

Recall that an HPNG can have at most one general one-shot transition, which is a generally distributed transition that can only fire once during the evolution of the system. The firing of the one-shot transition follows an arbitrary continuous probability distribution that depends on a continuous random variable. A *random variable* is a variable that can take on a set of possible values, where each possible value is associated with a probability. *Continuous random variables* reason about sets of intervals, since there are infinitely many possible values that the random variable can take on. An *continuous probability distribution* is often associated with a probability density function that describe all the probabilities of the possible values that the random variable could take on.

In the following the definition of HPNGs is introduced:

Definition 3.1 (Hybrid Petri nets with general one-shot transition)**.** A HPNG is defined as a tuple $\langle \mathcal{P}, \mathcal{T}, \mathcal{A}, \boldsymbol{m_0}, \boldsymbol{x_0}, \Phi \rangle$.

The set of places $\mathcal{P} = \mathcal{P}^D \cup \mathcal{P}^C$ are divided into two disjoint sets $\mathcal{P}^D$ and $\mathcal{P}^C$ for the discrete and continuous places, respectively. The discrete marking $\boldsymbol{m}$ is a vector that represents the number of tokens $m(P)$ for each discrete place $P \in \mathcal{P}^D$ and the continuous marking $\boldsymbol{x}$ is a vector that represents the non-negative level of fluid $x(P) \in \mathbb{R}_0^+$ for each continuous place $P \in \mathcal{P}^C$. The initial marking is given by $M_0 = (\boldsymbol{m_0}, \boldsymbol{x_0})$.

The finite set of transitions $\mathcal{T} = \mathcal{T}^I \cup \mathcal{T}^D \cup \mathcal{T}^G \cup \mathcal{T}^C$ is composed of the set of immediate transitions, the set of deterministically timed transitions, the set of generally distributed transitions and the set of continuous transitions, respectively. Note that in this thesis as in [20] the number of general transitions is restricted to $|\mathcal{T}^G| = 1$.

The finite set of arcs $\mathcal{A} = \mathcal{A}^D \cup \mathcal{A}^C \cup \mathcal{A}^I \cup \mathcal{A}^T$ consists of the set of discrete input and output arcs, the set of continuous input and output arcs, the set of inhibitor arcs and the set of test arcs, respectively. $\mathcal{A}^D$ connects discrete places and discrete transitions. $\mathcal{A}^C$ connects continuous places and continuous transitions. $\mathcal{A}^I$ and $\mathcal{A}^T$ connect discrete places to all kinds of transitions.

The tuple $\Phi = \langle \phi_b^{\mathcal{P}}, \phi_p^{\mathcal{T}}, \phi_d^{\mathcal{T}}, \phi_f^{\mathcal{T}}, \phi_g, \phi_w^{\mathcal{A}}, \phi_s^{\mathcal{A}}, \phi_p^{\mathcal{A}} \rangle$ contains 8 functions. The function $\phi_b^{\mathcal{P}} : \mathcal{P}^C \to \mathbb{R}^+ \cup \infty$ assigns an upper bound to each continuous place as in Definition 2.11. The function $\phi_p^{\mathcal{T}} : \mathcal{T}^D \cup$

$\mathcal{T}^I \to \mathbb{N}_0$ assigns a unique priority to the discrete and immediate transitions with the interpretation of Definition 2.15. The function $\phi_d^{\mathcal{T}} : \mathcal{T}^D \to \mathbb{R}^+$ assigns a constant deterministic firing time to the discrete transitions. The function $\phi_f^{\mathcal{T}}$ assigns a constant nominal firing speed to the continuous transitions. The function $\phi_g(s)$ is the cumulative probability distribution function (CDF) for a random variable s, where s is the firing time of the general transition. The function $\phi_w^{\mathcal{A}} : \mathcal{A}^D \cup \mathcal{A}^T \to \mathbb{N}_{>0}$ assigns weights to the arcs with the interpretation of Definition 2.9. The function $\phi_p^{\mathcal{A}} : \mathcal{A}^C \to \mathbb{N}_0$ assigns a priority to the arcs. If there is a conflicts between some continuous arcs the arc with the highest priority gets all the fluid. If the priorities are equal, then a share function adapts the firing rates $\phi_s^{\mathcal{A}} : \mathcal{A}^C \to \mathbb{R}^+$ according to [8]. ■

3.2 Water tower example with an HPNG model

Recall that HPNGs are useful for analysis of critical infrastructure. For instance, an HPNG model for a water tower with a pump that might break is such a critical infrastructure. The elements of HPNGs as depicted in 3.1 can be used to construct a representation for various HPNGs such as the following example HPNG model that elaborates such a water tower:

Example 3.2 (Water tower)**.**

Fig. 3.2: An actual water tower [11]

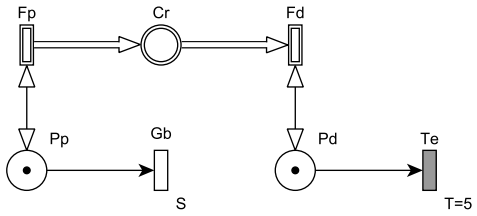

Fig. 3.3: A water tower HPNG model

A water tower, as depicted in Figure 3.2, provides water to households with a certain demand. Therefore, the water is stored in a large reservoir. In order to store and provide water from the reservoir, the water is pumped into the reservoir and pumped out of the reservoir with a certain flow rate.

Figure 3.3 shows an HPNG model for such a water tower with input and output pump. To be more precise, the reservoir is represented by a continuous place Cr. Pumps are represented by an ingoing continuous transition Fp and outgoing continuous transition Fd.

This model describes an ideal water tower with constant demand and input. In general, the water tower does not have a constant demand from the households and a pump can break. Assume that, for this water tower, the demand stops after 5 hours and the pump of the ingoing flow breaks according to a stochastic variable S.

The remaining part of the HPNG describes how the inflow stops according to a generally distributed pump breaking time modelled by a stochastic variable S and how the outflow stops after a deterministic time of 5 hours. When a token is fired from place Pp according to this pump breaking time the token

is removed from place Pp. As a consequence, the inflow stops, since transition Fp is not enabled any more due to the test arc connecting transition Fp and place Pp. In the same way, the outflow stops when transition Fd becomes disabled due to the removal of the token in place Pd. ■

3.3 States of HPNGs

Recall from 2.3 that states contain information that describe the changes in a system. In comparison with Definition 2.4, the states of HPNGs contain three additional elements as follows:

Definition 3.3 (States of a Hybrid Petri net with general one-shot transition)**.** The state of an HPNG is defined by the 5-tuple $\Gamma = \langle \boldsymbol{m}, \boldsymbol{x}, \boldsymbol{c}, \boldsymbol{d}, \mathcal{G} \rangle$. The vectors $\boldsymbol{m}$ and $\boldsymbol{x}$ are the markings for the discrete and continuous places. Vector $\boldsymbol{c} = (c_1, \ldots, c_{|\mathcal{T}^D|})$ contains a clock $c_i \in \mathbb{R}^+$ for each deterministic transition that represents the time that $Ti \in \mathcal{T}^D$ has been enabled. Vector $\boldsymbol{d} = (d_1, \ldots, d_{|\mathcal{P}^C|})$ indicates the drift, i.e., the change of fluid per time unit for each continuous place. Note that even though this vector $\boldsymbol{d}$ is determined uniquely by $\boldsymbol{m}$ and $\boldsymbol{x}$, it is included in the definition of a state to make it more descriptive. A general transition is only allowed to fire once, hence, the flag $\mathcal{G} \in \{0, 1\}$ indicates whether the general transition has already fired. So, the initial state of the system is $\Gamma_0 = \langle \boldsymbol{m_0}, \boldsymbol{x_0}, \boldsymbol{0}, \boldsymbol{d_0}, 0 \rangle$. ■

Changes in a system lead to different states, the next chapter discusses the representation of these states caused by changes in the system. More details of HPNGs can be found in [26].

Chapter 4

State Representation

The *system evolution* describes all reachable states starting from the initial state of an HPNG. These reachable states emerge from the initial state of an HPNG by firing transitions or reaching upper/lower boundaries of continuous places. The representation of the system evolution is important to understand the underlying algorithms, because a conditioning and deconditioning technique is applied.

Conditioning on the firing time of a generally distributed transition of an HPNG model results in a suitable state representation as explained in Section 4.1 for parametric locations and in Section 4.2 about Stochastic Time Diagrams (STDs). *Deconditioning* is the integration over the probability density function (pdf) for intervals derived from the parametric locations and Stochastic Time Diagram for a given time. In Chapter 5 and Chapter 6 intervals from more complex properties are derived and the deconditioning with STD is explained in detail for STDs.

4.1 Parametric reachability analysis

The first approach for reasoning about the system evolution is called *parametric reachability analysis* and was introduced in [26]. All reachable states from the initial state can be computed and are represented as a parametric location. A parametric location is an extension of an HPNG state as in Definition 3.3. An interval is added that describes for which possible firing time of the general transition the location is meaningful. A probability is added in order to solve conflicts between locations. The system evolution starts with the initial parametric location and expands to other parametric locations by events. An *event* is the firing of a deterministic or general transition or the reaching of an upper/lower bound of a continuous place. This approach allows to generate probability distributions of the fluid level of all the places by discretization over the support of the general distribution.

Graphically, the system evolution is represented with a treelike structure. Every node in the tree is a parametric location and every edge in the tree represents an event. So, the state-space is explored by the occurrence of events.

4.2 Region-based analysis

Similarly to the first approach using parametric reachability analysis, another approach called *region-based analysis* was introduced in [20]. This approach is proposed as an alternative that is exact and decreases the computation time with respect to the parametric reachability analysis. However, the approach currently has some limitations with respect to the models that can be analysed.

The region-based analysis has a 2-dimensional representation for the system evolution with on the x-axis the firing time of the general transition and on the y-axis the current time. To be more precise, an algorithm is considered that partitions the state space of an HPNG into regions with equivalent markings, depending on the current time t of the system and the firing time of the general transition s. This partitioning idea is similar to the partitioning of the state-spaces into zones of Timed Automata

[1]. Regions are used, because of the property that the state of the system does not change until an event occurs.

Figure 4.1 shows an example graphical representation of a Stochastic Time Diagram (STD). Each point in the STD represents a unique HPNG state as in Definition 3.3, which is denoted by $\Gamma(s,t)$.

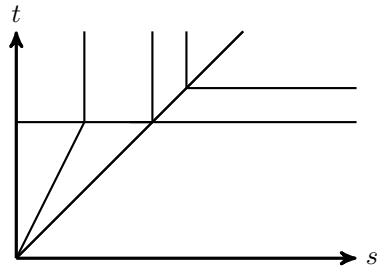

Fig. 4.1: A graphical representation of the system evolution in a STD

In [20] algorithms are introduced to iteratively generate a STD. The regions are represented by convex polygons. Initially, the STD is split into two regions: A deterministic area and a stochastic area. The area above the line $t = s$ in Figure 4.1 is the stochastic area and the area below the line $t = s$ is the deterministic area. A deterministic area is where the general transition is not fired, while in the stochastic area the general transition already has fired.

The further partitioning of the deterministic area consists of lines parallel to the x-axis. The intuitive reason is that these parallel lines are independent of s and only change by an event. The partitioning of the stochastic area, however, are all linear lines according to Proposition 1 in [20] and only changes by an event. In this case, only two types of events can occur due to the restriction of a one-shot general transition: The firing of a deterministic transition or the reaching of an upper/lower bound of a continuous place.

Conditioning and deconditioning is applied as illustrated in Table 4.1. Given an HPNG model with a general transition, the general transition fires according to a continuous probability distribution. *Conditioning* fixes a random variable from a support in order to retrieve sets of deterministic evolution. The *support* is the closure of the set of possible values of a random variable for a given probability distribution. Since no negative firing time for the general transition exists, the support is $[0, \infty)$. *Deconditioning* assigns probabilities to sets of evolutions according to a probability density function (pdf). This is achieved by integration over the pdf for intervals derived from the STD.

Conditioning	.HPNG	HPNG	→	STD
Deconditioning	STD	+ pdf	→	probabilities

Table 4.1: Conditioning and deconditioning

In the first place, the conditioning and deconditioning technique leads the HPNG models to transient probabilities, i.e. the probabilities to be in a certain state.

Chapter 5

Model checking properties

A classical *model checker* checks automatically if a model of a system meets a given specification. Figure 5.1 illustrates the model checking procedure. In fact, this illustration is based on [2].

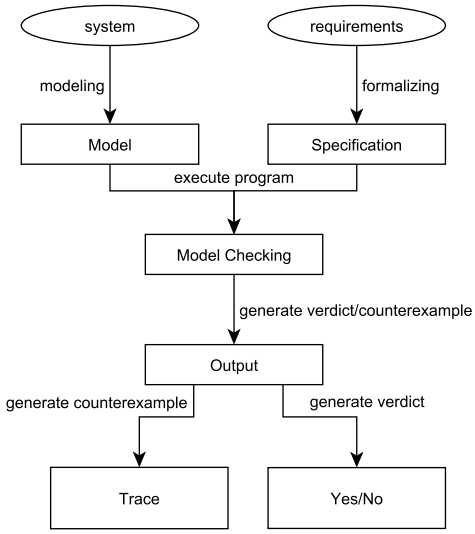

Fig. 5.1: The general case for model checking

In order to perform the model checking algorithms, the algorithms take as input a suitable model and specification. Hence, the system of interest is transformed into a suitable model in Figure 5.1, e.g. an HPNG model. Similarly, the requirements, i.e. properties that are either satisfied or not, are formalized into specifications that are handled to the model check. The model checking algorithms evaluate the specification over the model. In the end, the model checker returns a verdict, i.e. a yes/no answer. Moreover, if the verdict is no, a counter-example can be generated.

The definitions as given in this chapter are inspired by [21].

The model checker presented in this thesis is based on conditioning an HPNG to the underlying state-space representation of an STD with region-based analysis as explained in Section 4.2. To formalize requirements for HPNGs, a logic called Stochastic Time Logic (STL) is discussed in Section 5.1. In order to obtain the model checking verdict, a probability operator is introduced which is wrapped around a STL formula and compares the resulting probability of the STL formula with a given probability bound. In Section 5.2 the semantics of STL is given together with an example to illustrate the use of STL.

5.1 Syntax of Stochastic Time Logic

Having explained the syntax and semantics of the HPNG models in Chapter 3, in the following the STL specification for HPNGs is presented. In [21] this logic was introduced to reason on state-based and path-based properties, where a path is a sequence of reachable states. State-based properties are true in a certain state of the reachable states of the system. In general, path-based properties are true over an infinite or time-bound path. However, STL only considers time-bound paths, because analysis only supports time-bound paths.

Definition 5.1 (The syntax of STL)**.** Formulas in STL are defined by the following syntax:

$$\Psi := \ tt \ \mid \ x_{\mathcal{P}} \leq c \ \mid \ m_{\mathcal{P}} = a \ \mid \ \neg\Psi \ \mid \ \Psi_1 \wedge \Psi_2 \ \mid \ \Psi_1 \, \mathcal{U}^{[T1,T2]} \, \Psi_2$$

where $T1, T2, c \in \mathbb{R}^{\geq 0}$; $a \in \mathbb{N}^{\geq 0}$ ■

The state-based properties in STL are atomic properties and properties constructed by a conjunction operator ($\wedge$) of two STL properties or a negation operator ($\neg$) of one STL property. Atomic properties are indivisible, thus not constructed by other properties. The first atomic property is the continuous atomic property ($x_{\mathcal{P}} \leq c$), where x is the marking of a continuous place $\mathcal{P}$ and c is a constant for comparison with a marking. The continuous atomic property is satisfied when for a given HPNG model with a continuous place $\mathcal{P}$ and a given time to check t, the marking in that place $\mathcal{P}$ on time t is smaller than or equal to constant c. The second atomic property is the discrete atomic property ($m_{\mathcal{P}} = a$), where m is the marking of a discrete place $\mathcal{P}$ and a are constants for comparison with a marking. The discrete atomic property is satisfied when for a given HPNG model with a discrete place $\mathcal{P}$ and a given time to check t, the marking in that place $\mathcal{P}$ on time t is smaller than or equal to constant c. A constant is a fixed value and often expresses the same units as the marking, e.g. the marking of a continuous place is in m^3, then constants are often also expressed in m^3. A path-based property in STL is constructed with an until operator ($\mathcal{U}$) using a certain time bound from $T1$ to $T2$ and two STL properties Ψ_1 and Ψ_2. An until operator checks if a STL property Ψ_1 holds at least for a certain time bound until the other property Ψ_2 holds at the current or a future position.

Recall the conditioning and deconditioning technique from Section 4.2. First, a Stochastic Time Diagram is obtained from the firing time of the generally distributed transition of a given HPNG model by fixing a random variable from the support $[0, \infty)$. Next, the STD, STL formula and time to check lead to probabilities according to the probability density function associated with the firing time of the generally distributed transition of the given HPNG model. However, model checking questions require a verdict, i.e. a yes/no answer. Therefore, a probability operator has been introduced that compares the probability obtained with the conditioning and deconditioning technique with a fixed probability. Extended STL with an additional probability operator is defined as follows:

Definition 5.2 (Extended STL)**.** Formulas for verdicts in model checking are obtained by the following grammar, starting from the initial Φ:

$$\Phi := Pr \bowtie p \ (\Psi)$$

where $\bowtie \in \{\leq, <, >, \geq\}$, $0 \leq p \leq 1$ and $p \in \mathbb{R}^{\geq 0}$ ■

Note that, when this thesis refers in the following to the STL the probability operator is included.

5.2 Formalizing model checking questions

A continuous atomic property $x_{\mathcal{P}} \leq c$ holds in a certain state of the system iff the marking x of a continuous place in a HPNG model for a given time is less than or equal to a constant value c.

Similarly, a discrete atomic property $m_{\mathcal{P}} = a$ holds in a certain state iff the marking m of a discrete place in a HPNG model on a given time is equal to a constant value a. Intuitively, for DPNs this results in a yes/no answer. However for HPNGs, the answer is a probability due to the general transition which introduces stochasticity to the HPNGs.

Example 5.3 (Formalizing model checking questions to STL)**.** Consider the HPNG model of the water tower from Example 3.2. In order to understand what model checking questions can be formalized, several example model checking questions are formulated in the table below that transform the model checking questions into STL formula for each STL operators:

Question	**STL formula**
Can the amount of fluid in the reservoir be smaller than or equal to 5 m^3?	$Pr_{>0}(x_{Cr} \leq 5)$
Can the pump be broken?	$Pr_{>0}(m_{Pp} = 0)$
Can the pump be *working*?	$Pr_{>0}(\neg m_{Pp} = 0)$
Can the pump be broken *and* the amount of fluid be smaller than or equal to 5 m^3?	$Pr_{>0}(m_{Pp} = 0 \wedge x_{Cr} \leq 5)$
Can the amount of fluid be smaller than or equal to 5 m^3 *until* the pump is broken for the time interval 0 to 10 hours?	$Pr_{>0}(x_{Cr} \leq 5 \; \mathcal{U}^{[0,10]} \; m_{Pp} = 0)$

Table 5.1: Model checking questions with formulas

Note that since the system evolves, the marking over time might change, therefore, these questions can only be answered when the time is specified, the HPNG model is known and the places exist in these HPNG models. Assume that for these questions the time is specified, e.g. $t = 0$.

Consider the formula $Pr_{>0}(x_{Cr} \leq 5)$ from Table 5.1. A probability is obtained from the formula $x_{Cr} \leq 5$ with the given HPNG and the given time. The property $Pr_{>0}(x_{Cr} \leq 5)$ holds for a probability greater than zero iff there is a possibility that $x_{Cr} \leq 5$ holds. Therefore, a *Can ... be* occurs in the model checking questions. Assume that in the initial marking of the HPNG model from Example 3.2 for the continuous place Cr is 4, then the answer to the model checking question is yes, because $4 \leq 5$ at time $t = 0$ with a probability of 1 which is greater than the fixed probability 0. ■

Chapter 6

Model checking algorithms

This chapter explores the model checking algorithms, provided that, the algorithms take HPNG models and STL specification as input. For now, only algorithms for calculating a model checking verdict exist, so counter-examples are left out. The general model checking procedure results in the probability for a certain state-based STL formula to hold which is calculated by deriving the intervals from an *Stochastic Time Diagram* (STD) and an STL formula, and then deconditioning on the probability density function of the general transition over these intervals for each STL property. The model checking of the until operator requires a different algorithm for retrieving the intervals which is based on polygons in an STD.

This chapter explains the basic concepts of the underlying model checking algorithms. More details of model checking algorithms are provided in [20] and [21].

Section 6.1 describes the general model checking procedure and Section 6.2 describes the algorithms for model checking the until operator.

6.1 General model checking procedure

The general model checking procedure is based on the conditioning and deconditioning technique described earlier. First, a STD is generated, which is analysed to find the probabilities for a property to hold at a given time. The following figures depict the general model checking procedure for region-based analysis:

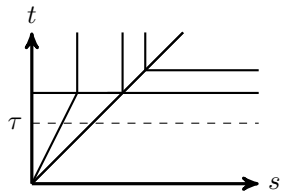

Fig. 6.1: Check at time τ

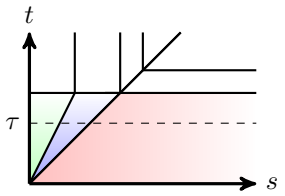

Fig. 6.2: Find the intersecting regions

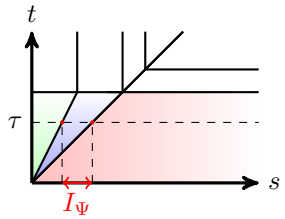

Fig. 6.3: Find the set of intervals I of the regions where property Ψ holds

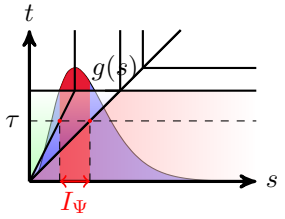

Fig. 6.4: Integrate over the probability density function *g(s)* for the set of intervals I_Ψ

First the underlying state-space representation is derived as shown in Figure 6.1. Figure 6.1 shows an STD with several regions, depicted by the solid lines, that is checked a time τ, depicted by a dotted line. The algorithm detects what regions are intersected by time τ as shown in Figure 6.2. Next, the algorithm determines for the regions where the property holds where a set of intervals I_Ψ is obtained as depicted in Figure 6.3. Figure 6.4 shows that by integration over the probability density function *g(s)* of the general transition for the set of intervals I_Ψ the probability that property Ψ holds at time τ is obtained.

A discrete atomic property ($m_\mathcal{P} = a$) either holds in an entire polygon of a region or not at all. However, the continuous atomic property ($x_\mathcal{P} \leq c$) may hold in a certain parts of the regions, because the fluid in a continuous place may be linearly dependant on s according to [20].

A conjunction ($\Psi \wedge \Psi$) results in a set of intervals where both properties hold. First, find the two intervals of the two properties. Next, find the intersection of these two intervals. The negation ($\neg\Psi$) results in a set of intervals where a specific property does *not* hold. First, find the interval of the property. Next, find the complement of this interval.

After each set of intervals is determined, the probability is calculated by integrating over the probability density function for the set of intervals. Lastly, the model checker calculates a verdict with the STL extension of Definition 5.2.

Having seen the intuitive meaning of interpreting STL formulas, a formal explanation is provided on how answers are obtained from STDs and a probability density function of the general transition. A satisfaction relation describe the relations between a single system state $\Gamma(s,t)$, i.e. a single point in the STD that contains the state of the system according to Definition 3.3, and an STL formula Ψ. The satisfaction relation between a single system state and an STL formula, according to Definition 5.1 is defined, as follows:

Definition 6.1 (Satisfaction on system states)**.**

$$
\begin{array}{lll}
\Gamma(s,t) \vDash^{s,t} & tt & \forall t, s, \\
\Gamma(s,t) \vDash^{s,t} & x_\mathcal{P} \leq c & \textit{iff } \Gamma(s,t).x_\mathcal{P} \leq c, \\
\Gamma(s,t) \vDash^{s,t} & m_\mathcal{P} = a & \textit{iff } \Gamma(s,t).m_\mathcal{P} = a, \\
\Gamma(s,t) \vDash^{s,t} & \neg\Psi & \textit{iff } \Gamma(s,t) \nvDash^{s,t} \Psi, \\
\Gamma(s,t) \vDash^{s,t} & \Psi_1 \wedge \Psi_2 & \textit{iff } \Gamma(s,t) \vDash^{s,t} \Psi_1 \wedge \Gamma(s,t) \vDash^{s,t} \Psi_2, \\
\Gamma(s,t) \vDash^{s,t} & \Psi_1 \, \mathcal{U}^{[T1,T2]} \, \Psi_2 & \textit{iff } \exists \tau \in [t+T_1, t+T2] : \Gamma(s,t) \vDash^{s,t} \Psi_2 \wedge (\forall \tau' \in [t,\tau] : \Gamma(\tau', s) \vDash^{s,t} \Psi_1).
\end{array}
$$

■

Since Definition 6.1 describes for all STL formulas when these formulas are satisfied in a single system state in the STD, the next step is to determine the satisfaction of the states on a specific time to check t. To be more precise, a set of satisfaction intervals I_Ψ is defined with the aid of a satisfaction relation between the interval on the support $[0, \infty)$ of the general transition and the STL formula Ψ, according to Definition 5.1, as follows:

Definition 6.2 (Set of satisfaction intervals)**.**
The set of satisfaction intervals $Sat^t(\Psi)$ is defined as the set of all intervals satisfying Ψ at time t, i.e., $Sat^t(\Psi) = \{I_\Psi : I_\Psi \vDash^t \Psi\}$, where the non-overlapping intervals have the following property:

$$I_\Psi \vDash^t \Psi \textit{ iff } \forall s \in I : \Gamma(s,t) \vDash^{s,t} \Psi$$

■

Probabilities can be found by deconditioning with the probability density function $g(s)$ of the general transition. The satisfaction relation of an STL formulas with a probability operator according to Definition 5.2 is as follows:

Definition 6.3 (Satisfaction of the probability operator)**.**
Let $\Gamma(t) = \{\Gamma(s,t)|s > 0\}$ be the set of possible system states at time t, then the satisfaction relation for the probability operator $\Phi := Pr \bowtie p$ is defined as:

$$\Gamma(t) \vDash \Phi := Pr \bowtie p \textit{ iff } Prob(\Psi, t) \bowtie p,$$

where

$$Prob(\Psi, t) = \sum_{I_\Psi \in Sat^t(\Psi)} \int_{I_\Psi} g(s)ds.$$

■

So, all STL formulas from Definition 5.1 and Definition 5.2 are interpreted by applying Definition 6.1, 6.2 and 6.3. First, the satisfaction relation is determined between a STL formula and a STD. In the end, a model checking verdict is given if a given STL formula according to 5.1 holds at time t for an HPNG model with a certain probability p.

6.2 Model checking an until formula

The algorithm for the until formula requires a special set of operations on sets of intervals and on sets of system states. The following definition, which is introduced in [21], describe an intersection and complement operator between sets of sets:

Definition 6.4 (Intersection and complement between sets of sets)**.** Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two sets of sets, the intersection operator between these two, and the complement operator of a set of sets, are defined as follows:

$$\begin{array}{ll} S \in \mathcal{S}_1 \sqcap \mathcal{S}_2 & \text{iff } \exists S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2 : S = S_1 \sqcap S_2, \\ S \in \neg\mathcal{S}_1 & \text{iff } \forall S' \in \mathcal{S}_1 : S' \cap S = \emptyset \wedge \nexists S'' \in \neg\mathcal{S}_1 : S'' \subset S. \end{array}$$

■

From definition 6.4 the union and relative complement operator are derived, respectively: $\mathcal{S}_1 \sqcup \mathcal{S}_2 = \neg(\neg\mathcal{S}_1 \sqcap \neg\mathcal{S}_2)$ and $\mathcal{S}_1 \setminus \mathcal{S}_2 = \mathcal{S}_1 \sqcap \neg\mathcal{S}_2$.

The algorithm for the until formula $(\Psi_1 \, \mathcal{U}^{[T1,T2]} \, \Psi_2)$ results in a set of intervals where the first property Ψ_1 has to hold at least for a certain time bound $[T1, T2]$ until the second property Ψ_2 holds at the current or a future position. The set of satisfaction intervals for the bounded until is defined as follows:

Definition 6.5 (Set of satisfaction intervals for bounded until)**.**
$Sat^t(\Psi_1 \, \mathcal{U}^{[T1,T2]} \, \Psi_2) = \bigsqcup\{I \subseteq \mathbb{R}_{\geq 0} | \exists \tau \in [t_0 + T_1, t_0 + T_2] : I \vDash^t \Psi_2 \wedge \forall t' \in [t_0, \tau] \wedge I \vDash^{t'} \Psi_2\}$ ■

Sets of non-overlapping satisfaction polygons are required for the formulas Ψ_1 and Ψ_2 instead of a set of satisfaction intervals. The set of satisfaction polygons is defined per region as follows:

Definition 6.6 (Set of satisfaction polygons)**.**
The set of polygons within region $\mathcal{R}$, where an STL formula according to Definition 5.1 holds is defined as:

$$\begin{array}{ll} \mathcal{P}^{\mathcal{R}}(tt) & = \{(s,t) \in \mathcal{R} \mid \forall s, t\}, \\ \mathcal{P}^{\mathcal{R}}(x_\mathcal{P} \leq c) & = \{(s,t) \in \mathcal{R} \mid \Gamma(s,t).x_\mathcal{P} \leq c\}, \\ \mathcal{P}^{\mathcal{R}}(m_\mathcal{P} = a) & = \{(s,t) \in \mathcal{R} \mid \Gamma(s,t).m_\mathcal{P} = a\}, \\ \mathcal{P}^{\mathcal{R}}(\Psi_1 \wedge \Psi_2) & = \mathcal{P}^{\mathcal{R}}(\Psi_1) \sqcap \mathcal{P}^{\mathcal{R}}(\Psi 2), \\ \mathcal{P}^{\mathcal{R}}(\neg\Psi) & = \mathcal{R} \setminus \mathcal{P}^{\mathcal{R}}(\Psi). \end{array}$$

■

Since the set of satisfaction polygons is determined for each region an algorithm handles three different situations in order to obtain the set of intervals for the until formula. First, find a set of intervals where the first property holds at the time to check. Next, move up and refine the intervals by omitting those intervals that violate the first property. Eventually, the time bound is reached or the second property is satisfied. After finishing the algorithms, deconditioning can be applied according to Definition 6.3.

More details about model checking algorithms for STL formulas with an until operator are in [21].

Chapter 7

Related Work

This chapter discusses the literature around the model checking tool which is presented in this master thesis. Therefore, *Fluid Petri nets* (FPNs) are considered which is related to *Hybrid Petri nets with a general one-shot transition* (HPNGs). Also, the *Zone Automata* (ZA) are considered for their relation with the region-based analysis. In the end, several tools for simulation, analysis and/or model checking are explored.

Section 7.1 is about the modelling and analysis of FPNs that are similar to modelling and analysis of HPNGs. Since FPNs and HPNGs are similar, a small comparison between these two is made to discover similarities and differences. In the end, several tools that support FPNs are discussed.

Despite the difference between *Timed Automata* (TA) and *Petri nets* (PNs), ZA inspired region-based analysis to use zones. Therefore, an explanation of ZA is given to extend knowledge on the region-based approach in Section 7.2.

Similarly, model checking tools for hybrid systems are currently under development such as SpaceEx and HyTech for *Hybrid Automata* (HA), Oris, GRIF Petri and SimHPN for *Hybrid Petri nets* (HPNs) and KB3 as a workbench for *Stochastic Petri nets* (SPNs). Despite the many similarities, the tools often use different algorithms, aim for different audiences and use other development environments. Therefore, this Section 7.3-7.7 explores this information for the model checking tools to discover where our tool is embedded and discuss relevant features of the tools.

Section 7.9 discusses what an HPNG model checker sets apart from the related work.

7.1 Fluid Petri nets

FPNs are an extension of PNs, where continuous places are added to regular discrete places. Two main branches have come forth from the FPNs which are HPNs and *Fluid Stochastic Petri nets* (FSPNs). Recall, HPNGs have an extension to support generally distributed firing of tokens by general transitions, i.e. HPNGs are HPNs with a general one-shot transition. FSPNs are extended with transitions that support negative exponentially distributed firing of tokens. HPNGs and FSPNs both made a trade-off between the number of continuous variables and the number of stochastic transitions. Hence, HPNGs support a large number of continuous variables and one general one-shot transition, while FSPNs support only a few continuous variables and a large number of negative exponentially distributed transitions. So, the most suitable models depend on the system.

After the first introduction of HPNs and FSPNs, a lot of extensions were added on the modelling level. [3] discusses a more detailed comparison and showed that there is no essential difference, since HPNs can be emulated as FSPNs and vice versa.

Several tools/techniques are proposed in [27] for the analysis of FPNs which are simulation and numerical solutions: FSPN [24], NuSMV [17], HyTech [32] and PRISM [33]. In order to use these tools, FPN models are transformed into appropriate models for these tools. The simulation and numerical

solutions of FSPN are obviously limited to FSPNs. The simulation technique is best suited for validation and testing of FSPN and the numerical solution for exact analysis of small models. In the case of NuSMV, verification of finite state systems is recommended, since NuSMV only supports finite state systems. HyTech supports up to linear HA. PRISM is used on Discrete Time Markov Chains. So, the usage of existing tools that can model and analyse similar models result in restrictions on FPN models.

So, HPNGs are embedded among FPNs and provide additional solution for analysis. Due to the scarcity of HPN tools, the introduction of a tool in this area seems meaningful.

7.2 Zone Automata

The theory of TA [1] is similar to timed PNs, since both theories are used to model and verify real time systems. TA is an extension of finite automata with clocks and allows to model real-time systems. Finite automata consists of states that are connected by state-transitions.

TA are a subclass of HA, since the state reachability problem for TA is decidable as shown in [1]. The difference between TA and HA is that TA has always a clock drift of exactly 1, while HA has various clock drifts. The *state reachability problem* of a TA is about the decision if a state of the system is reachable from a given initial state. More details about HA are in [29]. Note that HA models differ from HPNs models, a detailed description on how these two cohere is found in Chapter 5 of [9]. HPNs as introduced by [10] can be analysed with a tool such as PHAVer [19] by mapping the HPNs to the underling HA.

ZA are constructed from TA by grouping equivalent states that satisfy the same clock constraints together. Without grouping equivalent states, the TA might needs infinitely large models. Because of this reduction to finitely large models, ZA increases the efficiency for model checking and the new symbolic states contain information about time. Grouping states for analysis is an important intuition behind region-based analysis.

Several model checking tools have been developed for the theory of TA such as Uppaal [50] and Kronos [53] that both check TA against Timed Computation Tree Logic (TCTL).

7.3 SpaceEx

SpaceEx [54] is a verification platform for HA using a scalable reachability algorithm, which. The reachability algorithms are focused on computing approximations of reachable states for systems instead of other approaches that used more exact methods. The reachability algorithm for SpaceEx has an optimized approximation algorithm.

Scalability is a known problem for hybrid systems, therefore, SpaceEx provides a scalable approach due to the approximation. HA have continuous variables which are similar to combination of the markings for every continuous place and clocks for enabled transitions in HPNs. The reachability algorithm scales up to more than 100 real-valued state variables ranging from 0.7 seconds with 2 variables to 78.22 seconds for 198 variables. So despite the scalability problem with hybrid systems, SpaceEx provides a scalable solution to more than 100 real-valued variables in HA due to approximation.

SpaceEx separates the tool into three components: The analysis core, the web interface and the model editor. The analysis core takes a file via the command line with certain configurations and produces a series of output files. The web interface is a graphical user interface that provides some wrapper around the analysis core, while providing extra graphical output and easy configuration. All parts run separately, therefore, the web interface and the analysis core run with a client-server architecture. The

model editor allows the creation of models of hybrid systems with a graphical editor. The web interface and the model editor provide easier access to the analysis core, which is the most fundamental part of SpaceEx.

To conclude, scalability measures are useful to examine in view of scalability of newly introduced tool in this paper and the separation of the tool into components give some separation of essential parts or source code is useful.

7.4 HyTech

HyTech [32] is a symbolic model checker for linear HA. Linear HA, which has only linear clock drifts, is a subclass of HA. An exact definition of linear HA is in [31]. HyTech model checks linear HA against temporal-logic requirements using parametric analysis.

HyTech is a command-line tool that processes a file with an HA model and analysis commands written in C++. HyTech was originally written in Mathematica which was around 50-1000 times slower than the C++ version [28]. An input file consists of a textual description of linear HA and analysis commands with an analysis language for state-space exploration programs. The main analysis features are reachability analysis, parametric analysis, the conservative approximation of state assertions and the generation of error traces (counter-examples) which are all discussed in more detail in [31]. HyTech has a user guide [30] in order to ease the creation of input files.

A benefit of state-space exploration tools is the capability to generate an error-trace for state-based or path-based properties that are not satisfied. For instance, when a formula in temporal-logic is specified and the formula is not satisfied, it is often useful to know what sequence of states lead to an system state. HyTech generates error-traces which is of great benefit to multiple cases.

HyTech has been successfully used for systems with complex interactions between discrete and continuous components of the system. The tool has been applied for several use cases in control-based applications. Hybrid systems that consist of mainly discrete components with several clocks are better suited for TA tools.

In short, HyTech offers a wide scale of analysis capabilities on linear HA against temporal-logic requirements with the additional benefits of counter-examples. The tool is straightforward in the sense that it does not have many extra features to ease the modelling and analysis configuration for the user except that there is a user guide.

7.5 Oris

Oris [45] is a tool for qualitative verification and quantitative evaluation of *time PNs* (TPN). Moreover, stochastic preemptive Time Petri Nets (spTPNs) are considered [6]. TPNs associate an time interval for each transition, where either the transition may fire during that time interval or the transition fires in the end of the interval. The difference with Time**d** PNs (HPNGs are Time**d** PNs) is that the firing of Time**d** PNs occurs with a single firing time for each timed transition [41]. TPNs do not consider continuous places and continuous transitions in comparison to HPNGs.

Oris has a graphical user-interface (GUI) that provides file management, graphical representations of TPNs for model editing and analytical features. The file management helps the user to easily create and manipulate model files. The model editing is done graphically and therefore very intuitive for the user. The analytical features are a model checker for real-time temporal logic. The stochastic part is currently simulated and limited to the uniform and exponential distribution.

So, ORIS is a user friendly tool for TPNs with a GUI, file management, graphical representation of TPN models and model checking capabilities of TPNs against real-time temporal logic. The characteristic of TPNs are the time interval for each transition compared to characteristic of HPNGs that consider general distributions and continuous places and transitions.

7.6 GRIF Petri

GRIF [48] is a systems analysis software platform to determine quality aspects of reliability, availability and performance. GRIF PETRI is a module of GRIF allowing the user to model stochastic PNs. The tool is driven by a Monte Carlo simulation engine. This engine gives has interactive graphical simulation to validate the behaviour of the modelled system. The engine provides information about the mean, standard deviation and confidence intervals of the outcomes of the simulator. GRIF PETRI contains a model builder that provides an graphical drag-and-drop interface to create PN models.

An user-friendly graphical drag-and-drop interface for the creation of Petri net models is introduced. The graphical drag-and-drop interface was created within JAVA [38]. A tool bar shows the possible elements for drawing the tool that mostly consist of the elements of a stochastic PN. A big area is left open for drawing the Petri net model. This model is saved in a format recognized by the tool itself. Another tool bar shows buttons for the options for interactive graphical simulation. In short, the user-interface shows an interesting graphical drag-and-drop interface and a nice structure and tool bars for overall user-interfaces.

So there are useful insights in the treatment of PN models, however, internally a GRIF PETRI does not use analytical methods for PNs, but is based on simulation. The aim of the tool is to simulate PNs that often require multiple runs of a model of the system. An analytical approach often requires one run and is exact to a certain extent. However, analytical approaches are often hard to find and therefore have some limitations.

7.7 SimHPN

SIMHPN [34] is a collection of tools for simulation, analysis and design of discrete event systems for HPNs. The main features are simulation of HPNs, functionalities from graphical Petri net editors, visualization and analysis. The tool has a user manual to understand the capabilities of the tool [35]. The collection of tools are embedded in the MATLAB [47] environment. A PN editor is delivered separately from the toolbox.

SIMHPN has a GUI to perform the simulations and analysis procedures. The GUI consist of a MATLAB figure window for plotting the markings, adjust the options and management of the models. The output data is accessible in the MATLAB environment.

In short, SIMHPN is a tool mostly for simulation of HPNs inside the MATLAB environment with a GUI.

7.8 KB3

KB3 [15] is a workbench for analysis of reliability, availability, productivity and costs from reliability models, such as fault trees, Markov graphs and Monte Carlo simulations, and a graphical representation of the system based on a generic knowledge base. The workbench automatically builds reliability models, such as fault trees, Markov graphs, Monte Carlo simulation models, from graphical model of a

system that is based on a knowledge base, which is a repository of information that can be manipulated. A knowledge base is adapted to the problem with a generic description of components. KB3 has a description of knowledge bases written in a domain-specific language FIGARO.

Also for SPNs a knowledge base description is introduced, i.e. a description in FIGARO of knowledge base for Petri nets where the transitions fire according to a probability distribution determined by a random variable. Next, the graphical SPN models describe the system that is analysed with a Monte Carlo simulator called YAMS, which allows interactive analysis of the SPN models.

KB3 has a GUI that allows structurally linking and importing a knowledge base for studies of systems that be defined in the model editor. Knowledge bases written in FIGARO can be imported and linked to a study in the GUI. A study can be edited graphically in a model editor and analysed with a reliability model. The knowledge base determines what models are supported by the model editor.

In summary, KB3 allows to define a knowledge base such as a stochastic PN in FIGARO, model actual stochastic PNs in a graphical model editor and analyse these SPNs with simulation with a Monte Carlo simulator in YAMS. A more detailed description of KB3 is provided in [5].

7.9 An HPNG model checking tool

An HPNG model checking tool sets itself apart from this related work by focussing on model with many continuous variables, providing a model checker and quick computation times. The HPNG model checking tool introduced in this master thesis is trade off between allowing a large number of continuous variables, but restricting the number of stochastic transitions. The HPNG model checking tool sets itself apart by providing an actual model checker and calculation of transient probabilities based on exact analysis that provide valuable insight into system with critical infrastructures. Simulation often support more models for analysis compared to exact analysis, but exact analysis often provides lower computation times. So far, such a model checking tool for exact analysis of HPNG models does not exists.

Part II

Tool Development

Chapter 8

Domain analysis and requirements

Designing software requires thorough preparations before implementing any code. Therefore, this chapter studies the domain and the requirements of the tool.

In Section 8.1 a short description of the tool is provided. A tool is designed for a specific audience and judged by experts in that domain. For this reason, the target groups and domain experts are described in Section 8.2. Section 8.3 informs about the state-of-the-art. The requirements are obtained from the domain analysis in Section 8.4 and Section 8.5.

8.1 Tool description

The goal of this thesis is to develop a tool that model checks Hybrid Petri nets with one general one-shot transition (HPNGs) as explained in Definition 3.1 against Stochastic Time Logic (STL) as in Definition 5.1 and Definition 5.2. The main functionality of the tool is to generate a verdict with model checking as explained in Chapter 5. First, an HPNG file is parsed and an STL formula is parsed that is specified by direct input from the user. Next, the model checking algorithms are executed according to the algorithms as in Chapter 6. In addition, the tool gives some insight into the model checking procedure by displaying Stochastic Time Diagrams as explained in Section 4.2. Also, the tool generates probability plots for continuous atomic properties derived from Definition 5.1 for different moments in time. Only continuous atomic properties are currently implemented, however, the generation of plots depend on the model checker, the generation can easily be extended to an arbitrary STL formula. The tool has a graphical user interface (GUI) for editing HPNG models, configuring the model checking procedure and providing feedback or results after execution of the implemented algorithms.

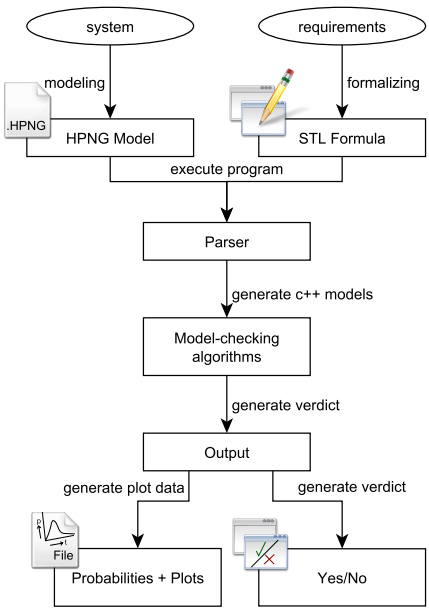

Fig. 8.1: Tool design for model checking

Figure 8.1 shows an illustration elaborating on Figure 5.1. The system is formalized into an HPNG model and combined with a formalized requirement of an STL formula specification that together form the input of the model checking tool. The input to the tool is parsed, i.e. an instance of a data structure is constructed from an input HPNG model file and input STL formula. Then the model-checking algorithms use these instances to generate a verdict or some intermediate results depending on the configuration of the tool. There are two output options which are a verdict or plots, that provide insight into the algorithms by displaying the probabilities and allowing to check a property over a range of time.

8.2 Target groups and domain experts

The target groups, i.e. the end users of this tool, are expected to come from both the academic world and business world. In the academic world tools are often used to demonstrate and show the feasibility of theoretical concepts, to educate students and colleagues and to study use cases from practice. Therefore, the tool is open to use cases from the business world with critical infrastructures such as systems with electricity, gas, oil, telecommunication, water, health, transportation, finances and security. In practice, the tool is mostly used and promoted by the domain experts themselves during the development. The domain experts are experts in the area of subjects like: Hybrid Petri nets, logic, model checking, tool development and validation.

8.3 State of the art

Before the development of the tool, the state of the art is provided to understand the exact contributions of this research and some restrictions of the tool.

Several papers form the basis for the algorithms behind the tool which elaborate parametric analysis, region-based analysis, calculation of transient probabilities, HPNG models, STL properties, Stochastic

Time Diagrams (STDs) and model-checking algorithms. The Background part of this master thesis discusses these most important theoretical concepts as introduced in [20], [21] and [26].

Furthermore, essential C [43]/C++ [46] code existed for most of the algorithms:

HPNG data structure A data structure for HPNG models was provided. This data structure had structs in C to store an arbitrary HPNG model.

Input file to HPNG model Functionalities were provided to transform an input file into an HPNG model instance of a data structures. So, an HPNG model file was provided as input that was transformed into an instance of a data structure that can be used to interact with, e.g. a Stochastic Time Diagram could be calculated.

Partitioning of the state space An implementation for the partitioning of the state space was provided that results in STDs. This implementation was based on the algorithms described in [20].

Calculation of simple STL probabilities The implementation of the calculation of transient probabilities were based on algorithms that are explained in Section 6.1. The implementations supported functions for conjunction, negation and continuous atomic STL properties. For the discrete atomic STL properties no implementation existed.

Calculation of until formula probabilties The implementation supported methods to obtain probabilities from until formulas with two atomic properties. A function existed to calculate sets of intervals where an until property holds and a function existed to calculate the probability by integrating a probability density function of the generally distributed transition over the sets of intervals.

The C++ code was compiled in Eclipse using the GCC compiler. The code was executed in the terminal and resulted in 3D plots of transient probability distributions in GNUPlot or some textual probabilities for checking the until operator.

Despite that there was a data structure and functionalities, which were mostly written in C, for retrieving an HPNG model from these data structure from model files, the HPNG models did not have proper error handling except for segmentation faults to be thrown at runtime. Furthermore, the models contain a lot of useless information (e.g. a value needs to be added for the discrete marking in a continuous place). The useless information is discarded, so this is still a bit confusion to the end-user.

The most important missing theoretical part was the nested STL formulas for model checking, which allows the user to insert and evaluate arbitrary STL formula on execution level. The existing code only supported STL formulas that existed inside the source code without a very clear data structure compared to the HPNG models, i.e. a user was not able to insert arbitrary STL formula to the program on execution level. Nested STL formulas brings challenges with combining the simple STL formulas that reason about sets of intervals and until formulas that reason about polygons. Furthermore, an arbitrary STL formula should be supported on execution level, i.e. a user should be able to define an arbitrary STL formula according to a language specification.

8.4 Functional requirements

The functional requirements are derived from domain analysis in the previous sections of this chapter. The functional requirements describe the functions of the tool, which are the following:

The tool must support models for HPNGs. Obviously, the input to the tool should support HPNG models as described in Chapter 3 as the core models for the tool.

The tool must support formula specifications for STL. The input of the tool should support STL formulas as described in Chapter 5.

The tool should allow the user to adjust the models and formulas. An editor is required for the user to increase the usability of the tool. So, it should be the case that various models and formulas are supported without knowledge of the core implementations.

The tool must parse a model and a specification. The model and specification are defined by a user in a certain language according to a grammar. This requirement demands that the input of a user is parsed with some input such as a file or an input field, i.e. the input of the user is build into a usable format for the model-checking algorithms.

The tool must execute the model checking algorithms based on [20] and [21]. Obviously, the implementations are based on the algorithms based on the papers about region-based analysis as described in Chapter 6.

The tool has to generate Stochastic Time Diagrams. Stochastic Time Diagrams should be internally created as explained in Section 4.2.

The tool should generate probabilities of STL formulas to hold at a specified time. The probabilities of STL formulas, excluding the probability operator, should be generated as intermediate results of model checking.

The tool must generate a verdict for an input model and formula specification. A model checking verdict is required for an input model and formula. However, these models and formulas may have certain restrictions and are described by Chapter 3 and Chapter 5.

The tool should export and import HPNG files as a model. HPNG model files need to be opened and to be saved in order to increase the usefulness of the editor.

The tool should export Stochastic Time Diagram images. Stochastic Time Diagrams should be externally exported to a graphical representation of these diagrams as in Figure 4.1.

The tool should export data and figures with the corresponding probabilities. It should be possible to compare the probabilities of STL formulas by plotting them into one figure and it should report all generated values to the user.

Having seen the functional requirements, an overview to illustrate the functionalities is given as follows:

Algorithm	**Input**	**Output**
Model checking	HPNG model STL formula Time to check	
Stochastic Time Diagram	HPNG model	
Transient probabilities	HPNG model Discrete/Continuous place Specific constant Time range	
	HPNG model Continuous place Constant range Time range	

Table 8.1: In- and output with corresponding algorithm

Table 8.1 gives an overview of the most fundamental functionalities by illustrating the input and output for each of the algorithms. The most important feature is model checking, where the input is an HPNG model, STL Formula and a time to check and the output is an verdict. Additionally, Stochastic Time Diagrams provide insight into the state space as in Section 4.2. Transient probabilities are calculated either over a range of constants or for a specific constant that result in respectively a 2D plot and a 3D

plot over a range of time, i.e. the 2D plots and 3D plots represent transient probability distributions that are calculated for a set of times to check with either a specific constant or for a set of constants. The 2D plots and 3D plots are in the first case only available for the discrete and continuous atomic properties. However, the tool should be easy to extend to 2D plots and 3D plots for properties with negation, conjunction, until operator and nested STL formulas.

8.5 Non-functional requirements

Having seen the functional requirements, the non-functional requirements give constraints to these functionalities, i.e. non-functional requirements define the criteria for operation of the system, which are the following:

The tool must be object-oriented. Object-oriented design structures code into 'objects' that have attributes that describe the objects and procedures. These objects function as computer programs that interact with each other.

The tool must be extendable. An extendable tool is achieved by applying design patterns, documentation and a development environment that ease the developing of new features.

The tool must use the core code written in C/C++. The existing code is used as a starting point as described in Section 8.3. This requires cooperation of the tool with existing C/C++ implementations.

The tool must be executed on a 64-bit Linux machine. This requires at least a tool that is compiled so it can be executed on a 64-bit Linux machine. Since the tool should be extendable, other operating systems and 32-bit processors are taken into account for future development.

The tool should work mostly free of bugs for the end-user. A bug produces an incorrect or unexpected result and should therefore be avoided. The scope for this requirements lays on showing the feasibility of the tool by a case study and feedback by users.

The tool must execute as a GUI. A graphical user-interface increase the usability of the tool with an intuitive GUI. A graphical user-interface allows the user to interact with a system through graphical icons and secondary notation (e.g. position, color and symmetry).

Chapter 9

Tool development environment

In order to develop a tool, first a suitable *Software Development Kit* needs to be chosen. An SDK is a set of software development tools that allow the creation of a certain software application. The contents of the SDK is discussed that lead to various SDK alternatives. After various SDK alternatives are considered, an SDK is chosen, which is also directed by the requirements of the tool.

The SDK depends on the requirements of the tool as discussed in Section 8.4 and Section 8.5. Next, the various types of tools inside the SDK are discussed in Section 9.1. In Section 9.2 the most obvious alternatives are discussed. Section 9.3 explains the chosen contents for the SDK that has been used in this master thesis.

9.1 The required development tools

The SDK needs a *compiler* for compiling the source code to a target machine. Furthermore, the software development kit uses several *libraries* for the graphical user interface and producing graphical plots. For cooperation purposes, a project page, a subversion server and documentation tool are added to the SDK. An *Integrated Development Environment* (IDE) is added to the SDK, because an IDE provides the developers with an editor, automated compilation and linking to libraries, a debugger, and a(n) documentation/Application Programming Interface (API) of the programming language. Often there are even more features in these IDEs to support software developers.

9.2 Development tools alternatives

First, the compiler should be chosen that is mostly determined by the programming language. Next, a GUI library should be chosen that depends on the programming language. In the end, the most suitable IDE should be chosen that depend on the chosen compiler and GUI library. Therefore, the following SDK alternatives are discussed:

	Compiler	**GUI library**	**IDE**
Java alternative	GCC [22] Java [38]	Swing [39]	Eclipse [14]
gtkmm alternative	GCC	gtkmm [16]	Eclipse
Qt library alternative	GCC	Qt [12]	Eclipse
Qt IDE alternative	GCC	Qt	Qt Creator [13]

Table 9.1: GUI software development kit options

Our SDK consists of the C/C++ compiler GCC, QT and QT Creator out of several options as mentioned in Table 9.1. Now the reasons for choosing this SDK are elaborated.

The source code is written in C++, since the existing code was also written in C/C++. Other options for the source code would be highly inefficient, since it requires to rewrite the entire existing code or use messaging. The advantages of a source code written in C++ is that it is object-oriented, has many libraries and the source code written in C still operates inside source code written in C++. C++ is a low-level programming language that allows to control a machine directly, manage memory and write programs, libraries and device drivers. The disadvantages of source code written in C++ is that it is difficult to learn, because it demands memory management and direct control of the machine. However, the memory management and direct control of the machine also offer an advantage when efficient programs are required.

Next, a GUI library in C++ needs to be chosen. The most recommended GUI libraries are gtkmm and Qt (gtkmm alternative & Qt library alternative from Table 9.1). These options have a large community, documentation and plenty of examples available on the Internet, i.e. this makes both libraries easy to learn. Both are compatible to most desktop Operating Systems. Qt is even compatible to mobile Operating Systems. So at this point, the choice is based on the taste of the developer.

Besides the similarity of these libraries, Qt offers Qt Creator (Qt IDE alternative from Table 9.1). Qt Creator is an IDE build for the Qt library. Qt creator is similar to Eclipse with some added features. In Qt Creator there is an easy drag-and-drop design editor and compilation of Qt is automated without installing any additional plug-ins. Despite the advantages of Qt Creator, moving from Eclipse to Qt creator also requires some effort, since the existing code was developed in an Eclipse environment. However, installing a Qt plug-in in Eclipse is barely supported. So to get Eclipse working with Qt requires a lot of workarounds.

Another option for the GUI library is considered (Java alternative from Table 9.1) for a well-known GUI library called Swing in the object-oriented language Java. This option is considered, because Java with Swing is more familiar to most developers and Java with Swing does not require to learn an additional language, because knowledge on Java with Swing is present. Furthermore, Java with Swing is very portable, since it is supported by most of the operating systems. However, the source code would be written in C++ and Java, so this requires two compilers. Additionally, a messaging between C++ and Java needs to be established. Since Java has a garbage collector for automated memory management measurements on algorithms are more difficult, since the computation speed of the algorithm might be influenced by memory operations. Also, the effort of learning a new GUI library is considered smaller than the effort of writing a messaging protocol between C++ and Java.

9.3 Contents of the Software Development Kit

After taking all these options from the previous section into consideration, the best alternative is GCC with Qt and Qt Creator. The main effort is learning a new GUI library and moving to another IDE. Figure 9.3 refers to the contents of the software development kit for this master thesis.

Fig. 9.1: GCC Compiler

Fig. 9.2: Qt GUI Library

Fig. 9.3: Qt Creator IDE

The SDK also contains some tools for cooperation. Doxygen [52] is used as standardized documentation. Doxygen automatically generates documentation in HTML [55] and LaTeX [37] format for the comments following the Doxygen documentation conventions. Furthermore, a SVN [18] server is added for version control. Also, a project page is added for information, releases and bug tracking [42].

Chapter 10

Tool architecture

An object-oriented and extendable tool requires a well chosen architecture. In order to obtain a well chosen architecture, software design patterns have been developed to add useful structure to the code. Software design patterns often provide a reusable solution to common problems in software development.

The Model-View-Controller (MVC) pattern is such a software design pattern and is explained in Section 10.1 followed by the explanation of the Facade pattern in Section 10.2. In Section 10.3 the conceptual design of the tool is elaborated including the application of patterns.

10.1 Model-View-Controller pattern

The *MVC pattern* separates the code into three parts: The *Model* represents the knowledge of the system, the *View* represents the visual representation of the models and the *Controller* reacts to events by affecting the views and the models. The interaction between parts of the code is illustrated as follows:

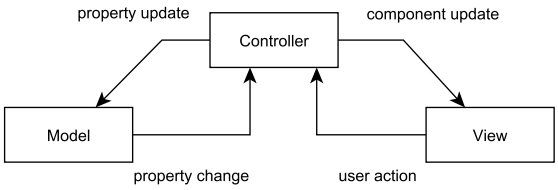

Fig. 10.1: Model-View-Controller pattern

Figure 10.1 shows in abstraction how the model, view and controller interact with each other. The view reports the *user action* to the controller when the user interacts with the visual representation. The controller often passes changes on to the model by *updating properties* in the model. The model performs internal algorithms as a response to the controller and passes the *property changes* to the controller. However, a model can also call the controller by model changes such as an internal clock. The controller tells the view to be updated by a *component update*.

The MVC pattern is often used in the context of user interfaces. Separation of concerns is the main reason for this, which is a design principle in software development for separating distinct parts of the code. Besides the separation of concerns, another advantage of MVC is *code reusability*. This refers to the likelihood of a piece of code to be reused under certain conditions. In the case of MVC the code is resuable, because a developer can easily add new models, views and controllers. A disadvantages is extra overhead in the extra number of classes.

10.2 Facade pattern

The *Facade pattern* provides a simpler interface for a subsystem consisting of a larger body of code. A Facade in a piece of code often consists of a class taking the responsibility of serving clients by interacting with a set of classes and providing answers and feedback to the clients.

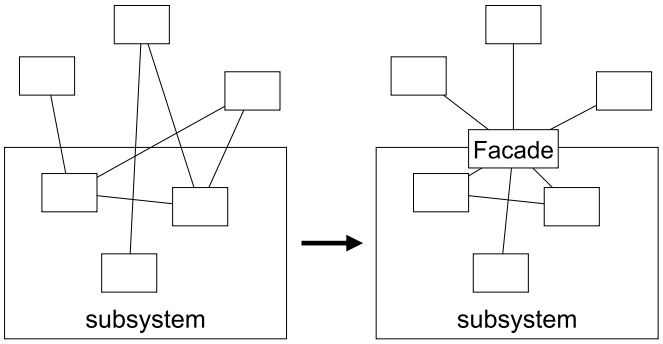

Fig. 10.2: Facade pattern

Figure 10.2 shows the improvement of an architecture by the Facade pattern. The rectangles represents the classes that interact with each other which is represented with lines. The left part illustrates the system without the Facade pattern. The right part of the figure shows the application of the Facade pattern by an additional Facade class. This class takes all responsibilities for the interactions between the subsystem and the clients.

An advantage of the Facade pattern is that it is easier to access the subsystem. When using the Facade pattern, note that the subsystem is not hidden from the client, i.e. the client can still interact with the whole subsystem. As with the MVC pattern code reusability is again an advantage, since a client can reduce the amount of code, because a Facade provides a simpler interface. In practice the client can reduce interaction to the subsystem to only one request to the Facade class. A disadvantage is that another class is added.

10.3 Conceptual design

Overall, model checking is performed by users interacting with a graphical user-interface that is driven by a software architecture with an MVC pattern and a Facade pattern. So, users see a graphical user-interface that they can interact with which is generated by the view package. Users interact with the user-interface and activate the model checking algorithms via the controller package and a FACADE class to the model package. The model package provides feedback to the user again via the controller package. So, the total interaction of the user with the model checker and user is a well chosen architecture that is driven by frequently used software concepts.

The current implementation of the tool combines the patterns from Section 10.1 and Section 10.2 as follows:

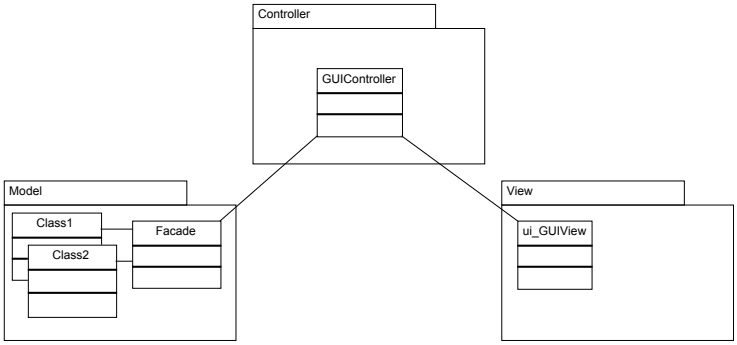

Fig. 10.3: Implementation of MVC and Facade pattern

Figure 10.3 gives a summary of the MVC and Facade implementation. The code is separated into a model, view and controller package which is a set of classes and structs. The core model checking algorithms and data structures for HPNG models and STL formulas are inside the model package. A Facade class is added to the model package to provide an interface between the model and the controller.

The MVC pattern is used to separate data structures and algorithms from the graphical user interface. The views are generated C++ code with the drawing design tool in Qt Creator. The controllers are the glue between the views and the models, i.e. the controller react to events by affecting the views and the models.

The Facade pattern is implemented by providing a simpler interface for the controller. This reduces code complexity for the developers. The Facade interacts with the model by executing the model checker, generating STDs, generating transient probabilities for 2D plots and 3D plots.

The model consists of the existing code, the additional functionalities for model checking and the data structures for STL formulas. The content of the model packages is as follows:

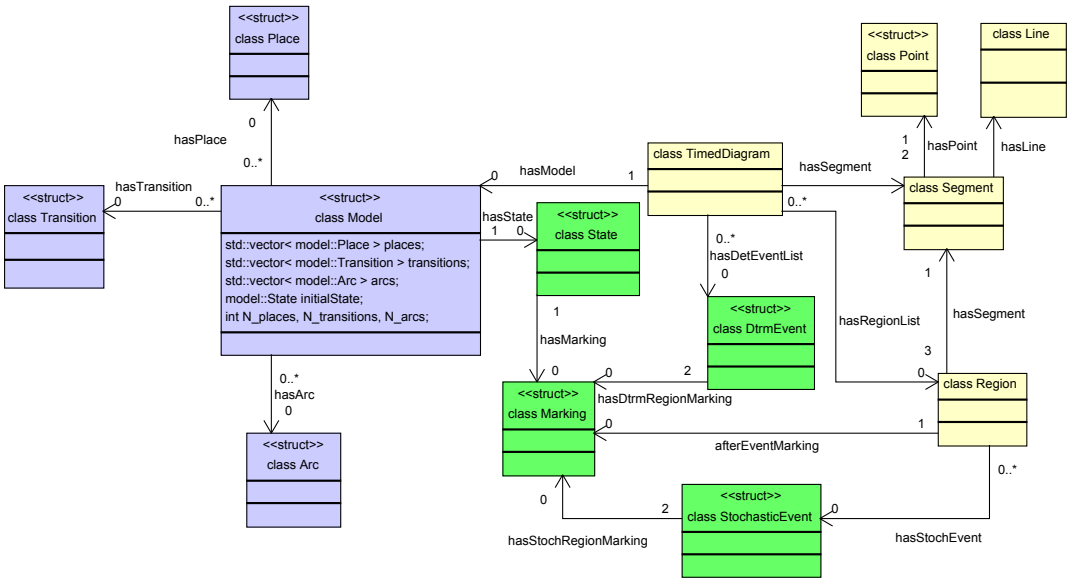

Fig. 10.4: Global design of existing implementation

Figure 10.4 shows the global design of the existing code, that form a great part of the core of the tool, with a UML class diagram. The classes for the data structures of the HPNG models are displayed on the left side of the diagram. In fact, these classes are structs, since they are written in C and C does not support classes. There is no great advantage by converting the structs into classes than consistency and a more object-oriented code. The state of the HPNG models are in the middle. The classes on the right are for generation of STDs.

10.4 Detailed design

Having discussed the conceptual design, the detailed design gives more insight into the actual implementation of the added classes. In this section attributes and operations of added to model classes and controller classes are discussed, i.e. the variables and functionalities of the added classes. The view classes are not discussed in detail, since the view classes are generated by Qt Creator.

First, the details of the Facade (≈ 550 lines of C++ code) and ModelChecker (≈ 850 lines of C++ code) class are discussed and are depicted with a class diagram as follows:

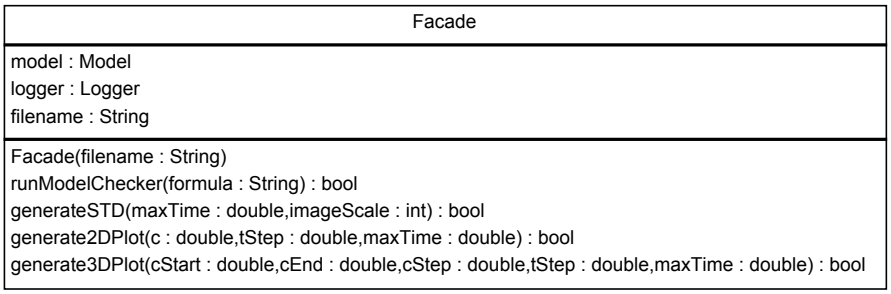

Fig. 10.5: Detailed design of Facade class

Figure 10.5 shows the Facade class which has three main functionalities: Generating STDs in the method generateSTD(...), generating plots in the method generate2DPlot(...) and the method generate3DPlot(...) and model checking formulas in the method runModelChecker(...). The parameters represent the configuration options for the functionality. The Facade class creates an instance of one Model class, String towards a file name and one Logger class. The Logger class is an interface for textual feedback to the controller. A controller implements this interface and passes the class towards the Facade, so it can provide the user with feedback.

Figure 10.6 shows the ModelChecker class that has methods for parsing STL formulas which is ParseFML(...) and for traversing formulas which is TraverseFML(...) and TraverseISetFML(...). Traversing has for every operator in STL a special function ISet...(...) that copes with the interpretation of that type of formula where the set of intervals of the subformulas are known. The resulting set of intervals is written into res for each of these functions for the operators.

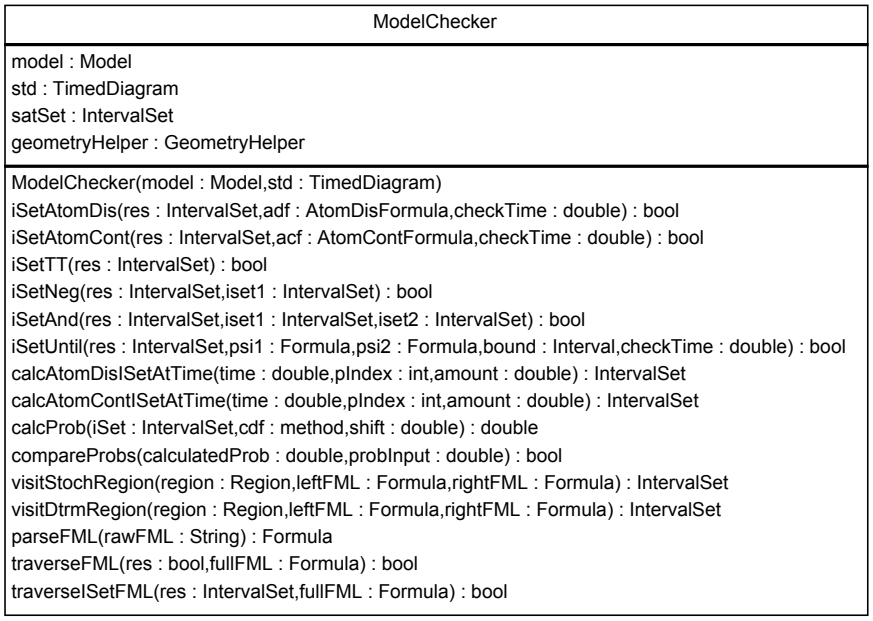

Fig. 10.6: Detailed design of ModelChecker class

The atomic properties have special functions calcAtomDis...(...) and calcAtomCont...(...) that cope with the interpretation according to the existing implementations. The until operator has additional helper functions acquired from the existing implementations: visitStochRegion(...) and visitDtrmRegion(...). For the probability operator some additional functions are added to find a verdict by deconditioning as in Section 4.2 which is calcProb(...) and the comparison of the probability to the calculated probability which is compareProb(...).

In order to understand the full sequence of actions performed, the interactions between processes of the tool is depicted below.

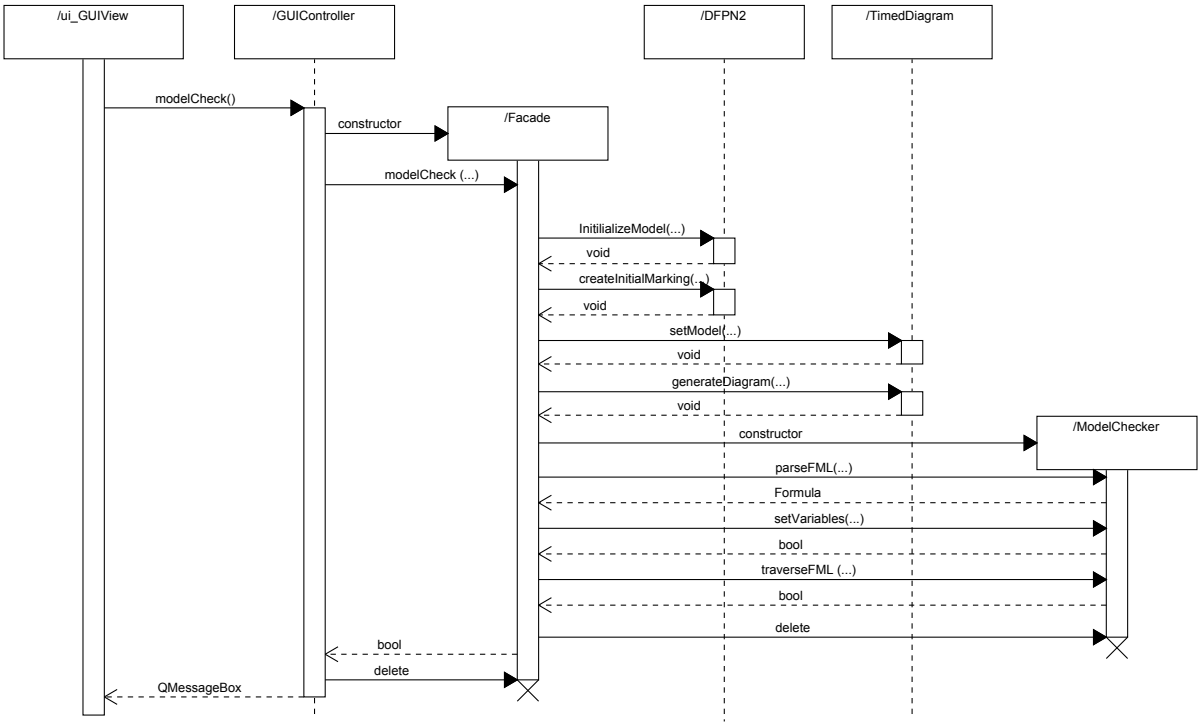

Fig. 10.7: Model checking illustrated in a sequence diagram

Figure 10.6 is a sequence diagram that shows what function are called when the user attempts to model check an STL formula. First, the GUIController ($\approx$ 675 lines of C++ code) class passes the request on to the Facade class. The Facade class loads the model and the current state of the HPNG file. Next, the STD is generated as a preparation for model checking. Now the model checking takes place by lexing and parsing formulas and traversal of the data structure of STL formulas. The exact algorithm is discussed in Chapter 11.

The same procedure holds for the plots, which can be seen as model checking a continuous atomic formula for different moments in time for the generation of 2D plots. Similarly, the generation of 3D plots works along the same lines as for 2D plots, but additionally varies over a range of constants for the continuous atomic formula. Generation of the STD is performed by the first part without the model checker.

Chapter 11

Implementation of the model-checking algorithms

The *general case* for model checking aims at allowing arbitrary STL formulas, as explained in Chapter 5, to be evaluated and analysed. The general case requires the combination of algorithms of formulas with the until operator, conjunction, negation and/or atomic properties. For each of these formulas there was at least an algorithm or an informal description of how individual formulas should be treated available. So, the challenge for this master thesis lays in the evaluation of combined formulas.

Knowing that the evaluation of combined formulas is important, the global strategy for evaluation reuses the algorithms by separating the larger problems into smaller problems that can be solved. To be more precise, the solvable smaller problems are the known algorithms and the larger problem is a combined formula. Section 5.1 mentions the fact that the atomic properties are indivisible. Similarly, the true formula always holds. So, the atomic properties and the true formula are the formulas that are evaluated first. The other operators always depend on other formulas, so such a formula can be evaluated as soon as the formulas that a formula depends on are evaluated.

Therefore, Section 11.2 introduces a lexer and a parser to transform arbitrary formulas into data structures. In more detail, Section 11.1 uses a common data structure to divide the larger STL formula into smaller formulas. Section 11.3 introduces an traversal algorithm through these data structure for evaluating the formulas.

11.1 Data structure of STL formulas

A *data structure* organizes information in such a way that it gives better algorithmic efficiency [4]. The data structure of an STL formula is an *Abstract Syntax Tree* (AST). An AST gives a treelike structure of STL formulas by separating the formula into *subformulas* which is a formula that is a part of the formula. The actual formula is in the root (top) of tree. ASTs can easily be constructed after parsing, which is explained in Section 11.2. Furthermore, ASTs can be traversed easily, because of the treelike structure. An AST can be represented with nodes and edges, where the nodes represent the formulas and the edges point to a subformula of that formula.

Next, a formula is represented as an AST in the following example:

Example 11.1 (AST of an example formula)**.** The formula $Pr \leq p\ (m_{\mathcal{P}} = a \wedge x_{\mathcal{P}} \leq c)$ is represented in with a graphical representation of an AST below.

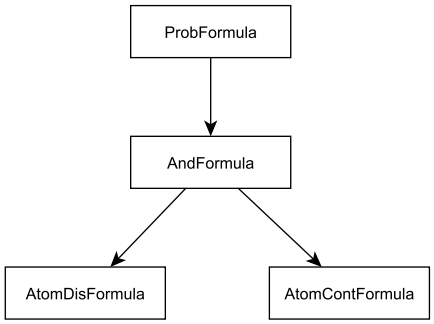

Fig. 11.1: AST of the formula $Pr \leq p\ (m_{\mathcal{P}} = a \wedge x_{\mathcal{P}} \leq c)$

ProbFormula is a formula with a probability operator, that has the highest level of precedence, and which is the top/root of the AST. The highest level of precedence tells which procedure comes first in a mathematical expression. ProbFormula has a child AndFormula. The AndFormula is a formula with an and operator with the highest level of precedence and it has two children: AtomDisFormula as the left child and AtomContFormula as the right child, which are the formulas with only an atomic discrete formula or only an atomic continuous formula. ■

The above example shows an instance of the following structure:

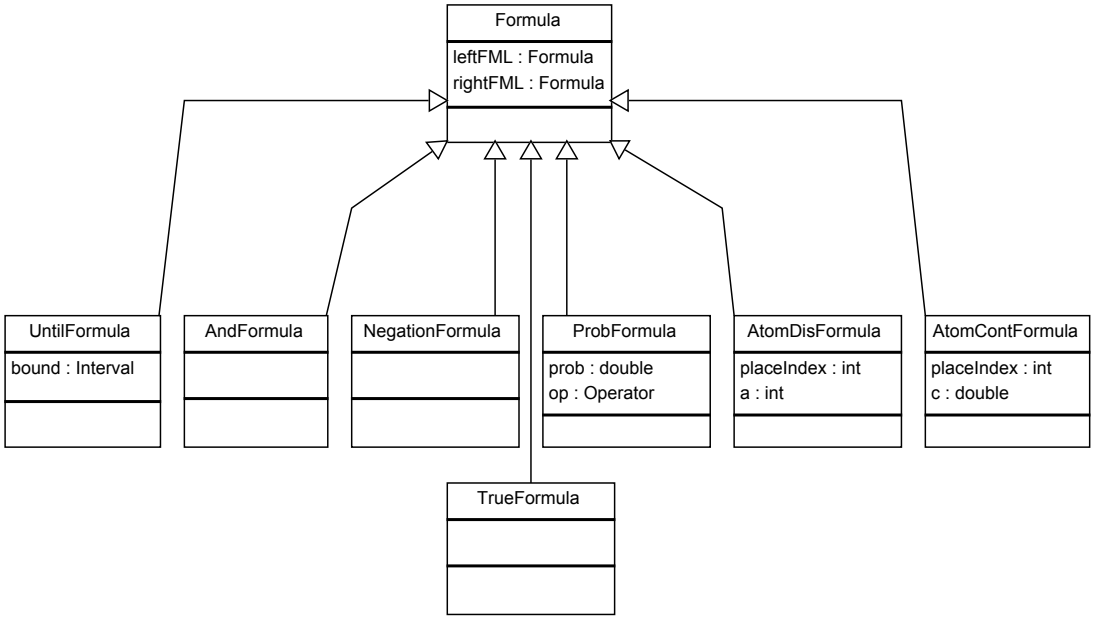

Fig. 11.2: Class diagram of formula data structure

Figure 11.2 shows a class diagram of the implemented AST structure for formulas. All of the formulas have a certain type and by inheritance at most two subformulas (children), i.e. UntilFormula, And-

FORMULA, NEGATIONFORMULA, TRUEFORMULA, PROBFORMULA, ATOMDISFORMULA and ATOMCONTFORMULA are all a FORMULA with a left child (LEFTFML) and a right child (RIGHTFML) where respectively the until operator, and operator, negation operator, true property, probability operator, atomic discrete property and atomic continuous property have the highest level of precedence. Additional information is saved for the time bound of the UNTILFORMULA, the comparison operator and probability for the PROBFORMULA and the place indices and constants for ATOMDISFORMULA and ATOMCONTFORMULA.

11.2 Lexing and parsing STL formulas

Having seen that an AST is a desired data structure, the user input of an arbitrary textual formula needs to be transformed into an AST. The same principles can be found in the design of compilers. Therefore, a lexer and a parser are introduced to perform this transformation. To be more precise, a *lexer* tokenizes a sequence of input characters and a *parser* takes the tokens and builds an AST from these tokens and during this process the syntax of the input is checked.

Writing a lexer and a parser by hand is rather complicated, so a lexer analyser generator for C called FLEX [40] and parser generator for C called BISON [23] are used. The generator uses special languages to describe the lexer and the parser. Obviously, writing in this special language is much shorter than writing formulas hard-coded in C/C++.

The most important part of the special language of Flex is the description of tokens corresponding to certain sequences of input characters. Describing sequences of input character is shortened with the aid of regular expressions. For BISON a grammar for the language of STL is described in Definition 11.2. This is done in such a way that every arbitrary STL formula according to Definition 5.1 and Definition 5.2 is supported.

Definition 11.2 (Grammar for STL formulas)**.**

```
main     : 'PR' '<=' double (expr)
         | 'PR' '<' double (expr)
         | 'PR' '>' double (expr)
         | 'PR' '>=' double (expr)

expr     : expr 'AND' expr
         | 'NEG' expr
         | expr 'UNTIL' '[' double ',' double ']' expr
         | 'M.' variable '=' double
         | 'X.' variable '<=' double
         | 'TT'
         | '(' expr ')'

double   : (([0-9]+(\.[0-9]*)?)|([0-9]*\.[0-9]+))
variable : [a-z][a-z0-9]*
```

■

Definition 11.2 shows the grammar, i.e. a set of structural rules that describe a language, of STL formulas. The grammar begins with an initial non-terminal *main* and all the formulas supported by the tool are derived by this grammar. There is a rule for each possible mathematical operators ($\leq, <, >, \geq$) with the probability operator (PR). The *expr* allows nested expressions with and operators (AND), negation operators (NEG), until operators ($UNTIL$), atomic properties ($M.$ and $X.$) and a true formulas (TT). The *double* and the *variable* are represented with a regular expression that is recognized by the lexer.

The grammar above leaves out the details of writing a lexer in FLEX such as ignoring white spaces, ignoring tabs and tokenizing. The parser constructs an AST by recognizing parts of these grammar and executing some C code, e.g. when the parser recognizes 'NEG expr' an instance of the NEGATIONFORMULA class is constructed in C++ according the AST data structure from Figure 11.2. The regular expressions of the doubles and grammars are put in the lexer and describe the formats of doubles and variables.

11.3 Traversing the data structure of the STL formula

Provided that an AST is created with the lexer and parser, the next step is solving the formulas and subformulas. Therefore, a bottom-up approach is used that evaluates the indivisible subformulas at the bottom and goes up to the root of the AST, which represents the actual formula that should be evaluated.

To be more precise, post-order traversal is used to solve the formula in the AST. *Post-order traversal* first visits the left subformula, then the right subformula and finally the formula itself. In fact, post-order traversal results in a recursive algorithm beginning at the root of the AST.

Algorithm 1 Recursive traversal algorithm for model checking

Require: CurrentFML is a parsed formula according to the grammar of Definition 11.2 that gives an instance of the Formula class as shown in Figure 11.2.

Ensure: The result is a boolean which is the verdict that is generated by evaluation of all subformula of this formula.

```
1:  function traverseFML(Formula currentFML)
2:      IntervalSet res ← traverseISetFML(currentFML.left)
3:      resProb ← calcProb(res)
4:      return compareProbs(resProb,currentFML.prob,⋈)
5:  end function
6:  function traverseISetFML(Formula currentFML)
7:      IntervalSet res;
8:      if currentFML.type = continuous then
9:          res ← iSetAtomCont(currentFML)
10:     else if currentFML.type = discrete then
11:         res ← iSetAtomDis(currentFML)
12:     else if currentFML.type = true then
13:         res ← iSetAtomTT()
14:     else if currentFML.type = and then
15:         IntervalSet leftISet = traverseISetFML(currentFML.left)
16:         IntervalSet rightISet = traverseISetFML(currentFML.right)
17:         res ← iSetAnd(leftISet,rightISet)
18:     else if currentFML.type = neg then
19:         IntervalSet leftISet = traverseISetFML(currentFML.left)
20:         res ← iSetNeg(leftISet)
21:     else if currentFML.type = until then
22:         res ← iSetUntil(currentFML)
23:     end if
24:     return res
25: end function
```

Algorithm 1 provides the pseudo code of the algorithm for traversing the AST in post-order. In this algorithm the functions are called recursively when the subformulas need to be evaluated before

the current formula can be evaluated. Mainly, this is the case for conjunction, negation, the until operator and the probability operator. Moreover, the traversal of most of the formulas works with sets of intervals. Recall Section 6.1, the probabilities are calculated with deconditioning when the probabilities are required for comparison with the probability operator.

Assume that TraverseFML in Algorithm 1 is initially called, a verdict is calculated for an STL formula according to the grammar of Definition 11.2. TraverseISetFML is called in order to retrieve the set of intervals before deconditioning on the model over an probability distribution. Line 2 calculates the set of intervals that satisfy the subformula. The right child of the formula in line 2 is always empty, since this should be a probability operator. In order to calculate the probabilities, line 3 uses deconditioning. Line 4 compares the probabilities for the formula 'Pr $\bowtie$ currentFML.prob (currentFML.left)', where $\bowtie \in \{\leq, <, >, \geq\}$. Every formula type is treated separately in the lines 8-23. More specifically, the conjunction and negation do a recursive call to TraverseISetFML in the lines 15, 16 and 19, while the rest of the statements call algorithms for calculating the set of intervals for each type of formula which is elaborated in Algorithm 2 and 3.

First, the traversal of the until operator is described in the following algorithm:

Algorithm 2 Until operator traversal algorithm

Require: CurrentFML is a parsed formula according to the grammar of Definition 11.2 that gives an instance of the Formula class as shown in Figure 11.2.

Ensure: The result is a set of intervals which is generated by evaluation of all subformula of this formula.

```
1:  function iSetUntil(Formula currentFML)
2:      Polygon pol1 ← traversePolyFML(currentFML.left)
3:      Polygon pol2 ← traversePolyFML(currentFML.right)
4:      ...          ▷ Standard algorithm for until operator using two polygons pol1 and pol2 as in [21]
5:  end function
6:  function traversePolyFML(Formula currentFML)
7:      Polygon res;
8:      if currentFML.type = continuous then
9:          res ← polyAtomCont(currentFML)
10:     else if currentFML.type = discrete then
11:         res ← polyAtomDis(currentFML)
12:     else if currentFML.type = true then
13:         res ← polyTT()
14:     else if currentFML.type = and then
15:         Polygon leftPoly = traversePolyFML(currentFML.left)
16:         Polygon rightPoly = traversePolyFML(currentFML.right)
17:         res ← polyAnd(leftPoly,rightPoly)
18:     else if currentFML.type = neg then
19:         Polygon leftPoly = traversePolyFML(currentFML.left)
20:         res ← polyNeg(leftPoly)
21:     else if currentFML.type = until then
22:         res ← polyUntil(currentFML)
23:     end if
24:     return res
25: end function
```

Recall from Section 6.2 that the until operator returns sets of intervals, but requires polygons that represent the area where the respective subformula holds. Therefore, Algorithm 2 shows the special treatment that is required for analysing the until formula. Since formulas inside an until formula should return polygons for a given time bound, so for each operator and atomic property a special algorithm

is required for calculating the respective polygons. So, the Algorithm 2 is very similar to the traversal of Algorithm 1.

An overview of all functions is structured as follows:

Formula	**Input**	**Output**	**Corresponding algorithm**
Atomic prop. Ψ	Discrete Place Continuous Place	Polygon, Interval set Polygon, Interval set	polyAtomDis(), iSetAtomDis() polyAtomCont(), iSetAtomCont()
$\Psi \wedge \Phi$	2 Polygons 2 Interval sets	Polygon Interval set	polyAnd() iSetAnd()
$\neg\Psi$	Polygon Interval set	Polygon Interval set	polyNeg() iSetNeg()
tt		Polygon Interval set	polyTT() iSetTT()
$\Psi\,\mathcal{U}^{[t1,t2]}\,\Phi$	2 Polygons, $[t1, t2] \mid t1, t2 \in \mathbb{N}$	Interval set	iSetUntil()
$Pr \leq p(\Psi)$	Intervalset, $p \mid 0 \leq p \leq 1$	boolean	calcProbs(), compareProbs()

Table 11.1: Necessary implementations

Table 11.1 shows several required functions for the algorithms that have been elaborated so far. For each type of formula there are several functions derived from Definition 5.1 and Definition 5.2. So each function is capable of generating polygons and sets of intervals, except for the until operator which is restricted to no nested until. The functions with polygon input and output are required by the subformulas inside a (sub)formula that has an until operator with the highest level of precedence. The probability formula obtains a boolean by deconditioning on the general distribution of an HPNG model and by comparing these values to a user-defined probability.

All functions were implemented except for the functions with polygons, since these have restrictions on the type of formulas that can be handled due to concave polygons, i.e. non-convex polygons, which are polygons that have one or more interior angles greater than 180°. The exact problem with concave polygons is that calculating a polygon from a conjunction formula or a negation formula can only be done when the polygon is convex. *Polygon clippers* exists to determine the intersection, union, difference and exclusive-or of two polygons such as the polygons with operators from Table 11.1. However, polygon clipping has difficulty handling concave polygons. Therefore, these functions that require polygon clipping are left open for investigation. So, only atomic properties can be calculated. Furthermore, the until operator gives additional challenges with regard to nested until operators. For now, the actual implemented part of the until operator is up to the atomic properties. However, Algorithm 2 shows how all subformulas inside until operator should be treated differently.

The algorithms for each function that is required according to Table 11.1 are described as follows:

Algorithm 3 Atomic properties and operator algorithms return a set of intervals

Require: CurrentFML is a parsed formula according to the grammar of Definition 11.2 that gives an instance of the Formula class as shown in Figure 11.2.

Ensure: The result is a set of intervals which is generated by evaluation of all subformula of this formula.

```
1: function iSetAtomCont(Formula currentFML)
2:     ...
3:     return An IntervalSet.
4: end function
5: function iSetAtomDis(Formula currentFML)
6:     ...
7:     return An IntervalSet.
8: end function
9: function iSetAtomTT(Formula currentFML)
10:    return [0, ∞)
11: end function
12: function iSetAnd(IntervalSet leftISet,IntervalSet rightISet)
13:    return leftISet ⊓ rightISet;
14: end function
15: function iSetNeg(IntervalSet leftISet)
16:    return iSetAtomTT() − leftISet;
17: end function
18: function iSetUntil(Formula currentFML)
19:    ...                                          ▷ According to Algorithm 2
20: end function
```

Algorithm 3 shows how the different functions, required for traversal, should be treated. So, each function in this list should return a set of intervals. The continuous (iSetAtomCont) and discrete (iSetAtomDis) atomic properties follow the thoroughly described algorithms of [20], which are also briefly explained in Section 6.1. The algorithms for the True formula (iSetAtomTT) is a special case of an atomic property, since it is constant for which the set consists of the largest interval possible in the STD, which is always $[0, \infty)$. Also explained in Section 6.1 are the negation (iSetNeg) and conjunction (iSetAnd). When two formulas are conjuncted, the intersection of the two sets of intervals is taken. When a formula is negated, the largest interval is subtracted from the set of intervals of that formula. The Until formula (iSetUntil) follows the description from Algorithm 2. In the actual implementation of the tool, the Until formula follows the algorithms only for the continuous and discrete atomic properties due to concave polygons.

The combination of the algorithms as in 1 and 2 are considered to be the best option, because this option keeps the existing algorithms and implementation and it does not save information that is unnecessary. However, some other serious options were taken into consideration.

Example 11.3 (Post-order traversal example)**.** The figure below illustrates the traversal of the formula of Example 11.1 with post-order traversal according to the pseudo code from Algorithm 1, 2 and 3. First, the discrete atomic formula is evaluated and returns a set of intervals after *step 1*. Next, the right subformula is visited and evaluated after *step 2* which returns a set of intervals. After *step 3* the subformula ($m_{\mathcal{P}} = a \wedge x_{\mathcal{P}} \leq c$) is evaluated which also results in a set of intervals. Eventually after *step 4*, the probability operator does the deconditioning on the generally distributed transition of the HPNG model and compares the calculated probability to a predefined probability bound which in the end return a boolean, i.e. a model checking verdict.

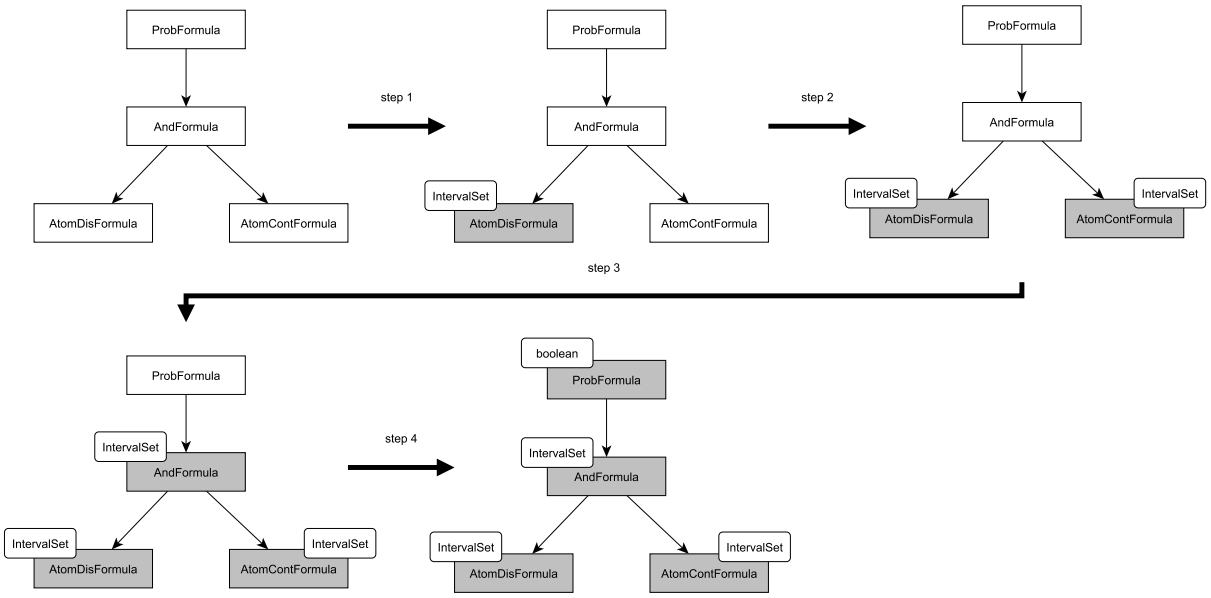

Fig. 11.3: Traversal of the formula $Pr \leq p\ (m_{\mathcal{P}} = a \wedge x_{\mathcal{P}} \leq c)$

If the formula would have been an UntilFormula instead of the AndFormula, Algorithm 2 is used which returns polygons for the underlying subformulas as illustrated below after *step 4*.

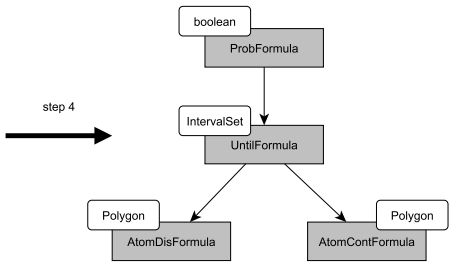

Fig. 11.4: Traversal of the formula $Pr \leq p\ (m_{\mathcal{P}} = a\ \mathcal{U}^{[T1,T2]}\ x_{\mathcal{P}} \leq c)$

The traversal of formulas is structurally performed such that the root, which is the actual formula of interest, can be evaluated with the aid of the functions as seen in Table 11.1 and algorithms mentioned earlier in this example. ■

Chapter 12

Tool presentation

This chapter shows how the tool is integrally presented to the end-user by showing input and output graphical user-interfaces and a connection between the graphical user-interface and the previous chapters. The name of the tool is Fluid Survival Tool (FST) which is named after the capability of the tool to provide insight into critical infrastructure with flows. FST has a graphical user-interface (GUI) which is a requirement from Section 8.5.

The graphical representation are provided for the input, which is elaborated in Section 12.1, and for the output, which is elaborated in Section 12.2. In order for the end-user to be able to interact with FST, an installation and execution is provided in Section 12.3.

12.1 Input representations

First the user interacts with a GUI through a main screen which is depicted as follows:

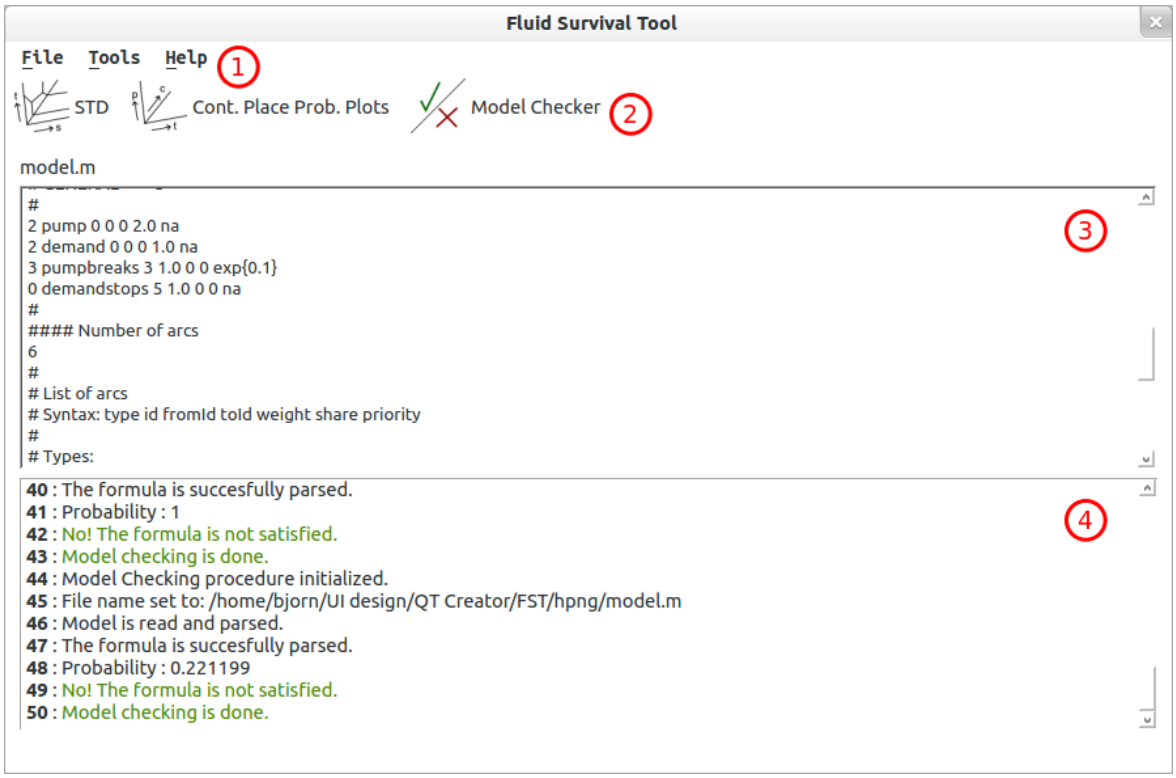

Fig. 12.1: Main screen

Figure 12.1 shows the representation of the tool which consists of a menu bar, buttons and input/output text boxes to support various functionalities. For instance, an HPNG model can be loaded in the menu bar **(1)** and/or edited in the editor **(3)**. The HPNG model that is shown in the editor of Figure 12.1 corresponds to the HPNG model of the water tower example as in Example 3.2. Each functionality has a button with an intuitive icon and textual description **(2)**. Furthermore, feedback is provided to the user with a built-in logger to report errors, successes and information messages **(4)**. The buttons open a dialog with additional configurations. All functionalities are also accessible via the menu bar with additional links to the project website [42] and an about dialog in the menu 'Help'.

The input structure of the HPNG models that is displayed in the editor **(3)** in Figure 12.1 corresponds to the format that is supported by the existing implementations as in Section 8.3. This structure allows to describe places, transitions and arcs of a model according to Definition 3.1. The tool includes several commented examples to demonstrate how these model files are structured.

When the model checking button **(2)** in Figure 12.1 is clicked a model checking configuration dialog pops up which looks as follows:

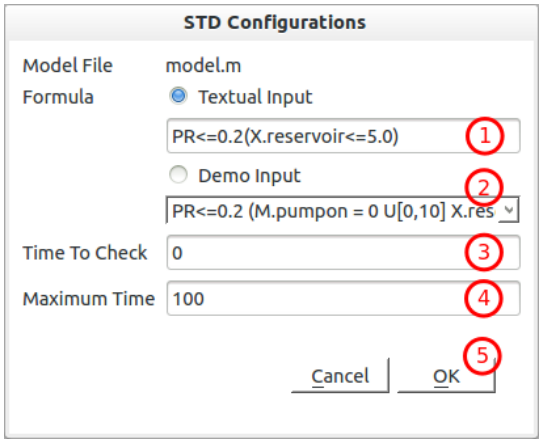

Fig. 12.2: Model checking configuration dialog

Figure 12.2 shows configuration options for model checking which are the formula of interest, the time at which this formula is checked and a maximum evaluation time for the STD. A formula as in Definition 11.2 is either set by textual input of the user **(1)** or a demonstration input is used **(2)** that works with one of the example files. The time at which the formula is checked is inserted in **(3)**. The maximum evaluation time of the generation of the STD **(4)** can influences the accuracy of the calculated probabilities where a higher maximum time can increases accuracy and computation time due to more information about regions in the entire STD. When the user is finished with the configuration, the 'OK' button activates the model checking with the configurations and the 'Cancel' button closes the configuration dialog without performing any model checking **(5)**.

When the continuous place probability plot button **(2)** is clicked from Figure 12.1, the plot generation configuration dialog is depicted as follows:

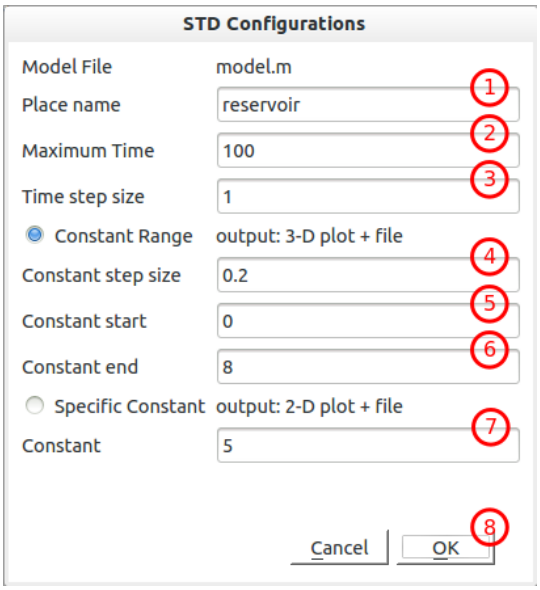

Fig. 12.3: Plot generation configuration dialog

Figure 12.3 shows the configuration options for generating plots which consist of a place name identifier, a maximum time for the plots, time steps and several settings for the constants. The atomic properties as in Definition 5.1 and Definition 5.2 can be checked over a range of time and/or a range of constants. For instance, the atomic property $x_{\mathcal{P}} \leq c$ requires a place name **(1)** and a constant configuration which is either a set of constants **(4)**, **(5)** and **(6)** or a specific constant **(7)**. The start **(5)** and the end **(6)** define the range of constants in combination with the step size **(4)** results in a set of constants. Furthermore, a maximum evaluation time for the plots **(2)** is required, because the time ranges from 0 to the maximum time. Also, the steps of time **(3)** are required, because the algorithms work for a specific time. Finishing the configuration and/or running the algorithms is activated with the 'OK' or 'Cancel' buttons **(8)**. Recall that the plot generation calculates the transient probability distributions over a range of time for either a specific constant that results in a 2D plot or a range of constants that results in a 3D plot as explained in Section 8.4.

When the STD button **(2)**, as shown in Figure 12.1, is clicked the STD configuration dialog is opened:

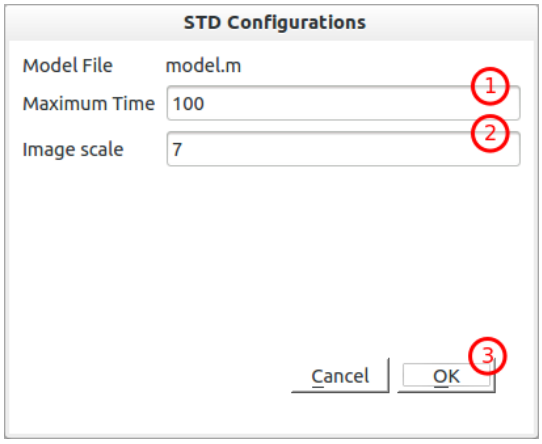

Fig. 12.4: Stochastic Time Diagram generation configuration dialog

Figure 12.4 shows the configuration options for the generation of STD plots which are a maximum evaluation time for STDs **(1)** and an image scale for resizing the output plot **(2)**. Also, an 'OK' and a 'Cancel' buttons are added to run the algorithms and/or finishing the configuration. Recall that the Stochastic Time Diagram generation calculates all the regions for an HPNG as explained in Section 4.2.

12.2 Output representations

The logging text box (4) in Figure 12.1 obtains output from the model via the Logger class as explained in Section 10.4. This provides an interface to send feedback from the algorithms. A controller has implemented this interface and adds the information to the view elements. So textual feedback is presented to the user. Additionally, the feedback by the model checker is a verdict and some probabilities. For all algorithms error messages are provided in case of failure.

When the 'OK' button is pressed in the model checking configuration dialog 12.2, a pop-up window is generated to provide the outcome of the verdict such as following outcome:

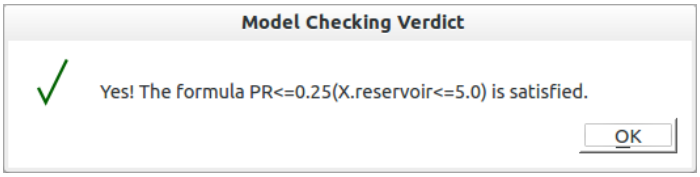

Fig. 12.5: Model checking verdict example output

Figure 12.5 shows the outcome of model checking a satisfying formula. The other outcome of model checking an unsatisfying formula is a similar representation that replaces the check mark with a cross. The same output is provided to the logging text box with some additional probabilities.

When the 'OK' button is pressed in the plot generation configuration dialog 12.3, either a 3 dimensional (3D) plot is generated or a 2 dimensional (2D) plot is generated. The representation of a 2D plot is as follows:

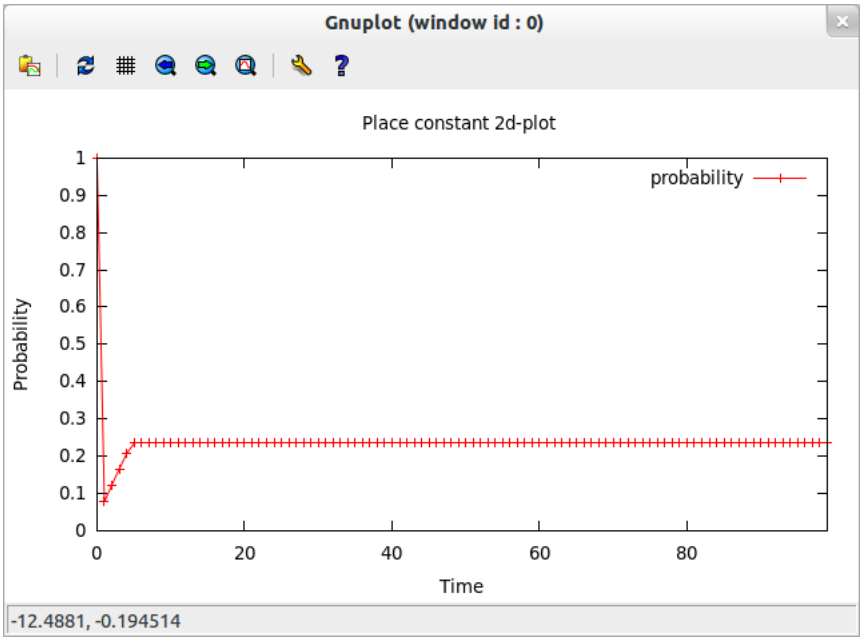

Fig. 12.6: 2D plot output for $x_{Cr} \leq 0.4$

Figure 12.6 shows a 2D plot of the HPNG model from Example 3.2, with one general one-shot transition that has an exponential distributed firing time ($\lambda = 0.1$). It shows the probabilities for a continuous atomic property that the fluid in the reservoir is smaller than or equal to 0.4: $x_{Cr} \leq 0.4$. The x-axis displays the time in hours and the y-axis displays the probability. The time ranges from 0 to 100 with a time step of 1. So, 101 times the atomic properties are calculated for various times.

Note that the actual calculated values are represented by '+', and the lines drawn in the plot between the '+'s are predictions based on the calculated values. So, unexpected values between the calculated values are neglected. The accuracy can be increased by decreasing the time steps or model checking for a specific time for the atomic property.

Initially, the 2D plot shows that there is no water in the reservoir. Obviously, the marking is smaller or equal to 0.4 m^3 at 0 hours with probability 1. The reservoir has an inflow of 2.0 m^3 and an outflow of 1.0 m^3, thus effectively exactly 1.0 m^3 is added each hour. Hence, the drop of the probability to nearly 0 after 1 hour. When time passes the probability that this atomic property holds increases due to the fact that the pump, as explained in Example 3.2, might break. Hence, the probabilities gradually increase over time. The demand stops after 5 hours, so, when the pump break there is effectively 0.0 m^3 added/substracted from the reservoir. Therefore, the probabilities calculated after 5 hours do not change any more and are represented as a flat line around 2.4.

Having explained the representation and intuition of a 2D plot, the representation and intuition of a 3D plot for a continuous atomic property is discussed.

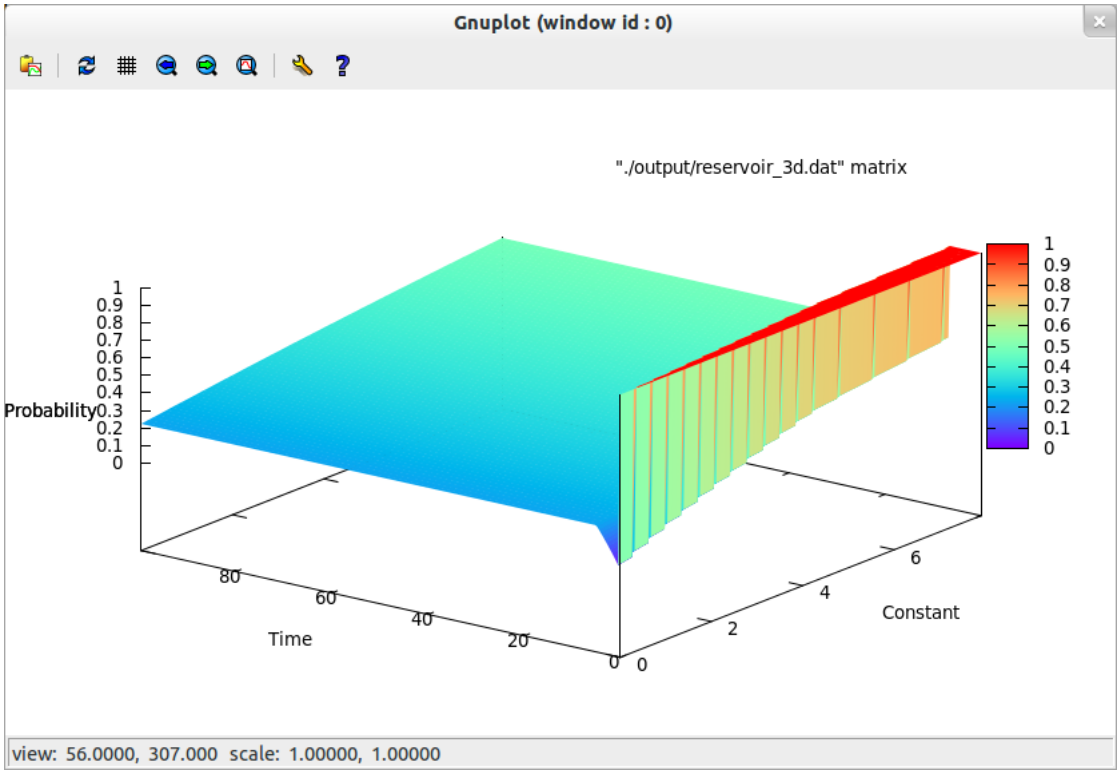

Fig. 12.7: 3D plot output for $x_{Cr} \leq c \mid c \in \{0.0, 0.2, \ldots, 8.0\}$

Figure 12.7 shows a 3D plot for the same HPNG model as for the 2D plot, however, additional options are given to determine the z-axis that display a set of calculated constants. In this 3D plot the range of the constants is from 0.0 m^3 to 8.0 m^3 with a constant step size of 0.2 m^3.

Intuitively, Figure 12.6 shows a slice of Figure 12.7 for the constant 0.4 m^3. Recall that effectively 1.0 m^3 was added to the water reservoir in Figure 12.6, the area where the probability in the 3D plot is 1, starts for this reason at the moment that the constants become greater than 1. The tool allows to rotate the 3D plot, therefore, another slice (2D plot) illustrates a constant 5 to discover the hidden area of the 3D plot with following figure:

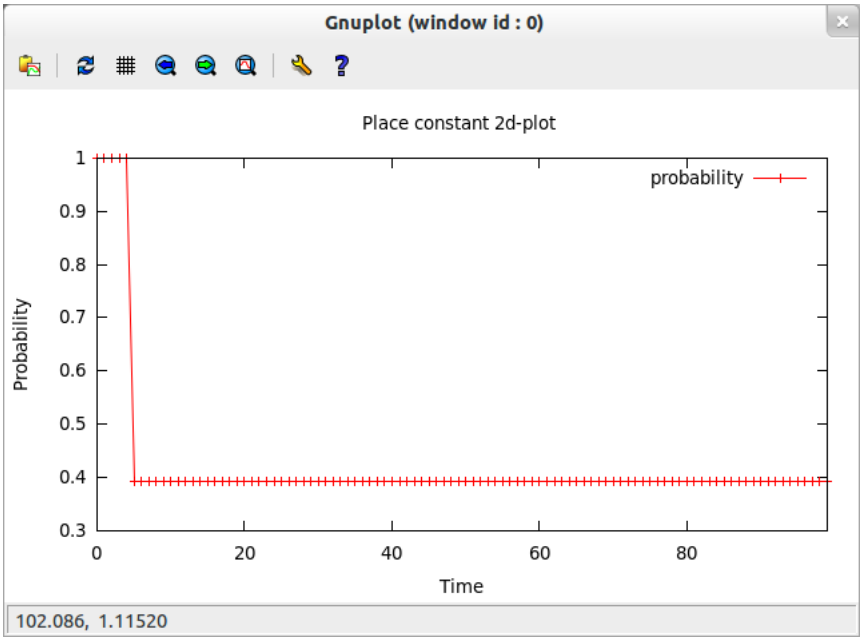

Fig. 12.8: 2D plot output for $x_{Cr} \leq 5$

Figure 12.8 differs from Figure 12.6 in that it does not gradually increase after the drop of the probabilities, the probabilities are 1 for a longer period of time and the values of the constants are higher. After 5 hours the demand stops, so for the constant 5 the probability of 1 directly drops to a constant probability value (≈ 0.38) and does not gradually increase as in Figure 12.6. So, this information and more is provided at once by rotating and reasoning about the 3D plot illustrated in Figure 12.7.

12.3 Installation and execution

An installation guide is provided in the following, in order to prepare the tool for execution. The installation is performed on a 64-bit LINUX machine and for the installation UBUNTU 12.04 LTS is recommended. When the system is started and the user is logged in, the latest release can be downloaded on the project website [42] in the download section or by downloading and compiling the source in QT CREATOR. In order to be certain that all features work properly, a number of libraries are required to be installed. The user executes the terminal and performs the following command:

```
$ sudo apt-get install libqtcore4 libqtgui4 gnuplot-x11 libmatheval1
```

This command installs all required libraries for QT (libqtcore4, libqtgui4), plotting (gnuplot-x11) and evaluation of mathematical expressions (libmatheval1). All other libraries are included in the release.

FST is executed in a terminal or in a file browser by double-clicking the executable. FST is executed in the terminal as follows:

```
$ sudo chmod +x ./FST
$ ./FST
```

As a consequence, the main window as in Figure 12.1 pops up and is ready to use.

Part III

Case Study & Conclusions

Chapter 13

Case Study: Sewage water cleaning facility

Recent news [49], [44] and [51] shows how flooding of a sewerage facility caused trouble in the homes and streets of Enschede, the Netherlands, and how three bachelor students utilize HPNs with general transitions, as explained in Section 7.1, to model the exact part of a sewage water cleaning facility, that caused this flooding, and evaluate a simulation program and the tool FST which is introduced in this master thesis. Figure 13.1 and Figure 13.2 show a flooded viaduct and a flooded residential area.

Fig. 13.1: Flooded viaduct in Enschede

Fig. 13.2: Flooded residential area in Enschede

The bachelor report [36] shows models of a sewage water cleaning facility as an HPN and evaluates using FST and simulations. FST cannot handle more than one general transition, so an HPNG, as elaborated in Chapter 3, version of the model is created to validate FST, which is based on region-based analysis, with the simulation program. Research from [36] is performed in cooperation with the validation of FST with the simulation program. This gave promising validation of FST, but also the other way around for the simulation program.

This chapter uses the basic water treatment HPNG model from [36] in order to discuss the scalability of HPNGs which shows additional feasibility of the tool. The basic water treatment model is scaled up with additional elements in the HPNG model. Especially measurements of the model checking part provides insight into the scalability of the HPNGs.

First, the basic water treatment with an extension for scalability is discussed in Section 13.1. Next, the measurements and discussion of the scalability are elaborated in Section 13.2. The validation performed in cooperation with the report [36] is explained in Section 13 together with some findings with respect to the existing code.

13.1 Water treatment HPNG models

When the sewage water cleaning facility cleans sewage water, there is a certain maximum capacity to the amount of water that can be processed. Therefore, water needs to be stored in a buffer first that is owned and maintained by the community of Enschede. In the case of heavy weather, the inflow increases so much that even the buffer can reach its maximum capacity, so flooding takes place. The HPNG model of the water treatment facility is modelled as follows:

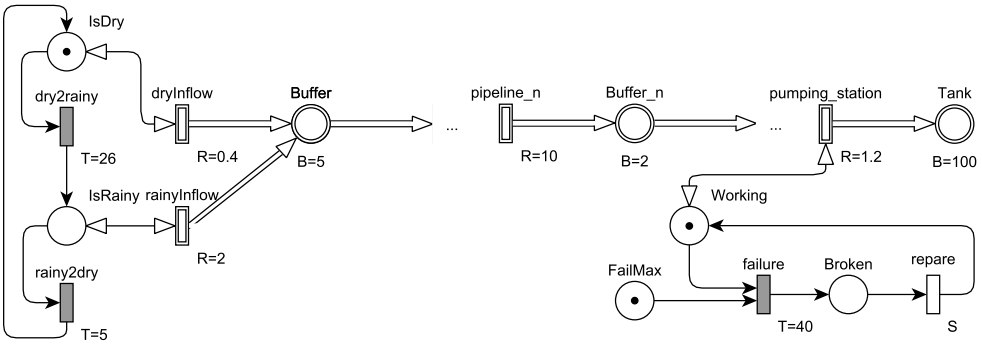

Fig. 13.3: Water treatment HPNG model

Figure 13.3 depicts part of the water treatment facility where the water is stored in a continuous place 'Buffer' with a capacity of 5 m^3 until enough water is transferred from the buffer to a cleaning tank through a pumping station. When the weather is dry the inflow of the continuous transition 'dryInflow' to the buffer is smaller than when weather is rainy as depicted by the continuous transition 'rainyInflow'. Therefore, a token is fired from transitions between the discrete place 'IsDry' in which the weather is dry and the discrete place 'IsRainy' in which the weather is rainy. Real weather conditions are very hard to model, so this model is not very realistic. However, we are currently mostly interested in specific situations. The weather changes after 5 hours ($T = 5$) from rainy to dry with the deterministically timed transition 'rainy2dry' and from dry to rainy after 26 hours ($T = 26$) with the deterministically timed transition 'dry2rainy'. The water goes into the tank, depicted by the continuous place 'Tank' with a maximum capacity of 100 m^3, by a pumping station with a pumping speed of 1.2 m^3. However, the pumping station can break, which is depicted by a discrete place 'Working' and 'Broken'. The pumping station breaks after 40 hours ($T = 40$) according to a deterministically timed transition 'failure' and the token is fired between the places 'Working' and 'Broken'. The reparation time of the pumping station is exponentially ($\lambda = 0.2$) distributed and depicted by the general transition 'repare'. This describes the basic HPNG model for water treatment as shown in [36].

Having seen the basic HPNG model for a water treatment facility, an extension is added to increase the number of continuous places and transitions which increases the complexity. For this reason, extra buffers are added before reaching the pumping station that moves the water into the tank. The extra storage of the buffers are depicted as the continuous places 'Buffer_n', where n is equal to the amount of added buffers. The buffers are connected by pipelines that have the advantages of gravity and are therefore very quick compared to the pumping station. The pipelines are depicted by the continuous transitions 'pipeline_n'. Note that 1 added buffer is associated with 1 additional place, 1 additional transition and 2 additional arcs in the HPNG models. The HPNG models for water treatment are now extended such that the model can be scaled for analysis of the execution times of the tool.

13.2 Scalability measurements

The number of elements of the HPNG model of a water treatment facility, as elaborated in Section 13.1, is scaled according to the description in this section. The measures of interest are the execution times for the various HPNG models of a water treatment facility that scale the number of buffer. The following table shows measurements for the HPNG model of a water treatment facility against the STL formula $Pr \leq 0.2(\mathit{Buffer} \leq 0.2)$, which model checks if the probability, that the buffer at specific time ($t = 45$) has at least 0.2 m^3, is smaller than or equal to the fixed probability 0.2, i.e. can we state with at most 20 % certainty that after 45 hours the buffer has at least 0.2 m^3:

#Buffers	#Regions	SGT	CTA	TET
1	444	1.02	0.00	1.05
2	539	1.08	0.00	1.27
3	681	2.02	0.00	2.03
4	847	2.11	0.00	2.43
5	984	3.01	0.02	3.05
6	1092	3.06	0.03	3.23
7	1237	4.00	0.06	4.04
8	1392	4.18	0.06	4.92
9	1489	5.01	0.08	5.11
10	1556	5.07	0.12	5.49
11	1608	5.94	0.17	6.21
12	1677	6.07	0.21	6.50
24	2228	12.43	1.01	13.06

Table 13.1: Model checking $Pr \leq 0.2(\mathit{Buffer} \leq 0.2)$

Table 13.1 shows the measures of interest for HPNG models ranging from 1 to 24 added buffers. The first column shows the number of buffers added to the HPNG model. The second column shows the number of regions generated for the STD that correlates with the complexity and the generation time of an STD. The *total execution time* (TET), which is either a run of the model checker or generation of a plot, is separated into two measures of interest: The *time for generation of an STD* (SGT) and the *computation time of the model checking algorithms* (CTA), which are all displayed in the remaining columns. All execution time measurements in this section are in *milliseconds*. The reason for the total execution time to be mostly larger than the sum of SGT values and the CTA values is that in the end some feedback is provided to the user and objects are created and deleted which are written and removed from the memory.

Computer often run other processes which influences the accuracy execution times, but also make these execution times relative and dependant of the environment and settings. Hence, the settings, accuracy and environment are taken into account. The time to check was 45, the maximum time was 100 and the general transition was exponentially distributed ($\lambda = 0.2$). The measurements of the execution time are the averages of 200 runs for each row to increase the accuracy of the measurements. All measurements are performed on a machine equiped with a 2.30GHz Intel® Core™i7-3610QM CPU, 8 GB of RAM and Ubuntu 64-bit.

A clear significant correlation between the number of added buffers and the number of regions, SGT values, CTA values and TET values is found by statistical testing if there is a significant linear correlation. This is tested by calculating Pearson's correlation coefficient ρ for the four combinations of the number of added buffers and the number of regions, SGT values, CTA values and TET values and comparing ρ to critical values cv to a certain level of significance α that depends on the degrees of freedom ($df = n - 2 \mid n =$ sample size). A linear correlation exists, when $\rho_{Regions}, \rho_{SGT}, \rho_{CTA}, \rho_{TET} \geq cv$ holds for an α. This is the case, since $\rho_{Regions} \approx 0.946, \rho_{SGT} \approx 0.998, \rho_{CTA} \approx 0.922, \rho_{TET} \approx 0.998, cv =$

$0.6835 \mid \alpha = 0.01, df = 11$). So with 99% confidence, a significant linear correlation exists between the number of added buffers and the number of regions, the number of added buffers and SGT values, the number of added buffers and CTA values and the number of added buffers and TAT values. This significant linear correlation shows that the computation time, generation time of the STD, execution time and the number of regions grow linear with the number of added buffers with a confidence of 99%.

Model checking is done for various operators which influence the computation time, because the traversal takes more time and more calculations need to be made for one execution. First an STL formula with a conjunction $Pr \leq 0.2(\mathit{Buffer} \leq 0.2 \wedge \mathit{Buffer\_1} \leq 0.2)$ is model checked and the execution times are measured for the HPNG model for water treatment.

#Buffers	SGT	CTA	TET
1	1.12	0.01	1.28
12	6.08	1.07	7.20

Table 13.2: Model checking $Pr \leq 0.2(\mathit{Buffer} \leq 0.2 \wedge \mathit{Buffer\_1} \leq 0.2)$

While scaling the number of buffers in Table 13.2 a large computational time compared to the simple STL formula $Pr \leq 0.2(\mathit{Buffer} \leq 0.2)$ from Table 13.1 is found (≈ 1.07 ms vs. ≈ 0.21 ms), while the simple cases hardly show a difference (≈ 0.01 ms vs. ≈ 0.00 ms).

Next, an STL formula with a negation $Pr \leq 0.2(\neg \mathit{Buffer} \leq 0.2)$ is model checked and measured with the same HPNG model as follows:

#Buffers	SGT	CTA	TET
1	1.04	0.00	1.05
12	6.05	1.01	6.85

Table 13.3: Model checking $Pr \leq 0.2(\neg \mathit{Buffer} \leq 0.2)$

Also, an STL formula with a single until operator is model checked and measured as follows:

#Buffers	SGT	CTA	TET
1	1.01	0.00	1.02
12	6.06	1.10	7.18

Table 13.4: Model checking $Pr \leq 0.2(\mathit{IsDry} = 1 \; \mathcal{U}^{[0,10)} \, \mathit{Buffer} \leq 0.2)$

Table 13.3 and Table 13.4 show larger computation time compared to the measurements from Table 13.1 with 12 additional buffers similar as stated about Table 13.2.

A larger STL formula $Pr \leq 0.2(\neg(\mathit{Buffer} \leq 0.2 \wedge (\mathit{IsDry} = 1 \; \mathcal{U}^{[0,10)} \, \mathit{Buffer\_1} \leq 0.2)))$ is model checked and measured for the HPNG model for a water treatment facility as follows:

#Buffers	SGT	CTA	TET
1	1.01	1.00	2.01
12	6.09	4.09	10.17

Table 13.5: Model checking $Pr \leq 0.2(\neg(\mathit{Buffer} \leq 0.2 \wedge (\mathit{IsDry} = 1 \; \mathcal{U}^{[0,10)} \, \mathit{Buffer\_1} \leq 0.2)))$

Table 13.5 shows the execution times for two cases with 1 extra buffer and with 12 extra buffers. The number of regions is left out, since these are the same as in Table 13.1. Note that the SGT is not exactly the same, this is probably due to processes that run in the background during execution. The measures of interest in this table are the CTA values that is increased due to extra computations for the larger STL formula compared to Table 13.1, where the computation time is .

Originally, the existing code only had the exponential distribution as probability distribution for the general transition. However, the strength of HPNGs is that it supports all continuous probability distribution functions. Therefore, in this research the uniform, folded normal, gamma and deterministic probability distributions are added to the source code. Additionally, a general function can be entered with a variable s to describe an arbitrary cumulative distribution function. FST requires an extra library to evaluate mathematical functions for general functions. The following table shows the execution times for using the various probability distributions:

Prob. distr.	**Parameters**	**SGT**	**CTA**	**TET**
exponential	$\lambda = 0.2$	6.07	0.21	6.50
uniform	$a = 0, b = 100$	6.10	0.18	6.50
folded normal	$\mu = 0, \sigma = 100$	6.05	0.22	6.50
gamma	$K = 10, \lambda = 100$	6.04	0.16	6.48
deterministic	$d = 10$	6.07	0.14	6.51
general function	$1 - e^{-0.2 \cdot s}$	6.07	0.27	6.64

Table 13.6: Model checking $Pr \leq 0.2(\mathit{Buffer} \leq 0.2)$ with various probability distributions

The paper on region-based analysis [20] states that when a closed form of a CDF exists, the choice of the distribution does not influence the complexity of the region-based algorithm. Table 13.6 shows that this is actually the case, because the execution times (SGT, CTA and TET) are not influenced much for various probability distributions. For this measurement 12 extra buffers were added to the HPNG model.

Currently, most of the execution times in Table 13.1 are for the generation of the STD, however, when generating 2D and 3D plots for transient probability distributions, the execution time is much longer due to writing data to files and plotting all the data into a plot. The measurements for generating a 2D and 3D plot for exactly 1 run are as follows:

	2D plot			**3D plot**		
#Buffers	**SGT**	**CTA**	**TET**	**SGT**	**CTA**	**TET**
1	1	3	158	1	102	277
2	1	3	180	1	128	360
3	2	4	153	1	172	406
4	2	4	172	3	218	397
5	2	5	166	3	261	399
6	2	7	176	3	300	468
7	3	7	170	5	351	527
8	3	8	171	5	396	543
9	3	9	178	5	425	614
10	5	12	172	5	445	671
11	5	13	185	5	466	700
12	6	14	174	6	507	747
24	12	19	182	13	708	896

Table 13.7: 2D & 3D plot generation

Table 13.7 shows an increase in the computational time and the total execution time in comparison to Table 13.1 due to writing data files and plotting the data. Each calculated probability is written into a file which increases the CTA values. When all probabilities are found, a plot is generated from the data with the aid of a plotting library which results in the overhead in TET. Another increase in measured when scaling the buffers for the CTA and TET values. Note that, that subtracting the CTA values from the TET values (2D: $\approx$ 160 ms, 3D: $\approx$ 200 ms) gives the time required for generating a

2D or a 3D plot. The CTA values increase for the 2D plot and the 3D plot around 6-7 times and the SGT values increase around 12-13 times when comparing the HPNG model with 1 extra buffer with the HPNG model with 24 extra buffers. This means that the STD generation time increase with ≈ 0.5 ms for each added buffer and increase ≈ 0.3 ms for each added buffer with the computation time due to linear growth (significant linear correlation).

The main advantage of generation of 2D and 3D plots, that calculates transient probability distributions over a range of time, is that generation is based on the model checker class which makes FST extendable. At least the generation of 2D plots with syntax corresponding to Definition 5.1 should be a trivial to implement in the future. Another advantage of obtaining 2D plots from the model checker algorithms is avoiding of duplicate code.

The most important remarks found in this section are the significant linear correlation of 99% confidence between the increase in the buffers and the number of regions and execution times.

13.3 Tool validation

The data generated by the simulation program is analysed and compared to the data generated for 2D plots of FST for the water treatment facility HPNG model of Figure 13.3 without added buffers, but with an increased buffer capacity equal to 20. This did not only show the validation of the simulation program, but also the validity of FST presented in this research. An 2D plot with a general transition with a deterministically distributed firing time 10 hours ($d = 10$), i.e. a general transition that behaves like a deterministically timed transition, is shown below:

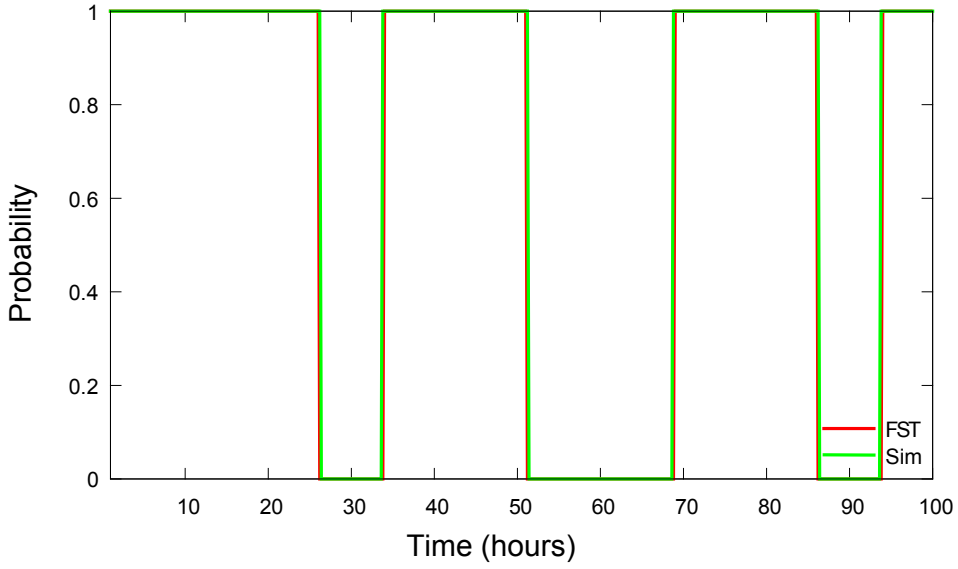

Fig. 13.4: FST vs. simulation program with a deterministic distributed firing time

Figure 13.4 shows that a fully deterministically timed model with the continuous atomic property $\mathit{Buffer} \leq 5.0$ obtain the same probabilities, since the lines for FST and the simulation (Sim) overlap. Intuitively, the 2D plot shows the cycle between rainy weather, where the calculated probability is 0, and dry weather, where the calculated probability is 1. Between 50 and 70 hours the probability remains 0 for a longer period of time, which is due to fact that the pump is broken after 40 hours and more water is pumped into the buffer. When the rain starts the buffer easily exceeds 5.0 m^3 and since the capacity is around 20 m^3 it takes more time to pump the water away. This is the reason why the probability is 0 during a period where the weather is dry.

More challenging is a model with a general transition with a uniform distributed firing time. The accuracy of the simulation program is determined by the number of runs. Therefore, an 2D plot of executions of such a model with a uniform distributed firing time ($a = 0, b = 20$) and various number of runs versus an execution of the same model in FST is shown below:

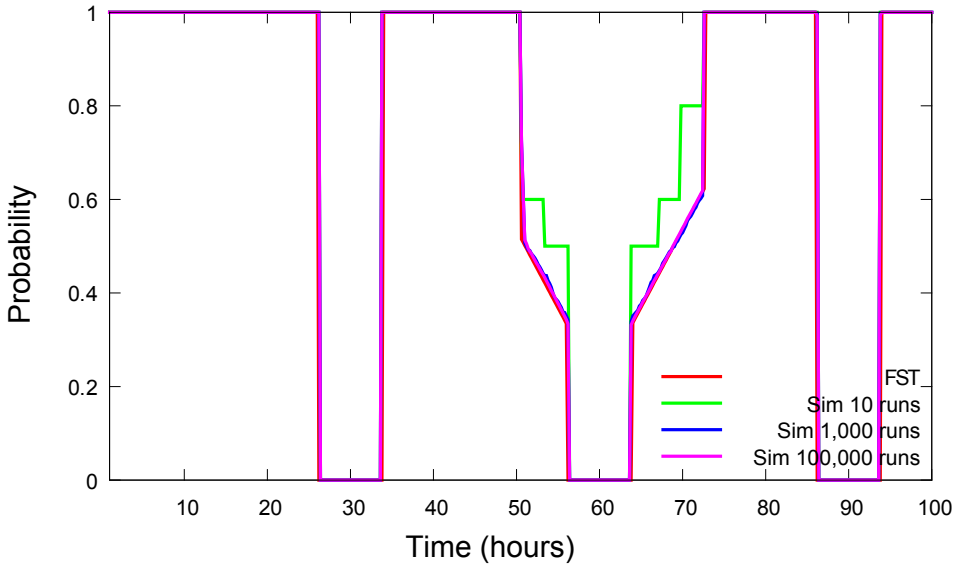

Fig. 13.5: FST vs. simulation program with a uniform distributed firing time

Since the reparation time is according to a uniform probability distribution, various probabilities are depicted for the transient probability distributions.

Also, a 2D plot with a general transition with an exponentially distributed firing time ($\lambda = 0.05$) for various number of runs versus FST is shown below:

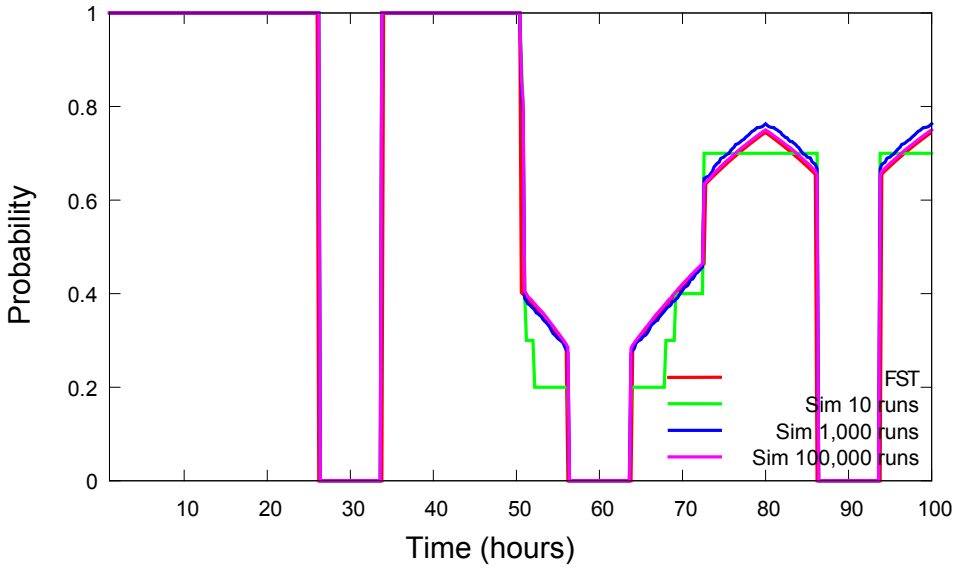

Fig. 13.6: FST vs. simulation program with a exponentially distributed firing time

Figure 13.6 and Figure 13.5 shows a 2D plot of the calculated probabilities for transient probability

distributions over range of time $Buffer \leq 5$ for FST, 10, 1,000 and 100,000 simulation runs. The overlap of the simulations and FST become more accurate when the number of simulations increase.

The report [36] states that 10 % more computation time is required for the simulation program with 100,000 runs, which is sufficient for a confidence interval of 95 %, compared to FST with 1 run, which is exact. In contrast, the simulation program can support more than one general transition.

More details are discussed in Chapter 6 of [36]. For instance, validation with a general transition with uniform distributed and deterministically distributed firing times.

During the tool design useful feedback was provided to solve bugs while using FST.

Also during development, an inconsistency was found in the existing code with respect to [20] on calculating a probabilities through integration. The integration is from 0 to ∞ in the paper, while in the existing code it is up to maximum time for which the STD is generated. So, the existing code is adjusted to infinity, otherwise probability mass is lost.

A bug occurred during the generation of 3D plots due to adjustments made in the existing code for model checking the until operator. After debugging, the bug was removed. The problem is solved by removing a piece of code that probably was necessary to handle some special cases, that looked unfamiliar.

Another bug emerged from the existing code when running FST on an European machine, since the notation for digits is different from an American machine, e.g. European 2,00 vs. American 2.00. So, HPNG model files could not be transformed into instances of HPNG model data structures. So, FST now forces the use of the dot from the American machine.

Chapter 14

Conclusions

This master thesis introduces the Fluid Survival Tool (FST) for model checking HPNG models against STL properties. Requirements and domain analysis helped to identify the needs of FST before implementing any code. The tool has been implemented with a graphical user-interface and a source code following the Model-View-Controller and Facade patterns as explained in a global and detailed design. Furthermore, a general algorithm for model checking STL formulas has been introduced that supports nesting for all operators except the until operator. Additionally, the support of additional probability distributions have been added to the HPNG models. FST has been tested and in cooperation with this research used for validation by bachelor students from the mathematical department which shows the feasibility of FST. Moreover, a case study has shown the feasibility of FST by scaling an HPNG model for sewage water treatment.

Section 14.1 describes the state of the art when this research was started. Next, Section 14.2 describes the overall results of the tool design. Section 14.3 concludes that full STL support is obtained from traversing a data structure and applying the algorithms as explained in Chapter 11. FST has been validated with a simulation program and a case study that shows the scalability which is elaborated in Section 14.4. Suggestions for future work on FST, analysis and algorithms are elaborated in Section 14.5.

14.1 Literature and implementations basis

Literature [21] described model checking algorithms and analysis of HPNG models against STL specification. For the HPNG models the definitions existed together with the data structure implementation and parsing of HPNG model files. However, HPNGs are restricted to exactly one general one-shot transition. Parametric reachability analysis [26] and region-based analysis [20] are efficient analysis methods that separate the deterministic and the stochastic evolution of the system and allows to evaluate transient probability distributions with the aid of a conditioning and deconditioning technique. STL specification follows from this analysis by adding logical operators to analyse additional state and path properties in combination with the transient probability distributions. An probability operator raises the probabilities found by these STL formulas back to the level of HPNG models. The model checking algorithms from [21] describe individual algorithms for treating the state and path properties. The existing implementations have the parsing and data structures of HPNG models and individual algorithms that obtain probabilities from these HPNG data structures.

14.2 Model checking tool design

An actual model checking tool has been developed by applying well-known software engineering principles. The software engineering process consists of exploration of requirements, domain analysis, global design, detailed design, implementation and validation. This process lead to an extendable, object-oriented model checking tool with a graphical user-interface (GUI), a software development kit (SDK)

focused on the developer which is executable on a 64-bit LINUX machine. FST is based the papers and implementation [26], [20] and [21] as described earlier in Section 14.1. The model checking tool allows to parse HPNG models and STL specifications from a textual input file (model) and a textual input field (specification). A built-in editor allows the user to modify, load, save, create and close HPNG models easily. FST generates, stores data and stores plots of verdicts, plots of transient probability distributions over a range of time and plots of Stochastic Time Diagrams. Internally, the GUI and core algorithms cooperate with the aid of the Facade and Model-View-Controller patterns as explained in Chapter 10.

14.3 Nested STL support

The analysis of STL formulas with nested structures are not described in existing literature and therefore, algorithms for traversing an STL abstract syntax tree (AST), which is a data structure, has been introduced using algorithms from [20] and [21]. Before any instances of the data structure were traversed, a textual user input field containing an STL formula needs to be transformed into the data structure. So, a parser has been designed and implemented with the aid of defining a grammar in a parser and lexer generator (FLEX/BISON). Next the ASTs are traversed that leads to two main branches: Traversal of nested Until formula and traversal of all other formulas. The traversal algorithms are ready for polygon algorithms, but these algorithms are currently still under development. Normal traversal results in probabilities from sets of intervals, while Until formulas requires to reason on polygons in the Stochastic Time Diagrams. So, the STL support allows nested STL formulas where the Until formula is currently supported up to two atomic properties.

14.4 Tool feasibility

The feasibility of FST is shown by evaluating different STL formulas and analysing the scalability of FST. Furthermore, results in FST have been validated with a simulation program in cooperation with bachelor students from the mathematical department. An elaborated case study on an HPNG model of sewage water treatment shows similar probabilities. The simulations took considerable longer than FST. The most accurate simulation run took approximately 10 times longer than FST. The analysis on the scalability shows a linear correlation between the computation times and the number of regions when increasing the number of continuous variables, with a confidence of 99 %. Moreover, two bugs and an inconsistency were exposed in the existing code and have been solved throughout the conduction of this research.

14.5 Suggestions

Two areas of suggestions are presented in the following, namely suggestions for the tool and suggestions for the algorithms. The tool suggestions are as follows:

XML/Language model files XML/Language model files provide advantages compared to the current HPNG model files. The HPNG model files contain a lot of useless information and feedback provided on from the models was often too coarse. During development better feedback was added, however, a lot of issues are still hard to trace. When parsing the HPNG model file is not treated like a language, but rather like a file with lines that instruct the parser (e.g. every time a place, transition or arc was added a counter in the model needs to be adjusted). XML/Language model files are easier to modify, reduce unnecessary information and provide better feedback to

the user. Furthermore, XML files are currently widely used to install tools as components of a greater system.

Extend user-interfaces Currently, a graphical user-interface is implemented on top of a Model-View-Controller pattern that allows to implement additional Views and Controllers for the same model. The tool design is extendable and therefore provides the capability of extending to other user-interfaces such as command-line, textual user-interfaces, web-based and mobile user-interfaces. Even the graphical user-interfaces can be extended for different groups of users that all use the same core.

Extend to other platforms and add 32-bit support Currently, at least a 64-BIT UBUNTU machine is supported. The software development kit (SDK), that contains QT, QT CREATOR and GCC, supports multiple platforms. The SDK can easily be transported to other platforms by putting some effort in compiling or finding libraries used for the tool and configuring the QT CREATOR project files. The advantage of extending to other platforms is that it increases the group of potential users.

Plots for STL formula Currently, transient probability distributions over a range of time have 2D and 3D plots which could be easily extended to STL formulas as in Definition 5.1. The plotting code is developed in such a way that it makes use of the model checking class. So, extending towards other STL formulas to provide more insight into the probabilities over a range of time for certain STL formulas is easy to implement.

Embed simulation program in FST The related research by the bachelor students from the mathematical department presented a simulation program that could be embedded into FST. Simulation can be used for models that are not yet supported by FST (e.g. a model with more than one general transition). Also, simulation is useful for validation of extensions to the analysis and algorithms.

Tool maintenance A tool needs maintenance which is a suggestion in order to keep the tool smoothly running and useful in the hands of the users. Keeping the tool up to date to recent research and relevant case studies is important for the tool to remain valuable. Important to keep in mind with tools is that publications such as web pages are important to keep up to date, especially when the tool becomes outdated. This important suggestion is often neglected which was often encountered during a study on related works.

Note that tool implementations are also required for most of the following suggestions on the algorithms:

Parallel computations of subformulas The traversal algorithms as proposed in this research can be optimized with the aid of parallel computation. Parallel computation of subformulas of the abstract syntax tree decreases the computation speed. Note that, parallel computation enables to get the most out of the hardware available and does not improve the complexity.

Optimize interpretation of logical operators Another improvement of the traversal algorithms would be to finish evaluation as soon as the set of intervals are known during traversal. For instance, the conjunction operator might evaluate a discrete atomic formula. Therefore, the set of intervals of the formula consists either of the entire support $[0, \infty)$ or it is the empty set $\emptyset$. So, when one of the subformula under a conjunction operator returns $\emptyset$, the outcome of the formula with the conjunction will always return the same $\emptyset$, since the underlying algorithm intersects the sets of intervals of these subformulas. Such an example shows that the algorithms for evaluation of logical operators can be optimized even more than currently stated and decrease the computation time.

Remove duplicate evaluation of equal STL subformulas When an equal subformula is detected during parsing the evaluation could be shared such that the amount of computations are reduced. For instance, a formula has three times the same subformula, then each subformula is evaluated separately. However, when a during parsing some pointers group or point to some other formula that does exactly the same calculation, the formula reduces the computation speed with two calculations of the subformula. Therefore, the removal of duplicate evaluation of equal STL subformulas decreases the computation time.

Counter-examples Model checkers often generate counter-examples that offer a trace through the state space to track what events caused the system to end up in an unsatisfying state. Counter-examples can be harder to find, since the state representation is different from normal automata. So, there is no certainty if counter-examples can be given in the case of model checking HP-NGs against STL, however, since counter-examples are important parts of model checkers it is recommended to investigate counter-examples.

Add support for other logical operators Other logical operators are often used by users, such as disjunction and implication, which might be more intuitive for a user to apply in certain situation. It is often relatively easy to add these operators, since these can be transformed with rules of logical equivalence. For instance, De Morgan's law states the following: $\neg(P \vee Q) \iff (\neg P) \wedge (\neg Q)$. Therefore, when we apply De Morgan' law together with a double negation on a disjunction the following formula can be internally evaluated for an or operator, $P \vee Q \iff \neg\neg(P \vee Q) \iff \neg((\neg P) \wedge (\neg Q))$. This is one solution, however, the interpretation needs to be checked, since De Morgan's law applies to propositional logic and boolean algebra.

Add nesting inside until formulas One of the main restrictions in this research is the nesting inside formulas with until operators. The children subformulas of the until formula are restricted to atomic properties, since polygon clipping is limited to convex polygons as explained in Section 11.3. Polygon clipping uses polygon algorithms to determine a new polygon from two polygons with operations, such as the intersection between two polygons. Algorithms and libraries exists to calculate these polygons, but mostly up to convex polygons, while the polygons derived from the subformulas can be non-convex, i.e. concave. Even harder is the nesting of until formulas inside until formulas which is not yet supported by the existing algorithms. Therefore, additional research need to be conducted on the support of nesting of formulas inside formulas with an until operator.

References

[1] Rajeev Alur and David L Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, April 1994. ISSN 03043975. doi: 10.1016/0304-3975(94)90010-8. URL http://linkinghub.elsevier.com/retrieve/pii/0304397594900108. 17, 26

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*, volume 950. The MIT press, 2008. ISBN 978-0-262-02649-9. doi: 10.1093/comjnl/bxp025. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11481. 18

[3] Matthias Becker and Thomas Bessey. Comparison of the modeling power of fluid stochastic petri nets (FSPN) and hybrid petri nets (HPN). In *Proceedings of Institute of Electrical and Electronics Engineers International Conference on Systems, Man and Cybernetics*, volume 2, pages 354–358. IEEE, 2002. ISBN 0-7803-7437-1. doi: 10.1109/ICSMC.2002.1173437. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1173437. 25

[4] Paul E Black. data structure. *Dictionary of Algorithms and Data Structures*, 2004. URL http://www.nist.gov/dads/HTML/datastructur.html. 45

[5] Marc Bouissou, Sibylle Humbert, Sabine Muffat, and Nathalie Villatte. KB3 tool: feedback on knowledge bases. In *Proceedings of the 11th European Safety and Reliability Conference*. ESREL, 2002. 29

[6] Giacomo Bucci, Laura Carnevali, Lorenzo Ridi, and Enrico Vicario. Oris: a tool for modeling, verification and evaluation of real-time systems. *Journal of International Journal on Software Tools for Technology Transfer*, 12(5):391–403, May 2010. ISSN 1433-2779. doi: 10.1007/s10009-010-0156-8. URL http://link.springer.com/10.1007/s10009-010-0156-8. 27

[7] Stephen H Conrad, Rene J LeClaire, Gerard P O'Reilly, and Huseyin Uzunalioglu. Critical national infrastructure reliability modeling and analysis. *Journal of Bell Labs Technical Journal*, 11(3):57–71, November 2006. ISSN 10897089. doi: 10.1002/bltj.20178. URL http://doi.wiley.com/10.1002/bltj.20178. 1

[8] René David and Hassane Alla. Continuous and hybrid Petri nets. *Journal of Circuits, Systems and Computers*, 8(1):159–188, 1998. doi: 10.1142/S0218126698000079. 14

[9] René David and Hassane Alla. On Hybrid Petri Nets. *Journal of Discrete Event Dynamic Systems*, 11(1-2):9–40, 2001. ISSN 0924-6703. doi: 10.1023/A:1008330914786. URL http://dx.doi.org/10.1023/A:1008330914786. 26

[10] René David and Hassane Alla. *Discrete, Continuous, and Hybrid Petri Nets*. Springer, Berlin, Heidelberg, 2010. ISBN 978-3-642-10668-2. doi: 10.1007/978-3-642-10669-9. URL http://www.springerlink.com/index/10.1007/978-3-642-10669-9. 5, 26

[11] Sander de Jong. Watertower Noorderbinnensingel in Groningen, the Netherlands, 2007. URL http://commons.wikimedia.org/wiki/File:Watertoren_Noorderbinnensingel_2.jpg. 14

[12] Digia. Qt, 1992. URL http://qt-project.org/. 36

[13] Digia. Qt Creator, 2009. URL http://qt-project.org/. 36

[14] Eclipse Foundation. Eclipse, 2001. URL http://www.eclipse.org/. 36

[15] EDF. KB3, 2002. URL http://research.edf.com/research-and-the-scientific-community/software/kb3-44337.html. 28

[16] Murray Elstner and Daniel Cumming. gtkmm, 2002. URL http://www.gtkmm.org/. 36

[17] FBK-IRST, University of Genova, and University of Trento. NuSMV, 2002. URL http://nusmv.fbk.eu/. 25

[18] Apache Software Foundation. Subversion, 2000. URL http://subversion.apache.org/. 38

[19] Goran Frehse. PHAVer, 2005. URL http://www-verimag.imag.fr/~frehse/phaver_web/. 26

[20] Hamed Ghasemieh, Anne Remke, Boudewijn Haverkort, and Marco Gribaudo. Region-Based Analysis of Hybrid Petri Nets with a Single General One-Shot Transition. In *Proceedings of 10th International Conference on Formal Modeling and Analysis of Timed Systems*, pages 139–154. Springer, 2012. ISBN 978-3-642-33364-4. doi: 10.1007/978-3-642-33365-1_11. 1, 2, 12, 13, 16, 17, 21, 22, 33, 34, 51, 65, 68, 69, 70

[21] Hamed Ghasemieh, Anne Remke, and Boudewijn Haverkort. Survivability evaluation of fluid critical infrastructure using hybrid Petri nets. In *Proceedings of the 19th Institute of Electrical and Electronics Engineers Pacific Rim International Symposium on Dependable Computing*. IEEE, 2013. 1, 2, 18, 19, 21, 23, 24, 33, 34, 49, 69, 70

[22] GNU. GCC, the GNU Compiler Collection, 1987. URL http://gcc.gnu.org/. 36

[23] GNU. Bison, 2002. URL http://www.gnu.org/software/bison/. 47

[24] Marco Gribaudo. *Hybrid formalism for performance evaluation: Theory and applications*. Technical report, phd thesis, Universitadi Torino, 2001. 25

[25] Marco Gribaudo and Anne Remke. Hybrid Petri nets with general one-shot transitions for dependability evaluation. In *Proceedings of the 12th Institute of Electrical and Electronics Engineers International Symposium on High-Assurance Systems Engineering*, pages 84–93. IEEE, 2010. URL http://doc.utwente.nl/75311/. 12

[26] Marco Gribaudo and Anne Remke. *Hybrid Petri nets with general one-shot transitions: model evolution*. Technical report, University of Twente, 2012. URL http://130.89.10.12/~anne/pub/tecrep.pdf. 1, 2, 15, 16, 33, 69, 70

[27] Marco Gribaudo, András Horváth, Andrea Bobbio, Enrico Tronci, Ester Ciancamerla, and Michele Minichino. Fluid Petri Nets and Hybrid Model-checking: A comparative case study. *Journal of Reliability Engineering & System Safety*, 81(3):239–257, 2003. ISSN 0951-8320. doi: 10.1016/S0951-8320(03)00089-9. URL http://www.sciencedirect.com/science/article/pii/S0951832003000899. 25

[28] T.A. Henzinger and Howard Wong-Toi. HYTECH: the next generation. In *Proceedings of the 16th Institute of Electrical and Electronics Engineers Real-Time Systems Symposium*, pages 56–65. IEEE, 1995. ISBN 0-8186-7337-0. doi: 10.1109/REAL.1995.495196. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=495196. 27

[29] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000. ISBN 978-3-642-64052-0. doi: 10.1007/978-3-642-59615-5_13. 26

[30] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to HyTech. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–71. Springer, 1995. 27

[31] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *Journal of International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997. ISSN 1433-2779. doi: 10.1007/s100090050008. URL http://dx.doi.org/10.1007/s100090050008. 27

[32] Tom Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech, 1997. URL http://embedded.eecs.berkeley.edu/research/hytech/. 25, 27

[33] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM, 2006. URL http://www.prismmodelchecker.org/. 25

[34] Jorge Júlvez, Cristian Mahulea, and Carlos-Renato Vázquez. SimHPN, 2012. URL http://webdiis.unizar.es/GISED/?q=tool/simhpn. 1, 28

[35] Jorge Júlvez, Cristian Mahulea, and Carlos-Renato Vázquez. SimHPN: A MATLAB toolbox for simulation, analysis and design with hybrid Petri nets. *Journal of Nonlinear Analysis: Hybrid Systems*, 6(2):806–817, 2012. doi: 10.1016/j.nahs.2011.10.001. URL http://www.sciencedirect.com/science/article/pii/S1751570X11000586. 28

[36] Loes Knoben, Maarten Otten, and Ruben Fransen. De waterzuivering als Petrinet. Technical report, Universiteit of Twente, 2013. 1, 61, 62, 68

[37] Leslie Lamport. LaTeX, 1984. URL http://www.latex-project.org/. 38

[38] Oracle. Java, 1995. URL http://java.com/. 28, 36

[39] Oracle. Swing, 1998. URL http://docs.oracle.com/javase/tutorial/uiswing/. 36

[40] Vern Paxson. flex: The Fast Lexical Analyzer, 1987. URL http://flex.sourceforge.net/. 47

[41] Louchka Popova. On time Petri nets. *Journal of Information Processing and Cybernetics*, 27(4):227–244, 1991. 27

[42] Björn F Postema and Hamed Ghasemieh. Fluid Survival Tool Project Page, 2013. URL https://code.google.com/p/fluid-survival-tool/. 38, 54, 59

[43] Dennis Ritchie. C, 1973. 33

[44] RTV Oost. Overijssel Vandaag 21-7-2013, 2013. URL http://www.rtvoost.nl/tv/uitzendinggemist.aspx?uid=290892. 1, 61

[45] STLab. Oris, 2010. URL http://www.stlab.dsi.unifi.it/oris/. 1, 27

[46] Bjarne Stroustrup. C++, 1983. 33

[47] The MathWorks Inc. MATLAB. URL http://www.mathworks.nl/. 28

[48] Total. GRIF, 2010. URL http://grif-workshop.com/. 1, 28

[49] TV Enschede FM. TV Enschede Nieuws 21-06-2013, 2013. URL http://www.youtube.com/watch?v=DRIB6JTNvhA. 1, 61

[50] Uppsala University and Aalborg University. Uppaal, 1995. URL http://www.uppaal.org/. 26

[51] UT Nieuws. Wanneer kun je kanoën op de Auke Vleerstraat?, July 2013. URL http://www.utnieuws.nl/studenten/wanneer-kun-je-kanoen-op-de-auke-vleerstraat. 1, 61

[52] Dimitri van Heesch. Doxygen, 2001. URL http://www.stack.nl/~dimitri/doxygen/. 38

[53] Verimag. Kronos, 1996. URL http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/. 26

[54] Verimag. SpaceEx, 2011. URL http://spaceex.imag.fr/. 26

[55] W3C and WHATWG. HyperText Markup Language, 1997. URL http://www.w3.org/TR/1999/REC-html401-19991224/. 38