Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

USING SIMULATION TO IMPROVE THE PRECISION OF HYBRID SYSTEMS REACHABILITY ANALYSIS

Lyudmila Vatskicheva

Examiners:
Prof. Dr. Erika Ábrahám
apl. Prof. Dr. rer. nat. Thomas Noll

Additional Advisor:
Stefan Schupp

Aachen, March 25, 2019

Abstract

A hybrid system combines discrete and continuous behavior. In order to analyze a hybrid system and verify its safety against a predefined set of unsafe states, we utilize classical reachability analysis via flowpipe construction. Due to its over-approximative nature, this method can only prove safety of a hybrid system.

In this thesis we extend our existing reachability analysis tool by a simulation-based approach in order to be able to provide counterexample runs in case the system is unsafe. In our approach we exploit previously gained information from flowpipe construction in a CEGAR-like fashion to guide the simulation of finitely many single executions of the system with the aim to find a counterexample. We evaluate our approach and the impact of the utilized sampling heuristics for simulation on some benchmarks.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Lyudmila Vatskicheva
Aachen, den 25. März 2019

Acknowledgements

I would like to express my gratitude towards Prof. Dr. Erika Ábrahám and Prof. Dr. Thomas Noll for their continuous support and constructive advices throughout my thesis. I am particularly grateful for the assistance given by Stefan Schupp. His patience and willingness to give his time so generously and being always available for fruitful discussions have been very much appreciated.

In addition, I wish to thank my parents Malina and Aleksandar, and my sister Petra for their continuous support and encouragement throughout my study despite the long distance between us. Many thanks to my friends and boyfriend Boris who managed to bring up a smile on my face even throughout the toughest times.

This thesis would not have been possible without your support. Thank you!

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Contents

Chapter 1

Introduction

In the last decade the usage of technology has drastically increased. Humanity has gained trust in computer-made decisions which provide convenience and better efficiency. The higher demand has led to developing applications and systems with increasing complexity. Systems became more automated and independent from the need of human supervision. Examples for automated systems, which are used in our everyday life are automatic transmissions in cars, the thermostats that are used in houses or the automated railway crossings. However, the increasing complexity of automated systems typically involves a higher risk of system malfunction or unexpected behavior. The software for controlling a railway crossing has to be highly reliable and error-free. An essential task of software engineers is to ensure the accurate behavior of such safety-critical applications and to verify the correctness of the software.

All of the examples above are hybrid systems, i.e. systems with combined discrete and continuous behavior. A discrete system has a finite set of states and its evolution can be described as a sequence of discrete state changes. For instance, consider the example of a digital controller for a thermostat whose goal it is to maintain a certain room temperature. The thermostat has two possible discrete states: `{on}` and `{off}`. On the other hand, the temperature in a room changes continuously over time. This is a continuous evolution of a physical component. In the thermostat example the physical component is `temperature`. Trajectories of the evolution of such systems can be computed and their development over time can be observed. Reachability analysis of a hybrid system is in general undecidable which presents a challenging task for its safety verification.

A formal method to prove a hybrid system of its safety is flowpipe simulation reachability analysis [ÁC15, LGG09, HKPV98, ACH+95]. This method is semi-decidable, since the system evolution is computed up to predefined time bound and number of discrete jumps. Its goal is to determine, whether a set of bad states is reachable during the execution of a certain system from a set of initial states. Bad states are defined as a set of states which violate the property that is to be proved. Starting from the initial set, the evolution of a hybrid system is computed until given time has passed. At each step the computed set is usually over-approximated and is tested for an intersection with the bad states. If the intersection is empty, the system is safe. If the intersection is not empty, bad states might be reachable which leads to the conclusion that the system safety cannot be guaranteed. However, there exists a trade-off between accuracy and required computational resources for the computation. The

given time, until which the system evolution is been computed, can be separated into smaller steps in order to approximate its exact development. Also, the choice of the time step size has an impact on the precision of the obtained solution. The bigger the time step, the faster the computation, but it delivers a more over-approximated solution.

During the computation there are several choices for the geometrical representation of the generated states of a hybrid system. Usually, the more complex the geometry is, the more storage it needs, the more difficult is it to apply operations on the sets and the more time it takes for the computation, but a better approximation can be achieved. For instance, boxes are fast and efficiently computed. However, they deliver very inaccurate and over-approximated solution, which can lead in some cases to indecisive results.

In case of an unsafe system reducing the over-approximative error by utilizing finer settings does not contribute to proving safety. Therefore, instead of proving safety, our method numerically calculates candidates for counterexamples of the system safety and thus tries to prove unsafety. In this thesis, we introduce numerical simulation of a hybrid system evolution using sampling, where the state set representation is a point. The idea is to take one or a few points from the initial set and to precisely calculate their behavior over time. This method provides an insight into the exact evolution of the system and finds a counterexample candidate in case of an unsafe system.

Outline. One of the main goals of this work is providing results, which are understandable for researchers outside of the theoretical computer science field. Therefore, the next Chapter 2 features all necessary background information for understanding this topic. Chapter 3 focuses on the solution design and displays all implementation details of our method. Afterwards, Chapter 4 investigates the applicability of our method in the context of hybrid systems safety verification and evaluates its performance in terms of required computational time and memory resources. Finally, the results are summarized and a potential future work and ideas are discussed, which can be of future interest.

Chapter 2

Preliminaries

In this chapter we present preliminary knowledge required for the rest of this work. First, we give a formal definition of a hybrid automaton. In the section after that we discuss various methods for representing a state set and consider their characteristics about operation efficiency and representation accuracy. Furthermore, the term of refinement setting along with its attributes is introduced, as well as its combination into strategies, which is used as a a powerful mechanism to determine the accuracy and efficiency of the approximation of a system under computation. Further, we introduce the essence of the already implemented reachability worker, which computes a classic forward reachability of a model in the context of the tool HyDRA.

In order to verify a system for its safety, computing a run from the initial state set to the set of bad states should not be possible. On the other hand, if such run of the system can be found, the system safety cannot be guaranteed. This verification method is referred to as safety verification for hybrid systems.

2.1 Hybrid automata

In order to verify a hybrid system using formal methods, an abstraction is required which models the system in a way, that suits its behavior. There are several representation methods, which consider different aspects of the system behavior. Hybrid automata (HA) as a model for hybrid systems have been used extensively in the past decades. A formal definition of the syntax of a HA, as introduced in [ACH+95] and in [SÁ18b], is:

Definition 2.1.1. *A hybrid automaton H is a tuple (Loc, Var, Lab, Edge, Act, Inv, Init) where:*

- *Loc is a finite set of locations or control modes;*
- *Var = x_1,..., x_d is a finite ordered set of real-valued variables; sometimes we use the vector notation $x = (x_1, ..., x_d)$. The number d is called the dimension of H. By Var we denote the set $\{\dot{x}_1, ..., \dot{x}_d\}$ of dotted variables (which represent first derivatives during continuous evolution), and by Var' the set $x'_1, ..., x'_d$ of primed variables (which represent values directly after a discrete change). Furthermore, given a variable set X, let $Pred_X$ denote a set of predicates with free variables from X;*

- *Lab is a finite set of synchronization labels that contains the stutter label* $\tau \in$ *Lab;*
- *Edge is a finite set of edges called transitions. Each transition* $e = (l, a\mu, l')$ *consists of a source location* $l \in$ *Loc, a target location* $l' \in$ *Loc, a synchronization label* $a \in$ *Lab, and a transition relation* $\mu \subseteq V^2$*;*
- *Act is a function, assigning a set of activities* $f : \mathbb{R}_{\geq 0} \to V$ *to each location which are time-invariant, meaning that* $f \in Act(l)$ *implies* $(f+t) \in Act(l)$ *where* $(f+t)(t_0) = f(t+t_0)$ *for all* $t_0 \in \mathbb{R}_{\geq 0}$*, and*
- *Inv is a labeling function that assigns to each location an invariant;*
- *Init is a finite set of initial predicate assignments to each location.*

The different components of the HA are described based on the following thermostat example, as in [ACH+95]. A thermostat has two possible locations: `on` and `off`. When the thermostat is `on`, the room temperature `x` increases until it reaches a degree between $22°C$ and $23°C$. Afterwards the thermostat switches to `off` until the temperature `x` drops down between $17°C$ and $18°C$. Note that this model is non-deterministic, since the control can change its location anytime, when the guard is satisfied, for example anytime between the interval $[22°C; 23°C]$. The dynamics of the temperature are defined by the following differential equations:

$$\dot{x} = \begin{cases} -0.1x + 5 & \text{if the heater is } \texttt{on}, \\ -0.1x & \text{if the heater is } \texttt{off}. \end{cases}$$

Initially the room temperature is $20°C$ and the heater is `on`. Turning the heater `on` and `off` is a discrete change of the system (Figure2.3). The temperature evolution over time has a continuous behavior (Figure2.2). We can model the thermostat system by the following hybrid automaton:

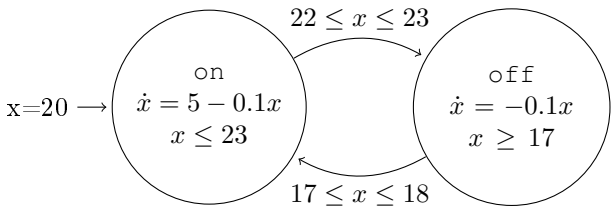


Figure 2.1: Graphical representation of a hybrid automaton, modelling the behavior of a thermostat.

Formal model:

- *Loc* is a finite set of locations, such as {`on`, `off`};
- Var is a finite set of real-valued variables. Such is the temperature {`x`};
- *Lab* is a finite set of synchronization labels, such as $\{17 \leq x \leq 18, 22 \leq x \leq 23, \tau\}$ with τ implicit self loops, also called stutter transitions, on each location, which are usually not displayed;

- *Edge* is a finite set of explicit edges between locations and implicit edges (self loops) τ:

$$\begin{aligned} Edge = \{&(\texttt{on}, 22 \leq x \leq 23, \texttt{off}),\\ &(\texttt{off}, 17 \leq x \leq 18, \texttt{on}),\\ &(\texttt{on}, \tau, \texttt{on}),\\ &(\texttt{off}, \tau, \texttt{off})\} \end{aligned}$$

- *Act* is a function, which models the flow at each location:

$$\begin{aligned} Act(\texttt{on}) &= \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall x \in \mathbb{R}_{\geq 0}. f(x) = 5 - 0.1x + c\}\\ Act(\texttt{off}) &= \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall x \in \mathbb{R}_{\geq 0}. f(x) = -0.1x + c\} \end{aligned}$$

- *Inv* is a function assigning an invariant to each location. Invariants restrict the variable assignment at each location into a defined bound and enforce the control to jump in another location:

$$\begin{aligned} Inv(\texttt{on}) &= \{v \in V \mid v(x) \leq 23\}\\ Inv(\texttt{off}) &= \{v \in V \mid v(x) \geq 17\} \end{aligned}$$

- *Init* is a set of initial states:

$$Init = \{(\texttt{on}, v) \in \Sigma \mid v(x) = 20\}$$

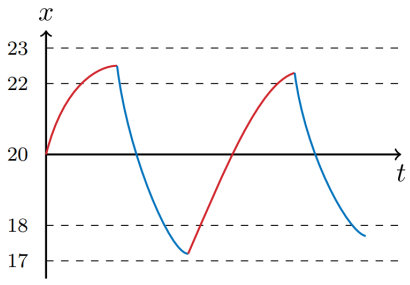


Figure 2.2: Continuous behavior of the thermostat. The room temperature increases or decreases continuously over time. Red: increasing temperature; blue: decreasing temperature. [ÁC15]

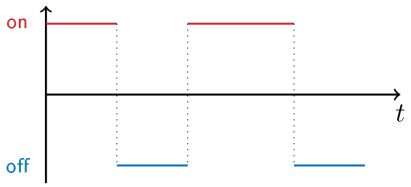


Figure 2.3: Discrete behavior of the thermostat. Regarding the room temperature, the heater is turned on (red line) or off (blue line). [ÁC15]

Figure 2.2, however displays only one possible behavior of the system. Since the control can switch locations anytime if the temperature x is in the interval $[17°C; 18°C]$ and $[22°C; 23°C]$ respectively, there are infinitely many possible runs, that have not been considered yet. In order to prove a system, it is required to compute all possible states of the system while checking whether there is a non-empty intersection with a set of predefined bad states. The computation of all reachable states is referred to as reachability analysis.

2.2 Reachability analysis

The following paragraph explains how reachability analysis is generally done and the hybrid automaton from Figure 2.1, which describes the functionality of a thermostat, will be taken as an example for a better understanding of this approach. In addition to that, with the help of Figure 2.4 an introduction to flowpipe analysis takes place, as it is a fundamental part of this thesis.

First the computation starts from the initial state ($x = 20$), which is the initial valuation of the variable x in the first location (`on`). On Figure 2.4 the initial state set is depicted by the blue rectangle. Then, taking into consideration the dynamics ($\dot{x} = 5 - 0.1x$) in this location, the continuous evolution of the system is computed (also referred to as flow). This calculation step is done as long as the location invariant ($x \leq 23$) holds, or the guard ($22 \leq x \leq 23$) allows the control to take a discrete jump to another location (`off`). The guard is represented on Figure 2.4 by the red area. The computation of the flow for one time step results in a segment of the flowpipe. The very first segment of a flowpipe represents the geometrical representation of the initial state set and each successive segment is a linear transformation of the previous one.

When taking a discrete jump with a reset function which are marked on Figure 2.4 with a green arrow, the current variable valuation initializes the variable in the next location, where the jump points to. Reset functions on discrete transitions map the guard satisfying states to a different location. In the case of the thermostat from Figure 2.1 a variable valuation can be $x = 22$. The flow is then further computed in location `off` with the new dynamics $\dot{x} = -0.1x$. The result of this calculation is a flowpipe which represents the flow at location `off`.

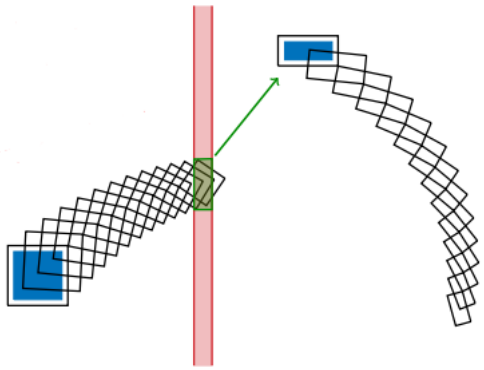


Figure 2.4: Depiction of the omputation of a flowpipe which intersects with a guard and takes a discrete jump with a reset to continue another flowpipe construction. [ÁC15]

The flowpipe computation alternates between discrete steps and continuous steps until a predefined time bound or set of bad states is reached. The general algorithm for reachability analysis is illustrated in Algorithm 1.

Input of the algorithm is a set of initial states $Init$ and the output is a set of reachable states R. The set R^{new} contains all states, that have to be analyzed. At the beginning R^{new} contains the whole set of initial states, whereas the set of reachable

Algorithm 1 Forward reachability analysis algorithm [ÁC15]

```
1: function REACH(Init)
2:     R^new := Init
3:     R := ∅
4:     while R^new ≠ ∅ do
5:         R := R ∪ R^new
6:         R^new := Reach(R^new) \ R
7:     return R
```

states R is empty. Then a loop is entered as long as there are still any states, that have to be analyzed. Iteratively the set of reachable states R is extended by computing the union of itself and the calculated successors. The set of states to be analyzed R^{new} is updated by taking away the previously analyzed set R. Note that the algorithm does not necessarily terminate.

Although the general reachability analysis of hybrid systems is undecidable [HKPV98], the problem can be transformed to become (semi-)decidable by limiting the time horizon and therefore the flow duration and the number of discrete jumps are bounded. The smaller the time step, the more accurate the flow is, but also more computational effort is required.

Flowpipe reachability analysis which is done on the previous thermostat example from Figure 2.1 is shown on Figure 2.5 with a time horizon of 10s, a time step size 0.01s and maximum of 2 jumps.

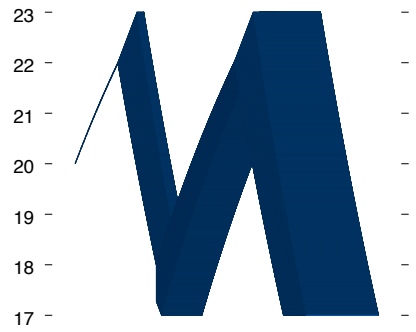


Figure 2.5: Forward reachability analysis of a thermostat [ÁCK+18]. It depicts the temperature evolution over time.

2.3 State set representations

Representing a state set of a hybrid automaton is needed to illustrate graphically its evolution over time. Usually we cannot compute the actual state set, therefore the representation can be an over- or under-approximation of the state set.

Under-approximation. An under-approximation does not consider all possible states of the system, since at each computational step possible runs are excluded. This approach is can be used when proving a system to be unsafe. Additionally, using an under-approximation approach reduces the required computational power.

Over-approximation. On the other hand, an over-approximation considers more states than the system actually can reach and it is ensured, that all reachable sets are computed. A non-empty intersection with a set of bad states leads to the conclusion that the simulated system might be unsafe. However, there are no insights whether the system is in fact unsafe or the possible run into the unsafe states is caused by the over-approximation of the state set. Therefore, there are various possibilities for representing the state set with diverse complexity levels and exactness of the state over-approximation.

$$\text{under-approximation} \subseteq \text{actual set} \subseteq \text{over-approximation}$$

Requirements on approximation sets. In order to compute reachability analysis of a hybrid system, there are several requirements on the state set representation: to efficiently calculate operations as union with previous state sets, intersection with guard, exclusion, membership relation, test for emptiness, etc. [SÁ18a].

An over-approximation of the state set of a hybrid automaton has to contain all required sets of the current computational step. By using the convex hulls of sets a computational efficiency is achieved. There are various geometric representations of a convex hull, such as boxes, ellipsoids, polytopes [LG09], oriented rectangular hulls [SK03], zonotopes [Gir05], as well as symbolic representations such as support functions [LGG09]. Each of them has its advantages and disadvantages over the other representation methods.

All state set representations are implemented in a C++ library, called HyPro [SÁMK17]. In the following, a few of them are compared and trade-offs between accuracy and required computational power are observed.

- **Boxes** are one of the simplest class of sets. They are efficiently computed, since only a pair of upper and lower corner bounds for each variable are required, as depicted on Figure 2.6. The set operations mentioned above are also efficiently calculated. However, this representation method is in most cases relatively inaccurate.

- **Polytopes** are a bounded intersection of a finite set of half-spaces, as depicted on Figure 2.7. There are two methods of representing polytopes: as a convex hull of finite set of vertices, referred to as $\mathcal{V}-$polytope, or as a convex hull of finite set of closed half-spaces, referred to as $\mathcal{H}-$polytope.

  The computation of a geometrical operation has a different complexity regarding whether they are calculated based on a $\mathcal{V}-$ or $\mathcal{H}-$polytopes, as listed in Table 2.1.

  Operations, which are efficiently calculated on a set of vertices, such as Minkovski sum or union, are computed in a polynomial complexity for $\mathcal{V}-$polytopes, since

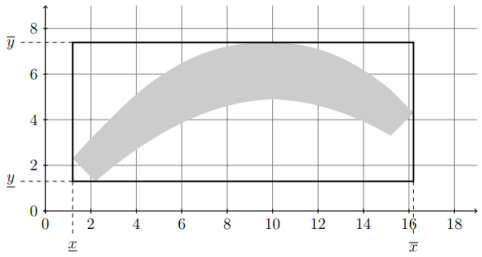


Figure 2.6: Over-approximation of the state set (grey) by a box [LG09].

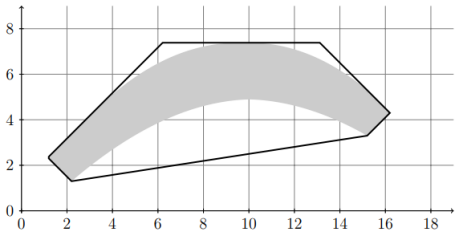


Figure 2.7: Over-approximation of the state set (grey) by a polytope [LG09].

	$x \in P$	$A \times P$	$P_1 \oplus P_2$	$P_1 \cap P_2$	$P_1 \cup P_2$
$\mathcal{V}-$polytope	easy	easy	-	-	-
$\mathcal{H}-$polytope	easy	hard	-	-	-
$\mathcal{V}-$polytope and $\mathcal{V}-$polytope	-	-	easy	hard	easy
$\mathcal{H}-$polytope and $\mathcal{H}-$polytope	-	-	hard	easy	hard
$\mathcal{V}-$polytope and $\mathcal{H}-$polytope	-	-	hard	hard	hard

Table 2.1: Complexity overview of geometrical operations on polytopes (membership relation, linear transformation, Minkowski sum, intersection and union)[ÁC15].

they are characterized by a set of vertices. For instance, convex hull of the union of two $\mathcal{V}-$polytopes can be computed by calculating the union of their vertices. On the other hand, there is no polynomial algorithm known [ÁC15, ACH+95], which calculates the intersection of $\mathcal{V}-$polytopes, whereas the intersection of two $\mathcal{H}-$polytopes can be computed in polynomial complexity by taking the union of their set of half-spaces. Moreover, the conversion of a $\mathcal{H}-$polytope into a $\mathcal{V}-$polytope or the other way around, cannot be computed in polynomial complexity.

The choice of representing the state sets by a polytope has an advantage on the exactness of the over-approximation over a box representation, but requires more computational power for calculating some geometrical operations.

- **Sampling** is the main objective in this thesis. The simulation of a single run of the system is done very efficiently, since the state set is represented only by a point, hence sample. Its accurate evolution in time is computed in order to get insights of the exact system behavior.

  One (or many) points from the initial state set are chosen based on heuristics, which are introduced in Chapter 3. Depending on the valuation derivation of the current location, the sample evolution in time is calculated and results in another point (sample). It is the initial variable valuation for computing the next computational step.

The hybrid systems reachability analysis includes computation of a flowpipe, which represents all reachable state sets of the system. The flowpipe contains segments, whose state sets are represented by various representation methods (see Section 2.3). In order to efficiently represent all reachable states, an over-approximation of those

states is suggested. In the case of reaching the predefined unsafe states, a refinement of the calculation on the relevant path is applied. There are several refinement settings, which the user can define and apply.

2.4 Refinement Settings

The segments computation has several parameters, which allow adjustments on the calculation accuracy and efficiency, as well as on the required computational time. Such parameters are:

- **Time horizon:** also called time bound, describes the execution time covered by the flowpipe construction. It is considered as a constraint and bounds the flow computation to a certain time limit. We denote the time horizon by T and measure it in seconds. This parameter is usually defined in the model that we use.
- **Time step:** is denoted by $\delta = \frac{T}{n}$, where n is the number of time steps and T is the defined time horizon. Therefore δ describes the size of one time step. The more time steps, the smaller their size is and the more accurate the flowpipe construction is. However, the required computational effort increases with the calculation accuracy. Discretizing T in large steps offers less precision, but a faster calculation.

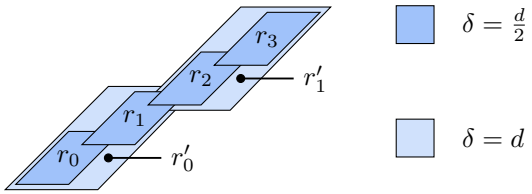


Figure 2.8: Discretisation of the time step δ into 4 smaller time steps (darker blue) or in 2 bigger time steps (lighter blue). [SÁ].

- **Representation:** as discussed above, is a geometrical object such as box, polytope [LG09], zonotope [Gir05] and ellipsoid, or a symbolic representation such as support function [LGG09] or Taylor model [ÁCS12]. Their characteristics are discussed in Section 2.3. An example for representing a state set with different representation methods is shown on Figure 2.9.

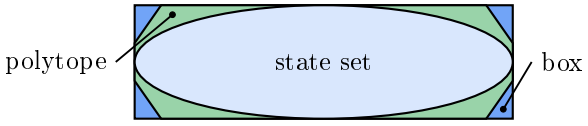


Figure 2.9: An ellipsoid state set over-approximatively represented by a polytope and by a box. [SÁ].

- **Aggregation and clustering:** during reachability analysis it is possible, that multiple sets satisfy a guard. Each of these sets is an initial state set for further computation of a segment of the flowpipe. Such branching requires more

resources for storing and processing of the successive flowpipes. A mechanism to overcome such branching is to aggregate all sets in the guard intersection by building their convex hull. In order to reduce the over-approximation factor of the aggregation, clustering is suggested. This mechanism combines multiple sets together by building their convex hull. The result is a reduction on the total number of sets which are utilized as initial state sets for further flowpipe construction.

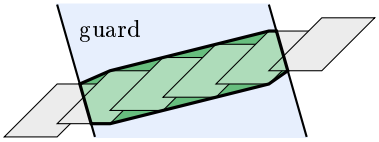


Figure 2.10: A guard intersection that contains multiple sets, which are aggregated into one set. [SÁ].

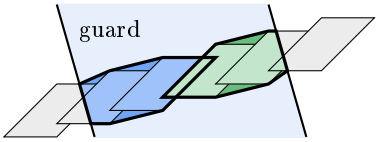


Figure 2.11: A guard intersection that contains multiple sets, which are clustered into two different sets. [SÁ].

2.4.1 Strategies

By sequentially ordering the refinement settings together into strategies, one can influence the accuracy of the delivered flowpipe and the required resources for its computation. A strategy contains various refinement settings with time steps, representation types, aggregation and clustering settings used for the flowpipe construction. It is useful to define the first `RefinementSetting` to be more general than the upcoming ones in order to cheaply get insights into the system behavior. Each following `RefinementSetting` gets more complex and expensive to compute and requires more computation resources.

An example for a strategy is depicted in Listing 2.4.1.

```
Strategy {
  RefinementSetting1 {
    timeStep: 0.1,
    representation: box,
    aggregation: yes,
    clustering: no
  },
  RefinementSetting2 {
    timeStep: 0.1,
    representation: polytope,
    aggregation: yes,
    clustering: no
  },
```

```
  RefinementSetting3 {
    timeStep: 0.01,
    representation: box,
    aggregation: no,
    clustering: no
  },
  RefinementSetting4 {
    timeStep: 0.1,
    representation: sample,
    aggregation: yes,
    clustering: no
  }
}
```

Listing 2.1: Strategy containing four different refinement settings with increasing accuracy.

The first reachability analysis of a hybrid system is computed using $\texttt{RefinementSetting}_1$ of the current `Strategy`. In case of intersecting the bad states a simulation takes place using the finer $\texttt{RefinementSetting}_2$. Otherwise, in case of safety no more simulations of the system are needed and the reachability analysis of the hybrid system has proven it to be safe. The refinement

settings which use sample as representation, and all other refinement settings are used similarly, but have different verification goals:

The goal of the over-approximative refinement settings is to show safety of a model by proving that there is no intersection of the flowpipe and the set of bad states even with using an over-approximation. If such intersection after various refinements is still calculated, this leads to the conclusion that the reachability analysis result is indecisive, since at each computational step the system behavior is over-approximated. There is no proof that the system is really unsafe or the intersection is caused by the over-approximation. The combination of the refinement settings into strategies allows to efficiently generate a flowpipe by over-approximating the system evolution.

The goal of the refinement settings using sampling is to show unsafety by finding a counterexample run from the initial states to the bad states, as depicted on Figure 2.12. In case of indecisiveness from the reachability analysis it is suggested to apply simulation to retrieve more insights in the system behavior. By computing a counterexample run from the initial set into the bad states, we prove that the hybrid system is unsafe. Additional information can be retrieved in order to examine which behavior causes such a run. In the case a counterexample run cannot be found, the sampling method is also indecisive, as depicted on Figure 2.13, since not every single system run is computed.

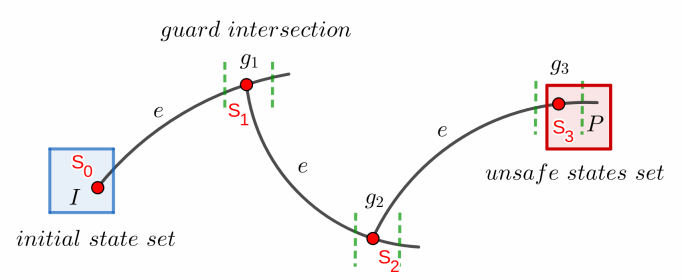


Figure 2.12: A run of the system from the initial state set into the set of bad states implies unsafety of the simulated system.

These different goals imply that as soon as simulation is triggered, the verification goal changes and a mode switch occurs. While before the aim was proving safety, now is used simulation to prove unsafety. For instance, we define a strategy containing 6 refinement settings, where the fourth and the sixth one use sample as a representation:

$$\underbrace{R_1 \to R_2 \to R_3}_{\text{prove safety}} \to \Big| \; \underbrace{S_1 \to R_4 \to S_2}_{\text{prove unsafety}} \qquad \text{mode switch} \tag{2.1}$$

First the computed reachability analysis aims at proving safety of the given model. If after all three refinements an intersection with the unsafe states is still calculated, the next refinement is triggered. From the point on of triggering simulation sampling, the goal changes to looking for a counterexample run and trying to prove unsafety. The vertical line indicates the mode switch.

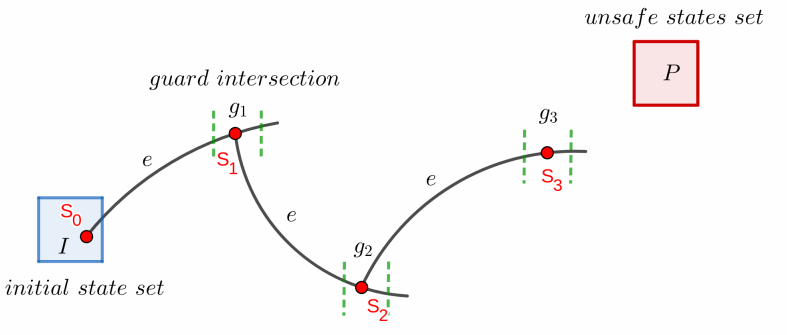


Figure 2.13: A run of the system from the initial state set which passes by the set of bad states implies indecisiveness about the simulated system. Only one single possible run is computed and not all of them.

The "prove safety" mode suggests, that finding a non empty intersection with the unsafe states is considered to be a failure, whereas the "prove unsafety" mode considers it to be a success.
As part of this thesis the described additional second mode has been implemented. Chapter 3 explains in detail how simulation of samples is realized. Beforehand we introduce sampling heuristics, which are used to pick samples from the initial state of the hybrid system and from the interval, in which a guard is being enabled for taking a discrete jump.
The next section presents HyDRA, which is the reachability analysis tool and simulation environment that is being used to compute a flowpipe from the given state sets. Explained is the process of building such a flowpipe, as well as the dynamic creation of the structure, which stores the already computed segments with their successors.

2.5 HyDRA

HyDRA [NÁGS16] is a tool which is in a prototypical development state run by the Hybrid Systems Chair at the RWTH University. The acronym stands for **Hy**brid **D**ynamic **R**eachability **A**nalysis. The tool realizes reachability analysis for linear hybrid systems and allows for applying refinement settings on already computed flowpipes. It is also responsible for the distribution of tasks among workers and defines how the workers solve these tasks. A parallel analysis based on multi-threading allows to process different tasks at the same time [SÁ18b].

Task collects all information that is required for computing a flow and jump successors of a state [SÁ18b]. There are two types of tasks depending on safety or unsafety proving mode, as depicted on Figure 2.14: a forward reachability task to show safety, and a counterexample task which aims at showing unsafety.

Worker is responsible for the execution of tasks. There are various types of workers, which are implemented to behave differently according to their task. In the current state of HyDRA there are two types of workers.

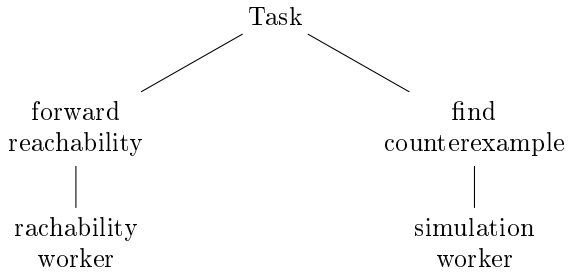


Figure 2.14: Task and worker system in HyDRA. For each forward reachability task there is a reachability worker assigned for computation. For each counterexample task there is a simulation worker which computes it.

Reachability worker implements the algorithm for forward reachability analysis by utilizing state-of-the-art method for computing flowpipes and jump successors [SÁ18b]. The worker gets a task assigned and according to the current strategy and the current mode (safety or unsafety proving) it computes each segment of a flowpipe, starting from an initial set.

Simulation worker utilizes forward analysis as well, but instead of computing each segment of a flowpipe, it samples its initial set and computes the flow of this sample accordingly until a given time. It aims at finding a counterexample run of the hybrid system and to prove it unsafe. The functionality of this worker is the main contribution of this thesis. The sampling method and its implementation are presented in detail in the next Chapter 3.

Reachability tree represents the data structure in which the initial state sets of the flowpipes are stored as nodes during the computation of reachability analysis. The root node includes all initial states and the children of a node include all discrete successors of the flowpipe which starts from this node. Multiple children can also occur due to non-determinism of the analyzed hybrid system. A node in the reachability tree can also store several state sets, each computed at a different refinement level, but referring to the same initial state of a flowpipe. The tree is computed until a set of bad states is hit or until the predefined time horizon is reached.
The computation of such search tree is essential for applying further refinement settings on the simulated system, since it contains crucial information about timings, safe and unsafe paths.

Path Refinements A path π is an ordered sequence of alternating time and discrete steps:

$$\pi = \underbrace{[t_{01}, t_{02}]}_{\text{time step 0}}, \underbrace{loc \to loc[t_{03}, t_{04}]}_{\text{discrete step 1}}, \underbrace{[t_{11}, t_{12}]}_{\text{time step 1}}, \underbrace{loc \to loc[t_{13}, t_{14}]}_{\text{discrete step 2}}, \underbrace{[t_{21}, t_{22}]}_{\text{time step 2}}, ...$$

A path which starts at the initial states is called an "initial path". The simulation worker is applied on the initial paths, which potentially end at the set of bad states. A time step interval describes the global time interval in which the control has been

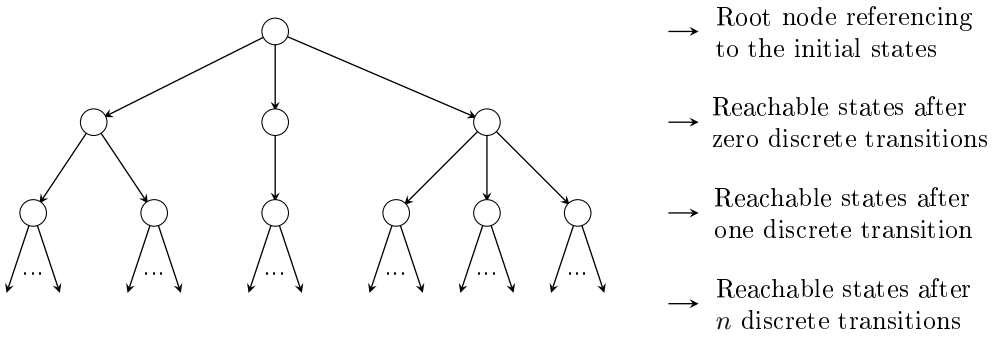


Figure 2.15: Segments computation resulting in a tree structure [NÁGS16].

in a certain location before taking a discrete jump. Seen from the point of flowpipe construction view this interval describes the time interval covered by the flowpipe. For instance in the path π, the first flowpipe is constructed in $t_{02} - t_{01}$ time. A discrete step contains source and target location, as well as a time interval describing at which global time interval the guard was satisfied in order to take a jump.
The path which contains a run into the unsafe states (see Figure 2.16)is called a 'critical path'. It is analyzed again, where a more precise state set representation is chosen in order to reduce the over-approximation error.

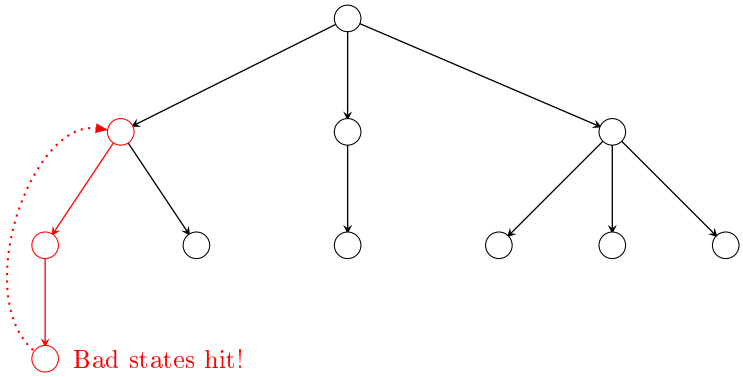


Figure 2.16: Path in the tree reaches set of unsafe states. Refinement on the red path is to be applied. [NÁGS16].

The reachability analysis of hybrid systems aims at verifying a system to be safe by proving that there is no intersection with the set of bad states. However, it is possible that after multiple analyzes of the critical path with various state set representations such intersection still exists. This leads to the conclusion that safety cannot be proven and the reachability analysis algorithm is undecidable.
In that case it is useful to find a counterexample for the system safety. By using simulation sampling we try to find a run from the initial state set into the set of bad states. When such a counterexample run is computed, a candidate for proving the system to be unsafe is found.

Chapter 3

Simulation in hybrid systems reachability analysis

The objective of this chapter is to introduce the algorithms, logic and heuristics behind the implementation of the simulation worker. In the first section we introduce various heuristic approaches for state set and interval sampling. Section 3.2 presents two modes of using simulation in hybrid systems safety verification: as a classical simulation in which one or more concrete system executions are computed, or as a counterexample validation method which is used as a `RefinementSetting` as a part of a `Strategy`.

Trace simulation is the first approach and it simulates the hybrid system by computing a single trajectory by utilizing the forward reachability analysis Algorithm 1. Starting from a single point (or a set of points), a concrete system execution is computed by means of time- and discrete steps, which are calculated until a time or a jump bound is reached, or an intersection with the bad states is reached. This approach is implemented very similarly to the classical reachability analysis. Nevertheless, it computes only one possible run of the system and a statement about the system safety cannot be derived.

Counterexample validation is the second approach. It is triggered when an intersection with the set of unsafe states is already computed by the classical reachability analysis and the system safety cannot be guaranteed. This approach relies upon information which is computed previously by over-approximating the system behavior. The information consists of a critical path π which is a sequence of time intervals and transitions and describes a trajectory, which may lead to the set of bad states. The path π is analyzed again by utilizing the simulation worker which aims at finding a counterexample run into the set of bad states and to prove the system unsafety.
Throughout the simulation π provided by the classic reachability analysis is followed. The starting point of the algorithm behind the simulation worker are the initial states. Originally, they are represented by a geometrical state set, as described in Chapter 2.3. The simulation, however, uses a point as a state set representation. This means, that

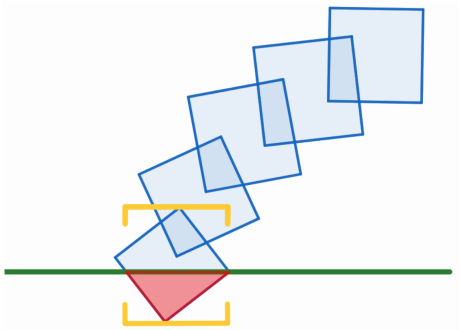


Figure 3.1: A system evolution which is over-approximated by blue boxes. The guard inequation (in green) is partially satisfied by the last box. Its part which does not satisfy the guard is colored in red. The time interval derived by the intersection of the over-approximated state sets with the guard is also over-approximated and is depicted with yellow borders.

a point (also called sample) from the initial states should be chosen as a representative for the state set.
When choosing a sample, its time evolution is computed, as in the classical reachability analysis. The difference is, that computing each time step is not required anymore, since the critical path provides a time interval, in which the guard for a discrete transition is enabled and a jump can be taken. Within this time interval, the sample should also take the jump and follow the trajectory of the over-approximated evolution of the system. Such an interval is depicted on Figure 3.1.
By knowing the time interval when a guard is enabled for a transition, computing each time step with the chosen points is not required anymore. We can let the point evolve until a certain time t_0 from the interval $[t_a, t_b]$. For choosing a time sample there are various heuristics, which are described in the next section.

3.1 Heuristics

In order to simulate a system sampling in both spatial and temporal dimensions has to be calculated. There are two kinds of sampling methods required: one for n-dimensional set (spatial) sampling and another for one dimensional interval (time) sampling. This chapter proposes various heuristic approaches for both sampling methods.

3.1.1 Heuristics for state set sampling

`VERTICES`

The states of the system are represented by different geometric representations, examples for which can be found in Chapter 2. All geometric representations that we defined in HyPro are convex. Therefore, in order to illustrate the outlines of a system flow it is sufficient to compute the evolution of the vertices of its initial state sets. This heuristic approach delivers the vertices of the initial state set representations and utilizes them during the sampling procedure. For example, the state set on Figure

3.2 is represented by a 2-dimensional polygon with 5 vertices depicted as blue dots. Note that this heuristic is not well scalable for higher dimensions. For instance, a two dimensional box has 4 vertices, whereas a three dimensional cube has 8 vertices. Therefore, the number of delivered vertices grows exponentially in the number of dimensions.

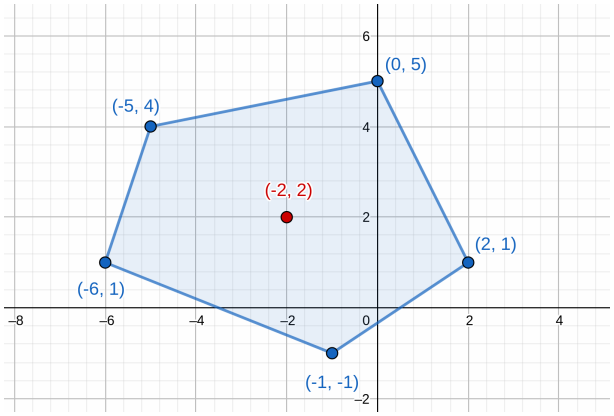


Figure 3.2: Calculating the center of gravity (the red point) of 2-dimensional polygon with 5 vertices.

CENTER

An alternative heuristic is to use the center of gravity for set sampling. This method delivers the average value of all initial state vertices by summing up component-wise all vertices together and dividing each component of the result by their total count. The output of this calculation delivers a spatial sample which is utilized for the further computation of the system flow. The formal definition for calculating the center of gravity is:

$$\frac{\sum_{i=1}^{n} \begin{pmatrix} x_{1i} \\ \vdots \\ x_{ki} \end{pmatrix}}{n}$$

where n is the number of vertices and k the dimension of the geometrical object. For example, on Figure 3.2 the center of gravity is calculated as follows:

$$\frac{\sum_{i=1}^{5} \begin{pmatrix} x_{1i} \\ x_{2i} \end{pmatrix}}{5} = \frac{\begin{pmatrix} -5 \\ 4 \end{pmatrix} + \begin{pmatrix} -6 \\ 1 \end{pmatrix} + \begin{pmatrix} -1 \\ -1 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 5 \end{pmatrix}}{5}$$
$$= \frac{\begin{pmatrix} -10 \\ 10 \end{pmatrix}}{5} = \begin{pmatrix} -10/5 \\ 10/5 \end{pmatrix} = \begin{pmatrix} -2 \\ 2 \end{pmatrix}$$

RANDOM-K

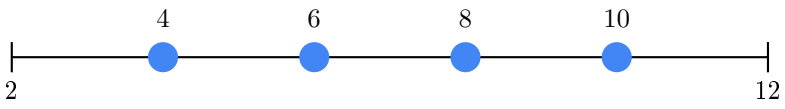
Unlike the methods proposed above this heuristic approach is non-deterministic. It randomly selects k uniformly distributed samples from a state set by using a random generator. Each dimension of the geometrical representation is once randomly sampled and the result is a coordinate of a point at the according dimension. The outputted samples of this heuristic are then utilized for further computation of the system flow.
For example, on Figure 3.2 the first dimension of the geometric object $x_1 \in [-6, 1]$ is once randomly sampled which results in a coordinate e.g. -3. The second dimension $x_2 \in [-1, 5]$ is also once sampled and the result is e.g. 3. The output of the heuristic `RANDOM-1` is in this case $\binom{-3}{3}$.
In Figure 3.3 `RANDOM-1` is applied. A randomly placed sample from the given initial set is chosen to represent the initial state of the analyzed system.

3.1.2 Heuristics for interval sampling

As the name suggests, these heuristics sample the temporal and not spatial position of the samples. Sampling an interval delivers a time sample, which describes the time of evolution of a spatial sample until reaching the guard to take a discrete transition. By applying the matrix exponential to the initial sample according to the time sample we can compute the spatial position of the chosen sample based on its initial position and the current flow dynamics.

UNIFORM-K

The interval, when a guard is enabled for a transition is a one-dimensional time interval. From that interval a given k number of equidistant points within this time interval are selected. They provide the initial state for calculating the next time steps. The given interval is divided into $k + 1$ smaller intervals of equal size. The upper bound of each small interval is picked as a sample. However, the upper bound of the last small interval is not considered, since it lies on the upper bound of the whole interval. We considered only samples which lie strictly within the interval, since it is known that the provided interval is over-approximated and its borders are most likely to lead to invalid spatial samples which do not satisfy the guard condition.
For instance, let $[2, 12]$ be an interval on which the UNIFORM-K heuristic is to be applied and let $k = 4$. The heuristic divides the interval into 5 smaller ones: [2,4], [4,6], [6,8], [8,10], [10,12]. Their upper bounds are chosen, without the last one. The choice of 4 equidistant points on the interval delivers the set $\{4, 6, 8, 10\}$.



BORDERS

This heuristic picks only the upper and the lower bound of the given interval. By computing the borders of the interval, an outline of the system flow is derived. For example, given the interval $[2, 12]$ only the samples 2 and 12 will be chosen.

RANDOM-K

From the intersection with the guard k randomly placed samples are picked for further computation. The heuristic works similarly to the one for spatial sampling with the difference that this one only samples in one dimension: time.

Note: k is a number, selected from the user. The larger the number, the more samples are chosen and the bigger the generated tree gets. Therefore, the simulation of the system takes longer to compute and requires more memory. However, if k is selected to be smaller number, it is possible that a counterexample run cannot be found and the simulation is indecisive. The choice of k in each heuristic should be taken into serious consideration and represent a thoroughly researched decision.
After choosing samples additional checks are carried out in order to verify them against both the guard and the invariant condition. In case of returning an invalid sample, the required number of verified samples cannot be achieved anymore. Therefore, re-sampling is applied with twice the samples as before. However, the required number of samples remains the same as in the beginning. If more samples than the required are valid, only those are taken which are in the nearest proximity of the guard. A possible behavior is that this process is iterated multiple times until the required number of valid samples is delivered. By the end of the sampling process, only valid samples are returned, which satisfy both the guard and the invariant condition.

3.2 Counterexample validation

After all heuristic approaches for temporal and spatial sampling are defined, they are implemented into the logic for simulation. The main role of the heuristics lies within utilizing them throughout following the critical path.
The counterexample validation approach follows a critical path provided by the classical reachability analysis by using an over-approximative representation type. Such path is analyzed again, until unsafety of the model is proven, or the end of the path is reached. By choosing multiple samples branching in the search tree occurs. Therefore, multiple runs of the hybrid system can be computed at the same time. However, in order to use this mode additional information about timings is required. For instance, the interval when a guard for a discrete jump is enabled or when a flowpipe has intersected the set of bad states. Such information cannot be retrieved by simulation sampling itself since it does not compute any time steps. Therefore, this approach relies on the search tree. In order to use sampling, we need to first use an over-approximative representation variant to derive such information. The sampling approach as defined in HyDRA is outlined as follows:

Following a path from a previous flowpipe computation

1. **Spatial sampling.** The result is a set of samples, which are chosen for computing their further evolution in time. For set sampling different heuristics can be applied, as described above.

2. **Compute the time evolution of the chosen samples to a certain point in time.** Regarding the flow dynamics in the current location, the matrix exponential of each sample is calculated. The matrix exponential describes the

time evolution of the given sample in the given location for a certain point in time. Multiplying the initial sample with the exponential matrix delivers the sample evolution after a given time period.

The equation for calculating the matrix exponential is:

$$e^{\delta A} = \sum_{k=0}^{\infty} \frac{(\delta A)^k}{k!} \tag{3.1}$$

where δ is the time step. Many time steps can be taken at once by using multiple multiplications of the matrix exponential. For example, let x_0 be a sample from the initial set. Its evolution for three time steps with size $0.1s$ would result in $x_1 = x_0 \cdot e^{0.3 \cdot A}$.

For instance, let $A = \left(\begin{smallmatrix} 0 & 2 \\ 0 & 0 \end{smallmatrix}\right)$ describe the flow in a certain location, $I = \left(\begin{smallmatrix} 3 \\ 4 \end{smallmatrix}\right)$ be the coordinates of the initial sample for this location and the time step length be $\delta = 1$. The matrix exponential is:

$$e^{1 \cdot A} = \sum_{k=0}^{\infty} \frac{A^k}{k!} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 & 2 \\ 0 & 0 \end{pmatrix} + 0 = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

The position of the initial state after one time step according to the flow is:

$$\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 11 \\ 4 \end{pmatrix}$$

This is, however, the theoretical definition of the matrix exponential. It is an endless sum, therefore, in practice it is numerically approximated [War77].

3. **Temporal sampling.** For this step the interval heuristics from above apply. The interval is derived from the intersection of the flowpipe and the guard condition. After choosing samples from the interval additional checks on the correctness of the samples are carried out, such as invariant satisfaction or containment in the bad states `P`. It is ensured that the exact required number of valid samples is delivered.

Repeat steps 2 and 3 until a counterexample run into the unsafe states `P` is computed, or until a predefined time bound or the end of the path is reached. In the case when a counterexample run is computed, there is a candidate found which disproves the system safety. Otherwise the result of the sampling method is inconclusive. The following Figure 3.3 visualizes the described simulation sampling approach. UNIFORM-3 interval sampling heuristic is applied and from each interval 3 samples are picked for further time computation. In this case the sampling method results in proving unsafety of the system.

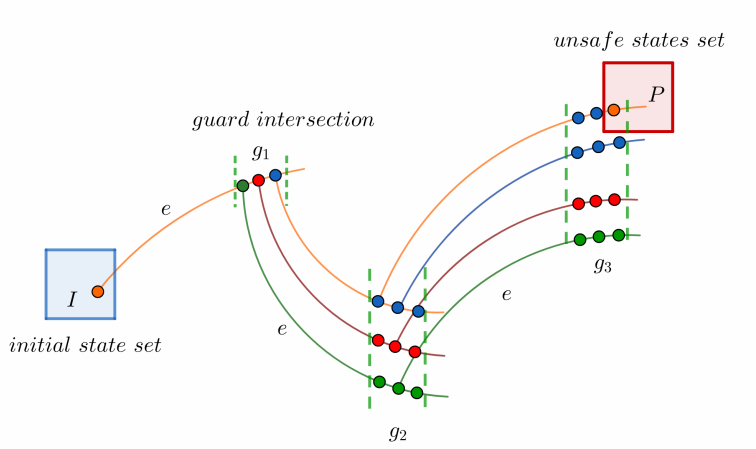


Figure 3.3: Sample the initial state I by picking a random point that is contained in the set I (1. step). Compute its time evolution until a guard g_1 is satisfied (2.step). From the guard intersection choose 3 equally distant from one another samples and compute their further time evolution (3. step). Steps 2 and 3 are repeated 2 times. Then a counterexample run is computed, which proves the system to be unsafe.

However, this approach is only successful when the last sample happens to be contained in the set of bad states `P`. This is often not the case. Then a refinement on the simulation is applied. Its objective is to investigate why the sample has not landed into `P` and nevertheless try to find a counterexample run. There are several causes for such cases:

- the previous `RefinementSetting` was not precise enough and does not deliver enough precision about timings. This leads to much larger intervals of enabled guard transitions, which causes heuristics to choose samples that are not contained in the actual guard or do not satisfy the invariant.
- the last sample before running into `P` is calculated in the near proximity of `P`, but not contained in `P`.
- there is indeed no actual run into `P` and the system has been safe. In such cases the sampling method remains inconclusive, since no run into `P` can be computed. Reachability analysis of the system should be done again with finer `RefinementSettings` in order to prove safety.

In order to deal with these cases and to find a counterexample run regardless of their causes, we define refinements on simulation sampling. Such refinements are triggered when an intersection with `P` is calculated with the previous over-approximative `RefinementSetting` and the last flowpipe segment is taken into consideration, but still no run into `P` is computed. Figure 3.4 graphically depicts the concept step-by-step for a better understanding.

Simulation refinement

This part requires both types of workers to cooperate: The simulation worker determines the initial states for a flowpipe construction which is then calculated by the reachability worker.

1. **Backtrack to parent sample and construct a flowpipe out of it.** Each sample is stored as an initial state in a corresponding node in a tree data structure. By backtracking to the parent node we can retrieve the initial state for the further flowpipe construction.

2. **Refine guard.** By constructing such flowpipe we obtain more insights about the sample evolution and its timings. With the new information a refinement of the over-approximated interval of enabled guard transition is applied. The refined interval can either contain the child sample, or not, as presented on the following figure:

   - if the refined interval (depicted in blue) contains the child sample (depicted in green), return to step 1 and backtrack to parent again. This case leads to the conclusion, that the child sample is correctly chosen and satisfies the guard and the invariant. The indecisiveness is not caused by choosing this sample badly, but some of its parent samples.
   - if the refined interval does not contain the child sample (depicted in red), choose another sample from the finer interval that also satisfies the invariant. On this new sample apply simulation as described in Section 3.2.

3. **Check whether intersection is calculated**

   - if after all refinements are applied and still no run into the unsafe states is found, there can be no statement derived about the safety or the unsafety of the system. Therefore, its analysis is inconclusive.
   - if an intersection is computed the unsafety of the system is successfully proven by providing a counterexample run into the set of unsafe states.

3.2.1 Guard expansion

As simulation sampling suggests each state is computed regarding the size of a time step. This means that we only know the position of the sample state in a certain time and not throughout the whole system evolution. Hence satisfying a guard equation is nearly impossible. For instance, let a guard equation be $x = 0$. Computing a sample which lands exactly on the line 0 requires a specific size of the time step, which is not possible to derive.
A solution of this problem is expanding the guard and representing it as two inequalities which describe the upper and the lower bound of the guard. Such guard widening enforces samples to be able to satisfy the guard condition and to land within its expansion. For instance, let the guard from above be expanded by 0.1 at each side resulting into an interval $x = [-0.1, 0.1]$. It is still a considerable small guard, however, allows sample states to land into it and enables their further computation.

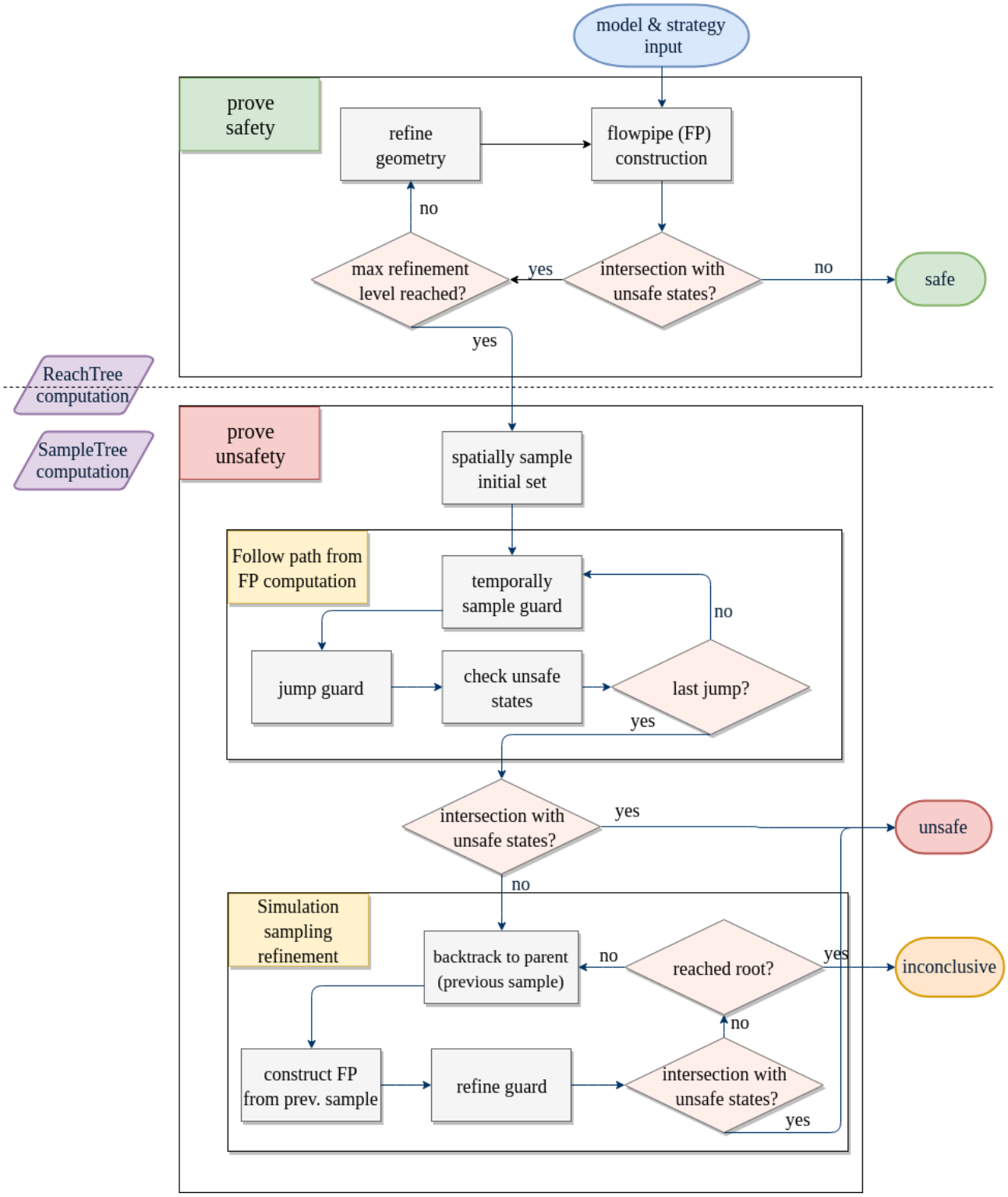


Figure 3.4: Functional concept of the workers.

Chapter 4

Evaluation

In this chapter we are going to evaluate our method against a selection of benchmarks to investigate its practical applicability. A benchmark consists of a model of a hybrid automaton, a defined set of bad states and a strategy. Several simulation strategies are applied on each model with different heuristics. The efficiency and effectiveness of the respective heuristics are evaluated, compared and summarized.

4.1 Bouncing ball

The model describes a bouncing ball dropped from a given height ($x = [10, 10.2]$) and with a certain vertical velocity ($v = 0$). Once released, the ball starts falling while loosing some of its potential energy. After reaching the ground ($x = 0$) the ball bounces up, reaches a certain height and starts falling again. The physical behavior of the ball can be represented by the following hybrid automation:

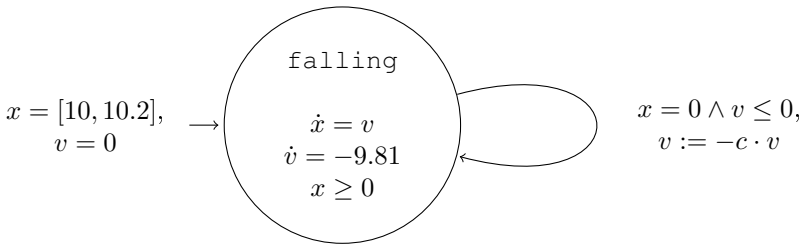


Figure 4.1: Graphical representation of a hybrid automaton, modelling the behavior of a bouncing ball.

In this model it is assumed that the ball has a mass of $m = 1$. Its movement is described by the following differential equation:

$$\dot{x} = v$$
$$\dot{v} = -9.81$$

While falling the ball is attracted to the ground by the Earth gravitational force $9.81 m/s^2$. The invariant $x \geq 0$ ensures that the ball stays above or at the ground. The guard label $x = 0 \wedge v \leq 0$ ensures that bouncing happens after the ball was

falling and as soon as it reaches the ground at $x = 0$. The reset function characterizes the loss of the ball's energy from the impact with the ground, which depends on the softness of the ball. This parameter is described as c and accepts values on the interval between $[0, 1]$. The smaller the number, the softer the ball is and the more energy is lost due to the impact with the ground. For the following evaluation c will be set to $c = 0.75$.
One location is sufficient to represent the movement of the ball, since only a sign change can describe whether the ball is falling or rising. We call such behavior symmetrical. The model is also deterministic, since for each trajectory at each point in time its location can be determined.
The set of bad states P against which the system is being verified to be safe or unsafe is selected to be a very small box with dimensions $x = [0, 0.1]$ and $v = [0, 0.2]$ in order to demonstrate the system behavior for a reasonably long time. The guard is being expanded in order to allow the samples to satisfy its condition, as presented in Section 3.2.1. The part of the guard from the model above where $x = 0$ is modified to be an interval $x = [-0.1, 0.1]$.

4.1.1 Trace simulation

The first strategy that we evaluate contains only one refinement setting with a sample as a representation:

```
Strategy computeTimeAndDiscrete{
  RefinementSetting {
    timeStep: 0.01,
    representation: sample,
    aggregation: no,
    clustering: no
  }
}
```

During the simulation execution one possible run of the system is computed, starting form the initial set. At each time step of size $0.01s$ the position of the ball is calculated based on the exponential matrix as seen in Equation 3.1. The simulation results in a trace describing the ball movement, as depicted on Figure 4.2.
This mode of sampling simulation is suitable for hybrid systems with a deterministic location of the control, i.e. at each point in time it can be determined at which location of the system the control currently is. Branching cannot be handled, which leads to excluding possible scenarios of system evolution and considering only one trajectory of one scenario. The mode does not use any heuristics about interval sampling, since we do not have such information. The jump is taken with the last state which satisfies the guard, since it was most suitable for our benchmarks, where the last state is calculated at the nearest proximity of the exact guard equation.

4.1.2 Counterexample validation

In this mode we evaluate two strategies where each of them has three refinement settings, as defined in Listings 4.1 and 4.2. The first refinement setting uses a box as an over-approximative state representation. The second configuration uses a point as a state set representation to indicate a mode-switch in the goal of verification: from safety to unsafety proving (see Equation 2.1). The third refinement setting is again a box, but has a finer time step than the previous refinements. This setting is used

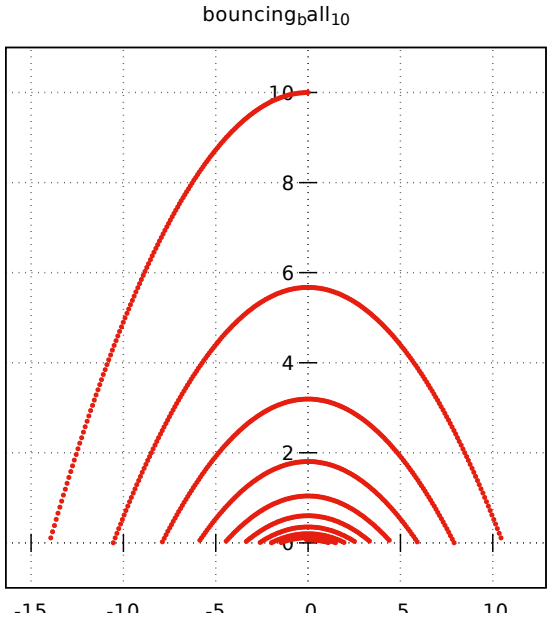


Figure 4.2: Simulation of bouncing ball model in mode 1 using `computeTimeAndDiscrete`.

to refine timing information which was obtained during safety verification in order to improve the simulation. It is utilized to calculate small flowpipes out of the samples in order to retrieve more exact information about timings (as described in Section 3.2).

```
Strategy sampleAggregation {
  RefinementSetting1 {
    timeStep: 0.1,
    representation: box,
    aggregation: yes,
    clustering: no
  },
  RefinementSetting2 {
    timeStep: 0.1,
    representation: sample,
    aggregation: yes,
    clustering: no
  },
  RefinementSetting3 {
    timeStep: 0.01,
    representation: box,
    aggregation: yes,
    clustering: no
  }
}
```

Listing 4.1: `sampleAggregation` strategy.

```
Strategy noSampleAggregation {
  RefinementSetting1 {
    timeStep: 0.1,
    representation: box,
    aggregation: yes,
    clustering: no
  },
  RefinementSetting2 {
    timeStep: 0.1,
    representation: sample,
    aggregation: no,
    clustering: no
  },
  RefinementSetting3 {
    timeStep: 0.01,
    representation: box,
    aggregation: yes,
    clustering: no
  }
}
```

Listing 4.2: `noSampleAggregation` strategy.

The difference between both strategies from Listings 4.1 and 4.2 is that `RefinementSetting`$_2$ uses aggregation on multiple samples which satisfies the guard and invariant condition. Aggregation ensures that a set of samples satisfying a guard condition is used as a basis to obtain exactly one sample as a discrete jump successor for the respective transition. The classical aggregation method is to build a convex hull out of the given state sets. However, building a convex hull out of multiple samples would not result in a single sample. Therefore the aggregation method for samples is to choose only one of them. The most accurate choice is the last sample which satisfies both conditions, since it is at the nearest proximity to the guard and also fulfills the invariant.
By applying `RefinementSetting`$_1$ and utilizing the reachability worker an over-approximation of the ball movement is calculated. On Figure 4.5 it is depicted by the blue boxes. The first two flowpipes do not intersect with the set of bad states which is considered a success, whereas the third one does intersect. In such case `RefinementSetting`$_2$ is utilized. Due to using samples as a representation, the system switches from trying to prove safety to proving unsafety of the model. The simulation sampling starts off by sampling the initial state, where various heuristics can be applied. Hereby we applied `CENTER` which results in a sample with coordinates $\binom{10.1}{0}$ and is represented by a red dot with the respective coordinates on Figure 4.5.

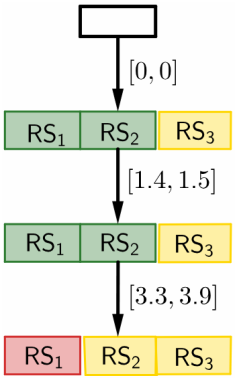
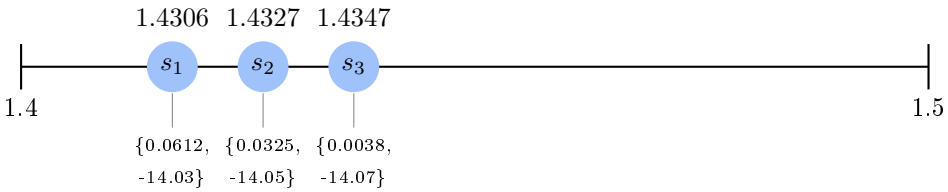


Figure 4.3: A search tree with three levels of refinement settings RS. The tree corresponds to the results from Figure 4.5.

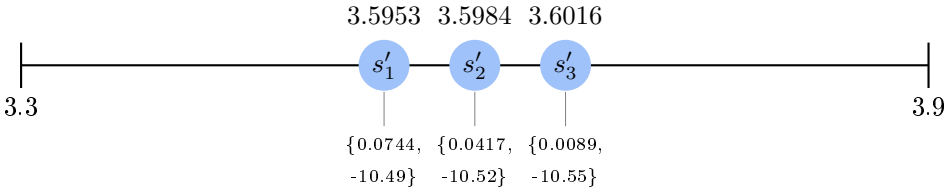
Due to the application of `RefinementSetting`$_1$ a search tree is created, which contains time stamps of all initial sets and whether a flowpipe is safe or unsafe, along with some other information. The tree which contains the results after applying all three refinement settings is depicted on Figure 4.3. The colors in the tree nodes indicate that the first two flowpipes generated from RS_1 and RS_2 have not intersected the set of bad states, hence green, whereas the third one has, hence the red color. The further refinements on the path are inconclusive, hence colored in yellow.
Furthermore, the critical path π storing all time and discrete steps of the previous simulation is utilized in order to retrieve information about the intervals of enabled guard transition. In the case of bouncing ball model the critical path is:

$$\begin{aligned}\pi = &[1.4, 1.5], \texttt{falling} \rightarrow \texttt{falling}[1.4, 1.5], \\ &[3.3, 3.9], \texttt{falling} \rightarrow \texttt{falling}[3.3, 3.9]\end{aligned}$$

The intervals which we take into consideration are these from the discrete steps. Following the path π, we apply interval sampling heuristics in the simulation worker in order to retrieve the last possible states in the according flowpipe. These states are required for taking a discrete jump and to calculate the initial set for the next flowpipes. Hereby we utilize `UNIFORM-3`, which delivers 3 samples from the interval it is applied on.
Sampling the first interval $[1.4, 1.5]$ of the path delivers 3 samples with respective coordinates {x, v}:



The three sample states satisfy the guard ($x = [-0.1, 0.1]$) and the invariant condition ($x \geq 0$). From each of these states a task is created and is put into the global queue in order to take its evolution into further consideration by utilizing the simulation worker. A discrete jump is taken with the samples and the resulting states are stored as an initial state of a node in the search tree. The next interval on the path which we sample is $[3.3, 3.9]$ by using the same heuristic.



The states which satisfies both the guard ($x = [-0.1, 0.1]$) and the invariant condition ($x \geq 0$) are depicted above. Their further evolution is computed until the point of previous intersection with the set of unsafe states. This information is stored in the nodes of the search tree and based on it we know at which time the intersection occurred. If at this point in time the calculated sample state is contained in the unsafe set P, then the model is proven to be unsafe and the system has provided a counterexample.
In our case the sample state did not intersect the unsafe set, which triggers the next $\texttt{RefinementSetting}_3$. It aims at investigating the reason why a counterexample run could not be provided. This refinement setting is utilized for creating small flowpipes out of each initial sample state with the purpose of obtaining additional information about the time intervals in which the guard condition was enabled. The flowpipes have finer time step which leads to deriving more precise information about the time intervals.
First a flowpipe is constructed from the last sample state which is not contained into the set of bad states P. The small flowpipe has the objective to check whether the sample state was computed in the near proximity of the unsafe set because of inaccurate previous interval sampling. Calculating an intersection of the flowpipe with the set of bad states is a prerequisite for finding a counterexample run and to prove the model to be unsafe. However, if such intersection is computed to be empty, the simulation sampling refinement from Section 3.2 is triggered and backtracking to the parent sample state is applied, as happens in the case of the bouncing ball.
Due to the tree structure and the clear child-parent relationship maintained in the tree finding the parent sample state is done very efficiently. It is utilized as the initial state for computing the next small flowpipe the objective of which is to derive an interval of the enabled guard transition. Due to the finer time step of the small flowpipe than of the over-approximated one by $\texttt{RefinementSetting}_1$ the derived interval is also finer and more accurate. The containment of the previously chosen sample state is

checked against the finer interval in order to examine its validity as a legitimate state which is indeed part of the system flow.
In the case when the sample state is not contained in the finer interval a conclusion is made that this sample was badly chosen, hence no run into the unsafe states could be computed. The more precise interval is then sampled by utilizing `UNIFORM-1` heuristic for interval sampling which delivers the center of the interval it is applied on. From this sample a forward simulation is triggered which aims at finding a counterexample run.
However, containment of the sample in the more precise interval leads to the conclusion that it is a legitimate choice for a sample, which is indeed part of the flow. Therefore, not being able to find a counterexample run is not because of the choice of this sample, as happens in the case of the bouncing ball, and backtracking to the parent sample state takes place again, until the root is reached. The sample choices are verified to fulfill the conditions of the invariant and of the time intervals of the enabled guard condition which leads to the conclusion that the simulation of the system is inconclusive, since $\texttt{RefinementSetting}_1$ uses a too large time step and a state set representation which introduces too much over-approximation errors.
By defining the set of bad states as a box with coordinates $x = [3, 2]$ and $v = [4, 2]$ the the system proves that the given model is unsafe regardless of the too large over-approximation of $\texttt{RefinementSetting}_1$, as depicted on Figure 4.4.

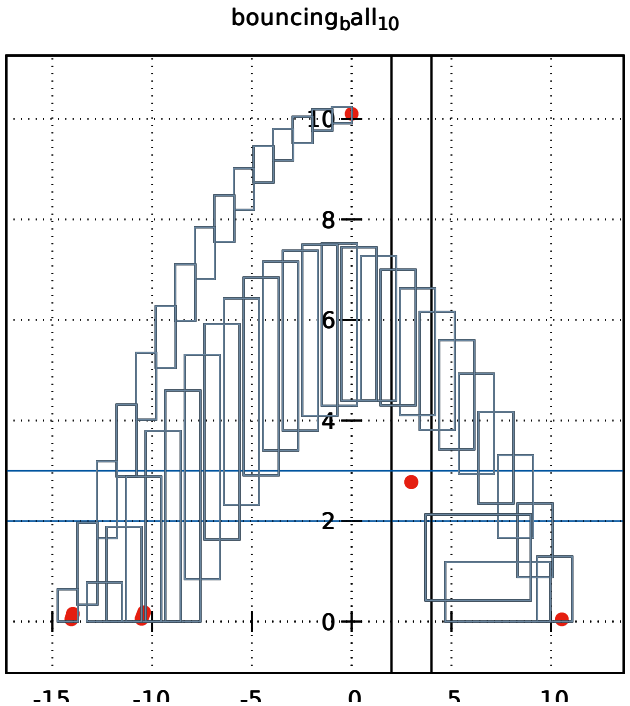


Figure 4.4: Simulation in mode 2 of the bouncing ball model using strategy `sampleAggregation` by utilizing `UNIFORM-3` interval sampling heuristic and `CENTER` set sampling heuristic. The simulation provides a counterexample candidate for proving the system unsafety.

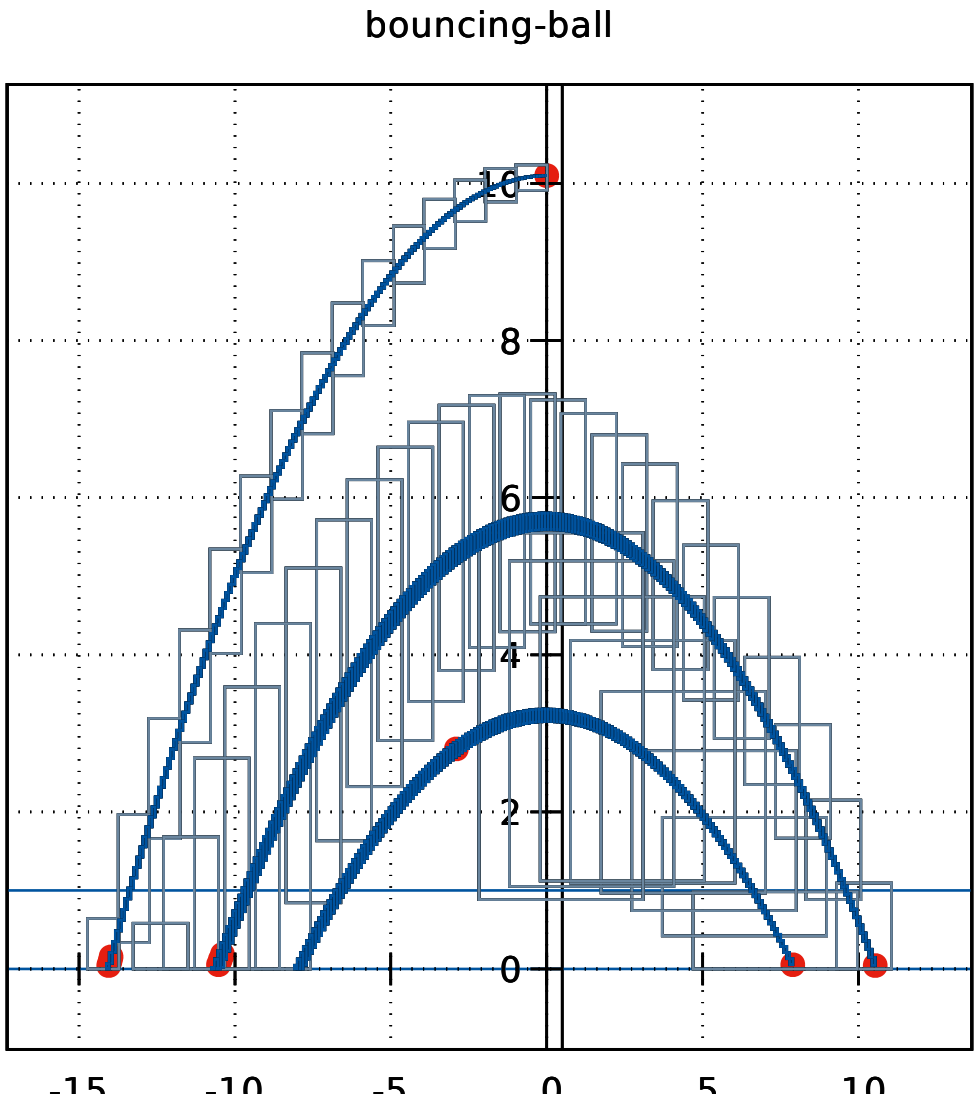


Figure 4.5: Simulation in mode 2 of the bouncing ball model using strategy `sampleAggregation` by utilizing `UNIFORM-3` interval sampling heuristic and `CENTER` set sampling heuristic. The result of the simulation is inconclusive.

4.2 Rod reactor

This model represents a reactor in a power plant which consists of a reactor tank and two rods with different cooling dynamics. By putting the rods into the tank the temperature in starts to lower according to the cooling dynamics of the rods. Without rods the temperature in the tank increases. There are two clocks c_1 and c_2 which are introduced in the system in order to measure the time of utilization of the respective rods. Each rod can be utilized for maximal time span of 20 seconds. The purpose of the system is to keep the temperature in the tank restricted between $510°C$ and $550°C$ and in case it exceeds, the system should be turned off for safety.

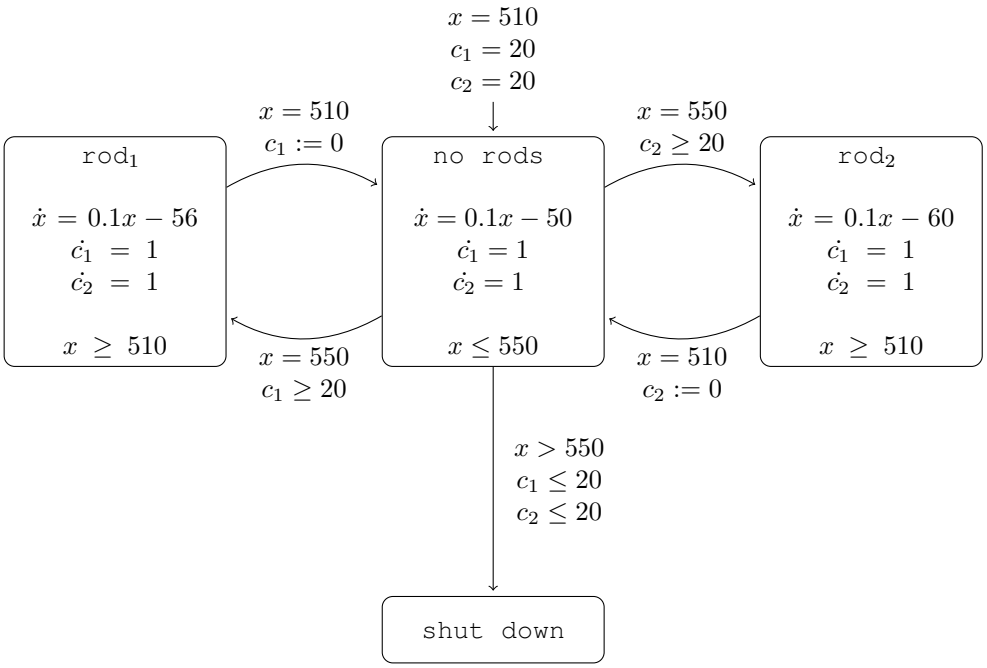


Figure 4.6: Graphical representation of a hybrid automaton, modelling the behavior of a rod reactor.

The initial temperature in the reactor tank is $510°C$ and we assume that both rods have already been utilized for 20 seconds each. At the beginning there are no rods in the tank and the temperature in it starts to increase as described by the following differential equation: $\dot{x} = 0.1 \cdot x - 50$. As soon as it reaches $550°C$ one of the rods is put into the tank in order to cool down the system. One of the rods has a cooling dynamics described by the differential equation $\dot{x} = 0.1 \cdot x - 56$ and the other one by $\dot{x} = 0.1 \cdot x - 60$, meaning that the first rod cools down the system slower than the second rod. By reducing the temperature in the reactor tank to $510°C$ the rod is pulled out and the system starts to heat up again.
This model is non-deterministic, since either rod_1 or rod_2 can be chosen to be put into the reactor tank at a time, however not both. The control of the system can nondeterministically decide which rod to take when the temperature increases to $550°C$.
All reachable states contained in location `shut down` are defined to be the bad states.

The guards $x = 510$ and $x = 550$ are extended by 0.1 at each side which delivers the intervals $x = [509.1, 510.1]$ and $x = [549.9, 550.1]$ respectively. Nevertheless, the temperature in the tank never drops under $510°C$, nor it exceeds $550°C$, since the invariants at each location do not allow this.
The model can be verified against a predefined set of unsafe states P. During the evaluation process the set of bad states P is defined to be any state which satisfies the following condition: after 70 seconds of utilization the temperature in the tank lies between $515°C$ and $520°C$.
Additionally, the applied strategy for the model evaluation is `sampleAggregation`, as introduced in Listing 4.1. The first refinement setting uses a box as a state set representation and has a time step of 0.1. Starting at time $c_1 = 20s$ and $c_2 = 20s$ in location `norod` the system starts to heat up. After ca. 16 seconds of exploitation the tank reaches a temperature of $550°C$ and one of the rods should be submerged into the tank in order to cool it down. From location `norod` the control can jump either into location `rod1` or `rod2`. The two possible scenarios are considered which results in branching, as depicted on Figure 4.7 at $c_1 = 36s$ and $c_2 = 36s$. All flowpipes for both possible scenarios are computed, until the set of bad states P is hit. The resulting search tree with its critical path is created during the computation and is hand over to the second refinement setting in the strategy. The critical path π for this model is:

$$\begin{aligned}\pi = &[36, 36.2], \texttt{norod} \rightarrow \texttt{rod2}[36, 36.2],\\ &[41.8, 42.1], \texttt{rod2} \rightarrow \texttt{norod}[41.8, 42.1],\\ &[57.8, 58.3], \texttt{norod} \rightarrow \texttt{rod1}[57.8, 58.3]\end{aligned}$$

The path indicates that switching the control at time interval [36, 36.2] to location `rod2` would hit the bad states at a future point in time.
The second refinement setting utilizes aggregated sampling. By following the critical path, the initial set and the following intervals are sequentially sampled and depicted as red dots on Figure 4.7. The chosen sampling heuristics are `CENTER` for set and `UNIFORM-3` for intervals. After computing the time evolution of the initial sample form the the last segment, it was found that at time $c_1 = 72s$ the sample is contained in the set of unsafe states P. By computing a run from the initial states to P it is proven that the model is unsafe with respect to the defined P.
In case of proving unsafety, triggering the third refinement setting from the strategy is not needed. The system evaluation has provided a clear result and further refinement of the state sets may not lead to new conclusions.
Applying strategy `noSampleAggregation` leads to the same result. The difference is that after each sampling a constant number of new samples is introduced for each existing sample leading to an exponential growth in the total number of samples. The paths from all samples result in the defined set of unsafe states. Applying different sampling heuristics also did not change the result. This unambiguity speaks for an already sufficiently refined approximation of the set of reachable states obtained by the preceding over-approximative refinement setting.

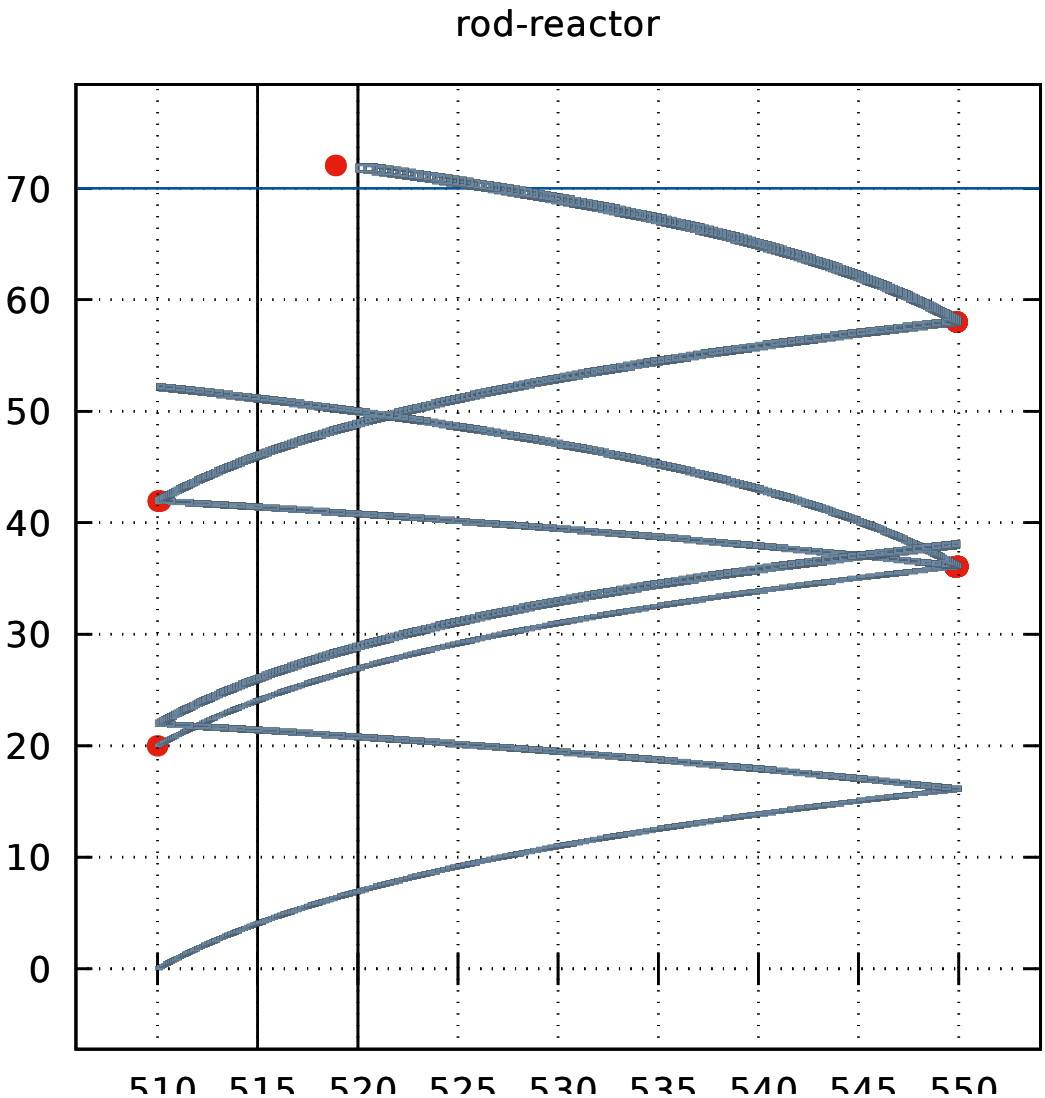


Figure 4.7: Simulation in mode 2 of the rod reactor model using `Strategy sampleAggregation` by utilizing `UNIFORM-3` interval sampling heuristic and `CENTER` set sampling heuristic. On the horizontal axis the temperature in the tank is depicted, whereas on the vertical the global time. The result of the simulation is a counterexample candidate for system unsafety.

4.3 Heuristics evaluation results

Based on the model examples above we evaluate the set and interval sampling heuristics which were introduced in Chapter 3.1. Various sets of bad states are defined in order to analyze the success rate of providing a counterexample run.

4.3.1 Set sampling heuristics

VERTICES:

This heuristic delivers multiple initial states, which leads to more initial nodes and therefore to a larger search tree. It produces more runs of the system. The probability of providing a counterexample run is therefore higher than with fewer initial states.

CENTER:

Only one initial state is delivered and is stored as a node in the search tree. Depending on the interval sampling heuristic the tree can remain narrow or start to expand further at each level. Although only one initial state is provided the probability of finding a counterexample run remains high, since the initial state is located at the exact center of the state set.

RANDOM-K:

This state sampling heuristic delivers k initial states. The probability of computing a counterexample run still remains high, since multiple samples can be chosen which are randomly spread across the whole initial state set.

In general we could observe, that the impact of the initial set sampling is rather small compared to the influence of the time interval sampling methods, since the set sampling is done only once at the beginning of the simulation. Choosing multiple spatial samples did not lead to significant improvement when proving unsafety, nor did require more computational time, since the samples evolution delivered the same over-approximated interval of enabled guard transitions.

4.3.2 Interval sampling heuristics

UNIFORM-K:

This interval sampling heuristic delivers a total of k states from the interval it is applied on, which leads to k branches in the search tree and produces therefore more runs of the system. The larger the number k the more iterations are needed to provide the required number of samples. In general, the probability of providing a counterexample run with larger k is higher than with fewer number of states. At each level of the tree the branching grows exponentially in the number of k. Consequently, the required computational power also grows with the number of runs, therefore the larger the k, the more time and computational resources are required for the simulation.

BORDERS:

The heuristic delivers only two samples at each side of the borders of the interval it is applied on. Since the timing information provided by a previous `RefinementSetting` deliver an over-approximated interval, the two samples cannot satisfy the conditions of the guard. Applying this heuristic on the models introduced above is only applicable, when the previous `RefinementSetting` delivers not an over-approximated interval, but an exact one. Only during a simulation where an under-approximative representation variant is chosen this heuristic is applicable and the samples would satisfy both the guard and the invariant condition.

RANDOM-K:

This heuristic delivers k number of valid interval samples from a given interval. Intuitively the higher the number k the higher the possibility of providing a counterexample run, but also the wider the search tree gets. Similarly to `UNIFORM-K` heuristic, the branching of the tree at each level grows exponentially in the number of k.

The choice of the applied interval sampling heuristic throughout the model simulation has an immense impact not only on the size of the tree but also on the required computational power. Since the algorithm of the simulation worker utilizes both set and sampling heuristics, such impact can only be measured by combining them together and evaluating the required time for computation.

4.3.3 Combination of interval and set sampling heuristics

This section includes the exact evaluation results of the Bouncing ball and the Rod reactor models. Their safety is checked against multiple definitions for sets of bad states. Moreover, we measured the computational time required for analyzing the models and we considered the success rate of providing a counterexample run. We compare the number of provided counterexamples (CEX) to the total number of runs which are generated and analyzed. During the evaluation process the average branching of the search tree, total number of processed tasks and total number of tree nodes are also taken into consideration.
Both models are analyzed by utilizing `sampleAggregation` from Listing 4.1 and `noSampleAggregation` from Listing 4.2 strategies.
From the evaluation of the benchmarks we expect to show that by utilizing aggregation on the samples before a discrete jump, the complexity of the computation increases only linearly, whereas when aggregation is not applied, the complexity increases in an exponential rate. The reason for the exponential complexity is the fact that at each discrete jump from one sample k more samples are generated. We also expect to detect deviations in the computational time and/or in the success rate which result from choosing odd or even numbers of samples. The odd numbers of samples always provide a sample which is placed exactly in the middle of an interval and is more likely to be as closest to the guard line. Therefore, delivering a counterexample run is more likely to succeed.

Evaluation environment

We evaluated our benchmarks on a machine with 16GB RAM and 4x Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz, 64-bit architecture.

Bouncing ball

The safety of the bouncing ball model is verified against two different sets of bad states:

$$P_1{}^{bb} = x \in [0{,}1] \wedge v \in [0, 0.5]$$

and

$$P_2{}^{bb} = x \in [2.7, 3.1] \wedge v \in [1.7, 3].$$

The first set $P_1{}^{bb}$ is chosen so that the over-approximation of $\texttt{RefinementSetting}_1$ from strategy `(no)sampleAggregation` hits the set of unsafe states, whereas the samples from $\texttt{RefinementSetting}_2$ are not calculated within $P_1{}^{bb}$. This means that $\texttt{RefinementSetting}_3$ is triggered. The goal of defining the set of bad states in this way is to show the worst case computation time required for the simulation and for the construction of the following flowpipes. It is the worst case, since a counterexample run could not be provided and from each sample a fine flowpipe should be constructed to show inconclusiveness. The results form the benchmark lead to the conclusion that $\texttt{RefinementSetting}_1$ delivered a too over-approximated computation, which does not provide sufficient information about the model behavior.
The second set $P_2{}^{bb}$ is defined in such a way that both the over-approximation of $\texttt{Refinement- Setting}_1$ from strategy `(no)sampleAggregation` and some of the samples from $\texttt{RefinementSetting}_2$ intersect with it. We aim to investigate which heuristic is more suitable and has a higher success rate of finding a counterexample run.
The first row in each of the following tables represents the required computational time for applying $\texttt{RefinementSetting}_1$. It establishes a baseline for calculating the time needed for the computation of the samples from the second refinement setting and the eventual flowpipe construction from the third one, which starts at each sample. This way we measure the time for the overall calculation process required to provide a counterexample run in a model.
The results provided in Table 4.1 indicate that the complexity of the computation indeed increases linearly with the number of samples. The computational time includes the required time for the calculation of all three refinement settings from the strategies. The various choice of heuristics did not succeed at providing a counterexample run as expected, since the delivered computation from $\texttt{RefinementSetting}_1$ is too over-approximated and does not deliver sufficient insights into the model evolution.
The number of nodes and the processed tasks is for each evaluation result in Table 4.1 the same, since aggregation is applied. These numbers are, however, in Table 4.2 different, since at each discrete jump the number of samples increases exponentially. The more samples are chosen, the bigger the branching factor. There is only one path when applying `CENTER` since only one run of the system is computed which starts from the center of the initial state set. By applying `VERTICES` there are four chosen samples from the initial state set, namely the vertices of the initial box, which initialize four paths.
Applying `UNIFORM-1` interval sampling heuristic requires the same amount of computational effort irregardless whether aggregation is applied or not. This is due to the

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1	3	3	-	0.102
CENTER	UNI−1	1	10	4	0% (0/1)	0.112
	UNI−2	1	10	4	0% (0/1)	0.119
	UNI−3	1	10	4	0% (0/1)	0.120
	UNI−20	1	10	4	0% (0/1)	0.154
VERT	UNI−1	1.6	35	16	0% (0/4)	0.147
	UNI−2	1.6	33	16	0% (0/4)	0.163
	UNI−3	1.6	33	16	0% (0/4)	0.188
	UNI−20	1.6	33	16	0% (0/4)	0.319

Table 4.1: Overview of the evaluation results of the bouncing ball model. The utilized strategy is `sampleAggregation` and the bad states are ${P_1}^{bb}$.

fact, that the heuristic delivers only one sample from the given interval which does not lead to branching.
Comparing `UNIFORM−1` with the other `UNIFORM−K` interval sampling heuristics when applying `VERTICES` as a set sampling heuristic we notice that the number of processed tasks is different, although the number of nodes stays the same. The interval which is delivered by the first refinement setting is over-approximated and the chosen by `UNIFORM−1` sample is not contained in the finer interval, derived by the third refinement setting. This requires resampling and calculating the trace of the new samples.

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1	3	3	-	0.102
CENTER	UNI−1	1	10	4	0% (1/1)	0.112
	UNI−2	1.27778	21	11	0% (0/9)	0.137
	UNI−3	1.4375	38	22	0% (0/20)	0.173
	UNI−20	1.8925	1452	958	0% (0/400)	2.942
VERT	UNI−1	1.6	35	16	0% (0/4)	0.149
	UNI−2	2.12222	91	54	0% (0/20)	0.284
	UNI−3	2.36	171	106	0% (0/44)	0.446
	UNI−20	3.02835	5829	3852	0% (N/A)	11.40

Table 4.2: Overview of the evaluation results of the bouncing ball model. The utilized strategy is `noSampleAggregation` and the bad states are ${P_1}^{bb}$.

The results from Tables 4.3 and 4.4 confirm our expectations for providing a counterexample run. The overall time required is less than in Tables 4.1 and 4.2, since the computation of the small flowpipes from $\texttt{RefinementSetting}_3$ is (in most cases)

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1	3	3	-	0.102
CENTER	UNI-1	1	6	3	100% (1/1)	0.102
	UNI-2	1	6	3	100% (1/1)	0.106
	UNI-3	1	6	3	100% (1/1)	0.109
	UNI-20	1	6	3	100% (1/1)	0.143
VERT	UNI-1	1.6	25	14	50% (2/4)	0.129
	UNI-2	1.6	15	12	100% (4/4)	0.120
	UNI-3	1.6	15	12	100% (4/4)	0.131
	UNI-20	1.6	15	12	100% (4/4)	0.278

Table 4.3: Overview of the evaluation results of the bouncing ball model. The utilized strategy is `sampleAggregation` and the bad states are ${P_2}^{bb}$.

not required.
An interesting result can be observed when applying `VERTICES` as set a sampling heuristic and `UNIFORM-1` as an interval sampling heuristic. The success rate of finding a CEX is only 50% in comparison with the other heuristics, which have a 100% success rate. This is due to the fact that `UNIFORM-1` provides less accurate samples, which is further from satisfying the invariant and the guard than the samples provided by the other interval sampling heuristics. This leads to a larger approximation error and deviation from the exact model behavior during the computation.

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1	3	3	-	0.102
CENTER	UNI-1	1	6	3	100%(1/1)	0.102
	UNI-2	1.41667	14	8	75% (3/4)	0.127
	UNI-3	1.75	23	15	78% (7/9)	0.141
	UNI-20	4.24821	673	569	N/A	1.454
VERT	UNI-1	1.6	25	14	50% (2/4)	0.129
	UNI-2	2.35	57	40	70% (14/20)	0.200
	UNI-3	3.06	91	70	81% (36/44)	0.273
	UNI-20	9.38392	2793	2350	N/A	5.420

Table 4.4: Overview of the evaluation results of the bouncing ball model. The utilized strategy is `noSampleAggregation` and the bad states are ${P_2}^{bb}$.

Another interesting result emerges when comparing Tables 4.3 and 4.4. Aggregating the samples leads to a 100% success rate and requires less computational effort. Not aggregating them allows for the paths to "spread out" and not always be able to reach the unsafe states. Moreover, choosing more samples provides for higher success rates.

However, the required computational effort increases exponentially with the number of samples.
The conclusion which can be derived from the results for the bouncing ball model is that aggregating the samples before a discrete jump is a useful mechanism, since the same result is delivered as without aggregation, but with less computational effort and irregardless of how over-approximated the calculation of the previous refinement setting is.
The evaluation results confirmed our expectations for a linear complexity when applying aggregation as well as an exponential complexity when not applying it. Moreover, by using sample aggregation the delivered result is derived not only faster, but also with a higher success rate.

Rod reactor

The safety of the rod reactor model is also verified against two different sets of bad states:

$$P_1{}^{rr} = x \in [515, 520] \wedge c_1 \geq 70$$

and

$$P_2{}^{rr} = x \in [525, 528] \wedge c_1 \in [69.8, 70.3].$$

The first set $P_1{}^{rr}$ is defined so that both the over-approximation of $\texttt{RefinementSetting}_1$ as well as the samples from $\texttt{RefinementSetting}_2$ from strategy `(no)sampleAggregation` hit the set of unsafe states. This means that $\texttt{RefinementSetting}_3$ would not be triggered, since all paths already lead in $P_1{}^{rr}$. Therefore, the required computational time in Tables 4.5 and 4.6 includes the computation of the flowpipe construction from the first refinement setting and the samples evolution from the second one, which represents the best case during a simulation. It is the best case, since no flowpipes are computed from the samples and the exact computational time for sampling simulation can be derived by subtracting the computational time for the reachability analysis from the time for the whole analysis.
The second set $P_2{}^{rr}$ is chosen so that the over-approximation of $\texttt{RefinementSetting}_1$ from strategy `(no)sampleAggregation` hits the set of bad states. Some of the samples from $\texttt{RefinementSetting}_2$ are also calculated within $P_2{}^{rr}$, however, not all of them. Therefore, the third refinement setting is triggered and more computational effort is required in order to construct small flowpipes out of them.
The first row in each of the following tables represents the required computational time for the reachability analysis calculated with $\texttt{RefinementSetting}_1$. It establishes a baseline for calculating the time needed for the computation of the samples from the second refinement setting and the eventual flowpipe construction from the third one, which starts at each sample. This way we measure the time for the overall calculation process required to provide a counterexample run in a model.
The results from the benchmarks are expected to confirm our assumptions, that, similarly to the bouncing ball model, the complexity of the calculation increases linearly when applying aggregation to samples at each discrete jump, whereas without aggregation the complexity increases exponentially.
The number of processed tasks in Table 4.5 is only influenced by the applied set sampling heuristic. Using `CENTER` heuristic delivers one sample, which is placed at the center of the three dimensional cube formed by defining the initial state set of the

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1.16667	8	8	-	0.125
CENTER	UNI-1	1	12	4	100% (1/1)	0.128
	UNI-2	1	12	4	100% (1/1)	0.128
	UNI-3	1	12	4	100% (1/1)	0.132
	UNI-20	1	12	4	100% (1/1)	0.169
VERT	UNI-1	1.77778	40	32	100% (8/8)	0.141
	UNI-2	1.77778	40	32	100% (8/8)	0.145
	UNI-3	1.77778	40	32	100% (8/8)	0.175
	UNI-20	1.77778	40	32	100% (8/8)	0.488

Table 4.5: Overview of the evaluation results of the rod reactor model. The utilized strategy is `sampleAggregation` and the bad states are ${P_1}^{rr}$.

model. A cube has 8 vertices, therefore, by utilizing the `VERTICES` heuristic, eight traces are computed.
The results from Table 4.5 confirm our expectation for linear complexity of deriving the solution by using aggregation, whereas the ones from Table 4.6 indicate exponential complexity when not applying aggregation.

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1.16667	8	8	-	0.125
CENTER	UNI-1	1.625	24	16	100% (1/1)	0.132
	UNI-2	1.875	36	28	100% (18/18)	0.137
	UNI-3	2.21429	87	79	100% (54/54)	0.172
	UNI-20	15.3253	17877	17869	N/A	13.85
VERT	UNI-1	2.88889	136	128	100% (72/72)	0.178
	UNI-2	3.33333	224	232	N/A	0.233
	UNI-3	3.93651	640	632	N/A	0.512
	UNI-20	27.245	142960	142952	N/A	368.9

Table 4.6: Overview of the evaluation results of the rod reactor model. The utilized strategy is `noSampleAggregation` and the bad states are ${P_1}^{rr}$.

After exceeding 100 paths, the search tree gets too big to be depicted by the setup, on which the evaluation is executed. Such cases mostly occur in Tables 4.6 and 4.8. All benchmarks also have a time limit of 20 seconds and by exceeding it their execution is interrupted. For the case of applying `VERTICES` and `UNIFORM-20`, as depicted in Table 4.6, we let the benchmark terminate, which took over 6 minutes of computation time. The delivered result has the same accuracy as the one derived by the other heuristics which have smaller number of samples. However, its computation

requires exponentially more effort.

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1.16667	8	8	-	0.127
CENTER	UNI-1	1	12	4	100% (1/1)	0.124
	UNI-2	1	19	5	0% (0/1)	0.239
	UNI-3	1	12	4	100% (1/1)	0.128
	UNI-4	1	12	4	100% (1/1)	0.134
	UNI-20	1	12	4	100% (1/1)	0.165
VERT	UNI-1	1.77778	40	32	100% (8/8)	0.136
	UNI-2	1.77778	96	40	0% (0/8)	1.023
	UNI-3	1.77778	40	32	100% (8/8)	0.169
	UNI-4	1.77778	40	32	100% (8/8)	0.199
	UNI-20	1.77778	40	32	100% (8/8)	0.475

Table 4.7: Overview of the evaluation results of the rod reactor model. The utilized strategy is `sampleAggregation` and the bad states are ${P_2}^{rr}$.

An intriguing result emerges when observing the results in Table 4.7. The number of samples provided by `UNIFORM-1` and `-2` is not sufficient for choosing suitable samples. Although only one sample is delivered by `UNIFORM-1`, the heuristic succeeds at providing a counterexample run, whereas `UNIFORM-2` does not. This anomaly is due to the fact that the one sample from the succeeding heuristic is chosen at the exact center of the interval it is applied on, which is more likely to be at the nearest proximity of the guard than the other two samples. Such a result indicates that by choosing an even number of a few samples is less likely to find a CEX run and therefore the required computational time increases in order to show inconclusiveness.

Set sampling heuristic	Interval sampling heuristic	Average branching	Nr proc. tasks	Nr of nodes	Success rate (CEX/total)	Comput. time (s)
-	-	1.16667	8	8	-	0.127
CENTER	UNI-1	1.625	24	16	100% (9/9)	0.129
	UNI-2	1.5	82	46	0% (0/18)	1.041
	UNI-3	1.8125	199	124	16.7% (9/54)	2.346
	UNI-20	-	-	-	-	timeout
VERT	UNI-1	2.88889	136	128	100% (72/72)	0.169
	UNI-2	2.66667	600	368	N/A	7.413
	UNI-3	3.22222	1536	992	N/A	17.44
	UNI-20	-	-	-	-	timeout

Table 4.8: Overview of the evaluation results of the rod reactor model. The utilized strategy is `noSampleAggregation` and the bad states are ${P_2}^{rr}$.

Comparing Tables 4.7 and 4.8 shows that applying aggregation is useful when having more samples delivered by the heuristics. The mechanism reduces the required computational effort tremendously. On the other hand, when the number of provided samples is too small, the above discussed anomaly occurs and the results from the benchmark can be inconclusive.
Overall, the results from the evaluation of the model supports our expectation for reducing the computational effort from exponential to linear complexity by applying aggregation. However, the applied heuristics can lead to anomalies when having too few and an even number of samples.
Simulating single traces is computationally cheap in comparison to a full reachability analysis. As a conclusion from the evaluation results, we consider that the provided samples from the heuristic should be at least three with respect to our benchmarks. The optimal choice for a interval sampling heuristic is therefore `UNIFORM-3`, since only with few samples a big part of the system evolution is observed. In general, aggregating the samples is also a good idea, since the same solution is provided, but with less computational effort.

Chapter 5

Conclusion

5.1 Summary

In the course of this thesis we extended the tool HyDRA for proving unsafety of a given model against a defined set of bad states. In order to do so, we defined various heuristics for sampling the initial states set as well as for sampling the time interval in which a guard is enabled for taking a discrete transition. The applicability of the heuristics are evaluated on various models against multiple definitions of sets of bad states and under usage of different analysis strategies. From the results we derive that aggregating the provided samples is a powerful mechanism to reduce the computational complexity from exponential to linear with negligible loss of solution accuracy. Having too few samples can lead to anomalies, therefore the amount of utilized samples should be sufficiently large compared to the state space dimension.
The computational time required to provide a counterexample run is relatively small compared to the time required for a reachability analysis. Nevertheless, the simulation relies upon information computed during reachability analysis of a model and cannot be used by its own.

5.2 Future work

Throughout this thesis multiple discussions on various topics were raised which aimed at improving the utilized algorithms for the simulation worker. We questioned whether our general approach of forward reachability analysis is the optimal one to use, or we can combine it with backward analysis for faster retrieval of results [Mit07]. The employment of heuristics for spatial and temporal sampling leaves room for further suggestions and ideas alongside the ones which were proposed in this thesis, as well as room for improving and adjusting the existing ones.
We structured our ideas into different categories: improvement of the heuristics, improvement of the guard expansion implementation and handling, as well as more general ideas about utilizing backward analysis for hybrid systems.

5.2.1 Improvement and completeness of heuristics

The main motivation for choosing the heuristics which are implemented as part of this thesis are their advantage in computational simplicity and intuitiveness in the way

they work. From the overall six suggested heuristics only `RANDOM-K` set sampling heuristic is incompletely implemented in terms of the state set representations it can be applied on. In the course of this thesis the heuristic can be utilized for a box. For the sake of completeness the heuristic can be extended to sample all other representation methods which are present in HyPro.
Overall the implementation of additional set and interval sampling heuristics is encouraged. Their performance in terms of running time and quality of the provided results can be measured in order to choose an optimal heuristic either for one concrete system model or for a class of models.

5.2.2 Improvement of guard expansion implementation and handling

In order to enforce samples to satisfy their local guard condition, we proposed a mechanism which widens the guard in case it is an equation by representing it by two inequalities and and relaxing them by some predefined error ε. This mechanism limits the over-approximation error to a defined tolerance bound. However, the samples that satisfy the expanded guard are not necessarily a part of the exact model behavior. Such samples can be labeled for improvement potential. When applying backtracking and refining the derived timing information the labels can be utilized as an evidence that such sample can be refined to satisfy the actual guard condition.
The size of the ε-interval throughout this thesis is absolute. Although it can be manually adjusted to suit the dynamics of the model it is applied on, it is nevertheless a fixed constant. A suggestion to achieve flexibility when defining ε-interval is to define it relative to the guard and invariant constraints, or to the time step. By doing so, the user is abstracted from making the decision about the size of the ε-interval and no previous knowledge about the model is required.
Additionally, we propose a mechanism for convergence to the actual guard condition. Its objective is to iteratively get closer to the guard by measuring the distance between the sample and the guard. The input of the algorithm is the guard equation that is to be satisfied and an over-approximated interval, derived from a previous refinement setting. The output delivers a sample which satisfies the guard up to some error bound ε. At each iteration the approximation error is reduced, so that the output lies within a closer bound to reaching the exact solution. The algorithm is outlined as follows in Algorithm 2 and an example execution is depicted in Figure 5.1.
The advantage of this approach is that the derived interval of the enabled guard transition can be arbitrary over-approximated. Furthermore, the algorithm delivers an exact sample while maintaining a reasonably small complexity.

5.2.3 Backward and forward analysis for hybrid systems

An alternative approach to forward reachability analysis is backward analysis, where the safety verification of the model starts from the predefined set of bad states. The evolution of the bad states is computed backwards in time and if an intersection with the initial set is computed, the model is proven to be unsafe [Mit07]. Otherwise, if no intersection is calculated, the model is considered to be safe. Both approaches can be used separately or simultaneously in order to prove safety of the system under analysis. Starting off by sampling the set of bad states and computing the flow of the sample backwards in time, the aim of the simulation is to calculate a run into the

Algorithm 2 Get a sample which satisfies the guard

```
1:  function REFINESAMPLE(Interval i, Guard g)
2:      size p, q                          ▷ distance from upper respectively lower bound to guard
3:      sample s                                                              ▷ exact sample
4:      sample s_p := i.getLower()          ▷ set s_p at the lower bound of the interval
5:      sample s_q := i.getUpper()          ▷ set s_q at the upper bound of the interval
6:      while !s_p.satisfy(g) || !s_q.satisfy(g) do
7:          p := s_p.getDistanceTo(g)       ▷ measure distance from s_p to the guard
8:          q := s_q.getDistanceTo(g)       ▷ measure distance from s_q to the guard
9:          s_p := e^{p·δ}                                          ▷ compute spatial samples
10:         s_q := e^{q·δ}
11:     if s_p.satisfy(g) then                                      ▷ get the optimal sample s
12:         s := s_p
13:     else
14:         s := s_q
15:     return s
```

initial states.
In order to utilize a combination of both approaches, samples from the initial and the bad state sets have to be chosen. The flow of the initial sample is calculated forwards in time until the set of bad states or a time bound is reached, whereas the flow of the sample from the bad states is calculated backwards in time, until the initial states or a time bound is reached. The method of combining both approaches leaves room for defining further heuristics, which aim at proving safety of the analyzed hybrid system.

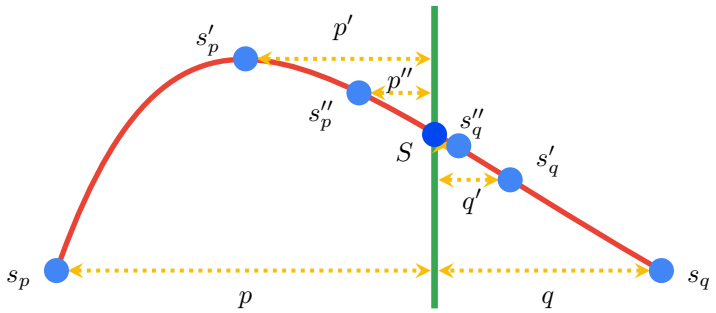


Figure 5.1: Flow of a system depicted as a red line for duration of the interval defined by $[s_p,s_q]$. The local guard is represented by a green line. By iteratively calculating the spatial samples with time step size according to their distance to the guard, a sample is derived which satisfies the guard. The respective distances are depicted as yellow dotted lines.

Bibliography

[ÁC15] Erika Ábrahám and Xin Chen. Modeling and analysis of hybrid systems: Lecture notes. In *Modeling and Analysis of Hybrid Systems*, Aachen, April 2015. Theory of Hybrid Systems, Informatik 2, Faculty of Mathematics, Computer Science, and Natural Sciences, RWTH Aachen University.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1): pages 3 – 34, 1995. Hybrid Systems.

[ÁCK+18] Erika Ábrahám, Xin Chen, Stefan Kowalewski, Ibtissem Ben Makhlouf, Stefan Schupp, and Sriram Sankaranarayanan. Hypro benchmarks. `https://ths.rwth-aachen.de/research/projects/hypro/benchmarks-of-continuous-and-hybrid-systems/`, 2018. [Online; accessed 10-May-2018].

[ÁCS12] Erika Ábrahám, Xin Chen, and Sriram Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 183–192, December 2012.

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *Hybrid Systems: Computation and Control*, pages 291–305, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1): pages 94 – 124, 1998.

[LG09] Colas Le Guernic. *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics*. Theses, Université Joseph-Fourier - Grenoble I, October 2009.

[LGG09] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In *Computer Aided Verification*, pages 540–554, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Mit07] Ian M. Mitchell. Comparing forward and backward reachability as tools for safety analysis. In *Hybrid Systems: Computation and Control*, pages 428–443, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[NÁGS16] Johannes Neuhaus, Erika Ábrahám, Jurgen Giesel, and Stefan Schupp. Develpoment of a modular approach for hybrid systems reachability analysis. `https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/theses/neuhaus_bachelor.pdf/`, 2016. [Online; accessed 14-May-2018].

[SÁ] Stefan Schupp and Erika Ábrahám. Efficient dynamic error reduction for hybrid systems rachability analysis. `https://ths.rwth-aachen.de/wp-content/uploads/sites/4/research/HyPro/presentation/poster.pdf`. [Online; accessed 24-July-2018].

[SÁ18a] Stefan Schupp and Erika Ábrahám. Efficient dynamic error reduction for hybrid systems reachability analysis. In *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*, LNCS. Springer, 2018.

[SÁ18b] Stefan Schupp and Erika Ábrahám. Spread the work: Multi-threaded safety analysis for hybrid systems. In *Software Engineering and Formal Methods*, pages 89–104, Cham, 2018. Springer International Publishing.

[SÁMK17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. Hypro: A C++ library of state set representations for hybrid systems reachability analysis. In *NASA Formal Methods*, pages 288–294, Cham, 2017. Springer International Publishing.

[SK03] Olaf Stursberg and Bruce H. Krogh. Efficient representation and computation of reachable sets for hybrid systems. In *Hybrid Systems: Computation and Control*, pages 482–497, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[War77] R. Ward. Numerical computation of the matrix exponential with accuracy estimate. *SIAM Journal on Numerical Analysis*, 14(4): pages 600–610, 1977.