hybr2d 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

ANALYSIS OF PROBABILISTIC HYBRID AUTOMATA IN HYPRO

Marvin Vogt

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

Additional Advisor:
Dr. Stefan Schupp

Aachen, 04.03.2021

Abstract

Hybrid systems combine discrete and continuous behavior to model for example systems that digitally control a physical process. Well-known applications of this are modern cars or industrial plants. Analysis methods for hybrid systems can be used to prove the correctness of such systems, usually to decide whether the system is able to reach a specific set of states. Some systems additionally exhibit random, probabilistic behavior, for instance unreliable transfer channels and sensors or actions that occur with a defined probability. In this case, probabilistic hybrid systems are required for modeling and the analysis task is to decide with which probability the system reaches a certain set of states.

In this work, first, the behavior of probabilistic hybrid systems is discussed before presenting two algorithm variants based on classic non-probabilistic hybrid analysis techniques. Additionally, an approach to unbounded analysis is developed. Finally, the functionality of all presented algorithms is demonstrated and compared to the existing tool ProHVer using a few existing case studies.

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.

Marvin Vogt
Aachen, den 04. März 2021

Acknowledgements

First, I would like to thank Prof. Dr. Erika Ábrahám for providing me with the opportunity to write this bachelor thesis and especially the fruitful discussions about probabilistic hybrid automata. Likewise, I want to thank the second examiner apl. Prof. Dr. Thomas Noll. Very helpful was my advisor Dr. Stefan Schupp, who regularly took the time to share his expertise, provide feedback on intermediate results and motivate me. Finally, I would like to thank Markus for listening to me and providing an outside perspective.

Contents

1 Introduction **9**

2 Preliminaries **11**
- 2.1 Mathematical Notations . . . 11
- 2.2 Hybrid Automata . . . 11
- 2.3 Markov Chains & Markov Decision Processes . . . 13
- 2.4 Probabilistic Hybrid Automata . . . 15
- 2.5 Reachability Analysis . . . 19
- 2.6 Search Trees . . . 20
- 2.7 Probabilistic Reachability Analysis . . . 21
- 2.8 Improving Approximations . . . 25
- 2.9 HyPro . . . 25

3 Algorithm and Implementation **27**
- 3.1 Simple Algorithm . . . 28
- 3.2 Improved, CEGAR-based Algorithm . . . 31
- 3.3 Improvements . . . 34

4 Experimental Results **37**
- 4.1 ProHVer: Bouncing Ball . . . 38
- 4.2 ProHVer: Water Level Control . . . 44
- 4.3 ProHVer: Autonomous Lawn-Mower . . . 44
- 4.4 ProHVer: Thermostat . . . 50
- 4.5 Refinement Demonstration . . . 50
- 4.6 Errors from Over-approximations . . . 53

5 Conclusion **57**
- 5.1 Future Work . . . 58

Bibliography **61**

A Extended Automaton Grammar for HyPro **63**

Chapter 1

Introduction

The field of safety analysis is an important part of automata theory, it is beneficial to know if a modeled system reaches an undesirable state. Imagine an autonomous car, if the distance to the car driving in front becomes too short an accident could happen. Therefore, it is undesirable to get to a state where that distance is less than a set minimum. This system could be modeled as an automaton, then the problem can be formalized as deciding whether a given state is reachable inside that model. For many types of models, this problem has already been solved. For many other types there exist various approximations. The goal of this work is to explore and extend reachability and thereby also safety analysis to the model of probabilistic hybrid automata.

One of the simpler classes of automata models are discrete systems. They are usually represented by a set of discrete states and the system changes between them according to the rules of the specific models. It is important to note that the system will always be in exactly one state and will change instantly between states. In the car examples one could for example differentiate between states *accelerating* and *braking* [AS20b].

For many systems and applications that abstract model may be enough, but almost any action in the real world employs continuous behavior. The car doesn't jump from position 1 to position 2, it gradually advances from 1 to 2 and for any position between 1 and 2, there exists a time point when the car is at that position. The velocity can be characterized similarly, when accelerating, the car won't instantly transition from $0\,\mathrm{m\,s^{-1}}$ to $100\,\mathrm{m\,s^{-1}}$. It will continuously increase its velocity until it reaches the targeted speed. Even time evolves continuously, there are no discrete time steps. All these examples can be summarized as continuous evolution of the system state (over time) [AS20b].

Therefore, almost any system that interfaces with the real world has to deal with continuous behavior. For some systems it may not be important, e.g. it does not matter how long the interior light of the car takes to activate as long as it does eventually. In contrast, the aforementioned position and velocity are safety-critical and a system controlling the throttle and brake should use a continuous model for this. The combination of these two modeling classes is called a hybrid system. A hybrid system uses instantaneous state changes like in discrete systems to model different kinds of changes in control, e.g. switching from accelerating to braking, while also considering the effect of these states as the continuous state evolutions. This idea is illustrated in Figure 1.1 [AS20b].

Until now, it was assumed that every state change happens either deterministically or non-deterministically, there is no way to specify that when switching from braking to accelerating in 0.0001 % of all cases the motor dies. An unintentionally stopped motor is an undesired

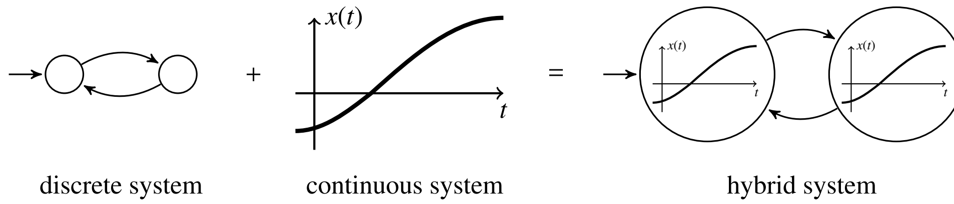


Figure 1.1: Intuitive illustration about the combination of discrete and continuous systems to form hybrid systems. This illustration is reproduced from [AS20a].

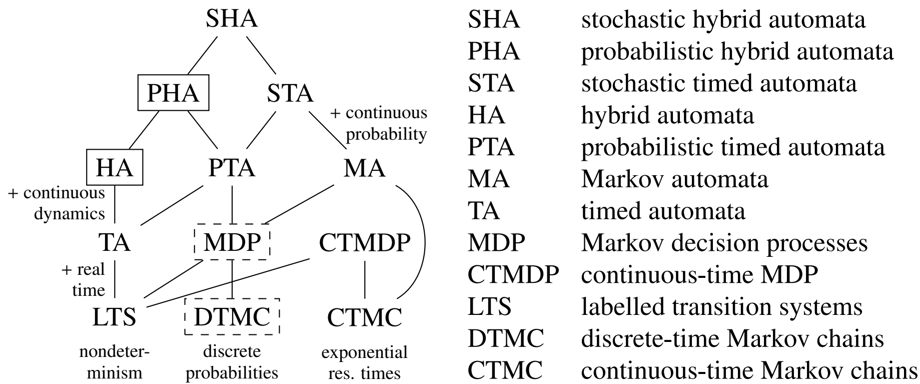


Figure 1.2: The family tree of automata-based quantitative formalisms by [HHH+19]. All for this work relevant automata models are highlighted.

state, but what is the probability of reaching that state? The extension of hybrid systems with (discrete) probabilities yields a probabilistic hybrid system, the corresponding automata model is called a probabilistic hybrid automaton. [meh, rework, this section is especially bad]

The context of (probabilistic) hybrid automata in the complete field of formal automata models is visualized in Figure 1.2.

Related Work One of the first works on probabilistic hybrid automata is by Jeremy Sproston in [Spr00], where the concept of probabilistic hybrid automata was introduced. It also presented analysis techniques for some subclasses of probabilistic hybrid automata. Based on this, [ZSR+10] developed an approach to analyze general probabilistic hybrid automata and compute an upper bound. Additionally, the authors implemented their algorithm in the tool ProHVer which will be used for comparison. The same tool was later extended to stochastic hybrid automata by [FHH+11].

Chapter 2

Preliminaries

This chapter serves as the basis for the remainder of this thesis. All relevant basic definitions and concepts will be introduced.

2.1 Mathematical Notations

Some mathematical constructs and symbols are used with varying meanings, all ambiguous symbols or concepts with several notations, which are used throughout this work, are defined in this section.

Definition 2.1.1 (Powerset ([Tho18]))**.** *For a set X, the powerset of X is defined by* $\mathcal{P}(X) := \{S \mid S \subseteq X\}$.

Definition 2.1.2 (Trivial intervals)**.** *An interval* $I \subseteq X$ *for a set X is called trivial if and only if* $|I| = 1$. *Additionally, it holds that a closed interval* $I = [l, u]$ *is trivial if and only if* $l = u$.

Consequently, an interval I is non-trivial if it contains $x, y \in I$ *such that* $x \neq y$.

2.2 Hybrid Automata

The automata model used to model hybrid systems is usually called a *hybrid automaton*. These terms will be used interchangeably where the exact difference is not important. For the purpose of this work, a composition of several classic definitions is used. Furthermore, the option for composition of two or more hybrid automata is disregarded, as it is not relevant here and the given definition can easily be extended to cover it. For this reason, synchronization labels are omitted as well.

Definition 2.2.1 (Syntax for Hybrid Automata ([ACH+95, AS20b]))**.** *A Hybrid Automaton (HA) can be formally defined by a 6-tuple* $\mathcal{H} = (Loc, Var, Edge, Act, Inv, Init)$ *where each component has the following meaning.*

- *Loc is a finite set of* ***locations****.*
- *Var is a finite set of real-valued* ***variables****.*
  A ***valuation*** $v : Var \to \mathbb{R}$ *assigns a real-value* $v(x)$ *to each variable* $x \in Var$. *The set of all valuation is called* V.
  A ***state*** *is a location-valuation-pair* $(l, v) \in Loc \times V$. *The set of all states is called* Σ.

- *$Edge \subseteq Loc \times \mathcal{P}(V^2) \times Loc$ is a finite set of edges called* ***transitions****. A transition $(l_1, \mu, l_2) \in Edge$ has a source location l_1 and a target location l_2. Each pair of valuations $(v_1, v_2) \in \mu$ defines a possible valuation v_1 for which this transition can be taken. After taking the transition the current valuation is set to v_2.*
- *$Act : Loc \to \mathcal{P}(\mathbb{R}^{\geq 0} \to V)$ is a function that assigns to each location $l \in Loc$ a set of* ***activities*** *$Act(l)$.*
  Every activity function, also called ***flow****, $f : \mathbb{R}^{\geq 0} \to V$ maps a non-negative real to a valuation.*
  The set of activities $Act(l)$ has to be time-invariant, for every activity $f \in Act(l)$ and $t \in \mathbb{R}^{\geq 0}$, $(f + t) \in Act(l)$ holds, where $(f + t)(t') = f(t + t')$ for all $t' \in \mathbb{R}^{\geq 0}$.
- *$Inv : Loc \to \mathcal{P}(V)$ is a function that assigns to each location $l \in Loc$ a set of valuations $Inv(l)$. This set is called the* ***invariant*** *of location l.*
- *$Init \subseteq \Sigma$ is a set of* ***initial states****.*

A hybrid automaton models discrete as well as continuous behavior, appropriately there are two types of steps linking two states together. First, the discrete-steps $\xrightarrow{e}$, these correspond to the transitions introduced earlier and change the location and valuation of a given state. A discrete-step executing a transition may also be called a *jump*. The second type of steps are time-steps $\xrightarrow{t}$ modeling the continuous behavior. During a time-step, the automaton stays a non-negative amount of time inside one location and the valuation changes according to the flow of that location. The union of both discrete- and time-steps is denoted by $\to$. This informal description of the semantics of a hybrid automaton can be formalized through operational semantics [AS20b, ACH+95].

Definition 2.2.2 (Semantics for Hybrid Automata ([ACH+95, AS20b]))**.** *Given a hybrid automaton $\mathcal{H} = (Loc, Var, Edge, Act, Inv, Init)$, the semantics may be defined based on a (usually infinite) transition system, where each state of $\mathcal{H}$ corresponds to a single state in the transition system. Every step and therefore state-change of $\mathcal{H}$ is mapped to a transition in the transition system.*

- *For a discrete-step $\xrightarrow{e}$ this is*

$$\frac{e = (l_1, \mu, l_2) \in Edge \quad (v_1, v_2) \in \mu \quad v_2 \in Inv(l_2)}{(l_1, v_1) \xrightarrow{e} (l_2, v_2)}$$

- *and for a continuous time-step $\xrightarrow{t}$ with $t \in \mathbb{R}^{\geq 0}$*

$$\frac{f \in Act(l) \quad f(0) = v_1 \quad f(t) = v_2 \quad t \geq 0 \quad \forall 0 \leq t' \leq t. f(t') \in Inv(l)}{(l, v_1) \xrightarrow{t} (l, v_2)}.$$

The concept of single steps linking pairs of states can be extended to several steps linking several states. A (non-empty) finite or infinite sequence of this kind $\sigma_0 \to \sigma_1 \to \sigma_2 \to \dots$ is called a *run* if it starts in an initial state, that is if $\sigma_0 = (l_0, v_0) \in Init$ and $v \in Inv(l_0)$. A state is called *reachable* if there exists a run leading to that state. All initial states are trivially reachable by a run consisting of only that initial state if the valuation of that state satisfies the invariant of the corresponding location [AS20b].

A transition is called *enabled* for a given state if it is possible to take a discrete-step using that transition.

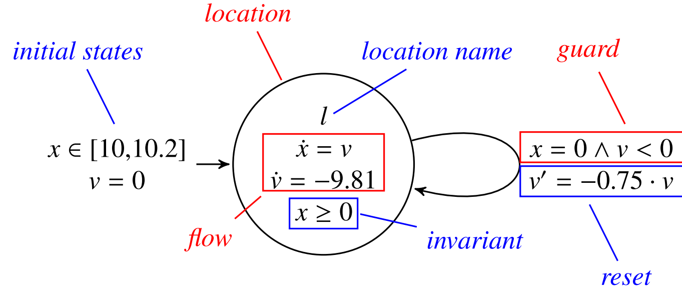


Figure 2.1: The hybrid automaton for Example 2.2.1 (Bouncing Ball). This automaton simultaneously serves as a reference for graphical hybrid automata specification.

A hybrid automaton $\mathcal{H} = (Loc, Var, Edge, Act, Inv, Init)$ specified using Definition 2.2.1 and 2.2.2 may include the use of infinite sets for transitions, activities, invariants and initial states. Each transition in the *Edge*-set may have an infinite amount of valuation-pairs, for a location $l \in Loc$ the set of activity functions $Act(l)$ may be infinite and while the set of locations *Loc* is finite, the initial state set *Init* takes the initial valuation into account, thus $\mathcal{H}$ may have an infinite amount of initial states. For this reason, alternative specification methods are commonly used. A transition $(l, \mu, l_2) \in Edge$ can be represented using a guard and reset function. The set $\{v_1 \mid (v_1, v_2) \in \mu\}$ needs to be satisfied by a valuation in order for the transition to be enabled and is called a *guard*. The *reset* is a mapping of each valuation v_1 satisfying the guard to a set of possible valuations $\{v_2 \mid (v_1, v_2) \in \mu\}$ after taking the transition. During execution one valuation is chosen from this set non-deterministically. Activities describe how valuations in a location evolve when time passes and are usually expressed by ordinary differential equations. Invariants and initial states both define conditions, which can be similarly specified as guards for transitions [AS20b].

Example 2.2.1 (Bouncing Ball ([Sch19])). *One of the most widespread examples is the bouncing ball automaton. It models a simplified physical process where a ball is dropped from a defined height (here* 10 m – 10.2 m*). When the ball is released, it accelerates due to gravity (here using the conventional standard for gravitational acceleration on Earth which is approximately* $9.81\,\mathrm{m\,s^{-2}}$*). On impact with the ground (height* 0 m*), the ball loses some kinetic energy in sound and heat. For this example, it is assumed that the ball bounces up with* 75 % *of the velocity it had on impact with the ground. All other physical effects are neglected.*

This behavior can be modeled like in Figure 2.1 using a single location and a single transition with the same source and target location. This example also serves as an introduction to the graphical notation of hybrid automata.

2.3 Markov Chains & Markov Decision Processes

Markov chains (MC) are the result of extending transitions of transition systems with probabilistic distributions. In a transition system, each transition has a defined target state. By comparison, in a MC the target state is chosen from a probability distribution over states. That probability distribution may be defined as in Definition 2.3.1.

Definition 2.3.1 (Probability Distribution ([Kat20, Sto02]))**.** *For a set X, a probability distribution over X is a function* $\mu : X \to [0,1]$ *with* $\sum_{x \in X} \mu(x) = 1$.
The set of all probability distributions over X is called $Distr(X)$.

Given a (possibly infinite) set X, the support $sup(\mu)$ *of a probability distribution* μ *is the set of all elements of X with a probability in* μ *strictly larger 0. That is* $sup(\mu) = \{x \in X \mid \mu(x) > 0\}$. *All relevant probability distributions will have finite support.*

The Markov chain model can easily be defined analogously to classic transition systems using probability distributions.

Definition 2.3.2 (Markov Chains ([Kam19, Kat20]))**.** *A (Discrete-Time) Markov Chain (MC) is a 3-tuple* $\mathcal{D} = (\Sigma, \sigma_0, P)$ *where each component has the following meaning.*

- Σ *is a countable set of* ***states***.
- $\sigma_0 \in \Sigma$ *is an* ***initial state***.
- $P : \Sigma \to Distr(\Sigma)$ *is a* ***transition probability function***. *Each state is mapped to a probability distribution over all states* Σ.

The behavior of a MC is equivalent to the behavior of a transition system with one small difference in transition logic. If a MC $\mathcal{D}$ is in state $\sigma \in \Sigma$, $\mathcal{D}$ will probabilistically choose a state from the distribution $P(\sigma)$ and advance to that state [Kam19, Kat20].

As for transition systems, a run π of MC $\mathcal{D} = (\Sigma, \sigma_0, P)$ is a (possibly infinite) sequence of states $\sigma_0, \sigma_1, \sigma_2, \ldots$, with $\sigma_i \in \Sigma$, such that it starts with the initial state and the transition function holds for every step, i.e. for any $k > 0$ in run π, let $\mu := P(\sigma_{k-1})$ denote the probability distribution for state σ_{k-1}. Then $\mu(\sigma_k)$ should have a value, in this case probability, larger than 0. In summary, $P(\sigma_{k-1})(\sigma_k) > 0$ should hold for all succeeding states σ_{k-1}, σ_k with $k > 0$ in run π [Kam19, Kat20]. Each run has an assigned probability consisting of the probability of the taken transitions. The probability $P(\pi)$ of an (again possibly infinite) run $\pi : \sigma_0, \sigma_1, \ldots$ can be computed using

$$P(\pi) = \prod_{k=1} P(\sigma_{k-1})(\sigma_k).$$

The probability $P(G)$ of $\mathcal{D}$ eventually reaching a set of goal states $G \subseteq \Sigma$ is now the sum of all probabilities of all paths reaching G for the first time [Kam19, Kat20], where * follows Kleene-star [Gro18] notation

$$P(G) = \sum_{\pi \text{ is a path over } (\Sigma \setminus G)^* G} P(\pi).$$

There exist different interpretations of probabilistic behavior in the context of non-determinism. This work will consider probabilistic behavior and non-deterministic behavior two strictly different effects.

Hybrid automata include non-determinism by design. Therefore, it would be beneficial to already include non-determinism in Markov chains. The next definition introduces a generalization of Markov chains, afterward, the semantics is investigated. In contrast to the usual definitions by e.g. [Kam19], this definition doesn't require each action to be present in each state.

Definition 2.3.3 (Markov Decision Processes (based on [Kam19, Spr00, HKHH13]))**.** *A Markov Decision Process (MDP) is a 4-tuple* $\mathcal{D} = (\Sigma, \sigma_0, A, P)$ *where* Σ *and* σ_0 *are defined as for Markov chains. A is a set of* ***actions*** *and the* ***transition probability function*** $P : \Sigma \to \mathcal{P}(A \times Distr(\Sigma))$ *is augmented with the action set A.*

The behavior is similar to the behavior of a Markov chain, but if MDP $\mathcal{D}$ is in state $\sigma \in \Sigma$, $\mathcal{D}$ will first non-deterministically choose an action and distribution from $P(\sigma)$. Only afterward $\mathcal{D}$ will probabilistically choose a state from the chosen distribution and advance to that state [Kam19, Spr00].

The concept of a run is the same as for MCs while adapting the different transition probability function. But the defined probability for a path requires reconsideration.

Consider a MDP $\mathcal{D} = (\Sigma, \sigma_0, A, P)$. Each state can have multiple pairs of actions and probability distributions over states. Let $\sigma \in \Sigma$ be such a situation where $P(\sigma) = \{(a_1, \mu_1), \ldots, (a_n, \mu_n)\}$ with $n > 1$. There are at least these options to resolve this non-determinism:

- One option is using a pre-distribution. If there are $|P(\sigma)| = n$ actions and distributions over states, choose each pair with probability $1/n$.

- Alternatively, employ a scheduler, which chooses an action-distribution-pair ((a_i, μ_i)) based on defined criteria. For example according to a priority list, where each pair has a unique priority [AS20b].

- If the goal is to compute minimal and maximal bounds of reaching an accepting (or unsafe) state for a given input word, there is another option. Evaluate each pair separately and combine the resulting intervals.

Each option has its own uses for different applications. For now, schedulers are introduced.

Definition 2.3.4 (Schedulers ([Kam19, AS20b, HKHH13])**.** *Given a Markov decision process $\mathcal{D} = (\Sigma, \sigma_0, A, P)$, a scheduler is a function mapping each path of $\mathcal{D}$ to an action-distribution-pair available at the last state of the path. That is, for a path $\pi : \sigma_0, \ldots, \sigma_n$, a scheduler will select an element from $P(\sigma_n)$ based on σ_n and possibly all states leading up to σ_n.*

Each scheduler for $\mathcal{D}$ induces a Markov chain $\mathcal{D}'$, where each non-deterministic choice has been resolved by that scheduler.

There are several sub-classes of schedulers, one noteworthy example is the *memoryless* or *history-independent* scheduler which will choose an action-distribution-pair based solely on the last state σ_n of the path [Kam19, AS20b].

In general, the probabilities for a MDP are therefore only well-defined in the context of a scheduler and the behavior is equal to the behavior of the scheduler-induced Markov chain.

2.4 Probabilistic Hybrid Automata

While a hybrid automaton combines discrete and continuous behavior, a probabilistic hybrid automaton augments this with additional probabilistic behavior similar to MDPs. Probabilistic hybrid automata are effectively hybrid automata, where the discrete steps derive properties of transitions of a MDP. The presented definition differs to [Spr00] and [HKHH13] mainly in the declaration of the probabilistic transition, here a combined version which is closer to the style of the definition for hybrid automata (Definition 2.2.1) is used.

Definition 2.4.1 (Syntax for Probabilistic Hybrid Automata (based on [Spr00, HKHH13])). *A Probabilistic Hybrid Automaton (PHA) is a 6-tuple* $\mathcal{H} = (Loc, Var, Prob, Act, Inv, Init)$ *where Loc, Var, Act, Inv and Init are defined as for hybrid automata. The finite transition set Edge is replaced by a finite* ***probabilistic transition*** *set*

$$Prob \subseteq Loc \times \mathcal{P}(V) \times Distr((V \to \mathcal{P}(V)) \times Loc \times Idx).$$

An additional restriction applies to probabilistic transition set Prob. For all $(l_1, G, \mu) \in Prob$*, for all* $(r, l_2, i) \in sup(\mu)$ *and for all* $v_1 \in G$ *it has to hold that* $r(v_1) \subseteq Inv(l_2)$*.*

Each transition-distribution $(l_1, G, \mu) \in Prob$ has a source location l_1. G is the guard set. For a state $(l, v) \in \Sigma$, the transition can be taken if $l_1 = l$ and $v \in G$. μ is a probability distribution over a triple (r, l_2, i) consisting of a valuation mapping r, a target location l_2 and an index i from an index set $Idx \subset \mathbb{N}$. It is assumed that each triple has a unique index $i \in Idx$, where the support $sup(\mu)$ of each μ is finite. Furthermore, it holds that $\sum_{(l_1,G,\mu)\in Prob} |sup(\mu)| = |Idx|$. Based on the current system valuation, the valuation mapping r describes a set of possible target valuations of which one valuation is non-deterministically chosen when this transition is taken. The triple $(r, l_2, i) \in sup(\mu)$ specifies a single transition inside the transition-distribution. To be more explicit, the quintuple (l_1, G, r, l_2, i) may also be used when describing a single transition option of $\mathcal{H}$.

The restriction from the definition can now be described as follows. It is required that for a transition-distribution $(l_1, G, \mu) \in Prob$ and a target location l_2 all valuations $v_2 \in r(v_1)$, that may occur after taking a transition to l_2, satisfy the invariant of the target location l_2, that is $v_2 \in Inv(l_2)$ for all $v_2 \in \{v_2' \mid v_2' \in r(v_1), (r, l_2, i) \in sup(\mu), v_1 \in G\} \subseteq V$. The reason for this constraint is motivated after the semantics have been introduced.

The semantics for probabilistic hybrid automata can be defined similarly as an extension of the semantics of hybrid automata (Definition 2.2.2).

Definition 2.4.2 (Semantics for Probabilistic Hybrid Automata (based on [Spr00, HKHH13])). *Given a probabilistic hybrid automaton* $\mathcal{H} = (Loc, Var, Prob, Act, Inv, Init)$*, the semantics for a continuous time-step* $\xrightarrow{t}$ *with* $t \in \mathbb{R}^{\geq 0}$ *is similar to the semantics for continuous time-steps of hybrid automata. The only difference is that a Markov decision process is used instead of a transition system as the underlying model. Each continuous time-step* $\xrightarrow{t}$ *for a timestep* $t \in \mathbb{R}^{\geq 0}$ *becomes a transition* $\xrightarrow[1]{t}$ *with probability 1 and the timestep t as the action.*

The semantics for a discrete-step $\xrightarrow[p]{e}$ *with probability p can then be defined like this*

$$\frac{\begin{array}{l} e = (l_1, G, \mu) \in Prob \quad v_1 \in G \\ T = \{(r, l_2, i) \mid r \in V \to \mathcal{P}(V), v_2 \in r(v_1), (r, l_2, i) \in sup(\mu), i \in Idx\} \\ I = \{i \mid (r, l_2, i) \in T\} \quad p = \sum_{(r,l_2,i)\in T} \mu(r, l_2, i) \quad p > 0 \quad v_2 \in Inv(l_2) \end{array}}{(l_1, v_1) \xrightarrow[p]{e,I} (l_2, v_2)}.$$

The set I in the rule for discrete-steps in the above definition is the subset of the index set that is relevant for moving from state (l_1, v_1) to state (l_2, v_2) using transition-distribution e. The probability of all involved transitions, that is all transitions whose index is in I, is summed up to form a single transition in the resulting MDP. The action of the transition in the MDP is the transition-distribution e. Additionally, each transition is annotated with the relevant index subset I for reference.

The concepts of runs and reachable states are the same as for HA. A transition-distribution is enabled if it is possible to take a discrete-step using a transition from that transition-distribution.

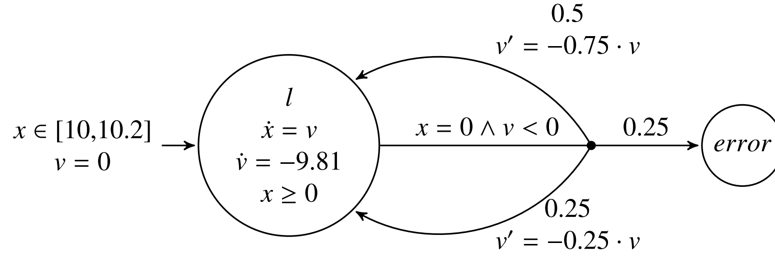


Figure 2.2: The probabilistic hybrid automaton for Example 2.4.1 (Probabilistic Bouncing Ball).

Example 2.4.1 (Probabilistic Bouncing Ball ([Sch19, ZSR+10])). *The probabilistic bouncing ball automaton as shown in Figure 2.2 is an extension of the bouncing ball automaton from Example 2.2.1 for a probabilistic setting. The main setting remains unchanged, only the ball is made up of different sections with different material properties. In this case,* 50 % *of the surface is the same material as before (with* 75 % *velocity conservation),* 25 % *is a softer material with only* 25 % *velocity conservation and the remaining* 25 % *is an even softer material which fully absorbs the impact with the ground. The side with which the ball hits the ground is assumed to be completely random and only dependent on the amount of surface covered with that material.*

Similar to HA, transitions for probabilistic hybrid automata can also be specified using guards and resets. While the guard has to be equal for all transitions in a distribution, the reset may be unique for each transition. This notation can be used to explain the remark at the end of Definition 2.4.1. Formally, for each transition in a transition-distribution, it is required that each valuation that satisfies the guard of the distribution together with the invariant of the source location l_1 satisfies the invariant of the target location l_2 of that transition after being transformed according to the reset of the transition. The following equation summarizes this statement.

$$v \text{ satisfies } Inv(l_1) \wedge guard \implies reset(v) \text{ satisfies } Inv(l_2)$$

Technically, this condition is already enforced by Definition 2.4.2 for each discrete step. Still, without this condition the probability space [AS20b] for distributions would not necessarily be well-defined anymore. Consider Example 2.4.1, but add an additional invariant $v \leq -5$ to location *error*. This invariant will only be satisfied for the first few bounces. For later bounces this transition to location *error* therefore becomes impossible and only the two transitions back to location l can be taken. With a probability of 50 % the transition for the harder material and with a probability of 25 % the transition for the softer material is used. The behavior in case of the remaining 25 % is not specified. If the satisfaction of the invariant is already enforced through the guard and reset (here not present or rather the identity), this case can not occur and all transitions of a distribution are enabled if the guard of the transition-distribution is satisfied.

Since the semantics are defined in Definition 2.4.2 based on MDPs, the problem of MDPs with non-determinism directly carries over to PHAs. Intuitively, non-determinism in PHAs can be classified in discrete and continuous non-determinism as displayed in Table 2.1.

Some of these sources of non-determinism may be mitigated by clever modeling. While this might not necessarily reduce the expressivity of the modeling class of PHA, it would restrict the modeling options and is therefore not further considered.

Table 2.1: Sources of non-determinism in PHAs and their type of non-determinism. DND is short for discrete non-determinism and CND for continuous non-determinism.

DND	CND	Source of non-determinism
✓		Several transition-distributions are enabled in the same state.
	✓	For a state at least one transition-distribution is enabled and time-evolution is possible.
	✓	A transition has a non-deterministic reset.
✓		Several initial states in different locations have a valuation satisfying the corresponding invariant.
	✓	Several initial states in the same location have a valuation satisfying the invariant.

For now the behavior of a PHA $\mathcal{H} = (Loc, Var, Prob, Act, Inv, Init)$ in a state $(l, v) \in \Sigma$ taking a transition is considered to be

1. non-deterministically choose a transition-distribution μ from the set of enabled transition-distributions for this state,
2. probabilistically decide on one transition from the chosen transition-distribution,
3. non-deterministically choose a reset from the reset set for this transition.

In the beginning of each run an initial state $(l,v) \in Init$ with $v \in Inv(l)$ is non-deterministically chosen from the set of initial states. This covers both discrete and continuous non-determinism in initial states.

As before, a scheduler as introduced in Definition 2.3.4 is used to resolve all non-deterministic choices [Spr00, AS20b]. While the schedulers used here are defined on MDPs, it would not be a big difference to first apply a scheduler to the PHA and afterward map the non-determinism-free automaton to the corresponding MC. The non-determinism-free PHA only has a single initial state and for every reachable state in the automaton at most a single transition is enabled as well as never both a transition is enabled and time-evolution is possible.

Using a scheduler, each run on PHAs is only dependent on probabilistic choices, therefore the probability of a path for a specific scheduler can be computed as for MCs. The same holds for the probability of reaching a set of goal states using a specific scheduler.

Each probabilistic hybrid automaton induces a hybrid automaton by replacing all instances of probabilistic choices with non-deterministic ones. The set of reachable states stays the same, only the probability of reaching a specific state is unknown.

Definition 2.4.3 (Induced Hybrid Automaton of a PHA ([ZSR+10])). *Given a probabilistic hybrid automaton $\mathcal{H} = (Loc, Var, Prob, Act, Inv, Init)$, it induces a hybrid automaton $ind(\mathcal{H}) = (Loc, Var, Edge, Act, Inv, Init)$, where for each element $(l_1, G, \mu) \in Prob$ and $(r, l_2, i) \in sup(\mu)$ the set $S := \{(v_1, v_2) \mid v_1 \in G, v_2 \in r(v_1)\}$ is computed and the transition $(l_1, S, l_2) \in Loc \times \mathcal{P}(V^2) \times Loc$ is added to Edge.*

2.5 Reachability Analysis

Until now, the focus has been on the automata models themselves. The actual reason why these automata models are relevant has only briefly been described. The next sections aim to resolve this by introducing reachability analysis as it is already known for hybrid automata and afterward adapting it to probabilistic hybrid automata.

During the motivation in the introduction, it was already established that is interesting to know whether a system can reach a specific set of states. This is usually a set of undesired states which should never be reached under any circumstances. Usually these states are therefore called *unsafe* or *bad states*. Going back to the non-probabilistic bouncing ball example (Example 2.2.1), a possible set of unsafe states could be every state with a height between 4 m and 5 m and a velocity between $-1\,\mathrm{m\,s^{-1}}$ and $1\,\mathrm{m\,s^{-1}}$, since it could be known that if the ball reaches that configuration, it is going to break a nearby flower pot. Now there are two important questions.

- Will the ball never reach that configuration, that is, is the flower pot safe?
- Will the ball reach that configuration, that is, is the flower pot going to break?

In the first case one tries to prove that the system will never reach the defined states to show the system's safety. For the second case one tries to prove that the system will reach a defined state set. While these two questions are opposite to each other, each one can be shown differently. The focus of this work is proving the safety of a system, or more specifically the safety of a probabilistic hybrid system.

One widely used approach is forward reachability. It is summarized in Algorithm 1. The main idea is to compute every state reachable from an initial state. For this, the corresponding algorithm starts with the initial state set and computes all states reachable in either a discrete step or a continuous step. This computation is repeated for every state that was not already handled before. If the computation yields no new state, the algorithm had computed all reachable states.

A system is safe if no unsafe state is reachable, in this case, if the returned set of all reachable states and the set of unsafe states are intersection-free.

Input: Set of initial states *Init*.
Output: Set of reachable states.
$R^{\text{new}} := \mathit{Init}$
$R := \emptyset$
while $R^{\text{new}} \neq \emptyset$ **do**
$R := R \cup R^{\text{new}}$
$R := \textsc{Reach}(R^{\text{new}}) \setminus R$
end while
return R

Algorithm 1: General forward reachability algorithm ([AS20b]).

In general this algorithm is undecidable for hybrid systems [ACH+95] due to function Reach. To still show the safety of a hybrid system, over-approximations and bounds on the number of iterations are usually used. If the over-approximation is safe, that is the over-approximation of all reachable states does not include an unsafe state, the actual set of all reachable states is safe as well. The decision of how coarse the over-approximation can be

while still providing useful results and the actual representation of a state set is very complex and not that relevant for this work, for more detailed information we refer to [Sch19].

The number of iterations is usually limited through a *local time-horizons* and a *maximum jump-depth*. The local time-horizon restricts automata which allow for an infinite amount of time to pass in a single location. It limits the amount of time that may pass without taking a discrete jump and should be chosen to capture all relevant behavior of the automaton. Similarly, automata with runs with an infinite number of jumps exist. This will be addressed by the maximum jump-depth. Again, this limit should be chosen in a way to still capture all relevant behavior of the automaton. Analysis will only consider states reachable under both limits. The interpretation of reachability analysis thus becomes the question of reaching an unsafe state within the local time-horizon and the maximum jump-depth. If no unsafe state is found to be reachable within the bounds, then this does not necessarily imply the safety of the whole system. A reachable unsafe state may still exist beyond the specified bounds.

Before this approach is extended for the probabilistic setting, it makes sense to take a quick look at how all computed state sets can be organized. While the relationship between two different state sets is not important for reachability in HA, it aids the computation to know which state set can be reached by which state set. This also allows to quickly generate counterexample-candidates in case of a safety violation by following the path through all state sets that leads to the state set with an unsafe intersection. For PHA, the relationship between two state sets is intuitively important as the probability of reaching a state is directly dependent on the path to that state. Therefore, in the next section, one approach to model such a relationship during computation is described. Afterward, in Section 2.7, reachability analysis for PHAs is covered.

2.6 Search Trees

As already motivated at the end of the previous section, this section aims to cover how all reachable states can be ordered and managed in a structured way during the computation of all reachable states. It is assumed that there exists a selection of different ways of representing a (usually convex) set of states, for further information on state set representations we refer to e.g. [Sch19]. The approach summarized here is called *flowpipe-construction-based reachability analysis* and is described in more detail in Section 2.9.

A state set in this setting contains all states reachable only through continuous time-evolution from a start state set. Every discrete jump will lead to a new start state set from which a new state set is constructed through time-evolution. When the initial state set of a hybrid automaton is used for the start state set of the first state set, this directly induces a kind of search tree, where each node is associated with a set of states and all states reachable by time-evolution. Each edge between two nodes models a discrete jump. For this work, only a smaller less complete definition is given in Definition 2.6.1, for a more complete definition we refer to [Sch19]. It is important to emphasize that the search tree will satisfy the tree property, that is, even if the same state set is reachable by two different paths, the nodes corresponding to those state sets will be separate and not associated with each other.

Definition 2.6.1 (Search Tree ([Sch19])).
Given a hybrid automaton $\mathcal{H} = (Loc, Var, Edge, Act, Inv, Init)$, a search tree for this automaton is a 5-tuple $S = (Nodes, Root, Succ, State, Trace)$ where each component has the following meaning.

- *$Nodes$ is a finite set of **nodes**.*
- *$Root \in Nodes$ is a **root node** for search tree S.*
- *$Succ \subseteq Nodes \times Nodes$ is a **node relation**.*
- *$State : Nodes \to (Loc, \mathcal{P}(\mathbb{R}^{|Var|}))$ is a function assigning the represented state set $State(n)$ to each node $n \in Nodes$.*
- *$Trace : Succ \to Edge$ is a function that assigns to each edge $e = (n_1, n_2) \in Succ$ the transition that is used to get from node n_1 to node n_2, or more specifically to get from a state subset of $State(n_1)$ to the state set $State(n_2)$.*

For a search tree $S = (Nodes, Root, Succ, State, Trace)$ and a node $n \in Nodes$, the set $\{n' \in Nodes \mid (n,n') \in Succ\}$ is the set of all *successors* of node n and similarly the set $\{n' \in Nodes \mid (n',n) \in Succ\}$ is the set of all *predecessors* of node n. Furthermore, search tree S has to satisfy the tree property, that is, there should exist no node which has more than one distinct predecessor. Similarly, S can not contain any cycles and the root node $Root$ is the only node which has no predecessor. Without further explanation it is assumed that each node can store arbitrary attributes used in algorithms.

While this defines search trees exclusively for hybrid automata, nothing is preventing it from being lifted to probabilistic hybrid automata. From now on search trees defined using Definition 2.6.1 will be used for both non-probabilistic hybrid automata and probabilistic hybrid automata.

Applying the search tree concept to the general forward reachability algorithm (Algorithm 1), the algorithm can be rewritten in a recursive way like in Algorithm 2. While this is not the most intuitive form, it allows for an easier extension to the probabilistic case. The main idea is to call the function ANALYZE once for each node. It will handle all possible continuous time-steps (REACHTIME) as well as find all discrete jumps leading to successors of this node (REACHDISCRETE). Previously both of these functions were combined in function REACH. After completion, it has constructed the complete search tree for the given automaton.

This modified algorithm now allows a simple extension to support probabilistic hybrid automata.

2.7 Probabilistic Reachability Analysis

Reachability or more specifically safety analysis for hybrid automata answers whether a set of unsafe states can be reached. Assuming this question would be fully decidable, this would lead to a binary outcome, either the state set can be reached or it can not be reached. Recalling the concept of a probabilistic hybrid automaton, each state can be reached by the sum of the probabilities of all paths starting in an initial state and ending in that state. Consequently, the reachability question changes for probabilistic hybrid automata. Instead of asking if a state set can be reached, it is interesting what is the probability of reaching said state set. With that the answer is no longer 'yes'/'no', but a probability value from $[0,1]_{\mathbb{R}}$. An outcome of 1 corresponds to every path in the automaton leading to the unsafe state set, while an outcome of 0 represents no path of the automaton ever reaching a state from the set of unsafe states.

Input: Hybrid automaton with set of initial states *Init*.
Output: Search tree of reachable states.

$S := (Nodes, Root, Succ, State)$
$rootNode := Node(Init)$
ANALYZE($rootNode$)
return S

function ANALYZE($node$)
  $R :=$ REACHTIME($node$)
  for all $(R_J, Jump) \in$ REACHDISCRETE(R) **do**
    $node := Node(R_J)$
    ANALYZE($node$)
  end for
end function

Algorithm 2: Recursive version of the general forward reachability algorithm (Algorithm 1).

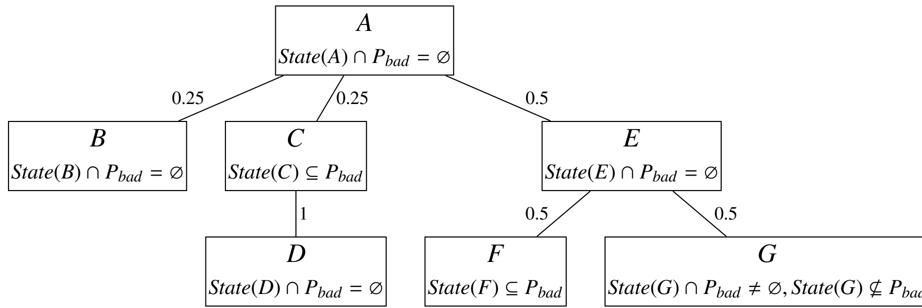


Figure 2.3: Search tree for Examples 2.7.1 and 2.7.2, the probability at each edge is the probability of taking the transition leading to this node. Below each node the relationship to the unsafe state set P_{bad} is given. For the purpose of these examples the continuous behavior is omitted.

Applying normal non-probabilistic reachability analysis to a hybrid automaton induced by a probabilistic hybrid automaton is in general not equivalent to directly using probabilistic reachability analysis on the PHA and checking if the resulting probability is strictly greater than 0, since paths reaching an unsafe state with a probability of 0 could exist. Compared to Markov chains [Kat20], the existence of safety-critical paths with a probability of 0 seems to be rare, therefore for many relevant automata this still holds.

With that, it is now possible to construct an algorithm similar to the non-probabilistic case. And because the previous section introduced an algorithm constructing the search tree from a given hybrid automaton, this part can simply be reused to construct a search tree of the induced hybrid automaton. The probability of reaching an unsafe state is then the sum of all probabilities of all paths leading to a node in the search tree containing an unsafe state. The probability of an edge in the search tree is given by the probability of the transition associated with that edge. The final important step is the first one, the PHA contains non-determinism which prevents the computation of a distinct probability, to mitigate this, a given scheduler is applied to the PHA before constructing the search tree based on the underlying HA. The complete process is outlined in Algorithm 3 and illustrated using Example 2.7.1.

Input: Probabilistic hybrid automaton $\mathcal{H}$, scheduler *Scheduler*, set of unsafe states P_{bad}.
Output: Probability of reaching a state in P_{bad}.

```
S := constructSearchTree(ind(Scheduler(H)))
return computeProbability(S.Root)

function computeProbability(node)
    if State(node) ∩ P_bad ≠ ∅ then
        return 1
    else
        probabilitySum := 0
        for all n ∈ {n' ∈ Nodes | (node, n') ∈ Succ} do
            p := computeProbability(n)
            probabilitySum := probabilitySum + probability((node,n)) · p
        end for
        return probabilitySum
    end if
end function
```

Algorithm 3: Recursive forward reachability algorithm for PHA using search trees, where function constructSearchTree is Algorithm 2.

Example 2.7.1 (Computation of a Probability Upper Bound). *Figure 2.3 depicts a (constructed) search tree as it could result from Algorithm 2. Now the main idea is to recursively call* computeProbability *for each node. Starting with the root node A, A does not contain an unsafe state, therefore the function is evaluated for each successor-node. Node B does not contain an unsafe state and has no children. Thus, it evaluates to 0. Node C does contain an unsafe state and therefore directly returns 1. In consequence node D is skipped. Node E handles like node A and the successors are considered. Both successor-nodes F and G return 1 due to containing unsafe states. Therefore, node E evaluates to* $0.5 \cdot P_F + 0.5 \cdot P_G = 1$. *Finally, node A computes the probability* $0.25 \cdot P_B + 0.25 \cdot P_C + 0.5 \cdot P_E = 0.25 \cdot 0 + 0.25 \cdot 1 + 0.5 \cdot 1 = 0.75$. *In summary, the probability of reaching an unsafe state in this search tree is upper bounded by 0.75.*

The main result is that reachability analysis for probabilistic hybrid automata can use reachability analysis for hybrid automata to achieve the main part of the work. To support probabilistic hybrid automata only a post-processing step is needed.

This also implies that this algorithm is just as undecidable as the algorithm for non-probabilistic hybrid automata. Again over-approximations are used to alleviate this. But in contrast to the non-probabilistic case, the result is not binary but a real number. The probability of reaching an unsafe state using the over-approximation is therefore at least as high as reaching an unsafe state in the actual automaton. Conversely, given only the probability of the over-approximation, the actual probability has to be at most that probability. This results in an upper bound P_u, but using a technique from [AS20b], it is possible to fix a lower bound P_l as well. To achieve this, the complementary event of reaching an unsafe state is observed. While the probability of reaching an unsafe state is less or equal than the probability P_u of reaching an unsafe state in the over-approximation, the probability of not reaching an unsafe state is less or equal than the probability of not necessarily reaching an unsafe state, that is, the sum of all paths that do not include an unsafe state. This statement is equivalent to the probability of reaching an unsafe state is greater or equal to the complementary probability

of not necessarily reaching an unsafe state. Combining both results yields a lower bound P_l and upper bound P_u as summarized below for the finite case where p is the actual probability of reaching an unsafe state. If both bounds $P_l = P_u$ are equal, then the probability of reaching an unsafe state is exact with $p = P_l = P_u$. Both bounds are different if nodes exist which contain both unsafe states and states which don't lead to unsafe states, in the simplified Figure 2.3 this would be nodes like G. The process of deriving a lower bound and the subsequent combination is illustrated in Example 2.7.2.

$$P_l = 1 - \sum_{\text{path } \pi \text{ contains no unsafe state}} P(\pi) \leq p \leq \sum_{\text{path } \pi \text{ contains an unsafe state}} P(\pi) = P_u$$

Example 2.7.2 (Computation of a Probability Lower Bound)**.** *This example is a continuation from Example 2.7.1 using the same search tree in Figure 2.3. The difference is that now a probability lower bound is derived, for which the probability of the complementary event of not reaching an unsafe state is computed. The first step is to calculate the probability of not reaching an unsafe state. Node A does not contain an unsafe state, therefore its successors are considered. Node B does not contain an unsafe state and has no successor-nodes. Thus, it returns 1. Node C has only unsafe states, Consequently, its successor-node D is skipped and node C directly returns 0. For node E again both successor-nodes are evaluated since it does not contain an unsafe state. Node F has no state which is not unsafe and therefore returns 0 while node G has at least one state which is not unsafe and thus evaluates to 1. This results in E returning 0.5. Finally, node A computes* $0.25 \cdot P_B + 0.25 \cdot P_C + 0.5 \cdot P_E = 0.25 \cdot 1 + 0.25 \cdot 0 + 0.5 \cdot 0.5 = 0.5$*. The complementary probability is now 0.5 as well. To summarize, the probability of reaching an unsafe state in this search tree is lower bounded by 0.5. If this result is combined with the result from Example 2.7.1, the probability of reaching an unsafe state in this search tree is between 0.5 and 0.75.*

This describes deriving a probability value for a specific scheduler. In general, all possible schedulers and thus all possible non-deterministic choices are considered. The interesting question then evolves to whether the probability under every scheduler is below a certain threshold. Or conversely the maximum of all probabilities under every possible scheduler. By symmetry, it is also possible to consider if the probability under every scheduler is above a threshold or the minimum of all probabilities. For some applications, the probability interval might be interesting as well. The interpretation of that would be, depending on the non-deterministic choice in the automaton, with a probability inside the interval, an unsafe state is reached.

To limit the scope of this work, only a single fixed scheduler is considered. For now, this is the (history-independent) urgent-priority-scheduler [AS20b], a scheduler that will choose to take a transition as soon as it is enabled (urgency) and if more than one transition-distributions are enabled, the transition with the highest priority is taken. To facilitate this, a unique priority is assigned to every transition-distribution. While this takes care of most of the non-determinism revolving around taking a transition, the non-determinism in the reset and in choosing the initial state remains, for simplicity, it is assumed that all reset functions are deterministic and only a single initial state satisfying the invariant exists.

An approach to consider different schedulers or even all schedulers is presented in Optimal Scheduler in Section 5.1. As a side effect, this approach can also deal with more than one initial state.

2.8 Improving Approximations

Previously (over-)approximations were introduced in order to be able to derive results in an automated fashion. But depending on how good the approximation is, the expressiveness of the result changes. In the context of hybrid automata, an over-approximation might conclude that an unsafe state is reachable, whereas this does not hold for the actual automaton. It is possible to use a better over-approximation, but this would require more computation resources.

Based on this, counterexample-guided abstraction refinement (CEGAR) tries to combine the advantage of being less computationally expensive of coarser approximations with the advantage of providing better results of tighter approximations. The main idea is to first use the coarser approximation until a path to a reachable unsafe state is found. This is the counterexample-candidate possibly disproving the safety of the system. Then this counterexample-candidate is used to analyze the behavior of the system in the tighter approximation. Instead of looking at all possible paths, only the path of the counterexample-candidate in the search-tree is analyzed in the better approximation. The approximation is *refined* along that path. If the refined path does not reach an unsafe state anymore, the coarser approximation did include a path that does not exist for the actual automaton and is therefore not relevant for the safety of the system. Contrarily, if the refined path still reaches an unsafe state an even better approximation may be used.

With this approach of 'on demand'-refining, that is using a better, but computationally more expensive, approximation only when it has the potential to provide additional information, an algorithm can potentially save computation resources while still providing results that are as expressive as an algorithm that directly computes the tightest approximation. A more in-depth description and explanation can be found in [Sch19], the described strategy will be almost identical to what will be used for the remainder of this work. Only a small simplification is made, while [Sch19] stores all approximation improvements directly in the search tree, here a search tree for each approximation layer is used. This simplifies descriptions and allows focusing more on the actual purpose.

2.9 HyPro

HyPro is a C++ library for state set representations targeted to the analysis of hybrid systems. While the algorithms explained in this work are agnostic to the actual representation used, different state set representations of approximations may result in different performance results. HyPro simplifies the usage of different state set representations and already includes functions for computing time-evolution and determining discrete jumps. Amongst other helpful utilities like a parser for hybrid automaton specifications or a plotting service, HyPro already implements two analyzers for hybrid automata, one for a fixed approximation type, that is a fixed state set representation and configuration, and one implementing a CEGAR-based approach for a number of fixed approximation types. [Sch19] focuses more in detail on the capabilities of HyPro.

One of the more common analysis approaches, which is also employed by HyPro, is *flowpipe-construction-based reachability analysis*. The main principle is to approximate the reachable state set by several flowpipe-segments. In the simplest cases, these usually convex segments may be boxes or convex polytopes, for a selection of more complex representations we refer to [Sch19].

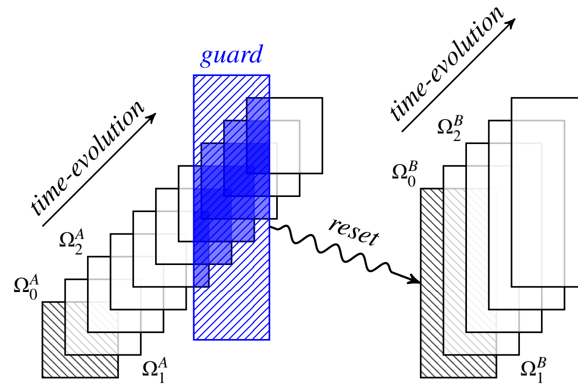


Figure 2.4: Simplified flowpipe-construction using a box-approximation with time-evolution, handling of a discrete jump and again time-evolution. Here flowpipe segment sections satisfying a guard are grouped together.

Using flowpipes, the two main processes, time-evolution and discrete jumps are illustrated in Figure 2.4. They can abstractly be described like this. For an initial segment Ω_0^A the algorithm iteratively computes next segments $\Omega_1^A, \Omega_2^A, \Omega_3^B, \ldots$, each approximating a small duration of time, until a defined limit has been reached. This results in a set of flowpipe segments, which together approximate all states only reachable through time-evolution from that initial segment Ω_0^A. For each flowpipe segment and each outgoing transition in the corresponding location, the guard satisfying subset of that flowpipe segment is generated. Depending on the settings all these subsets may be grouped into a single subset for each transition, otherwise, after the application of the transition reset function, these subsets form several initial sets Ω_0^B of the next iteration of time-evolution. This process is repeated until either a global time-horizon or a maximum number of time-jump-iterations is reached. More detailed information is available from [AS20b, Sch19].

Chapter 3

Algorithm and Implementation

The goal of this chapter is to describe the algorithm for safety analysis of probabilistic hybrid systems in more detail. A focus will be on interesting and important cases, especially regarding the differences to the non-probabilistic setting and to HyPro.

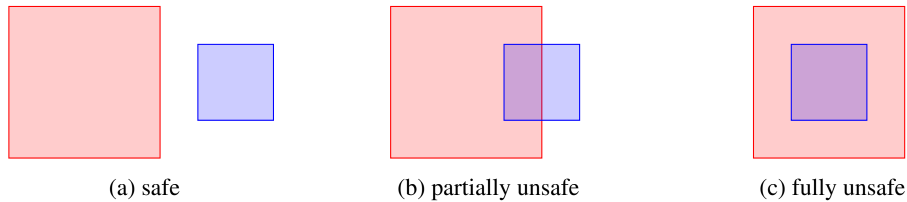
Before the actual algorithms can be described, the input grammar for automata in HyPro needs to be extended. A reference of the updated rules is provided in Appendix A.

First, the basic analysis approach for a fixed state set representation is covered before improving this algorithm with a CEGAR-based approach, but before going into the details, a few assumptions under which the algorithm operates need to be stated.

Algorithm Assumptions The presented algorithm assumes the usage of a fixed scheduler, as before it is assumed that an urgent-priority-scheduler is used. This has the consequence that the algorithm will not compute the correct result for all probabilistic hybrid systems. It is possible to construct an automaton for which the algorithm will compute an interval I for the probability of reaching an unsafe state, but the correct probability p for that automaton is not included in the interval, that is $p \notin I$. This situation occurs if a transition is only enabled in the used over-approximation, but not in the actual automaton, and is the first enabled transition for that state by the means of time-evolution. Therefore, it is assumed that this doesn't occur for any provided model and analysis parameter combination. An example of this problem is provided in Section 4.6. While this problem still occurs when considering not only one but all possible schedulers, it is not a problem in this situation since the correct behavior of the system is still included. The additional, non-existent path only decreases the accuracy of the result. This is also covered in Optimal Scheduler in Section 5.1.

The other restrictions to deterministic reset functions and a single initial state satisfying the invariant still apply.

Previously, analysis was mostly characterized in the context of all reachable states or an over-approximation of all reachable states. As already mentioned in Section 2.5, this needs to be restricted in practice. For this purpose, the two limiting bounds of a local time-horizon and a maximum jump-depth are used. Again, these limits should be chosen to still capture all relevant behavior of the automaton as the presented analysis techniques will only consider states reachable under both limits. The interpretation of reachability analysis in the probabilistic setting thus becomes the probability of reaching an unsafe state within the local time-horizon and the maximum jump-depth.



(a) safe (b) partially unsafe (c) fully unsafe

Figure 3.1: Possible relationships of a flowpipe segment (blue) with unsafe state set (red).

3.1 Simple Algorithm

The algorithm as it was presented back in Algorithm 3 is already very close to an actual implementation. The main differences result from adjusting to the structure of HyPro and integrating the scheduler into the analysis. The logic can be divided into two parts:

1. Constructing a search tree and
2. Computing probabilities based on the search tree from step 1.

Before both steps are covered in detail, there are general parameters needed for the analysis. It was already briefly mentioned that analysis is stopped when a maximum number of allowed jumps is reached and time-evolution continues only up to a local time-horizon. From now on it is assumed both parameters and an additional state set representation as well as a probabilistic hybrid automaton and unsafe state set are given.

Search Tree Construction Constructing the search tree for the given PHA under the given analysis settings is very similar to the non-probabilistic case. The basic structure follows closely after the ANALYZE function from Algorithm 2 except that a queue-worker-approach is preferred over a recursive function, while this currently doesn't make any difference, it will simplify reusing parts in the CEGAR algorithm.

The biggest difference compared to non-probabilistic analysis and something not previously explicitly mentioned is about the handling of flowpipe segments. In non-probabilistic analysis, these are computed and usually stored alongside the node they belong to. For the probabilistic case, these are complemented by additional information. For each flowpipe segment, it is stored how this segment interacts with the unsafe state set. There are a few cases worth distinguishing to which Figure 3.1 provides an additional visual reference:

1. The segment and the unsafe state set are *intersection-free*, also called *safe*.
2. The segment and the unsafe state set have an intersection, but the segment contains states not in the unsafe state set. This partial intersection will be called *partially unsafe*.
3. The segment is a subset of the unsafe state set, called *full intersection* or *(fully) unsafe*.

The other difference is concerning the stopping behavior, while the non-probabilistic setting stops as soon as it found a path to an unsafe state, the probabilistic analysis continues computation. Only once it has been guaranteed that a path will lead to an unsafe state, that specific path is abandoned. The analysis stops when all paths either lead to a full intersection with the unsafe state set, reach the maximum jump-depth or no jump is enabled in the last node's segment set.

Taking all this into account, the algorithm for processing a node will first compute all flowpipe segments up until the local time-horizon, the invariant doesn't hold any more or until the first (completely) unsafe segment is encountered. For each segment, the relationship to the unsafe state set is stored. This part is summarized in Algorithm 4. Afterward, the set of all enabled jumps is computed. This is the same as for non-probabilistic systems, for each outgoing transition and each segment the subset satisfying the guard is tested for non-emptiness. If it is non-empty the reset is applied to that subset and yields the starting point for the next nodes. The result from computing all jump successors is a set of triples consisting of transitions, next initial state sets and the segment number where this transition was enabled. The segment number corresponds to the order in which the segments were constructed, this implies a segment with a lower segment number covers an earlier time interval. Probabilistic transition-distributions that include several transitions are processed for each transition separately while marking them as contiguous. Based on the set of jump-successors a new node is created and inserted into the search tree. The non-probabilistic setting would now enqueue all nodes for processing, but in the probabilistic case only a subset is relevant. If all transition-distributions are grouped, all remaining successor-(set)s represent a non-deterministic choice. This non-deterministic choice has to get resolved by a scheduler. It was stated earlier that the main focus is on the urgent-priority-scheduler, therefore now the earliest transition-distribution is picked. A tie is resolved by the assigned priority. The earliest transition-distribution can be obtained by the smallest segment number. All nodes for the earliest transition-distribution with the highest priority are then enqueued if the jump-depth has not been reached yet. The complete process for handling a node is outlined in Algorithm 5.

An interesting case happens if the time-evolution stops because it found an unsafe segment. This does not necessarily imply that this processing path ends, if there exists an earlier jump, the fully unsafe segment may not be reached by a system governed by an urgent-priority-scheduler.

The probabilistic setting will construct an additional node layer compared to the non-probabilistic case. For this last layer, only the next initial segments are constructed, but the nodes are not processed. The only purpose is to provide information on when the earliest jump is enabled, because the flowpipe segments of the previous layer are only relevant until that jump is taken.

The analysis starts with a node constructed from the initial state already present in the queue. Then nodes are processed as long as nodes are present in the queue. If the queue is empty, this part completes.

Computing Probabilities This is based on function computeProbability in Algorithm 3 with only slight modifications. The remaining small difference is a result of using flowpipe segments, a scheduler and computing a lower and upper bound instead of an exact probability.

The recursive structure is kept, beginning with the root node, which was the node the previous step started with, the probability is recursively evaluated along the search tree.

If the node has no children, that is no successor-nodes, the probability depends completely on the flowpipe segments and their relationship to the unsafe states.

- If a fully unsafe segment exists, then the probability of reaching an unsafe state is in [1,1]. In this case, the probability is exactly 1.
- If no fully unsafe segment, but a partially unsafe segment exists, the probability is in [0,1]. Since this is an over-approximation, it is not possible to exclude the safe part of the segment nor the unsafe part.

Given: Local time-horizon as an analysis parameter and unsafe state set P_{Bad}.
Input: Start state set with location l and valuation-set S.
Output: Flowpipe segments and flowpipe relationships with unsafe state set.

```
segmentList := [ ]
segment := BuildFirstSegment(S)
unsafeIntersection := Intersect(segment, P_Bad)
segmentList.push([segment, unsafeIntersection])
if unsafeIntersection = P_Bad then
    return segmentList                              ▷ segment fully unsafe
end if
while below local time-horizon do
    segment := BuildNextSegment(segment, l)
    if segment = ∅ then
        return segmentList                          ▷ invariant
    end if
    unsafeIntersection := Intersect(segment, P_Bad)
    segmentList.push([segment, unsafeIntersection])
    if unsafeIntersection = P_Bad then
        return segmentList                          ▷ segment fully unsafe
    end if
end while
return segmentList                                  ▷ local time-horizon reached
```

Algorithm 4: Adapted time-evolution function ReachTimeProb for PHAs. The termination causes are listed as comments.

- If no fully unsafe and no partially unsafe segment exist, then the probability is in [0,0] or again exactly 0.

This covers the case for no children, now for the case where the node has successors. First, the subset of nodes relevant under the used urgent-priority-scheduler is determined. Then the contribution of this node to the probability is evaluated, for this the flowpipe segments up to and including the segment where the jump becomes enabled are relevant. They are processed as in the case for no children and result in a probability interval I_N. If this interval consists of the single value 1, it is known that the current path will always reach an unsafe state, the probability of the children won't change anything about this result and with a resulting probability of 1 it is possible to return early.

Otherwise, the probabilities for the relevant successors are computed and weighted according to the probability of taking the transition to that successor, the equation for this is given below, where p_i represents the probability of taking the transition to successor i, and l_i and u_i are the lower- and upper-probability-bounds of successor i. The lower bound can be derived over the complementary event, the maximum probability of not reaching an unsafe state. This computation is similar to following the processes outlined in Example 2.7.1 and Example 2.7.2 at the same time.

$$\left[1 - \sum_i p_i \cdot (1 - l_i), \sum_i p_i \cdot u_i\right] = \left[1 - \underbrace{\sum_i p_i}_{=1} + \sum_i p_i \cdot l_i, \sum_i p_i \cdot u_i\right] = \sum_i p_i \cdot [l_i, u_i]$$

Given: Probabilistic hybrid automaton with analysis parameters.
Input: Node *node* in search tree.

$Segments(node) := \textsc{ReachTimeProb}(StartStates(node), Loc(node))$
$newNodes := [\,]$
for all $(Jump, resetStates, sgmtIdx) \in \textsc{ReachDiscrete}(Segments(node))$ **do**
  $succNode := Node(Jump, resetStates, sgmtIdx)$
  $newNodes.push(succNode)$
end for
if below jump-depth **then**
  $relevantNodes := \textsc{UrgentPriorityScheduler}(newNodes)$
  $\textsc{AddToAnalyzerQueue}(relevantNodes)$
end if

Algorithm 5: Simple algorithm for processing a search tree node for a probabilistic hybrid automaton.

This results in a probability interval I_C which approximates the probability of reaching an unsafe state in the relevant successor nodes. Both intervals I_N and I_C are then combined to be the result of this node. It approximates the probability of reaching an unsafe state from the initial segment of this node. The interval combination can again be derived over the complementary event for the lower bound, where it is the minimum of both lower bounds.

The final result of reaching an unsafe state is then the return for the root node, it represents the probability of reaching an unsafe state in the analyzed probabilistic hybrid system. If the interval is a single value, the computation was exact, but this does not necessarily have to be the case.

3.2 Improved, CEGAR-based Algorithm

The previous section explained the simplest approach to analyzing probabilistic hybrid automata. The aim of this section is to extend this algorithm using the technique from section 2.8. Counterexample-guided abstraction refinement (CEGAR) selectively improves (refines) the approximation where a better approximation might lead to stronger results. In the case of safety analysis for hybrid systems, this is every time an unsafe state can be reached. For probabilistic hybrid systems, this is slightly different. If a set of states leads to an unsafe state with a probability of 1, that is, every path starting in one of those states will eventually reach an unsafe state, then every refinement will only restrict the set of allowed paths and have the same result that every path still leads to an unsafe state with a probability of 1. The same holds for a set of states reaching an unsafe state with a probability of 0, no path in any refinement can reach an unsafe state. Actually, for any set of states that reaches an unsafe state with a probability p that is a trivial interval, any refinement will not provide a better result. In all remaining cases, there exists a non-trivial probability interval I for reaching an unsafe state. This indicates that there exist paths that do not reach an unsafe state as well as paths that do reach an unsafe state. A refinement now has the possibility of excluding some of these paths, possibly resulting in a smaller probability interval. In an extreme case, one of the groups gets completely excluded and the probability interval condenses down to either 0 or 1. To summarize, a path leading to a node of the search tree where the probability of reaching an unsafe state is a non-trivial interval is a candidate for refinement, or in this context also called a *counterexample-candidate*. In Figure 3.2 both nodes would be counterexample-candidates

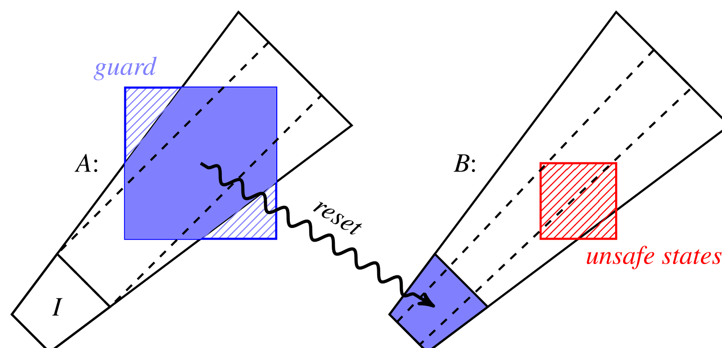


Figure 3.2: Simplified exemplary flowpipes connected through a discrete jump. Initial state set I expands to flowpipe A which in turn satisfies the guard of the jump. The states satisfying the guard become the initial states of flowpipe B after reset application. These states in turn form the initial set for flowpipe B. The expanded flowpipe B partially intersects an unsafe set of states, therefore the probability intervals of reaching an unsafe state are non-trivial for both flowpipes, respectively the nodes associated with these flowpipes. The dashed lines indicate flowpipes obtained using a better approximation, possibly in a refinement.

when considering the unbroken flowpipe lines.

This provides a basic rule for refinement candidates, but also includes situations where there is no benefit in computing a refinement, e.g. node A in Figure 3.2. If a node's successor probability of reaching an unsafe state amounts to a non-trivial interval and the flowpipe segments of the node itself are all safe, the probability interval for the node is still non-trivial. The probability interval in this case is completely determined by the successors. If one computes a refinement to this node, one will in general not get a better result that was not already covered by a refinement for another node. There are situations where refinements will detect transitions that were only enabled in the over-approximations, but these were excluded by assumption and could occur for any node. Continuing the example, if the probability interval is completely determined by the successors, it makes more sense to look at the successor(s) which introduced the ambiguity. For Figure 3.2 this would mean to refine node B instead of node A. This would lead to the dashed flowpipes, where flowpipe B completely crosses the unsafe states and the probability of reaching an unsafe state is 1 for both flowpipes respectively nodes. The general strategy now becomes constructing the search tree, but refining every time a node is detected where the relevant flowpipe segments up to the first transition do not provide a conclusive result, that is if a partially unsafe segment, but no fully unsafe segment is present. If the refinement does not provide a definite result, a higher refinement may be tried and if no higher refinement is present or the refinement results in a final safe/unsafe decision, the lower refinements are updated with more accurate information obtained in higher refinements (we refer to [Sch19] for how lower refinements can benefit from information from higher refinements). Afterward, the search tree is constructed as before.

As previously hinted, each refinement tier has its own approximation quality. The quality and therefore the required computational effort increases from lower to higher tiers. Besides, each tier maintains a separate search tree. All data relevant for computing the probability of reaching an unsafe state will be stored in the lowest tier search tree. From now on this lowest tier will be called the base tier and the accompanying search tree, the base search tree. That aside, the algorithm structure stays very similar to the simple algorithm from the previous section:

1. Constructing a search tree and
2. Computing probabilities based on the search tree from step 1.

While the second step stays completely the same, the first step differs. This also requires that the analysis parameters change slightly, while general analysis parameters like the maximum jump-depth and the local time-horizon are still required, instead of a single state set representation, several (possibly different) state set representations are needed. Each tier (including the base tier) requires a state set representation.

Improved Search Tree Construction Computation starts in the base tier for which the algorithm presented in the previous section is used. This base level tree is constructed until a refinement candidate is found, that is a path to a node which relevant flowpipe segments subset contains a partially unsafe, but no fully unsafe segment.

Following the CEGAR-principle, the path to that node is then refined in the next higher tier analysis. This is as well very similar to what happens for the base tier as described in the previous section but restricted to the single counterexample-path. A refinement analyzer is given a path to a node, based on this it will construct its own search tree, but only do analysis along the provided path. A more detailed differentiation for the non-probabilistic setting is available in [Sch19]. The only difference of non-probabilistic to probabilistic refinement analysis is again, just as for naive analysis, in the stopping behavior and keeping track of how each computed flowpipe segment interacts with the unsafe state set.

If the higher tier (refinement analysis) concludes with the same result, that the last node in the refinement path contains relevant partially unsafe flowpipe segments, but no relevant fully unsafe flowpipe segments, then again, the next higher tier is applied. If the refinement results in a more conclusive result, that is either all relevant flowpipe segments are safe or a relevant fully unsafe flowpipe segment is included, the refinement-process was a success. Next, all lower tiers are updated with the more precise information of the highest refinement level used. In the base search tree, this information is directly important for the following computation and finally for computing the probability of reaching an unsafe state. The other refinement tiers can make use of this more exact information in future refinements. The search trees of higher analysis tiers are not destructed after each refinement-run, since a future refinement might have a refinement path-prefix in common, which then can be directly reused. The last remaining case is what happens if the highest available refinement tier was reached and still no conclusive answer was found. Then, similar to the behavior with no refinements, the final answer might be a non-trivial probability interval. The process for achieving this is equivalent to the logic for a refinement success, all lower tiers are provided with the improved results, even though these results might not be the best results achievable under all possible approximations.

This already summarizes the complete algorithm for constructing a search tree using the CEGAR-approach. The final search tree is the search tree of the base tier, which in general also is the only search tree that is complete under the provided analysis parameters. As already stated, the next and final step is simply evaluating the probability of reaching an unsafe state using the same process described in the previous section for simple analysis.

Dealing with Detected Assumption Violations One of the assumptions was that every transition enabled in an approximation is not just enabled because of the over-approximation. While this assumption still needs to hold, in certain cases violations of this assumption are detected in refinement steps. Essentially, these cases arise if a used approximation includes

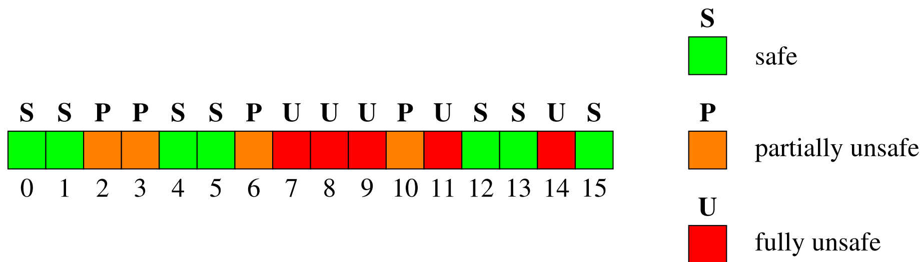


Figure 3.3: Segment list for an exemplary flowpipe on the left. Each segment is represented by a single box with its flowpipe index below. The relationship of a segment with the unsafe state set is labelled according to the legend on the right. Note that computation will usually halt after encountering the first fully unsafe segment.

such a transition. Then a refinement is made along a path that includes this transition and in a better approximation this transition is found to be not present anymore. An attempt at fixing this situation can be made by removing the subtree of the node reachable by this transition in the search trees of all levels. If any node contained in this subtree is currently in an analyzer queue, it is removed from there as well. In case this transition is part of a transition-distribution, this process is repeated for all other transitions parts. After the subtrees have been removed from the base level search tree, a new transition-distribution is selected by the urgent-priority-scheduler. Since the new transition might allow for a longer stay in the current location and thus more flowpipe segments become relevant, a refinement may provide additional results in case a partially unsafe flowpipe segment, but no fully unsafe flowpipe segment is relevant now. If a refinement can't provide better results, since either the newly relevant flowpipe segments are not interesting or a fully unsafe segment is already contained, analysis directly continues in the base level.

It is very important to remember that this approach only corrects cases in which the violation is noticed. Therefore, the assumption, that no transition can only be enabled in over-approximations, can only be relaxed to have to hold at the end of analysis. Essentially, analysis is correct only if either no such transitions exist for all used approximations or all such transitions are found during analysis.

3.3 Improvements

The above algorithm(s) can be improved in many ways. A few very simple already realized improvements are described below. These enhancements include reducing the required memory or extending the analysis to different situations.

Indexing Relationship of Flowpipe Segments with Unsafe States The algorithm as described in Section 3.1 and also the extension in Section 3.2, require that the relationship of a flowpipe segment with the unsafe state set is stored for every single segment. While this type-information is very small, if a flowpipe consists out of many individual flowpipe segments, this will still lead to a larger memory usage. Additionally, this data can already be preprocessed for easier subsequent handling. An abstract visualization of an arbitrary flowpipe is given in Figure 3.3, this flowpipe will be used as an example during the description of this optimization.

The main idea is to store important segment indices instead of the segment type for each segment. Since the only place where the segment type becomes relevant is Section 3.1, Computing Probabilities, all required information can be obtained based on the following two indices.

1. The index of the *first partially unsafe segment*.
   In the example, this would be segment 2.

2. The index of the *first fully unsafe segment*.
   In the example, this is segment 7.

Additionally, it is possible to derive the two indices listed below. While these are not strictly necessary, the first one allows for a quick plausibility check and the other one might be of relevance for other schedulers.

3. The index of the *last safe segment before the first partially unsafe or fully unsafe segment*.
   For the example, this is segment 1.

4. The index of the *last partially unsafe segment before the first fully unsafe segment*.
   In the example, this is segment 6.

Based on these 2 respectively 4 indices, the case distinction for computing the probability of reaching an unsafe state for a relevant flowpipe prefix as given in Section 3.1, Computing Probabilities can be adapted as below. The relevant prefix is given as the index i_{lr} of the last relevant flowpipe segment.

- A relevant fully unsafe segment exists iff the index of the first fully unsafe segment is below or equal to i_{lr}.
- No relevant fully unsafe segment, but a relevant partially unsafe segment exists iff the index of the first fully unsafe segment is larger than i_{lr} or such an index does not exist. Additionally, the index of the first partially unsafe segment has to be less or equal to i_{lr}.
- No relevant fully unsafe or relevant partially unsafe segments exist, i.e. only safe segments exist, iff both the indices for the first fully unsafe and first partially unsafe segments either don't exist or are larger than i_{lr}.

In summary, instead of storing the relationship with the unsafe states for each individual flowpipe segment, store two (or four in the extended case) indices along each search tree node to reduce memory consumption and future effort to find relevant segments.

Unbounded Analysis Both the algorithm in Section 3.1 and the extension in Section 3.2 compute an interval for the probability of reaching an unsafe state within a local time-horizon as well as a maximum number of jumps. This type of analysis is known as *bounded*-analysis [Sch19] and the standard for hybrid automata. It is perfectly suited if the goal is to analyze the execution of an automaton within these bounds or the automaton has no (relevant) behavior outside these bounds. But to derive a statement about behavior outside these bounds, either the limits have to be increased, which comes with an additional performance cost (refer to Chapter 4), or strategies from *classic unbounded*-analysis [Sch19] (e.g. fixed-point recognition) have to be implemented, which has a major performance cost and does not cover all situations.

The main idea behind this conservative approach to *unbounded*-analysis is to approximate the behavior behind the analysis bounds. This approach considers all reachable states, but unlike normal unbounded-analysis does not compute all reachable states. When encountering a path that crosses the analysis limits, bounded analysis assumes the probability of reaching an unsafe state is 0 for the path suffix beyond the limit. The interpretation is, that either no unsafe state is reachable beyond the bound or it is not relevant. The change to consider the path suffix beyond the border is very minimal. Since it is not possible to make a statement about this continuation, it is assumed its probability of reaching a bad state is the most general interval possible [0,1]. Therefore, every possible result from this path suffix is captured.

This will considerably widen the computed probability intervals in many cases. Especially if either the jump limit is low or local time-horizons are reached within a few jumps. Since many automata enforce a maximum duration to stay in a location without a jump, reaching the local time-horizon is less relevant. The more interesting case is reaching the jump limit. If the maximum jump-depth is high enough and paths include transitions from transition-distributions with more than one transition, in general, the probability $P(\pi)$ of a run π reaching the jump-depth limit becomes comparatively small. Therefore, the influence of approximating the continuation of this path with [0,1] is rather small as well. If only a limited number of paths cross this limit, for example, because all other paths lead to an unsafe state within the analysis bounds, the accumulated influence of all paths with an approximate continuation is again small compared to the influence of all paths which didn't reach the limits. An example of this is visible in Paragraph Unbounded Analysis of Section 4.1 when exploring the results obtained using this unbounded approach.

Only little changes are required to support this different analysis type. The time-evolution function (Algorithm 4) needs to return the reason why time-evolution stopped alongside the computed flowpipe segments. This additional information is then used when computing the probabilities for a flowpipe (Section 3.1, Computing Probabilities). For a flowpipe that is not restricted by taking only the relevant prefix and for which computation was stopped because of reaching the local time-horizon, the probability of reaching an unsafe state is [0,1] if no fully unsafe segment is included. This results in a flowpipe with only safe segments which reached the local time-horizon to have a probability of [0,1] instead of 0. The other change has to be made when determining the probability for a search tree node beyond the jump-depth. Previously a probability of 0 was used, but now it has to be replaced by [0,1] as well.

Chapter 4

Experimental Results

Both of the algorithms presented in Section 3.1 and Section 3.2 were implemented in library HyPro (Section 2.9). The aim for the first part of this chapter is to compare both approaches with the tool ProHVer [ZSR+10] which is based on PHAVer. In the second part, a few notable examples demonstrating the core behavior and algorithm-specific corner cases are explored.

The main idea behind ProHVer is to split the probabilistic hybrid automaton into a non-probabilistic hybrid automaton and a function mapping the transitions of the hybrid automaton to their stripped probabilities. The non-probabilistic hybrid automaton is obtained similarly to the induced HA of Definition 2.4.3. This HA is then processed by a modified version of PHAVer, which produces a transition system of reachable state sets. Afterward, the transition system is extended using the probability-mapping to obtain a Markov decision process. In the last step, the maximum probability of reaching an unsafe state is computed [ZSR+10].

Furthermore, ProHVer considers the maximum probability under an arbitrary scheduler [ZSR+10] while the presented approaches only take a single fixed scheduler into account. This leads to ProHVer being able to compute sensible results for more PHAs compared to the algorithms from this work. Some automata employ little or no non-deterministic behavior, in these cases, the presented approach is comparable to ProHVer. Other automata can be remodeled to use less non-determinism while preserving the original behavior as much as possible, for example for automata that use an additional non-deterministic jump to specify unsafe-regions in a subarea of a location, this non-determinism can be replaced by a direct specification of the unsafe region in the original location. Automata which heavily rely on non-determinism and which cannot be reasonably approximated with a fixed scheduler are usually not compatible with the presented approach.

The core functionality of analyzing a hybrid system is a comparison between the already implemented algorithms in HyPro [Sch19] and PHAVer [Fre05]. While HyPro provides a more exhaustive library of geometric representations, PHAVer implements other techniques like fixed-point detection where it recognizes if the automaton returns to a previously visited state [Sch19]. This fixed-point detection leads to ProHVer having to extract the probability from a Markov decision process which potentially includes cycles while the presented algorithm only has to deal with the simpler case of a strict tree.

For the comparison the case studies published for ProHVer [pro11] will be used. [ava10] also provides an interesting list of many models for comparison, 8 of which include probabilistic or stochastic behavior. Unfortunately, the only applicable, non-password-protected models appear to be the 4 PHAs of ProHVer.

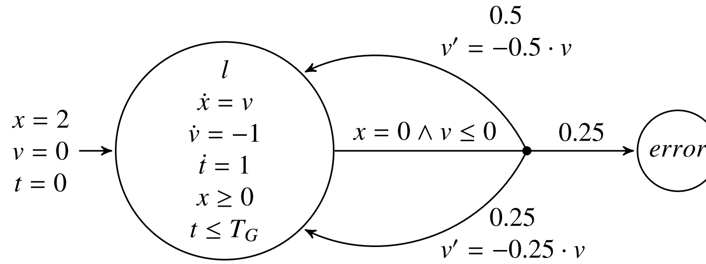


Figure 4.1: The modified probabilistic hybrid automaton for case study ProHVer: Bouncing Ball in Section 4.1.

The execution times for ProHVer are the time it took to build the abstraction, i.e. the produced transition system [ZSR+10], whereas for the presented implementation in HyPro every analysis step is included. Parsing and an eventual output step are excluded, thus the execution time covers constructing the search tree and computing the probabilities based on the search tree. Usually, the time-contributions of the excluded steps for ProHVer are minor, therefore the performance numbers are still comparable.

Additionally, ProHVer provides the number of states in the generated transition system. For HyPro the number of nodes in the (base level) search tree is recorded. To make for a fair comparison, only full-aggregation (for an explanation we refer to [Sch19]) is used, while the computed results should be the same, many states would otherwise be directly discarded. Furthermore, both the maximum jump-depth and local time-horizon are chosen large enough to not be restrictive unless stated otherwise.

4.1 ProHVer: Bouncing Ball

The first case study is a probabilistic version of the classic Bouncing Ball by [pro11]. This is Example 2.4.1 with slightly adjusted constants and an additional global timer. To enable compatibility with HyPro, the strict inequality in one of the guards is relaxed to a non-strict inequality. Furthermore, the intermediate locations where any time-evolution is prohibited are removed, because locations, where no time-evolution is allowed, are not always supported by HyPro. These changes don't affect the behavior of the automaton and the performance impact should be negligible. The final automaton is given in Figure 4.1.

The main idea of this automaton has already been described in Example 2.4.1, the additional global timer t is used in conjunction with a global time-horizon T_G to limit the execution of the automaton.

Bounded Analysis This automaton was analyzed with a range of global time-horizons T_G and several approximations. Before exploring results using the unbounded extension from Section 3.3 or comparing results with ProHVer, the results obtained using the original bounded-analysis are examined. For the simple approach, Table 4.1 and Table 4.2 cover results for two different types of approximations under different timesteps. Table 4.3 provides results for the advanced variant.

One of the first observations is that all computed probability intervals are trivial. This is a direct consequence of the unsafe states in the automaton being all states which are in location *error*. If all unsafe states are isolated to one location which furthermore is unsafe

Table 4.1: Experimental results for case study ProHVer: Bouncing Ball in Section 4.1 using the simple approach with bounded-analysis implemented in HyPro with 4 approximations based on boxes and varying timesteps. For each configuration, the computed probability (column *Probability*), the execution time (*Time in* ms) and the number of nodes in the search tree (*#Nodes*) is shown. Runs which exhibit Zeno-behavior are marked with *Zeno-behavior*. Trivial intervals of value $x \in \mathbb{R}$ are abbreviated as $[x]$. ^{F} indicates invalid executions violating the assumption that jumps are not allowed to only exist in an over-approximation.

T_G	Probability	Time in ms	#Nodes	Probability	Time in ms	#Nodes
	Box, timestep 1/10			Box, timestep $1/10^2$		
1	[0]	0.2	1	[0]	0.3	1
2	[0.25]	0.2	4	[0.25]	0.6	4
3		*Zeno-behavior*		[0.3125]	1.3	7
3.1		*Zeno-behavior*		[0.3125]	2.2	7
3.2		*Zeno-behavior*		$[0.328125]^F$	1.4^F	10^F
3.3		*Zeno-behavior*			*Zeno-behavior*	
3.4		*Zeno-behavior*			*Zeno-behavior*	
	Box, timestep $1/10^3$			Box, timestep $1/10^4$		
1	[0]	2.2	1	[0]	21.0	1
2	[0.25]	5.1	4	[0.25]	42.2	4
3	[0.3125]	9.1	7	[0.3125]	84.9	7
3.1	[0.3125]	11.1	7	[0.3125]	101.5	7
3.2	[0.3125]	10.6	7	[0.3125]	97.4	7
3.3	[0.328125]	11.6	10	[0.328125]	106.0	10
3.4		*Zeno-behavior*			*Zeno-behavior*	

Table 4.2: Experimental results for case study ProHVer: Bouncing Ball in Section 4.1 using the simple approach with bounded-analysis implemented in HyPro with 4 approximations based on support functions and varying timesteps. For each configuration, the computed probability (column *Probability*), the execution time (*Time in* ms) and the number of nodes in the search tree (*#Nodes*) is shown. Runs which exhibit Zeno-behavior are marked with *Zeno-behavior*. Trivial intervals of value $x \in \mathbb{R}$ are abbreviated as $[x]$.

T_G	Probability	Time in ms	#Nodes	Probability	Time in ms	#Nodes
	Support Function, timestep 1/10			Support Function, timestep $1/10^2$		
1	[0]	1.8	1	[0]	7.3	1
2	[0.25]	22.2	4	[0.25]	36.9	4
3		*Zeno-behavior*		[0.3125]	68.8	7
3.1		*Zeno-behavior*		[0.3125]	71.7	7
3.2		*Zeno-behavior*		[0.3125]	70.9	7
3.3		*Zeno-behavior*			*Zeno-behavior*	
3.4		*Zeno-behavior*			*Zeno-behavior*	
	Support Function, timestep $1/10^3$			Support Function, timestep $1/10^4$		
1	[0]	79.8	1	[0]	781.8	1
2	[0.25]	160.3	4	[0.25]	1630.6	4
3	[0.3125]	357.4	7	[0.3125]	3415.9	7
3.1	[0.3125]	395.3	7	[0.3125]	3644.7	7
3.2	[0.3125]	402.1	7	[0.3125]	3844.2	7
3.3	[0.328125]	470.7	10	[0.328125]	4210.3	10
3.4		*Zeno-behavior*			*Zeno-behavior*	

Table 4.3: Experimental results for case study ProHVer: Bouncing Ball in Section 4.1 using the advanced approach with bounded-analysis implemented in HyPro with 4 approximation-lists based on boxes and varying timesteps. For each configuration, the computed probability (column *Probability*), the execution time (*Time in* ms) and the number of nodes in the base level search tree (*#Nodes*) is shown. Runs which exhibit Zeno-behavior are marked with *Zeno-behavior*. Trivial intervals of value $x \in \mathbb{R}$ are abbreviated as $[x]$. ^{F} indicates invalid executions violating the assumption that jumps are not allowed to only exist in an over-approximation.

T_G	Probability	Time in ms	#Nodes	Probability	Time in ms	#Nodes
	(Box, timestep $1/10$), (Box, timestep $1/10^3$)			(Box, timestep $1/10^2$), (Box, timestep $1/10^4$)		
1	[0]	0.3	1	[0]	0.4	1
2	[0.25]	0.3	4	[0.25]	0.7	4
3		*Zeno-behavior*		[0.3125]	1.4	7
3.1		*Zeno-behavior*		[0.3125]	2.5	7
3.2		*Zeno-behavior*		$[0.328125]^F$	1.5^F	10^F
3.3		*Zeno-behavior*			*Zeno-behavior*	
3.4		*Zeno-behavior*			*Zeno-behavior*	
	(Box, timestep $1/10^3$), (Box, timestep $1/10^5$)			(Box, timestep $1/10^4$), (Box, timestep $1/10^6$)		
1	[0]	2.2	1	[0]	21.5	1
2	[0.25]	5.2	4	[0.25]	42.2	4
3	[0.3125]	9.7	7	[0.3125]	90.0	7
3.1	[0.3125]	10.6	7	[0.3125]	99.0	7
3.2	[0.3125]	17.0	7	[0.3125]	99.9	7
3.3	[0.328125]	11.2	10	[0.328125]	117.6	10
3.4		*Zeno-behavior*			*Zeno-behavior*	

for all valuations, partially unsafe flowpipe segments do not exist. This in turn leads to all intermediate results during computation of probabilities being trivial intervals. For the same reason the results for the extended algorithm (Table 4.3) almost match the results for the simple algorithm (Table 4.1) exactly when comparing the approximations of the first level, even the execution times are almost the same. The explanation continues the argument from before, if the search tree has at no point non-trivial probability intervals, then refinement is never used and thus the computation of both algorithms is almost the same as well. The important consequence is that the extended algorithm is only useful if the automaton can contain flowpipe segments which are not either safe or fully unsafe. A small additional overhead is observable, but even for small systems it is barely distinguishable from run-to-run variations.

Another unusual result from Table 4.1 is, that the probability for reaching an unsafe state under time-horizon $T_G = 3.2$ is exactly 0.328125 for timestep $1/10^2$ and exactly 0.3125 for timestep $1/10^3$. This obvious contradiction is resolved by the first result being obtained under an analysis that used a jump which is only available in the used approximation. Consequently, this execution violates the assumption from the beginning of Chapter 3. This is not detected and all computed results therefore need manual verification. In Section 5.1 two possible solutions are presented. The other used type of approximation (Table 4.2) does not express this kind of violation. It only occurs for certain approximation configurations depending on the automaton.

The direct comparison between the two approximation types (Table 4.1 and Table 4.2) yields not many interesting results, support functions are able to obtain better results for larger timesteps ($T_G = 3.2$, timestep $1/10^2$), but take a multiple of the time to run. For the

Table 4.4: Experimental results for case study ProHVer: Bouncing Ball in Section 4.1 using ProHVer with analysis parameters provided by [pro11]. For each configuration, the computed probability upper bound (column *Probability upper bnd*), the time to build the abstraction (*Time in* s) and the number of states in the generated abstraction (*Abstract states*) is shown. Runs which did not terminate in sufficient time are marked as such.

	Interval length 0.15			Interval length 0.15, convex hull			Interval length 0.05, convex hull		
T_G	Probability upper bnd	Time in s	Abstract states	Probability upper bnd	Time in s	Abstract states	Probability upper bnd	Time in s	Abstract states
1	0	0.2	38	0	0.2	17	0	0.7	56
2	0.25	0.9	408	0.25	0.7	59	0.25	3.1	185
3	0.5	4.5	2907	0.5	1.5	124	0.3125	7.3	347
3.1	0.5	6.1	4147	0.5	1.5	130	0.3125	7.6	356
3.2	0.5	8.2	5400	0.5	1.5	132	0.5	8.1	386
3.3	0.5	12.1	6838	*not terminating within one day*			0.5	8.6	399
3.4	0.5	16.3	8789	1	1.8	142	0.5	9.1	411
3.5	0.5	23.4	11 216	1	1.8	145	0.5	9.8	425
3.6	0.5	34.4	14 297	1	1.9	150	0.5	11.2	436
3.7	0.5	52.6	18 338	1	2.0	154	*not terminating within one day*		

selected examples even using the box approximation type with a smaller timestep is often preferable compared to using support functions with a larger timestep.

Now for the main reason why this automaton is interesting, it features a high amount of Zeno-behavior [Sch19]. For approximations that are too coarse, the guard of the single transition-distribution is already satisfied directly after the last jump. This automaton was specifically constructed to reach Zeno-behavior for any used approximation given the time-horizon T_G is chosen large enough. This is the reason why for example Table 4.1 can't obtain results for $T_G > 2$ with a timestep of $1/10$, but with a better approximation, which in this case is a smaller timestep, manages to compute correct results for time-horizons up to 3.3. None of the approximations implemented in HyPro will be able to compute a correct result for $T_G = 3.4$, since e.g. the path of the automaton which always chooses the transition with 25 % velocity conservation will never reach that time-bound. This path's value of variable t (the execution time of the automaton) will asymptotically go to $3.\overline{3}$.

ProHVer After interpreting the results obtained using the presented algorithms implemented in HyPro, the equivalent results for ProHVer are presented in Table 4.4. The obtained execution times are about 3-4 times faster than those given by [pro11], this is most likely the effect of faster hardware. Unusual behavior was observed for 2 specific runs, for which the PhaVer step does not terminate within one day. At some point, the generated log-files start to exhibit signs of cyclic behavior with characteristic debug information repeating in cycles of 9 and 13 iteration long blocks. The reason for this behavior is unknown.

For every time-horizon up to $T_G = 3.3$, the computations from HyPro were multiple orders of magnitudes faster than those by ProHVer and in the case of $T_G = 3, 3.1, 3.2$ and 3.3 even more accurate. For every time-horizon larger than 3.3, ProHVer is able to obtain useful results under at least some configurations. This is very likely the result of PHAVer's fixed-point detection.

Unbounded Analysis Up until now, only bounded-analysis was considered, now the same experiments as before are repeated for the unbounded-analysis variation presented in Section 3.3. With this change, the maximum jump-depth becomes relevant as well. Previously it was

just chosen large enough such that the time-horizon is hit before the maximum jump-depth. The time-horizon is chosen as before large enough to not be restricting. The probability and execution times results are presented in Table 4.5 for different configurations.

Comparing these results against the results under the same configurations, except for bounded-analysis (Table 4.1), the previously trivial probability intervals are not all trivial anymore. In any situation where the maximum jump-depth was not large enough to construct the complete search tree, the probability interval is not trivial anymore. The only cases where the maximum jump-depth can be large enough are for time-horizon $T_G = 3.3$, for which the complete search tree is at most 4 levels deep. But even then, larger timesteps ($1/10$ and $1/10^2$) are not able to compute the complete tree without exhibiting Zeno-behavior. The difference to before is, that smaller timesteps can compute an approximation when restricting the jump-depth. The result of $[0.25, 0.5]$ for $T_G = 3.3$, timestep $1/10$ and maximum jump-depth of 1 may not very accurate, but compared to the previous 'no result' this is a big improvement.

Comparing the results of a fixed time-horizon T_G across different timesteps and different maximum jump-depths, the intuitive assumptions hold, the interval borders get tighter with increasing maximum jump-depth and better approximations (decreasing timestep). The difference of both interval borders is here a direct measure for the accuracy. Using an infinite maximum jump-depth (and possibly an infinite timestep) would always asymptotically lead to a trivial interval.

Comparing the execution times for different time-horizons T_G for a fixed timestep and maximum jump-depth, the results are almost if not identical for smaller jump-depths, while following the expected relationship of increasing with increasing time-horizon for larger jump-depths. For smaller timesteps, all nodes in the search tree were restricted due to the jump-depth, therefore an increase in the time-horizon will result in the same search tree, which will take the same amount of time to compute. If not all nodes are limited due to the maximum jump-depth, but some are limited due to T_G, these nodes will get new successors for larger time-horizons. This in turn will result in a larger computation time.

The comparison of these results with ProHVer is not entirely fair, since this approach only considers the execution of the automaton up to some limit and approximates the execution beyond the limit. ProHVer however considers the complete execution, which is made possible by their fixed-point detection. Still, the results obtained using unbounded-analysis (Unbounded Analysis in Section 3.3) are not only more accurate but were also achieved in less time. Exemplarily comparing the largest listed time-horizon for ProHVer of $T_G = 3.7$, which claims the probability of reaching an unsafe state is below 0.5 to the result from HyPro, which states the probability is between approximately 0.407 and 0.426 already shows the improvement. Additionally, the unbounded-analysis approach can easily compute larger time-horizons, with very little to no additional required time, since as already previously motivated, once the search tree is complete up to a jump-depth, the time needed for analysis cannot increase further. This is illustrated in Table 4.6, with only slight increases in execution time, sensible results for previously impossible time-horizons $T_G = 4, 5$ and 6 can be obtained. For $T_G = 6$, the search-tree is already complete up to jump-depth 10, therefore the results for all larger time-horizons are practically equal but still correct. This approach enables the computation of an arbitrarily large time-horizon in finite time, in this example even probability bounds with quite good accuracy in very little time.

Table 4.5: Experimental results for case study ProHVer: Bouncing Ball in Section 4.1 using the simple approach with unbounded-analysis (Section 3.3) implemented in HyPro with 4 approximations based on boxes and varying timesteps and 4 different maximum jump-depths. Each cell contains the computed probability interval in the first row and the execution time in ms in the second row. Runs which exhibit Zeno-behavior are marked with *Zeno-behavior*. ^{F} indicates invalid executions violating the assumption that jumps are not allowed to only exist in an over-approximation.

T_G	max. jumps 1	max. jumps 3	max. jumps 5	max. jumps 7
Box, timestep 1/10				
3.3	[0.25, 0.5] 0.3	*Zeno-behavior*	*Zeno-behavior*	*Zeno-behavior*
3.4	[0.25, 0.5] 0.4	*Zeno-behavior*	*Zeno-behavior*	*Zeno-behavior*
3.5	[0.25, 0.5] 0.3	*Zeno-behavior*	*Zeno-behavior*	*Zeno-behavior*
3.6	[0.25, 1] 0.4	*Zeno-behavior*	*Zeno-behavior*	*Zeno-behavior*
3.7	[0.25, 1] 0.4	*Zeno-behavior*	*Zeno-behavior*	*Zeno-behavior*
Box, timestep $1/10^2$				
3.3	[0.25, 0.5] 1.3	[0.328125, 0.375] 3.8^F	*Zeno-behavior*	*Zeno-behavior*
3.4	[0.25, 0.5] 1.9	[0.328125, 0.375] 1.8	*Zeno-behavior*	*Zeno-behavior*
3.5	[0.25, 0.5] 1.2	[0.359375, 0.40625] 2.0	*Zeno-behavior*	*Zeno-behavior*
3.6	[0.25, 0.5] 1.2	[0.359375, 0.4375] 2.3	*Zeno-behavior*	*Zeno-behavior*
3.7	[0.25, 0.5] 1.3	[0.359375, 0.5] 4.7	*Zeno-behavior*	*Zeno-behavior*
Box, timestep $1/10^3$				
3.3	[0.25, 0.5] 9.2	[0.328125, 0.328125] 13.1	[0.328125, 0.328125] 12.2	[0.328125, 0.328125] 13.5
3.4	[0.25, 0.5] 10.4	[0.328125, 0.375] 12.0	[0.344727, 0.359375] 14.0	*Zeno-behavior*
3.5	[0.25, 0.5] 11.1	[0.359375, 0.40625] 12.7	[0.379883, 0.40625] 17.8	*Zeno-behavior*
3.6	[0.25, 0.5] 10.3	[0.359375, 0.40625] 13.1	[0.379883, 0.40625] 18.0	*Zeno-behavior*
3.7	[0.25, 0.5] 11.0	[0.359375, 0.4375] 13.3	[0.393555, 0.4375] 19.4	*Zeno-behavior*
Box, timestep $1/10^4$				
3.3	[0.25, 0.5] 91.7	[0.328125, 0.328125] 113.3	[0.328125, 0.328125] 107.2	[0.328125, 0.328125] 106.7
3.4	[0.25, 0.5] 94.4	[0.328125, 0.375] 112.8	[0.342773, 0.351562] 127.5	[0.346619, 0.351562] 120.9
3.5	[0.25, 0.5] 99.9	[0.359375, 0.40625] 142.4	[0.379883, 0.40625] 124.1	[0.391418, 0.40625] 172.4
3.6	[0.25, 0.5] 100.0	[0.359375, 0.40625] 126.3	[0.379883, 0.40625] 150.6	[0.391418, 0.40625] 155.5
3.7	[0.25, 0.5] 101.6	[0.359375, 0.4375] 151.3	[0.393555, 0.425781] 158.1	[0.407654, 0.425781] 157.8

Table 4.6: Experimental results for case study ProHVer: Bouncing Ball in Section 4.1 using the simple approach with unbounded-analysis (Section 3.3) implemented in HyPro. All results were obtained using a box-representation and a timestep of $^1/_{10^6}$ and a maximum jump-depth of 10. For each configuration, the computed probability (column *Probability*), the execution time (*Time in* s), the number of nodes in the search tree (*#Nodes*) and the maximum search tree depth (*Search tree depth*) is shown.

T_G	Probability	Time in s	#Nodes	Search tree depth
1	[0, 0]	2.6	1	1
2	[0.25, 0.25]	4.8	4	2
3	[0.3125, 0.3125]	9.5	7	3
4	[0.606229, 0.625]	18.6	3076	12
5	[0.781215, 0.8125]	25.3	4612	12
6	[0.943686, 1]	31.9	6142	12
10	[0.943686, 1]	32.2	6142	12
20	[0.943686, 1]	32.0	6142	12
100	[0.943686, 1]	32.4	6142	12

4.2 ProHVer: Water Level Control

This case study is again by [pro11] and models a water level control system of a tank where the water level is monitored by wireless sensors. This communication channel is assumed to be unreliable and the transmission time is determined using a (discrete) probabilistic distribution. The system reaches an unsafe state if the water level in the tank rises above a fixed limit or falls below another fixed limit. Unfortunately, this case study relies on strict inequalities for checking if the water level is outside the allowed region, which cannot be mapped to an equivalent automaton without strict inequalities for use with HyPro. Some brief experiments were made with mapping the strict inequalities to non-strict inequalities with a small offset, but the results were not useful.

4.3 ProHVer: Autonomous Lawn-Mower

The third case study by [pro11] deals with an autonomous lawn-mower placed on a rectangular patch of grass which is 100 m wide and 200 m deep. Whenever the mower reaches the border of the grass, it changes direction and returns. To ensure a variation in the movement, this change in direction is influenced by a probabilistic distribution. In area from 90 m to 100 m in x-direction and 170 m to 200 m in y-direction is a sheet of cloth according to [pro11]. For further description, it will be assumed there is a bed of beautiful flowers in this area instead of a piece of cloth. In any case, if the lawn-mower reaches this area, the flowers or the cloth will be shredded. Thus, all states in this area are declared as unsafe states for all locations of this automaton. The computed probability of reaching said unsafe states is then the probability of vandalizing the flowerbed. Initially, the mower is placed at 10 m in x-direction and 20 m in y-direction with a heading of diagonally into positive x-y-direction. The complete automaton is given in Figure 4.3 with a sketch of the full initial setting in Figure 4.2. Compared to the original automaton from [pro11], similar modifications were made as in Section 4.1. The single intermediate location for the initial state where any time-evolution is prohibited is removed and instead of a non-deterministic modeled destruction of the flowers, a direct approach is used, where the unsafe states are in the locations themselves. Just as for Section 4.1, an otherwise unused global timer t is added, to limit the execution of the automaton through the use of a global time-horizon T_G.

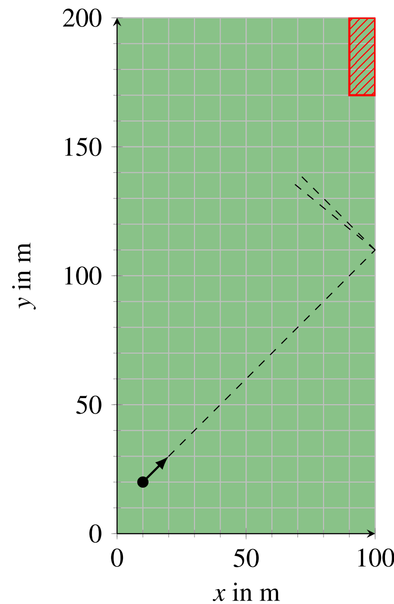


Figure 4.2: Sketch of the environment of case study PROHVER: Autonomous Lawn-Mower in Section 4.3. The complete grass region is colored green, the flowerbed is marked in red and the lawn-mower itself is located at the black dot with the arrow indicating the initial heading. The movement in the first few seconds is illustrated by the dashed lines.

Bounded Analysis Like before, this automaton was analyzed for the same range of global time-horizons as [pro11] and using different approximations. Again, first, the results for the bounded-analysis variant are discussed, before a comparison with PROHVER is made. Afterward, the unbounded approach from Section 3.3 is tested against this automaton. The result for bounded-analysis are listed in Table 4.7 for the simple algorithm, while Table 4.8 represents the advanced algorithm.

Unlike before, the computed probability intervals of destroying the flowerbed are in general not trivial anymore. This is a direct consequence of the existence of partially unsafe flowpipe segments.

If the mower reaches the grass-boundary close to a corner, the mower may hit two boundaries shortly after each other. If the flowpipe segment satisfying the guard of the transition-distribution of the first boundary directly satisfies the guard of the transition-distribution of the second boundary after applying the reset, the mower can get stuck in this corner with a non-realizable path due to Zeno-behavior. This is not a sufficient condition, for this situation to occur, the mower needs to reach the first boundary under a specific angle which changes depending on the analysis parameters. In general, this occurs less for better approximations. Additionally, in rare cases, the mower only gets stuck temporarily and leaves the corner after a certain amount of direction changes at both boundaries. But even if the mower leaves the corner again, it briefly included incorrect behavior and the result is thus invalid. None of the executions presented in this section exhibit this kind of behavior. The decreasing number of nodes in the base level search trees when using better approximations is mostly a result of a more exact time-horizon. If the lawn-mower stops very close to a border, this border might be reached in a less accurate approximation. While this is certainly surprising and in special

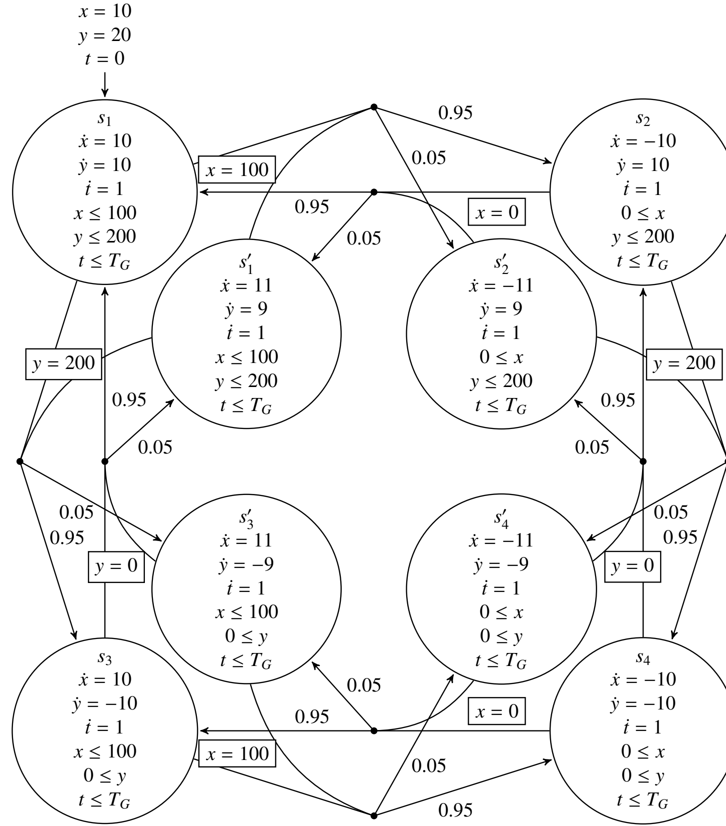


Figure 4.3: The modified probabilistic hybrid automaton for case study ProHVer: Autonomous Lawn-Mower in Section 4.3. Transition-distributions where only the source location differs were unified for brevity and the common guard of both transition-distributions is represented by a single guard in rectangular boxes. For instance in location s_1, if the guard $x = 100$ is satisfied, it is possible to take the top-most transition-distribution and go to s_2 with 0.95 probability or to s_2' with 0.05 probability. The same transition-distribution is available in location s_1', if the guard $x = 100$ is satisfied, the automaton can take this transition to go to s_2 with 0.95 probability or to s_2' with 0.05 probability.

cases even leads to execution time reductions when using better approximations, this does not affect the correctness of the result, only the accuracy. Redundant paths in lower approximations, which get resolved in better approximations, have a small contribution in search tree nodes which are leaves or predecessors of leaves. While the inclusion of these paths is technically incorrect, they can't be easily excluded and don't affect the result.

Examining each time-horizon T_G of the results for the simple algorithm in Table 4.7 separately, for $T_G = 10$, the probability result is already an exact value for the coarsest approximation used. As expected, this result does not change for better approximations, but the required time to compute the result increases. For $T_G = 70$, the coarsest approximation with timestep 1 is not able to derive a correct result anymore, while the approximation for timestep $1/10$ computes a correct probability interval. The analysis for timestep $1/10^2$ shrinks the interval, but only the approximation for timestep $1/10^3$ manages to compute an exact value. This increase in quality of course comes with a tradeoff in execution time. The time-horizon

Table 4.7: Experimental results for case study PROHVER: Autonomous Lawn-Mower in Section 4.3 using the simple approach with bounded-analysis implemented in HYPRO with 4 approximations based on boxes and varying timesteps. For each configuration, the computed probability (column *Probability*), the execution time (*Time*), the number of nodes in the search tree (*#Nodes*), the maximum search tree depth (*Search tree depth*) and the memory usage at the end of analysis (*Memory usage*) is shown. Runs which exhibit Zeno-behavior are marked with *Zeno-behavior*.

T_G	Probability	Time	#Nodes	Search tree depth	Memory usage
Box, timestep 1					
10	[0, 0]	0.4 ms	3	2	35 MB
70	-	*Zeno-behavior*	-	-	-
100	-	*Zeno-behavior*	-	-	-
110	-	*Zeno-behavior*	-	-	-
120	-	*Zeno-behavior*	-	-	-
Box, timestep 1/10					
10	[0, 0]	0.6 ms	3	2	35 MB
70	$[2.150\,82 \times 10^{-6}, 9.823\,53 \times 10^{-5}]$	560.0 ms	2023	11	50 MB
100	-	*Zeno-behavior*	-	-	-
110	-	*Zeno-behavior*	-	-	-
120	-	*Zeno-behavior*	-	-	-
Box, timestep $1/10^2$					
10	[0, 0]	0.004 s	3	2	0.04 GB
70	$[1.040\,73 \times 10^{-5}, 1.145\,29 \times 10^{-5}]$	2.8 s	1807	11	0.14 GB
100	$[1.040\,73 \times 10^{-5}, 1.145\,29 \times 10^{-5}]$	55.6 s	28 557	16	1.9 GB
110	$[1.833\,36 \times 10^{-4}, 3.130\,12 \times 10^{-4}]$	99.9 s	59 445	17	3.6 GB
120	$[1.833\,36 \times 10^{-4}, 3.130\,12 \times 10^{-4}]$	191.8 s	101 021	19	6.1 GB
Box, timestep $1/10^3$					
10	[0, 0]	0.04 s	3	2	0.04 GB
70	$[1.119\,84 \times 10^{-5}, 1.119\,84 \times 10^{-5}]$	25.4 s	1795	11	1.0 GB
100	$[1.119\,84 \times 10^{-5}, 1.119\,84 \times 10^{-5}]$	505.4 s	26 411	16	19 GB
110	$[2.786\,11 \times 10^{-4}, 2.818\,62 \times 10^{-4}]$	883.8 s	57 157	17	34 GB
120	$[2.786\,11 \times 10^{-4}, 2.818\,62 \times 10^{-4}]$	1496.8 s	86 817	19	58 GB

$T_G = 100$ is very similar to $T_G = 70$, while $T_G = 110$ and $T_G = 120$ show the same characteristics except for better approximations. For the used approximation with timestep $1/10^2$, analysis computes a probability interval, which is reduced by an approximation with timestep $1/10^3$. A trivial-interval as a result would require an even better approximation, for the used box-approximation a timestep of $1/10^4$ would be sufficient, but require even more time and memory. This is the exact problem the improved CEGAR-based algorithm (Section 3.2) tries to solve by only using the better approximation where it can provide better results.

Comparing the results from the improved algorithm in Table 4.8 with the previous results using the simple algorithm yields no improvement for time-horizon $T_G = 10$ and even the execution time and memory consumption is almost if not identical. Time-horizon $T_G = 70$ has the first interesting result, the simple algorithm has only non-trivial interval results for timestep $1/10$ and $1/10^2$ and requires timestep $1/10^3$ for an exact probability. The improved CEGAR-approach manages to compute an exact probability for an approximation-strategy, which first uses timestep $1/10$ and secondary $1/10^3$. The first approximation is good enough to be able to compute a result and the second approximation is good enough to be able to

Table 4.8: Experimental results for case study ProHVer: Autonomous Lawn-Mower in Section 4.3 using the advanced approach with bounded-analysis implemented in HyPro with 4 approximation-lists based on boxes and varying timesteps. For each configuration, the computed probability (column *Probability*), the execution time (*Time*), the number of nodes in the search tree (*#Nodes*), the maximum search tree depth (*Search tree depth*) and the memory usage at the end of analysis (*Memory usage*) is shown. Runs which exhibit Zeno-behavior are marked with *Zeno-behavior*.

T_G	Probability	Time	#Nodes	Search tree depth	Memory usage
(Box, timestep 1), (Box, timestep $1/10^2$)					
10	[0, 0]	0.9 ms	3	2	35 MB
70	-	*Zeno-behavior*	-	-	-
100	-	*Zeno-behavior*	-	-	-
110	-	*Zeno-behavior*	-	-	-
120	-	*Zeno-behavior*	-	-	-
(Box, timestep $1/10$), (Box, timestep $1/10^3$)					
10	[0, 0]	0.8 ms	3	2	35 MB
70	$[1.11984 \times 10^{-5}, 1.11984 \times 10^{-5}]$	6400 ms	1811	11	335 MB
100	-	*Zeno-behavior*	-	-	-
110	-	*Zeno-behavior*	-	-	-
120	-	*Zeno-behavior*	-	-	-
(Box, timestep $1/10^2$), (Box, timestep $1/10^4$)					
10	[0, 0]	0.006 s	3	2	0.04 GB
70	$[1.11984 \times 10^{-5}, 1.11984 \times 10^{-5}]$	10.0 s	1795	11	0.50 GB
100	$[1.11984 \times 10^{-5}, 1.11984 \times 10^{-5}]$	62.9 s	28 185	16	2.2 GB
110	$[2.81861 \times 10^{-4}, 2.81861 \times 10^{-4}]$	1079.5 s	57 623	17	50 GB
120	$[2.81861 \times 10^{-4}, 2.81861 \times 10^{-4}]$	1142.6 s	96 867	19	52 GB

compute a trivial probability interval. Unsurprisingly, the execution time of this analysis is longer than for the simple analysis with timestep $1/10$ which computed a non-trivial probability interval. The actual result is, that the execution time is only about a quarter of what a complete analysis for timestep $1/10^3$ took while requiring less memory. $T_G = 100$ would have a similar result when using timesteps $1/10^2$, $1/10^3$, but even using timesteps $1/10^2$, $1/10^4$ is about 8-times faster than doing a full computation with timestep $1/10^3$ and only uses an eighth of the memory. Time-horizons $T_G = 110$ and $T_G = 120$ paint a similar picture, while the analysis for $T_G = 110$ and timesteps $1/10^2$, $1/10^4$ is slightly slower than the simple algorithm for timestep $1/10^3$, it is able to obtain a trivial probability interval. In the same case for $T_G = 120$, the extended algorithm even was a quarter faster despite using a better approximation for parts of the analysis. Again, memory consumption roughly follows execution time.

The conclusion is that the extended CEGAR-based approach is quite useful for this automaton, both in terms of faster analysis and of higher accuracy.

ProHVer All of these results were obtained using HyPro, the equivalent results for ProHVer are listed in Table 4.9. Compared to the results given by [pro11], the obtained execution times are about 2-4 times faster, again most likely due to faster hardware. For $T_G = 10$, both presented approaches can be about an order of magnitude faster than ProHVer, which is not relevant if the computation is finished in about a millisecond. The opposite holds for all larger selected time-horizons, in the best cases, ProHVer is about 5000-times faster than the best result obtained using the presented approaches in HyPro, in the worst cases ProHVer

Table 4.9: Experimental results for case study ProHVer: Autonomous Lawn-Mower in Section 4.3 using ProHVer with analysis parameters provided by [pro11] which in this case are the default parameters for PhaVer. For each configuration, the computed probability upper bound (column *Probability upper bound*), the time to build the abstraction (*Time in* s) and the number of states in the generated abstraction (*Abstract states*) is shown.

T_G	Probability upper bound	Time in s	Abstract States
10	0	0.002	4
70	$1.119\,85 \times 10^{-5}$	0.460	632
100	$1.119\,85 \times 10^{-5}$	2.6	3022
110	$2.818\,67 \times 10^{-4}$	15.4	9076
120	$2.818\,67 \times 10^{-4}$	28.9	12 660
130	$2.818\,67 \times 10^{-4}$	102.2	25 962
140	$2.818\,67 \times 10^{-4}$	268.0	38 830

Table 4.10: Experimental results for case study ProHVer: Autonomous Lawn-Mower in Section 4.3 using the simple approach with unbounded-analysis (Section 3.3) implemented in HyPro. All results were obtained using a box-representation and a timestep of $1/100$ with time-horizon $T_G = 130$. For each configuration, the computed probability (column *Probability*), the execution time (*Time in* s), the number of nodes in the search tree (*#Nodes*) and the maximum search tree depth (*Search tree depth*) is shown.

Max. jumps	Probability	Time in s	#Nodes	Search tree depth
7	$[0, 1]$	0.62	511	9
8	$[7.8125 \times 10^{-10}, 1]$	0.96	1019	10
9	$[1.040\,73 \times 10^{-5}, 1]$	1.9	1807	11
10	$[1.040\,73 \times 10^{-5}, 1]$	3.8	3383	12
11	$[1.040\,73 \times 10^{-5}, 1]$	6.5	6535	13
12	$[1.040\,73 \times 10^{-5}, 1]$	12.8	12 839	14
13	$[1.040\,73 \times 10^{-5}, 1]$	30.6	25 447	15
14	$[1.115\,16 \times 10^{-5}, 1]$	59.8	45 107	16
15	$[1.833\,36 \times 10^{-4}, 1]$	80.3	59 445	17
16	$[1.833\,36 \times 10^{-4}, 1]$	118.1	88 121	18
17	$[1.833\,36 \times 10^{-4}, 1]$	176.4	145 921	19
18	$[1.833\,36 \times 10^{-4}, 1]$	254.6	260 625	20
19	$[1.833\,36 \times 10^{-4}, 3.130\,12 \times 10^{-4}]$	488.6	260 625	20

can compute results for time-horizons which would either take too long or require too much memory in HyPro. These long runtimes are a result of a large search tree due to a constant branching factor of 2 in every jump. It is probable that fixed-point detection provides a benefit when two paths that follow the same trajectory are combined and only analyzed once. We suppose this is the reason for ProHVer's lead.

Unbounded Analysis Even using the unbounded extension from Section 3.3 does not drastically improve results. In Table 4.10 an exemplary selection of varying maximum jump-depths for a fixed approximation and fixed time-horizon $T_G = 130$ is presented. These results do not drastically vary for different approximations or time-horizons. While the CEGAR-based improvement may be able to slightly improve the results, it has no larger effect on the appearing phenomena and has thus not been used. Two main results can be derived from this table. Firstly, the computed probability intervals always stay the same or get tighter. This is an obvious consequence of this technique. The second effect is specific to this automaton, the probability intervals only change every few increases in jump-depth and every change is

very small. Due to the nature of this automaton, every time the grass-boundary is reached, two possible path-continuations are available. After a few direction-changes at the boundaries, each path will have only a very small contribution to the probability result. Whenever the probability interval changes slightly, a few paths will have reached the flowerbed. But since their contribution is so small, only slight changes are observable. Additionally, most realizable paths of the lawn-mower are generally headed in the same direction at any time. If most are driving away from the flowerbed, the mower will reach the flowers for a few jumps, i.e. direction changes at the grass-border. The third result is a consequence that only a limited number of paths for the lawn-mower reach the flowers before the time-horizon is reached. Therefore, for most paths the decision if they have the chance to chew up the flowers is determined right before the time-horizon is reached and thus the probability interval has the largest change when the search tree is fully computed for the given time-horizon. The remaining probability interval is the consequence of paths where the mower either drove close by the flowerbed where it is not determined whether they steer clear or hit the flowers or which reached the time-horizon close to reaching the flowers, where it is again unclear whether the flowers are safe or are reached.

4.4 ProHVer: Thermostat

The last case study for PHAs by [pro11] is an extension of one variant of the classic thermostat system [ACH+95]. The system still has two distinct locations which are responsible for heating and cooling. The temperature is monitored by a temperature sensor, which fails with a probability of 5 % when checking the temperature. Every sensor failure is declared as an unsafe state and therefore the interpretation of the probability result is the probability of a sensor failure (within a fixed time-horizon). Unfortunately, this case study relies on non-deterministic behavior to choose between checking the temperature and initiating cooling, which can't be modeled in a meaningful way using an urgent-priority-scheduler. The direct usage of an urgent-scheduler will alternate between heating and cooling without ever checking the temperature and thus compute a probability of 0 of having a sensor failure.

4.5 Refinement Demonstration

The emphasis for the previous section has been on the performance of the presented techniques. The next sections each explore a specific part of the algorithms using constructed examples. For this section, the aim is to examine the refinement process as described in Sections 2.8 and 3.2 in a step-by-step manner. The PHA used is listed in Figure 4.4 and based on the automaton of Section 4.1, with only slight changes in constants to make for a simpler graph when plotting reachable states and a fixed time-horizon $T_G = 11$. The unsafe states are as for Section 4.1 all states in location *error*. Additionally, a set of unsafe states is added for location *l*, following an idea already briefly discussed in Section 2.5, all states in location *l* with a height between 3.5 m and 7 m and a velocity between $-1\,\mathrm{m\,s^{-1}}$ and $1\,\mathrm{m\,s^{-1}}$ are declared unsafe, since it is known that if the ball reaches that configuration, it is going to break a nearby flower pot.

Simple Algorithm Before dissecting the process for the improved algorithm, a reference is obtained using the simple approach. Although not relevant, in this case, a box-approximation with a timestep of $1/10$ is used. Both maximum jump-depth and local time-horizon are chosen large enough in order not to inhibit the automaton. The complete plot of all reachable states

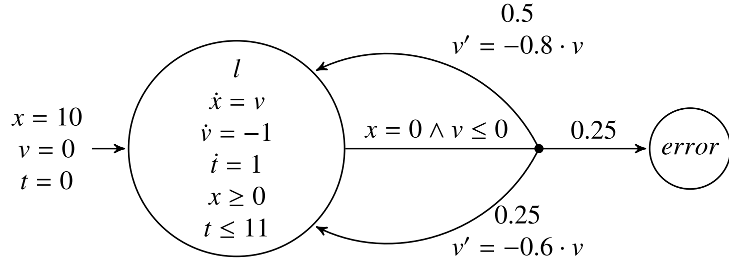
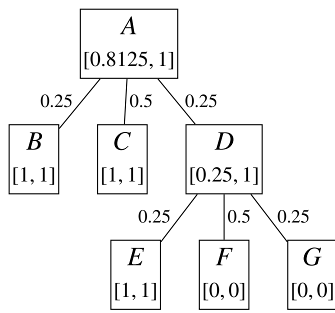
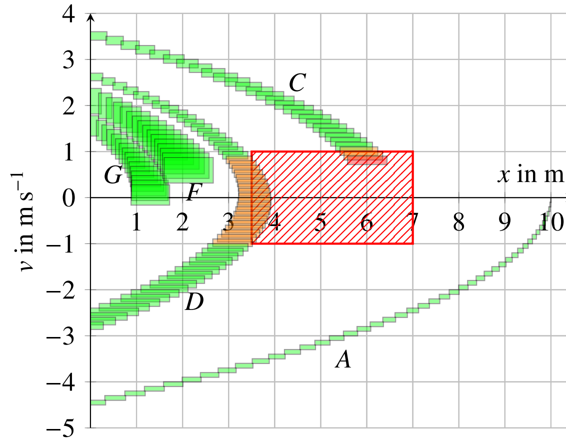


Figure 4.4: The probabilistic hybrid automaton for case study Refinement Demonstration in Section 4.5.



(a) Search tree, the probabilities at the edges is the probability of taking the transition leading to this node. The probability intervals below each node are the final probabilities after termination.

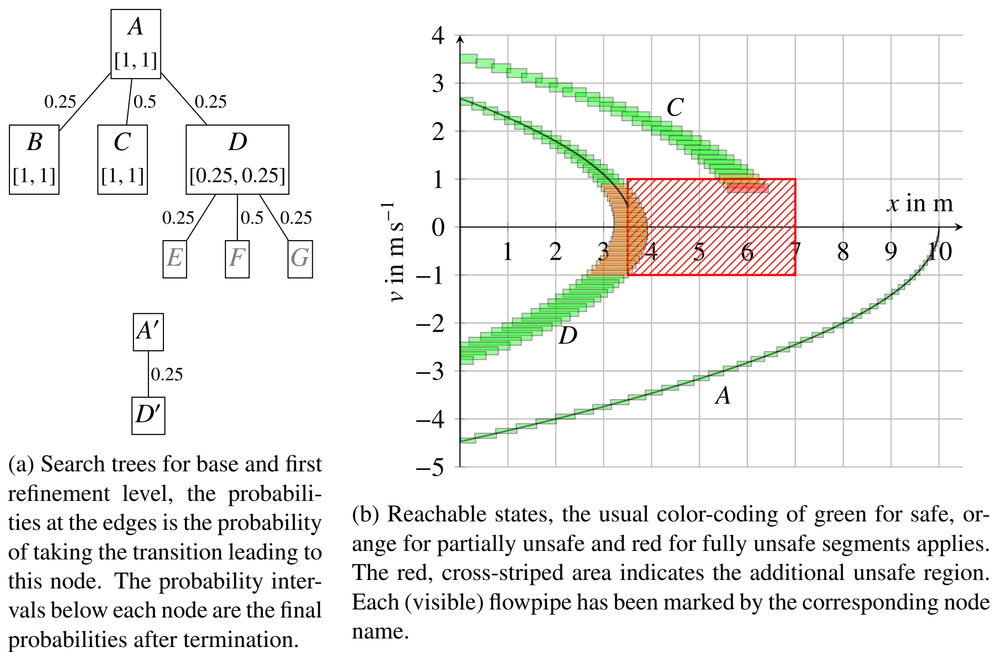


(b) Reachable states, the usual color-coding of green for safe, orange for partially unsafe and red for fully unsafe segments applies. The red, cross-striped area indicates the additional unsafe region. Each (visible) flowpipe has been marked by the corresponding node name.

Figure 4.5: Search tree and reachable states obtained using the simple algorithm (Section 3.1) for PHA of case study Refinement Demonstration in Section 4.5.

projected onto their values for x and v is given in Figure 4.5b. The additional dimension t has no further relevance as its main use was to limit the execution of the automaton. Furthermore, the search tree which is constructed during analysis is presented in Figure 4.5a as well. Below the main steps during the computation of these reachable states and the subsequent probability computation are outlined:

1. Root node A is created.
2. Node A is processed, during which first the flowpipe for node A is constructed and afterward the three successor-nodes B, C, D corresponding to each transition of the single transition-distribution are created.
3. Node B is processed. This is directly finished since B is the node which transition moved to location *error*. No flowpipe segments are visible.



(a) Search trees for base and first refinement level, the probabilities at the edges is the probability of taking the transition leading to this node. The probability intervals below each node are the final probabilities after termination.

(b) Reachable states, the usual color-coding of green for safe, orange for partially unsafe and red for fully unsafe segments applies. The red, cross-striped area indicates the additional unsafe region. Each (visible) flowpipe has been marked by the corresponding node name.

Figure 4.6: Search trees and reachable states obtained using the extended algorithm (Section 3.2) for PHA of case study Refinement Demonstration in Section 4.5.

4. Node C is processed. During the construction of flowpipe C, a fully unsafe segment is encountered and processing is aborted early. The guard of the single transition-distribution is not satisfied by any so far computed segment. No successor-node is created.

5. Node D is processed. Some flowpipe segments are partially unsafe, but none are fully unsafe. Successor-nodes E, F, G are created.

6. Nodes E, F, G are processed. E is again the simple case with location *error*, and F and G violate the global time-horizon before anything interesting occurs.

7. The last step is to recursively compute the probability of reaching an unsafe state, the final result is $[0.8125, 1]$.

The reason why this example results in a non-trivial probability interval when all intervals for Section 4.1 were trivial is the additional unsafe region or more precisely the existence of partially unsafe flowpipe segments.

CEGAR-based Algorithm The most interesting part of the analysis run is the processing of node D, a partial intersection with the unsafe state set is encountered, but due to a lack of precision, no decision can be made. This is the exact point where the improved algorithm makes a difference. All analysis parameters are kept the same, only an additional box-approximation with timestep $1/1000$ is used for the first refinement-level. Again, both the reachable states and the search tree have been plotted in Figure 4.6 to accompany the outline of the main analysis steps below:

1. Root node A is created.

2. Node A is processed, during which first the flowpipe for node A is constructed and the three successor-nodes B, C, D corresponding to each transition of the single transition-distribution are created.

3. Node B is processed. This is directly finished since B is the node which transition moved to location *error*. No flowpipe segments are visible.

4. Node C is processed. During the construction of flowpipe C, a fully unsafe segment is encountered and processing is aborted early. The guard of the single transition-distribution is not satisfied by any so far computed segment. No successor-node is created.

5. Node D is processed. Some flowpipe segments are partially unsafe, but none are fully unsafe. Successor-nodes E, F, G are created.

6. Since a partially unsafe flowpipe segment was encountered, the path to this encounter is refined with a better approximation. Thus, node A' with its higher precision flowpipe and node D' with its higher precision flowpipe are constructed. During the flowpipe computation for D', a fully unsafe segment is encountered. This information is propagated back into the base-level search tree for node D. Nodes E, F, G now became irrelevant.

7. The last step is to recursively compute the probability of reaching an unsafe state, the final result is $[1, 1]$.

This analysis results in a higher accuracy compared to the simple approach. The main point is that for a partially unsafe flowpipe segment, a better approximation can provide more information. Additionally, even though the same result can be obtained with a simple analysis-run completely in the better approximation, the better approximation was not needed for node C. This on-demand-approach can use only coarser approximations whenever they already provide expressive enough results.

4.6 Errors from Over-approximations

One of the assumptions described as part of the Algorithm Assumptions in Section 3 was that each transition enabled in a used over-approximation has to be enabled for the actual automaton as well. It is not allowed for a transition to be only enabled in an approximation. This section will exemplify this claim using a constructed example for clarity. The PHA as presented in Figure 4.7 is almost the same as for the previous Section 4.5, but now the global time-horizon $T_G = 10$ and an additional transition-distribution is introduced. For all states in location l where the ball has a height between 6.5 m and 7 m and a velocity between $-0.5\,\mathrm{m\,s^{-1}}$ and $0.5\,\mathrm{m\,s^{-1}}$, a transition can go to location *error* with a probability of 0.5 and back to location l while also increasing the balls height by 2 m. As previously mentioned, this additional jump does not have a meaningful physical interpretation. For the moment, the unsafe states are all states in location *error*.

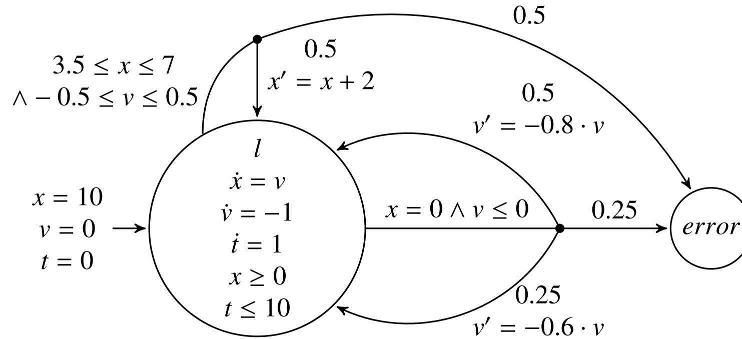
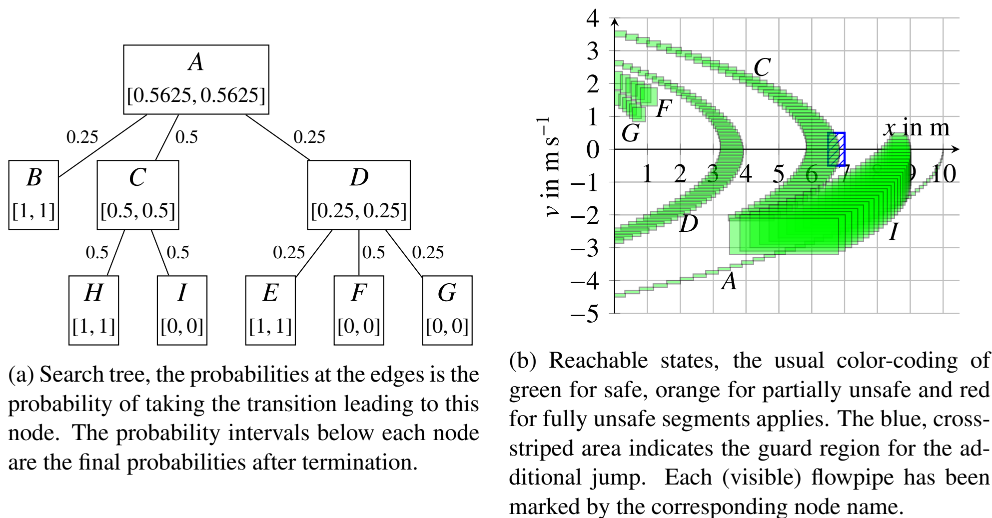


Figure 4.7: The probabilistic hybrid automaton for case study Errors from Over-approximations in Section 4.6.



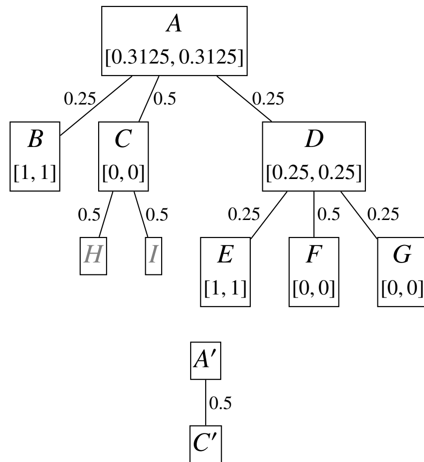
(a) Search tree, the probabilities at the edges is the probability of taking the transition leading to this node. The probability intervals below each node are the final probabilities after termination.

(b) Reachable states, the usual color-coding of green for safe, orange for partially unsafe and red for fully unsafe segments applies. The blue, cross-striped area indicates the guard region for the additional jump. Each (visible) flowpipe has been marked by the corresponding node name.

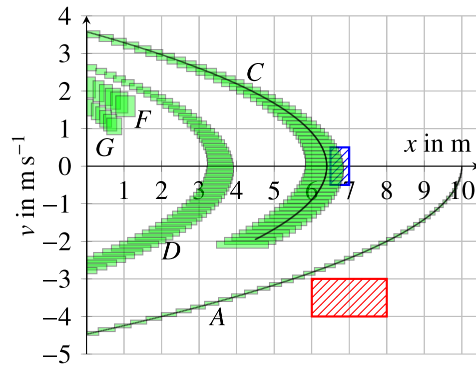
Figure 4.8: Search tree and reachable states obtained using the extended algorithm (Section 3.2) for PHA of case study Errors from Over-approximations in Section 4.6.

General Case In this section, all analysis runs will be made with the same configuration, although the exact details are not relevant for the presented phenomena, the parameters are listed here for completeness. All runs use the advanced algorithm with an approximation-list consisting of two box-approximations with timesteps of $1/10$ and $1/1000$. The maximum jump-depth and local time-horizon are chosen large enough to not inhibit the behavior of the automaton. For presentation of the results uses the same structure as the previous section with a plot of all reachable states projected onto dimension x and v in Figure 4.8b and the search tree(s) in Figure 4.8a. Following this is an outline of the main steps during the computation:

1. Root node A is created.
2. Node A is processed, during which first the flowpipe for node A is constructed and afterward the three successor-nodes B, C, D corresponding to each transition of the single transition-distribution are created.



(a) Search trees for base and first refinement level, the probabilities at the edges is the probability of taking the transition leading to this node. The probability intervals below each node are the final probabilities after termination.



(b) Reachable states, the usual color-coding of green for safe, orange for partially unsafe and red for fully unsafe segments applies. The blue, cross-striped area indicates the guard region for the additional jump, whereas the equally marked red region denotes the additional unsafe region. Each (visible) flowpipe has been marked by the corresponding node name.

Figure 4.9: Search trees and reachable states obtained using the extended algorithm (Section 3.2) for PHA of case study Errors from Over-approximations in Section 4.6 with additional unsafe region $6\,\mathrm{m} \leq x \leq 8\,\mathrm{m}, -4\,\mathrm{m\,s^{-1}} \leq v \leq -3\,\mathrm{m\,s^{-1}}$.

3. Node B is processed. This is directly finished since B is the node which transition moved to location *error*. No flowpipe segments are visible.

4. Node C is processed. After constructing the flowpipe C, some segments satisfy the guard of the additional jump, and the corresponding two successor-nodes, H for the transition to location *error* and I back to location l with additional height, are created. The global time-horizon is reached before the ball reaches the ground at height 0 m. But even then, the urgent-priority-scheduler would prefer the other, earlier enabled transition-distribution.

5. Node H is in location *error* and thus not interesting.

6. Node I is the node after the transition where the ball gained 2 m in height, the flowpipe is constructed as usual without satisfying any other guard before reaching the global time-horizon.

7. Node D is processed. Successor-nodes E, F, G are created and processed as previously.

8. The last step is to recursively compute the probability of reaching an unsafe state, the final result is $[0.6875, 0.6875]$.

Corner Case: Fault Detection Without any additional information, it is very hard to suspect something wrong. However, exact computation would reveal that the flowpipe of node C does not satisfy any guard of either transition. This might be revealed by an analysis using a better approximation (e.g. a box-approximation with timestep $1/1000$), but also by accident later in the analysis process. This was already briefly mentioned in Dealing with Detected Assumption Violations at the end of Section 3.2. Now, this is covered alongside an example. The previous example and PHA from Figure 4.7 are completely reused. The only change is an additional unsafe region in location l, all states in location l where the ball has a height between 6 m and 8 m and a velocity between $-4\,\mathrm{m\,s^{-1}}$ and $-3\,\mathrm{m\,s^{-1}}$ are declared unsafe. Again, both the reachable states and the search tree have been plotted in Figure 4.9 to accompany the outline of the main analysis steps below:

1. Root node A is created.
2. Node A is processed, during which first the flowpipe for node A is constructed and afterward the three successor-nodes B, C, D corresponding to each transition of the single transition-distribution are created.
3. Node B is processed. This is directly finished since B is the node which transition moved to location *error*. No flowpipe segments are visible.
4. Node C is processed. After constructing the flowpipe C, some segments satisfy the guard of the additional jump, and the corresponding two successor-nodes, H for the transition to location *error* and I back to location l with additional height, are created. The global time-horizon is reached before the ball reaches the ground at height 0 m. But even then, the urgent-priority-scheduler would prefer the other, earlier enabled transition-distribution.
5. Node H is in location *error* and thus not interesting.
6. Node I is the node after the transition where the ball gained 2 m in height, the flowpipe is constructed and partially unsafe segments are encountered.
7. The path to and including node I is refined using the better approximation. During this process, node A' and C' are created and processed. When testing the flowpipe of C' for the guard of the additional jump, it is found to be not enabled anymore. Thus, it was falsely enabled in the coarser approximation and node H and I are removed from the search tree.
8. Node D is processed. Successor-nodes E, F, G are created and processed as previously.
9. The last step is to recursively compute the probability of reaching an unsafe state, the final result is $[0.3125, 0.3125]$.

Since the additional unsafe region isn't even reached in the final search tree, both analysis runs should compute the same result. With this result, it is obvious that the previous result was incorrect, but this is not easily determinable when only the previous one is obtained. This also shows how hard it is to detect such occurrences in general. While the additional unsafe region accidentally resolved this issue in this constructed example, this exact behavior does exist for relevant automata as well. Unfortunately, this is only a strict subset and many cases are not detected in this way. In all such situations, the computed results need to be invalidated manually. There exist more ways to combat this problem, two of which are described in Section 5.1.

Chapter 5

Conclusion

This work presents two algorithms for the (bounded) analysis of probabilistic hybrid automata based on techniques for non-probabilistic hybrid automata. Additionally, a variant for unbounded analysis is described. First, hybrid automata were introduced, which were afterward extended to probabilistic hybrid automata with a discussion of the semantics. Subsequently, the analysis of non-probabilistic hybrid systems was covered before laying the groundwork for the analysis of probabilistic hybrid systems. The main part of this work is the presentation of two algorithms for the bounded analysis of PHA. The first being a simple algorithm while the second one improves the first algorithm through CEGAR-based ideas. Furthermore, an approach to unbounded analysis is covered. In the last part, both algorithms and the extension to unbounded analysis were implemented in HyPro and tested against the existing analysis tool ProHVer before exploring the behavior of the presented algorithms in two constructed examples.

The comparison of HyPro and ProHVer revealed both advantages and disadvantages in both tools. Since the effect of analyzing a probabilistic hybrid system is rather small compared to the analysis of a non-probabilistic hybrid system, the main differences in performance is a result of the different algorithms and improvements implemented in HyPro and the underlying tool for ProHVer, PHAVer. Here it shows, that in general HyPro manages to compute equally or even more accurate results in less time for less complex systems while ProHVer is able to analyze systems HyPro either failed to analyze or took far longer.

The main advantage to the presented approach in HyPro is the computation of both a lower and an upper bound to the probability intervals of reaching an unsafe state while ProHVer only provides an upper bound. Additionally, the extension to unbounded analysis can compute good approximations for systems where a large amount of the probability mass is determined within a limited number of jumps and time from the initial state.

One of the biggest disadvantages to the presented approach is its limitation to a fixed scheduler and the resulting problem with transitions which are only enabled in over-approximations. ProHVer considers all possible schedulers and avoids this type of problem completely.

The last advantage to HyPro is not based on the presented algorithms, but already existing functionality. HyPro incorporates several different approximations and specialized analysis functions for subtypes of hybrid systems. A probabilistic singular system therefore can make use of specialized analysis functions. This allows HyPro to be more future-proof compared to ProHVer, which uses the (discontinued) tool PHAVer.

5.1 Future Work

There exist a few ways to continue and improve this work. Most of which have already been briefly mentioned in previous chapters.

Transition Simulation This is a technique to resolve the problem of jumps which are only enabled in over-approximations as outlined in Algorithm Assumptions at the beginning of Chapter 3. For each taken jump, a simulation of the path to this state is made to prove that at least one state exists in which this transition is actually enabled.

Different Schedulers The current algorithms assume a fixed urgent-priority-scheduler. To allow the modeling of different systems with different behavior, different schedulers are needed.

Optimal Scheduler An alternative to the previous suggestion of Different Schedulers is to minimize and maximize probabilities, which will effectively always consider all possible path continuations. This was already described in Section 2.3 for Markov decision processes. The same idea is also applicable to probabilistic hybrid automata. A scheduler resolves non-determinism. A PHA has the non-deterministic choice between time-evolution, a transition-distribution, and if applicable a reset from a set of possible resets for the chosen transition-distribution. The concept of an optimal scheduler would now evaluate each available option separately and combine the resulting probability intervals. The same idea applies if more than one initial state is present. As a side effect, this technique also solves the main problem with transitions which are only enabled in over-approximations, since the computed probability intervals are no longer incorrect and just less exact. Therefore, this resolves the need for the previous suggestion Transition Simulation. While that technique can still improve results in certain cases, it would not be required to always obtain correct results.

Dynamic Bounds for Unbounded Analysis Currently, the presented approach to unbounded analysis computes reachable states up to fixed bounds, both in terms of the maximum jump-depth and the local time-horizon. At least the maximum jump-depth could benefit from a dynamic bound that stops the current path once a certain accuracy has been reached. This would allow the analysis to spend more time on paths that strongly affect the resulting probability interval and less time on paths that only have a slight impact. A path would be stopped when its effective probability mass is below a certain threshold. Alternatively, an approach similar to iterative-deepening may be employed to prioritize paths which promise the largest improvement in accuracy and analysis finishes once a certain overall accuracy has been reached.

Refinement Heuristics The refinement strategy as described in Section 3.2 is to refine as soon as a flowpipe with partially unsafe but no fully unsafe segments is encountered. In certain situations, e.g. if all path continuations have a fully unsafe segment after the next jump, the refinement is not necessary anymore, and it would have been more performant to postpone this refinement step. There are a few similar situations where choosing a different refinement should be preferred over another refinement. If it is possible to define more advanced rules when a refinement should occur, a few unnecessary refinements might be able to get skipped.

Fixed-Point Detection HyPro presently does not support the detection of fixed-points [Sch19], that is states which were already analyzed previously. If this feature is eventually added, the computed search trees do not satisfy the strict-tree property anymore. This requires solving a set of linear equations to compute the final probability of reaching an unsafe state. A simple approach is to construct a Markov decision process based on the search tree and apply common tools for the analysis of probabilistic systems to it, e.g. Storm [HJK+20]. A reference implementation of this would be ProHVer.

Parallelization Technically HyPro already supports the parallel analysis of different, unrelated nodes in the search tree [Sch19]. The queue-based approach naturally facilitates such a procedure. Some corner cases, e.g. when refinements provide more accurate information to nodes earlier in the path still need to be handled. The most obvious case is when nodes get removed from the search tree because they were created for a transition which is later found to be only enabled in a over-approximation.

Extension to Different Subtypes of Hybrid Automata All presented algorithms focus on the extension of non-probabilistic hybrid automata to the probabilistic setting. The core analysis functions for time-evolution, computing enabled transition-distributions and applying transition-resets are inherited from hybrid automata with at most minimal changes. Therefore, the presented approaches can easily be applied to other subclasses of probabilistic hybrid automata, for example, probabilistic rectangular automata or probabilistic singular automata. HyPro already provides specialized, more efficient analysis functions for some subtypes. The application of the presented approach to these analysis functions makes for a relatively easy extension, which still retains all benefits from using more specialized analysis functions.

Extension to Stochastic Hybrid Automata Stochastic hybrid automata are an extension to probabilistic hybrid automata where the discrete probabilistic behavior is augmented by continuous probabilistic behavior [FHH+11]. This is usually done through the use of continuous probability distributions in reset functions. [FHH+11] also demonstrated a technique to approximate stochastic hybrid automata by probabilistic hybrid automata, which can then in turn be analyzed by the presented algorithms.

Bibliography

[ACH+95] Rajeev Alur, Costas A. Courcoubetis, Nicolas Halbwachs, Thomas A.A. Henzinger, Peihsin Ho, Xavier Nicollin, Alfedo M. Olivero, Joseph Sifakis, and Sergio Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AS20a] Erika Ábrahám and Stefan Schupp. Lecture Notes for lecture Modeling and Analysis of Hybrid Systems, RWTH Aachen University, 2020.

[AS20b] Erika Ábrahám and Stefan Schupp. Slides for lecture Modeling and Analysis of Hybrid Systems, RWTH Aachen University, 2020.

[ava10] AVACS case studies. `http://www.avacs.org/fallstudien/`, 2010.

[FHH+11] Martin Fränzle, Ernst Moritz Hahn, Holger Hermanns, Nicolás Wolovick, and Lijun Zhang. Measurability and Safety Verification for Stochastic Hybrid Systems. In *HSCC'11*, pages 43–52. ACM, 2011.

[Fre05] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In *HSCC'05, Proceedings*, volume 3414 of *LNCS*, pages 258–273. Springer, 2005.

[Gro18] Martin Grohe. Slides for lecture Formale Systeme, Automaten, Prozesse, RWTH Aachen University, 2018.

[HHH+19] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Křetínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models. In *TACAS'19*, volume 11429, pages 69–92. Springer, 2019.

[HJK+20] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The Probabilistic Model Checker Storm. *CoRR*, abs/2002.07080, 2020.

[HKHH13] Ernst Moritz Hahn, Marta Kwiatkowska, Arnd Hartmanns, and Holger Hermanns. Model Checking for Probabilistic Hybrid Systems, CPSWeek, 2013.

[Kam19] Benjamin Lucien Kaminski. *Advanced Weakest Precondition Calculi for Probabilistic Programs*. Dissertation, RWTH Aachen University, 2019.

[Kat20] Joost-Pieter Katoen. Slides for lecture Probabilistic Programming, RWTH Aachen University, 2020.

[pro11] ProHVer case studies. `https://depend.cs.uni-saarland.de/tools/prohver/casestudies/`, 2010/11.

[Sch19] Stefan Schupp. *State Set Representations and their Usage in the Reachability Analysis of Hybrid Systems*. Dissertation, RWTH Aachen University, 2019.

[Spr00] Jeremy Sproston. Decidable Model Checking of Probabilistic Hybrid Automata. In *FTRTFT'00*, volume 1926 of *LNCS*, pages 31–45. Springer, 2000.

[Sto02] Mariëlle Stoelinga. An Introduction to Probabilistic Automata. *Bulletin EATCS*, 78:176–198, 2002.

[Tho18] Sebastian Thomas. Lecture notes for lecture Diskrete Strukturen, RWTH Aachen University, 2018.

[ZSR+10] Lijun Zhang, Zhikun She, Stefan Ratschan, Holger Hermanns, and Ernst Moritz Hahn. Safety Verification for Probabilistic Hybrid Systems. In *CAV'10*, volume 6174 of *LNCS*, pages 196–211. Springer, 2010.

Appendix A

Extended Automaton Grammar for HyPro

HyPro's grammar for the specification of input automata needs to be extended to probabilistic hybrid automata. The updated grammar rule for a normal transition to be completely compatible in the probabilistic setting is given in Figure A.1. The additional grammar rule for specifying probabilistic transition-distributions is illustrated in Figure A.2. Both rules use the railroad diagram notation.

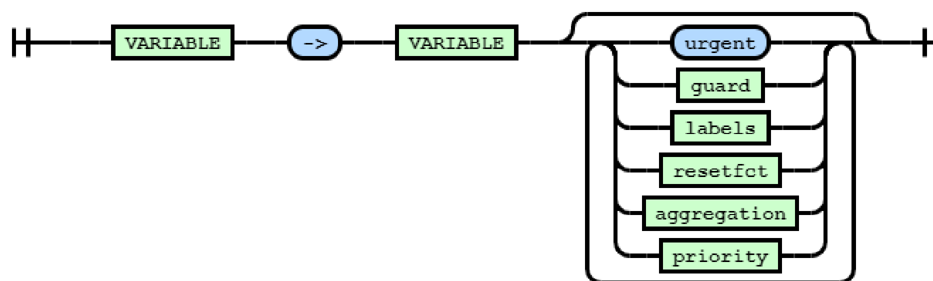


Figure A.1: Grammar rule for the specification of a non-probabilistic transition. The only change is the additional option to specify a priority to allow this as a shorthand notation for probabilistic transition-distribution with only a single transition. Both 'VARIABLE' fields denote locations, in the order of source and target location.

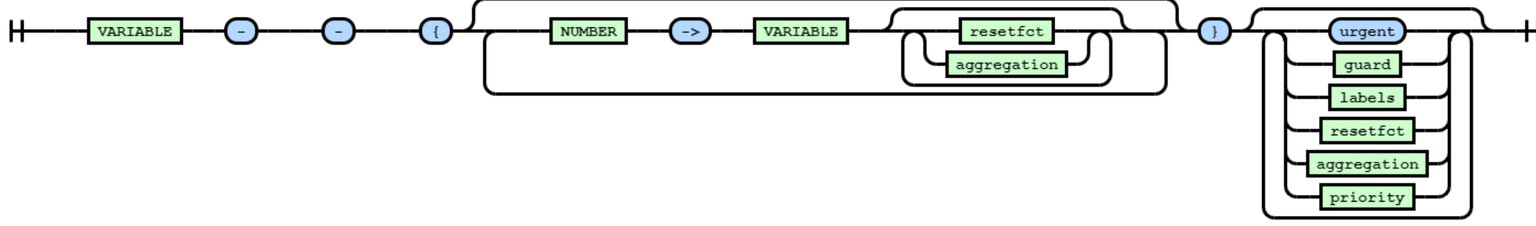


Figure A.2: Grammar rule for the specification of a probabilistic transition-distribution. The center part inside the curly braces specifies the list of transitions. Again, both 'VARIABLE' fields denote the source and target locations. The 'NUMBER' field in between indicates the probability of the particular transition. Each transition may define an individual reset function or aggregation setting. If unspecified the setting for the transition-distribution will be used (if provided).