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

Bachelor of Science Thesis

Extending flowpipe construction for compositional hybrid automata

Leander Behr

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

Additional Advisor:
Stefan Schupp

Aachen, October 30, 2020

RWTH AACHEN

Abstract

Faculty 1
Department of Computer Science

Bachelor of Science

Extending flowpipe construction for compositional hybrid automata

by Leander Behr

Hybrid systems are systems with both discrete and continuous behavior. To analyse them, a formal model, such as a hybrid automaton, must be created. Many computer controlled machines and devices can be modeled as hybrid systems, which makes their safety analysis increasingly interesting.

It occurs often that the behavior of a compositional system is of interest, where each part is a hybrid system in its own right. These compositional systems can lead to very large models, which makes them difficult to analyse. In this thesis we focus on compositional hybrid systems where the components interact at discrete points in time. For these systems we devise an algorithm that allows for their scalable analysis by using and composing the existing techniques for hybrid systems analysis.

Acknowledgements

First of all, thanks to Prof. Dr. Erika Ábrahám for giving me the opportunity to write this thesis and the effort she puts into her teaching. I also want to thank Prof. Dr. Thomas Noll for being my second supervisor. Especially helpful was Stefan Schupp who shared his knowledge and experience in a uniquely encouraging way, which I am very grateful for.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Compositional Hybrid Systems . . . 2
  - 1.2.1 Example System . . . 2
- 1.3 Related Work . . . 2
- 1.4 Our Approach . . . 3
- 1.5 Outline of the Thesis . . . 3

2 Flowpipe Construction for Hybrid Automata 5
- 2.1 Hybrid Automata . . . 6
  - 2.1.1 Fundamental Definitions . . . 6
  - 2.1.2 Composition . . . 8
    - Label Synchronization . . . 10
    - Jump Condition Synchronization . . . 10
  - 2.1.3 Linear Hybrid Automata . . . 11
- 2.2 Reachability Algorithm . . . 11
  - 2.2.1 Alternating Trajectories . . . 12
  - 2.2.2 General Reachability Algorithm . . . 12
  - 2.2.3 Over Approximation and Bounded Analysis . . . 13
  - 2.2.4 Concrete Reachability Algorithm . . . 14
    - Computation Tree . . . 14
- 2.3 Successor Computations . . . 15
  - 2.3.1 Continuous Successor Computation . . . 16
    - Valuation Set Representations and Operations . . . 16
    - Flowpipe Construction . . . 16
  - 2.3.2 Discrete Successor Computation . . . 18

3 Flowpipe Construction for Compositional Hybrid Automata 19
- 3.1 Compositional Computation Tree . . . 20
- 3.2 Precision . . . 20
- 3.3 Adapting Successor Operators . . . 21
  - 3.3.1 Continuous Successors . . . 21
  - 3.3.2 Discrete Successors . . . 22
    - Label Sets . . . 22
    - Computing the Set of Edges . . . 22
    - Computing the Valuation Sets . . . 24
    - Optimizing Edge Set Enumeration . . . 25
- 3.4 Urgent Edges . . . 26

4 Applications **27**
4.1 Robot Swarm Model 27
4.1.1 The Model 27
4.1.2 Analysis Parameters 27
4.1.3 Initial Results 28
4.1.4 Optimized Edge Set Enumeration 30
4.1.5 Conclusion 31
4.2 Hybrid Petri Nets 31
4.2.1 Petri Net Semantics 31
4.2.2 Hybrid Automaton Translation 32
4.2.3 Improved Translation 33

5 Conclusions **36**
5.1 Summary 36
5.2 Future Work 36
5.2.1 Stutter Jump Optimization 36
5.2.2 Explicit Time Dimension 37
5.2.3 Fixed Points 38
5.2.4 Advanced Discrete Synchronization 39
5.2.5 Extension to Discrete and One Step Reachability 39

Bibliography **40**

List of Figures

2.1 Production Robot . . . 7
2.2 Shared Variable Example . . . 9
2.3 Shared Variable Simulation . . . 10
2.4 Computation Tree . . . 15
2.5 First Segment Construction . . . 17

3.1 Compositional Flowpipe Construction . . . 21

4.1 Robot Swarm Model . . . 28
4.2 Hybrid Petri Net Example . . . 31
4.3 Discrete Place Translation . . . 32
4.4 Continuous Place Translation . . . 32
4.5 Discrete Transition Translation . . . 33
4.7 Improved Discrete Transition Translation . . . 34
4.6 Improved Discrete Place Translation . . . 34

5.1 Flowpipes with Stutter Optimization . . . 37

Chapter 1

Introduction

1.1 Background

When designing and building any kind of system or even just interacting with an existing one, the need to make predictions about its behavior may arise. Depending on the situation and what is at stake, different levels of confidence in those predictions will be required. Various techniques are available in different fields like testing or simulation. When a system is particularly critical however, formal verification of its behavior becomes attractive because it can give the strongest guarantees.

To be able to perform verification, the system in question has to be modeled in some formal language. The model may be discrete, like the states of an electric controller being modeled as states of an automaton. It may be continuous like a differential equation modeling the change of temperature in a reactor. The system may also have parts that are best modeled as discrete and others that are best modeled as continuous. Such a hybrid system is then best modeled using a formalism that accommodates for this.

We will be using hybrid automata for this task. They can be used to model a wide range of hybrid systems, such as networking setups, docking spacecraft or even the human heart [BHS17; Alt+18; And+17]. These models can then be analysed. Reachability analysis is one technique to prove that a model fulfills a specification. The specification must be expressed in terms of the states of the model. One may specify which states must be reached, which are allowed to be reached and which must not be reached. The set of reachable states is then computed and compared against the specification.

Depending on the type of hybrid automaton, the problem of computing the set of reachable states may be undecidable. To overcome this, an over or under approximation is computed and the behavior is only analysed for a bounded time frame. With an over approximation one can prove that a set of states definitely will not be reached while one can conversely show with an under approximation that a set of states will definitely be reached. We will work exclusively with over approximations. Properties like the absence of collisions can be modeled and verified like this.

To compute the set of reachable states both discrete an continuous successor states must be computed, i.e. states that are reachable from already known to be reachable states by either a discrete or continuous change. The method for computing the continuous successors that we will focus on is flowpipe construction. Its result also allows for the operations needed to compute discrete successors. This is described for example in [Fre16, Sec.2.2, 2.3].

1.2 Compositional Hybrid Systems

Hybrid systems often have multiple parts that are separate from each other but do interact in certain ways. This occurs, for example, when a controller with a controlled plant of some form is modeled. The system may also be comprised of many similar parts or agents. We will use such a system for examples from now on.

1.2.1 Example System

Assume we use robots in a production environment. A robot has continuous state like its position and battery level. To perform tasks it will likely have to go through multiple steps such as collecting materials, putting them into a machine and taking the product somewhere to store. These steps would be modeled as discrete states, or locations as they are called in hybrid automata.

We may then be interested in verifying that the robot completes its task in a certain amount of time and that its battery does not run out before it could reach a charging station. This specification can be given as states that should not be reached. The battery level should never drop to zero and, given we introduced a clock, the time taken should be smaller than some deadline until the goal state is reached.

The single robot we described so far would probably best be modeled as a single hybrid automaton. If we add multiple robots that interact with each other, we get a compositional hybrid system that is modeled by multiple hybrid automata. As we will see later, the hybrid automata modeling the robots implicitly define a single, compositional, hybrid automaton describing their combined behavior.

There are then different ways in which the robots may interact. The simplest is through discrete events like waiting on each other to finish a task or stop occupying a machine. This kind of interaction is what our work is focused on. They may also depend on each other's state at discrete points. For example by checking the location of other robots. This is modeled by shared variables. We will later briefly look at how this case can sometimes be reduced to the previous one. If their dependency is continuous, like their temperatures affecting each other, it is difficult to analyse them in any other way than explicitly constructing their composed model.

1.3 Related Work

The simplest way to analyse a compositional hybrid automaton is to explicitly construct its parallel composition, i.e. an automaton whose state space is the cartesian product of the state spaces of the composed automata (except for shared variables). The discrete part of this is similar to constructing the product automaton of multiple automata and thus also becomes exponentially more complex with the number of automata involved. One can then use existing algorithms to analyse the single resulting automaton. This construction has been implemented in the transformation tool Hyst [BBJ15]. The tool also offers the optimization of removing locations that are not connected to an initial location.

In [Fre05] and later [FZK04] abstractions of hybrid automata that preserve composability are explored. Then assume-guarantee reasoning is employed. As it is a form of compositional proof, it uses the property that results from the analysis of one component of a system can be used to find properties of the entire system.

The SpaceEx tool implements an analysis algorithm that accepts compositional hybrid automata as input without explicitly constructing the whole composition up front [Fre16, Sec.7].

1.4 Our Approach

Our goal is to design and implement an algorithm for the analysis of compositional hybrid automata that scales well with the number of automata involved. To be able to avoid combining their state spaces, we require that the synchronization between the automata is discrete. The main contribution lies in the composition and adaption of the continuous and discrete successor computations for single automata to obtain the successor states for the composed automaton. We implemented various optimizations of the handling of the discrete synchronization.

We consider, and try to avoid, two problems that may impede scalability. The growth of the number of locations and edges of the composed automaton and the growth of its dimension. We do this by exploring only the reachable parts of the composed automaton and by keeping the computations for each component separated from the others as much as possible.

The former problem may occur because the size of the composition of hybrid automata may be exponential in the number of automata. For example, the composition of n automata with 2 locations each has up to 2^n locations, if done naively. We combat this by lazily constructing the locations and edges that are discovered during the analysis and therefore constructing only the reachable subset of the control graph.

The severity of the latter problem depends on how well the underlying flowpipe construction scales with the dimension of the automaton. To avoid constructing a high dimensional flowpipe, we construct the flowpipe of each automaton separately and apply synchronizations explicitly. In [SNÁ17] it has been shown that reducing the dimension is an effective approach to speeding up computations.

The implementation builds upon flowpipe construction, but the algorithm could be adapted to be used with any over approximation of the set of reachable states that allows for the necessary operations and queries. That includes many variations of flowpipe construction but conceivably also different more precise and efficient procedures for rectangular hybrid automata or even fully discrete automata. The implementation is part of the HyPro library [Sch19] and uses the flowpipe construction algorithm implemented there.

1.5 Outline of the Thesis

First, in Chapter 2, we will describe the existing methods for reachability analysis of hybrid automata that our work depends on. To do so, we cover the fundamental concepts and definitions of hybrid automata and their composition in Section 2.1. Using those, we describe the reachability algorithm we will be extending and some of its properties in Section 2.2. The last Section, 2.3, contains the details of the algorithm's two main operations.

In Chapter 3, we describe our compositional algorithm in detail. We expand on its general structure and characteristics in sections 3.1 and 3.2. Following the same structure as before, Section 3.3 is a description of the adapted main operations of our algorithm.

To explore the strength and weaknesses of the developed algorithm, we apply it to two models in Chapter 4. The first, presented in Section 4.1, is a swarm of robots exhibiting emergent behavior. The next Section , 4.2, explores a simple translation of a hybrid Petri net to hybrid automata.

Finally, in Chapter 5 we discuss directions for further work and improvements to our approach, some of which arose from our observations in Chapter 4. We conclude the thesis with a summary of our findings.

Chapter 2

Flowpipe Construction for Hybrid Automata

After introducing a few notational conventions we will use, we describe the basic concept of hybrid automata and then present the reachability algorithm that our compositional algorithm builds on.

Notation 1 (Intervals)**.** For $S \in \{\mathbb{R}, \mathbb{N}\}$ and $l, u \in S$ we use the usual interval notation.

- $[l, u] := \{s \in S \mid l \leq s \leq u\}$
- $[l, u) := \{s \in S \mid l \leq s < u\}$
- $(l, u] := \{s \in S \mid l < s \leq u\}$
- $(l, u) := \{s \in S \mid l < s < u\}$

Notation 2 (Tuples)**.** Let $n, m \in \mathbb{N}$ with $n \leq m$ and let S_i be a set for all $i \in [1, m]$. Then $s \in S := S_1 \times \ldots \times S_n$ is a tuple. Analogously to $(S_1 \times \ldots \times S_n) \times (S_{n+1} \times \ldots \times S_m) = S_1 \times \ldots \times S_m$ we use $((s_1, \ldots, s_n), (s_{n+1}, \ldots, s_m)) = (s_1, \ldots, s_{n+m})$ and group values to signify their relationship. Additionally we use $x = (x)$ for any object x.

Notation 3 (Vectors)**.** Any tuple s may be interpreted as a column vector. We do this when the distinction between column and row vectors is relevant for arithmetic. When we then also write s elementwise, we use the transposition operator T to signify the column orientation, as in $s = (s_1, \ldots, s_n)^T$.

Notation 4 (Singleton Sets)**.** For any object x that is not a set we use x as the singleton set $\{x\}$ in set operations. For example $\{1, 2\} \cup 3 = \{1, 2, 3\}$ and $\{1, 2\} \cap 2 = \{2\}$.

Notation 5 (Multisets)**.** We use $\langle$ and $\rangle$ in place of $\{$ and $\}$ to denote a multiset. So $\langle 1, 2, 3, 2 \rangle$ contains 1 and 3 once and 2 twice and $\langle i \mod 3 \mid i \in [0, 5] \subseteq \mathbb{N} \rangle = \langle 0, 1, 2, 0, 1, 2 \rangle$.

Notation 6 (Quotient Sets)**.** For a set S and an equivalence relation $\sim$, we use $S/{\sim} := \{[s] \mid s \in S\}$ where $[s] := \{s' \in S \mid s' \sim s\}$ is the equivalence class of $s \in S$. When $\sim$ is applied to sets S, S', then $S \sim S'$ holds if and only if $s \in S$ implies that $s' \in S'$ exists with $s \sim s'$ and vice versa.

Notation 7 (Powerset)**.** For a set S we denote its power set by $\mathrm{Pow}(S) := \{S' \mid S' \subseteq S\}$. Another notation that is commonly used by other authors is 2^S.

Notation 8 (Objects and their Representation)**.** When denoting a mathematical object X, we will use $\widehat{X}$ to distinctly denote its representation. For different kinds of objects

we will define the relationship between abstract object and representation of the form $X := f(\widehat{X})$ for some function f.

If $\widehat{X}$ is defined in a context, we implicitly use $X := f(\widehat{X})$. If X is defined in a context, we implicitly use $\widehat{X} \in \{\widehat{X}' \mid f(\widehat{X}') = X\}$. We may for example define $\widehat{X} := A \in \mathbb{R}^{n\times n}, n \in \mathbb{N}$ with $X : \mathbb{R}^n \to \mathbb{R}^n, v \mapsto Av$. Here f maps matrices to their natural function equivalent.

2.1 Hybrid Automata

First we need definitions of hybrid automata, their semantics and their parallel composition. The definitions presented here are based on [Hen96].

2.1.1 Fundamental Definitions

Definition 1 (Hybrid Automaton)**.**
A hybrid automaton H is a tuple $(Var_H, Lab_H, Loc_H, Edge_H, Init_H, Flow_H)$. We additionally define $St_H := Val_H \times Loc_H$ where $Val_H := \mathbb{R}^d$ for the states space dimension $d \in \mathbb{N}$. The components are then defined as

- $Var_H \subseteq \mathbb{N}$, a set of d variables represented by indices from $\mathbb{N}$,
- Loc_H, a finite set of locations,
- Lab_H, a finite set of edge labels,
- $Edge_H \subseteq Loc_H^2 \times Lab_H \times \text{Pow}(Val_H^2)$, a set of edges between the locations with a label and a set of pairs of valuations.
- $Init_H \subseteq St_H$, a set of initial states,
- $Inv : Loc_H \to \text{Pow}(Val_H)$, assigning to each location $\ell \in Loc_H$ a set of valuations $Inv_H(\ell) \subseteq Val_H$,
- $Flow_H : Loc_H \to \text{Pow}(Map(\mathbb{R}_{\geq 0}, Val_H))$, assigning to each location $\ell \in Loc_H$ a set of functions $Flow_H(\ell) \subseteq Map(\mathbb{R}_{\geq 0}, Val_H)$.

The last component $cond \in \text{Pow}(Val_H^2)$ of an edge is often given as a *guard* and a *reset*. The guard constrains the valuations that enable the edge to be taken. The reset then describes how the enabling valuations are transformed before entering the target location.

It may be helpful to note that while we, for now, use the very general definition of $Flow_H$ given above, in practice the flow is often given in syntactical forms like systems of differential equations with limitations like linearity [Alt15, Sec.5] [Fre16, Sec.3.2].

Definition 2 (Discrete Successors)**.** Let H be a hybrid automaton, $s_1 := (v_1, \ell_1) \in St_H$, $s_2 := (v_2, \ell_2) \in St_H$ states and $e := (\ell_1', \ell_2', lab, cond) \in Edge_H$ an edge. Then $\xrightarrow{e}$ is the discrete successor relation via e and $s_1 \xrightarrow{e} s_2$ holds if and only if $\ell_1 = \ell_1'$, $\ell_2 = \ell_2'$ and $(v_1, v_2) \in cond$ and $v_2 \in Inv_H(\ell_2)$. We also define $\xrightarrow{\alpha} := \bigcup_{e \in Edge_H} \xrightarrow{e}$ to be the discrete successor relation via any edge.

Definition 3 (Continuous Successors)**.** Let H be a hybrid automaton, $s_1 := (v_1, \ell_1) \in St_H$, $s_2 := (v_2, \ell_2) \in St_H$ states and $r \in \mathbb{R}_{\geq 0}$. Then $\xrightarrow{r}$ is the continuous successor relation of duration r and $s_1 \xrightarrow{r} s_2$ holds if and only if

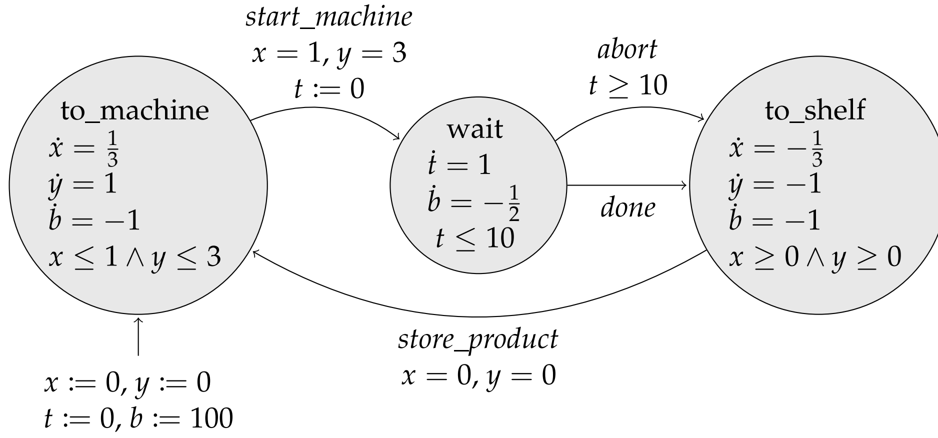


FIGURE 2.1: Hybrid Automaton modeling a single production robot.

- $\ell_1 = \ell_2$ and
- $f : \mathbb{R}_{\geq 0} \to Val_H \in Flow_H(\ell_1)$ exists with
  - $f(0) = v_1, f(r) = v_2$ and
  - $f(\varepsilon) \in Inv_H(\ell_1)$ for all $\varepsilon \in [0, r]$.

We also define $\xrightarrow{\tau} := \bigcup_{r \in \mathbb{R}_{\geq 0}} \xrightarrow{r}$ to be the continuous successor relation of any duration.

Finally the general successor relation is $\to := \xrightarrow{\alpha} \cup \xrightarrow{\tau}$ and a step from $s \in St_H$ to $s' \in St_H$ via the discrete or continuous successor relation is also called a jump or time step respectively.

Definition 4 (Trajectory)**.** A trajectory of a hybrid automaton H is a sequence $(s_i, \zeta_i)_{i \geq 0}$ with $s_i \in St_H$, $\zeta_i \in Edge_H \cup \mathbb{R}_{\geq 0}$ and $s_i \xrightarrow{\zeta_i} s_{i+1}$ for all $i \geq 0$.

- The trajectory is initial if and only if $s_0 \in Init_H$.
- A state $s \in St_H$ is reachable by $(s_i, \zeta_i)_{i \geq 0}$ in $i \in \mathbb{N}$ steps and $r \in \mathbb{R}_{\geq 0}$ time if and only if $s_i = s$ and $r = \sum_{j=0}^{i} d(\zeta_j)$ where

$$d(\zeta_i) := \begin{cases} \zeta_i, & \text{if } \zeta_i \in \mathbb{R}_{\geq 0}, \\ 0, & \text{if } \zeta_i \in Edge_H. \end{cases}$$

- The set of reachable states of H is the set R_H of states of H that are reachable by an initial trajectory of H.

Example 1 (Hybrid Automaton)**.** In Figure 2.1 you can see a hybrid automaton modeling the production robot we described in Section 1.2.1. Its single initial state is $((0, 0, 0, 100), \text{to_machine})$, assuming the variables are indexed in the order they are mentioned. Note that when a variable is omitted, it is implicitly declared to not change in flow and resets and to not be constrained in guards and invariants. The flow here allows each variable to change at a constant rate, i.e.

$$\begin{aligned} Flow_H(\text{to_machine}) = \{ & f_v \in \text{Map}(\mathbb{R}_{\geq 0}, Val_H) \mid v = (v_x, v_y, v_t, v_b) \in Val_H, \\ & f_v(t) = (v_x + \frac{1}{3}t, v_y + t, v_t, v_b - t) \text{ for } t \in \mathbb{R}_{\geq 0} \}. \end{aligned}$$

The invariant is $\{(v_x, v_y, v_t, v_b) \in Val_H \mid v_x \leq 1,\ v_y \leq 3\}$. The invariant and the jump condition $\{((v_x, v_y, v_t, v_b), (v'_x, v'_y, v'_t, v'_b)) \in Val_H^2 \mid v_x = 1,\ v_y = 3, v'_t = 0\}$ force a jump to the "wait" location once the position $(1, 3)$ is reached. Also, the value of the clock variable t is set to zero.

The robot then waits an unspecified amount of time until the machine is "done", but at most 10 time units, at which point the production is aborted. Then the robot moves back to the shelf at position $(0, 0)$, stores the product and repeats the cycle.

Example 2 (Trajectory)**.** A prefix of an initial trajectory of the hybrid automaton from Figure 2.1 is

$$((0,0,0,100),\text{to_m}) \xrightarrow{3} ((1,3,0,97),\text{to_m}) \xrightarrow{start_m} ((1,3,0,97),\text{wait}) \xrightarrow{6} ((1,3,6,94),\text{wait})$$
$$\xrightarrow{done} ((1,3,6,94),\text{to_s}) \xrightarrow{3} ((0,0,6,91),\text{to_s}) \xrightarrow{store_p} ((0,0,6,91),\text{to_m}).$$

Note that we can use labels here to identify edges because they are unique in this case. It is also worth noting that time steps can be split up into any number of shorter time steps here.

2.1.2 Composition

We will now define the parallel composition of hybrid automata. To do this we first introduce two extension functions on hybrid automata.

Definition 5 (Identity Jump Extension)**.** Let H be a hybrid automaton. Let $IdEdge_H := \{e_I(H, \ell) \mid \ell \in Loc_H\}$ with $e_I(H, \ell) := (\ell, \ell, I, cond)$, $cond := \{(v, v) \mid v \in Val_H\}$ and $I \notin Lab_H$. The identity jump extension of H is the hybrid automaton $\mathbb{E}_J(H)$, which is identical to H except for $Edge_{\mathbb{E}_J(H)} := Edge_H \cup IdEdge_H$.

So this extension adds a self loop that does not change the state of H to each location. We also call such self loops stutter jumps, including ones that are part of the original automaton. We will see below that these edges are necessary for synchronization with other automata.

Definition 6 (Variable Extension)**.** Let H be a hybrid automaton and $Var' \subseteq \mathbb{N}$ a set of variables represented by indices. Let $Var_{add} := Var' \setminus Var_H$ be the additional variables. We assume w.l.o.g. that $Var_H \cup Var' = \{x_1, \ldots, x_n, \ldots, x_m\}$ for some $n, m \in \mathbb{N}$ and $Var_H = \{x_1, \ldots, x_n\}$, $Var_{add} = \{x_{n+1}, \ldots, x_m\}$. The variable extension of H to $Var_H \cup Var'$ is the hybrid automaton $\mathbb{E}_V(H, Var') := H'$ with state space dimension $|Var' \cup Var_H|$, $Val_{H'}, St_{H'}$ as in Definition 1 and

- $Var_{H'} := Var_H \cup Var'$, $Loc_{H'} := Loc_H$, $Lab_{H'} := Lab_H$,
- $Edge_{H'} := \{(\ell_1, \ell_2, lab, \text{ext}(cond)) \mid (\ell_1, \ell_2, lab, cond) \in Edge_H\}$

  where $\text{ext}(cond) := \bigcup_{(v,v') \in cond} (v \times \mathbb{R}^{m-n}) \times (v' \times \mathbb{R}^{m-n})$.
- $Init_{H'} := \bigcup_{(v,\ell) \in Init_H} v \times \mathbb{R}^{m-n} \times \ell$,
- $Inv_{H'}(\ell) := Inv_H(\ell) \times \mathbb{R}^{m-n}$ for all $\ell \in Loc_H$,
- $Flow_{H'}(\ell) := \bigcup_{f \in Flow_H(\ell)} \{f_v : \mathbb{R}_{\geq 0} \to Val_{H'} \mid v \in \mathbb{R}^{m-n}, f_v(t) = (f(t), v)\}$.

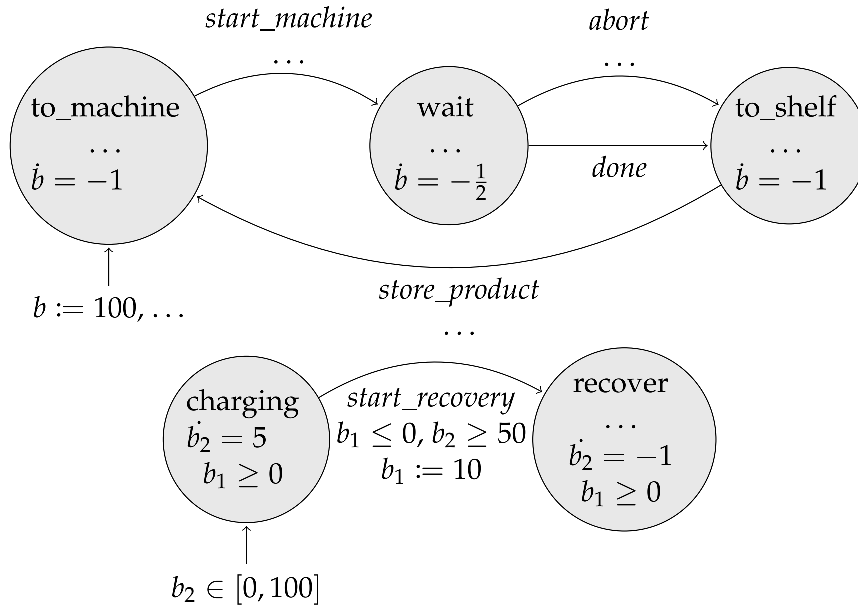


FIGURE 2.2: The automaton from Figure 2.1 with a backup automaton that checks its battery status, modeled using a shared variable.

Intuitively, the extension simply does not constrain the additional variables.

Definition 7 (Parallel Composition)**.** Let H_1, H_2 be hybrid automata and let $H_1' := \mathbb{E}_V(\mathbb{E}_J(H_1), Var_{H_2})$, $H_2' := \mathbb{E}_V(\mathbb{E}_J(H_2), Var_{H_1})$. Then their parallel composition is the hybrid automaton $H_1||H_2$ with

- $Var_{H_1||H_2} = Var_{H_1} \cup Var_{H_2}$,
- $Loc_{H_1||H_2} = Loc_{H_1} \times Loc_{H_2}$,
- $Lab_{H_1||H_2} = Lab_{H_1} \cup Lab_{H_2}$,
- $Edge_{H_1||H_2} \subseteq Loc^2_{H_1||H_2} \times Lab_{H_1||H_2} \times \mathrm{Pow}(Val^2_{H_1||H_2})$ with $((\ell_1, \ell_2), (\ell_1', \ell_2'), lab, cond) \in Edge_{H_1||H_2}$ if and only if there are
  - $(\ell_1, \ell_1', lab_1, cond_1) \in Edge_{H_1'}$ and
  - $(\ell_2, \ell_2', lab_2, cond_2) \in Edge_{H_2'}$

  with $cond = cond_1 \cap cond_2$ and
  - $lab = lab_1 = lab_2 \in Lab_{H_1} \cap Lab_{H_2}$ or
  - $lab = lab_1 \in Lab_{H_1}, lab_2 = I$,
  - $lab = lab_2 \in Lab_{H_2}, lab_1 = I$.
- $Init_{H_1||H_2} = Init_{H_1'} \cap Init_{H_2'}$,
- $Inv_{H_1||H_2}((\ell_1, \ell_2)) = Inv_{H_1'}(\ell_1) \cap Inv_{H_2'}(\ell_2)$ for all $(\ell_1, \ell_2) \in Loc_{H_1||H_2}$,
- $Flow_{H_1||H_2}((\ell_1, \ell_2)) = Flow_{H_1'}(\ell_1) \cap Flow_{H_2'}(\ell_2)$ for all $(\ell_1, \ell_2) \in Loc_{H_1||H_2}$.

This definition of parallel composition allows for synchronization in multiple ways. Notably, it does not allow behavior to be extended, i.e. the set of reachable states of the composition $R_{H_1||H_2}$, projected accordingly, is always a subset of R_{H_1} and R_{H_2}.

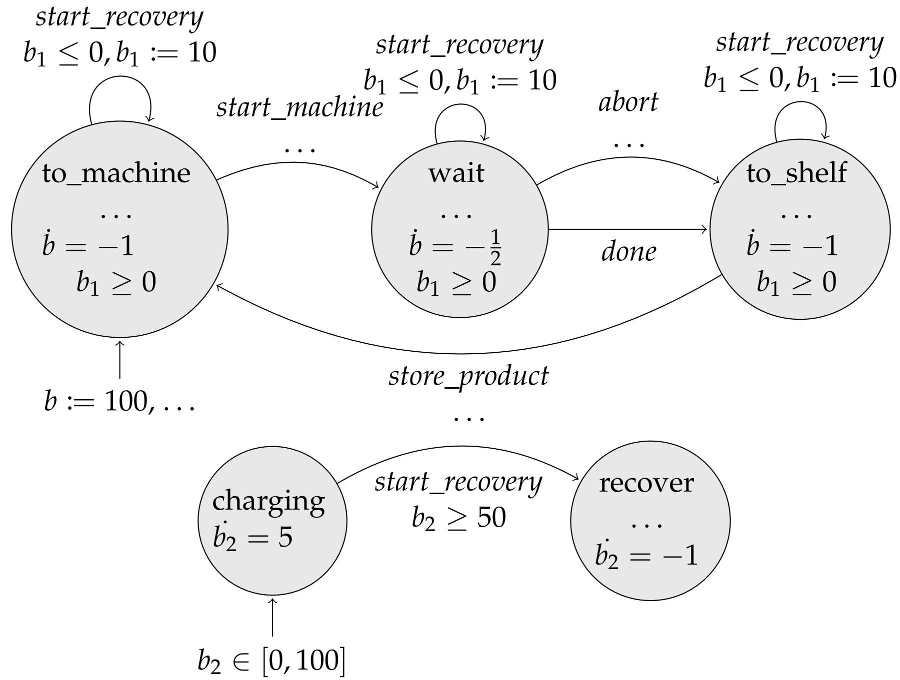


FIGURE 2.3: The automata from Figure 2.2 transformed to not share variables.

Label Synchronization

As mentioned in Section 1.4, we limit our analysis to compositional hybrid automata $H := H_1||H_n$, $n \in \mathbb{N}$ whose synchronization is discrete. We also require that the automata do not read or write each other's variables. Formally this means that $Var_{H_i} \cap Var_{H_j} = \emptyset$ holds for all $i, j \in [1, n]$. It is however worth mentioning another type of synchronization that may be transformed to meet this requirement.

Jump Condition Synchronization

Definition 8 (Unconstrained Variables)**.** Let H be a hybrid automaton and x_i the i-th variable in Var_H. The variable x_i is unconstrained by H if there is a hybrid automaton H' with $Var_{H'} = Var_H \setminus x_i$ such that $\mathbb{E}_V(H', x_i) = H$.

Let H_1, H_2 be hybrid automata. The behavior of $H_1||H_2$ can be simulated by $H_1'||H_2'$ with $Var_{H_1'} \cap Var_{H_2'} = \emptyset$ if H_1 and H_2 can be modified such that every variable $x \in Var_{shared} := Var_{H_1} \cap Var_{H_2}$ is unconstrained by either H_1 or H_2.

To achieve this, constraints over shared variables on edges can be removed and replaced by self loops with those constraints on every location of the other automaton. These self loops use same label as the changed edge so that they synchronize. Global invariants, i.e. invariants that are the same in all locations, can simply be moved to the other automaton.

These transformations are only possible if the constraints on the shared variables can be separated from those on other variables. Since we do not pursue this further, we will give an example and spare further formal definitions.

Example 3 (Simulating Jump Condition Synchronization)**.** In Figure 2.2 we see two automata that share a variable. The recovery robot has the constraint $b_1 \leq 0$ and

reset $b_1 := 10$ on its edge that models starting the recovery. To force the start of the recovery it adds the invariant $b_1 \geq 0$ in every location.

The invariant can simply be moved to the other automaton. Since the constraints $b_1 \leq 0$ and $b_2 \geq 50$ can be separated from each other, we can also replace the $b_1 \leq 0$ constraint and $b_1 := 10$ reset by self loops on each location of the other automaton with the *start_recovery* label. The result is shown in Figure 2.3.

2.1.3 Linear Hybrid Automata

Definition 1 is very general and does not specify any constraints on how a hybrid automaton may be given syntactically. Thus we need a more concrete definition to be able to perform analysis on hybrid automata. To that end, we now define Linear Hybrid Automata (LHA).

Definition 9 (Linear Hybrid Automata)**.** A linear hybrid automaton is a hybrid automaton H with state space dimension $d \in \mathbb{N}$ and the following properties.

- Each edge $e := (\ell, \ell', lab, cond) \in Edge_H$ is given by $\widehat{e} := (\ell, \ell', lab, \widehat{cond})$. The jump condition is given as guard and reset $\widehat{cond} := (\widehat{G}, \widehat{R})$. The guard is given by $m \in \mathbb{N}$ linear constraints $\widehat{G} := (A_\text{G}, b_\text{G})$ with $A_\text{G} \in \mathbb{R}^{m \times d}$, $b_\text{G} \in \mathbb{R}^m$. These constraints represent $G := \{v \in Val_H \mid A_\text{G} v \leq b_\text{G}\}$. The reset is given as an affine transformation $\widehat{R} := (A_\text{R}, b_\text{R})$ with $A_\text{R} \in \mathbb{R}^{d \times d}$, $b_\text{R} \in \mathbb{R}^d$. The transformation represents the function $R : Val_H \to Val_H, v \mapsto A_\text{R} v + b_\text{R}$. Finally $\widehat{cond}$ represents $cond := \{(v, R(v)) \in Val_H^2 \mid v \in G\}$.
- Initial states $Init_H$ are given as $\widehat{Init}_H := \{\widehat{Init}_H(1), \ldots, \widehat{Init}_H(n)\}$, $n \in \mathbb{N}$ where for each $i \in [1, n]$ there is a location $\ell \in Loc_H$, an $m \in \mathbb{N}$ and m linear constraints (A, b), $A \in \mathbb{R}^{m \times d}$, $b \in \mathbb{R}^m$, such that $\widehat{Init}_H(i) := (A, b, \ell)$. Similarly to before they represent the set $Init_H(i) := \{(v, \ell) \in St_H \mid Av \leq b\}$. Naturally the initial states are then $Init_H := \bigcup_{i=1}^{n} Init_H(i)$.
- For every $\ell \in Loc_H$, $Inv_H(\ell)$ is given as $\widehat{Inv}_H(\ell) := (A_\text{I}, b_\text{I})$ with $A_\text{I} \in \mathbb{R}^{m \times d}$, $b_\text{I} \in \mathbb{R}^m$, $m \in \mathbb{N}$ representing $Inv_H(\ell) := \{v \in Val_H \mid A_\text{I} v \leq b_\text{I}\}$.
- The flow $Flow_H(\ell)$ is given for each location $\ell \in Loc_H$ as $\widehat{Flow}_H(\ell) := A_\text{F} \in \mathbb{R}^{d \times d}$, encoding the linear differential equation $\dot{x} = A_\text{F} x$ where x is a vector of the variables from Var_H. The flow is then $Flow_H(\ell) := \{f_v \in \text{Map}(\mathbb{R}_{\geq 0}, Val_H) \mid v \in Val_H, f_v(t) = e^{A_\text{F} t} v \text{ for } t \in \mathbb{R}_{\geq 0}\}$.

We may use the matrices and vectors mentioned above without further qualification if the automaton H and location ℓ or edge e are unambiguously given by the context.

Note that affine dynamics of the form $\dot{x} = Ax + b$ can be simulated in a linear hybrid automaton H by adding an extra variable $y \notin Var_H$, setting its derivative to zero in all locations, not changing it on any jump and setting it to one initially. It can then be used in place of the constants from b.

2.2 Reachability Algorithm

After introducing hybrid automata, we now state the reachability algorithm that we will later extend in Chapter 3. To show its soundness for linear hybrid automata we first review some properties of their trajectories.

2.2.1 Alternating Trajectories

Definition 10 (Alternating Trajectory)**.** Let H be a hybrid automaton. A trajectory $(s_i, \zeta_i)_{i\geq 0}$ of H is alternating if and only if $\zeta_i \in \mathbb{R}_{\geq 0}$ for all even i and $\zeta_i \in Edge_H$ for all odd i.

Theorem 1 (Mergable Time Steps)**.**
Let H be a linear hybrid automaton and $(v_i, \ell) \xrightarrow{t_1} (v_{i+1}, \ell) \xrightarrow{t_2} (v_{i+2}, \ell)$ for some $v_i, v_{i+1}, v_{i+2} \in Val_H$, $\ell \in Loc_H$. Then $(v_i, \ell) \xrightarrow{t_1+t_2} (v_{i+2}, \ell)$ also holds.

Proof. $(v_i, \ell) \xrightarrow{t_1} (v_{i+1}, \ell)$ and $(v_{i+1}, \ell) \xrightarrow{t_2} (v_{i+2}, \ell)$ imply that $f, g \in Flow_H(\ell)$ exist with

- $f(0) = v_i$, $g(0) = v_{i+1}$,
- $f(t_1) = v_{i+1}$, $g(t_2) = v_{i+2}$,
- $f(\epsilon) \in Inv_H(\ell)$ for $\epsilon \in [0, t_1]$ and $g(\epsilon') \in Inv_H(\ell)$ for $\epsilon' \in [0, t_2]$.

For $t \in \mathbb{R}_{\geq 0}$ we obtain $f(t) = e^{A_F t} v_i$ and $g(t) = e^{A_F t} v_{i+1}$ from Definition 9. From this follows

$$f(t_1 + t) = e^{A_F(t+t_1)} v_i = e^{A_F t} e^{A_F t_1} v_i = e^{A_F t} f(t_1) = e^{A_F t} v_{i+1} = g(t).$$

This gives us all requirements for $(v_i) \xrightarrow{t_1+t_2} (v_{i+2})$.

- $f(0) = v_i$,
- $f(t_1 + t_2) = g(t_2) = v_{i+2}$,
- for $\epsilon \in [0, t_1] : f(\epsilon) \in Inv_H(\ell)$,
- for $\epsilon \in [t_1, t_1 + t_2] : f(\epsilon) = f(t_1 + \epsilon') = g(\epsilon')$ with $\epsilon' \in [0, t_2]$ thus
  for $\epsilon \in [t_1, t_1 + t_2] : f(t) \in Inv_H(\ell)$.

□

Theorem 2 (Alternating Reachability)**.** *Let H be a linear hybrid automaton. Then each reachable state $s \in St_H$ is reachable by an alternating trajectory, i.e. there is an alternating initial trajectory $(s_i, \zeta_i)_{i\geq 0}$ with $s_i = s$ for some $i \geq 0$.*

Proof. Since s is reachable there exists an initial trajectory $(s'_i, \zeta'_i)_{i\geq 0}$ with $s'_i = s$ for some $i \geq 0$. That $(s'_i, \zeta'_i)_{i\geq 0}$ can be transformed to an alternating trajectory while still reaching s follows from Theorem 1 and

$$s_i \xrightarrow{\alpha} s_{i+1} \xrightarrow{\alpha} s_{i+2} \Rightarrow s_i \xrightarrow{\alpha} s_{i+1} \xrightarrow{0} s_{i+1} \xrightarrow{\alpha} s_{i+2} \tag{2.1}$$

□

2.2.2 General Reachability Algorithm

After introducing the theory required to prove its soundness, we now discuss the reachability algorithm itself. Given a linear hybrid automaton H, the goal is to compute its set of reachable states R_H. The algorithm uses two operators on sets of states $S \subseteq St_H$.

- $T^+(S) := \{s' \in St_H \mid s \in S, s \xrightarrow{\tau} s'\}$.
- $D^+(S) := \{s' \in St_H \mid s \in S, s \xrightarrow{\alpha} s'\}$.

We compute sets S_i iteratively from for $i \in \mathbb{N}$ from $i = 0$ by

$$S_i := \begin{cases} T^+(Init_H), & \text{if } i = 0, \\ T^+(D^+(S_{i-1})), & \text{otherwise.} \end{cases} \tag{2.2}$$

Theorem 3. *The set $\bigcup_{i=0}^{\infty} S_i$ computed by Equation 2.2 is the set of reachable states R_H of H.*

Proof. We prove that for $i \in \mathbb{N}$ the set S_i contains all states that are reachable in $2i$ or $2i+1$ steps by an alternating trajectory. First we note that since zero duration time steps are allowed, $S \subseteq T^+(S)$ holds for any state set S.

Since $S_0 = T^+(Init_H)$, it contains $Init_H$, which are all states reachable in zero steps and the time successors of these states, which are all states reachable by the first step of an alternating trajectory.

Now assume the statement holds for an $i \in \mathbb{N}$. As noted $D^+(S_i) \subseteq T^+(D^+(S_i))$ holds and thus $S_{i+1} = T^+(D^+(S_i))$ contains all states that are reachable by a discrete jump or a discrete jump and a time step from a state in S_i. Thus the statement holds for $i+1$.

By induction $\bigcup_{i=0}^{\infty} S_i$ contains all states reachable by an alternating trajectory in any number of steps and by Theorem 2 these are all reachable states of H. □

Fixed-points. We reached a fixed point if $S_i \subseteq \bigcup_{j=0}^{i-1} S_j$. Then for all states $s \in S_i$ there is a $j < i$ such that $s \in S_j$ and thus s is also reachable in $2j$ or $2j+1$ steps. So s is reachable is less than $2i$ steps. Since all $s' \in S_{i+1}$ are reachable by a state from S_i in at most two steps, all s' are reachable is at most $2i+1$ steps and thus $S_{i+1} \subseteq \bigcup_{j=0}^{i} S_j = \bigcup_{j=0}^{i} S_j$. By induction the same holds for all following sets and the computation may be terminated. Note that such a fixed point may never be reached and thus the algorithm may not terminate.

2.2.3 Over Approximation and Bounded Analysis

That the algorithm may not terminate is a symptom of it solving the unbounded reachability problem for linear hybrid automata, which is undecidable. To overcome this there are three limitations put on the analysis.

We compute an over approximation $App(R_H)$ of the set of reachable states R_H such that $R_H \subseteq App(R_H)$ is guaranteed to hold. This means that we can use over approximations for both T^+ and D^+. Additionally we bound the number of jumps by some $j \in \mathbb{N}$ and compute the set $R_H[j] \subseteq R_H$ of states that are reachable by initial trajectories with j or fewer jumps.

The approach we describe below requires one more restriction to ensure termination. The time that may pass has to be bounded by a time horizon $T_g \in \mathbb{R}_{\geq 0}$ or $T_l \in \mathbb{R}_{\geq 0}$. The global time horizon T_g is a bound on the total time that may pass. Meaning that, using T_g, we compute the set $R_H[j, T_g]$ of states reachable in T_g or less time. The local time horizon T_l bounds the duration of consecutive time steps. So with T_l as a bound, we compute the set $R_H[j, T_l]$ of states reachable by trajectories that contain no sequence of time steps with a combined duration of more than T_l.

The results $App(R_H[j, T])$ where $T \in \{T_l, T_g\}$ are still useful since safety can be proven by $s \notin App(R_H[j, T]) \implies s \notin R_H[j, T]$. This allows us to prove that certain states are not reachable within a certain time and a certain number of jumps. The time horizon T may also imply the jump bound j or j together with invariants may imply T, so sometimes only one of them is necessary [Sch19, Sec.3.3].

A local time horizon can be imposed by adding a new variable and invariants. The variable has a derivative of one, is initialized to zero and reset to zero on every

jump. The invariants bound the variable from above by T_l. A global time horizon can be added similarly to a local one except that the variable is never reset. The local time horizon approach is specially handled in the implementation as we will mention below.

2.2.4 Concrete Reachability Algorithm

We now look at the concrete reachability algorithm for a linear hybrid automaton H. The algorithm effectively works with a partition of the sets S_i, $i \in \mathbb{N}$ by location. The state sets that it processes are represented by pairs $(\widehat{V}, \ell)$ or $(\widehat{F}, \ell)$. Here $\widehat{V}$ is a representation of a valuation set $V \subseteq Val_H$ as described in Section 2.3.1. A flowpipe $\widehat{F} := (\widehat{V}_1, \dots, \widehat{V}_c)$ with $c \in \mathbb{N}$ segments represents $F := \bigcup_{i=1}^{c} V_i$.

Initially, we construct pairs $(\widehat{V}, \ell)$ from $Init_H$. Since $Init_H$ is given as pairs of a set of linear constraints and a location, it lends itself well to this. Then we use two operators, similarly to the general algorithm, to compute their successors. The continuous successor operator T^+ on $(\widehat{V}, \ell)$ must satisfy $T^+_\ell(\widehat{V}) := \widehat{F}$ with

$$
\begin{aligned}
F \supseteq &\{f(t) \mid f \in Flow_H(\ell),\ f(0) \in V,\ t \in \mathbb{R}_{\geq 0},\ f(t') \in Inv_H(\ell) \text{ for } t' \in [0,t]\} \\
= &\{e^{At}v \mid \widehat{Flow_H}(\ell) = A,\ v \in V,\ t \in \mathbb{R}_{\geq 0},\ e^{At'}v \in Inv_H(\ell) \text{ for } t' \in [0,t]\}.
\end{aligned}
\tag{2.3}
$$

Let $Edge_H(\ell) := \{(\ell_1, \ell_2, lab, cond) \in Edge_H \mid \ell_1 = \ell\}$ be the set of edges originating in ℓ. The discrete successor operator D^+ on $T^+_\ell(\widehat{V}) = \widehat{F}$ is defined for each edge $e := (\ell, \ell', lab, cond) \in Edge_H(\ell)$. Let $(\widehat{G}, \widehat{R}) := \widehat{cond}$, $(A, b) := \widehat{G}$, $(A', b') := \widehat{R}$. The operator must satisfy

$$
\begin{aligned}
D^+_{\widehat{e}}(T^+_\ell(\widehat{V})) := \langle(\widehat{V}_1, t_1, \widehat{e}, \ell'), \dots, (\widehat{V}_n, t_n, \widehat{e}, \ell')\rangle &\text{ with } n \in \mathbb{N} \text{ and} \\
\bigcup_{i=1}^{n} V_i \supseteq &\{R(v) \mid v \in T^+_\ell(\widehat{V}), v \in G\} \cap Inv_H(\ell') \\
= &\{A'v + b' \mid v \in T^+_\ell(\widehat{V}), Av \leq b\} \cap Inv_H(\ell').
\end{aligned}
\tag{2.4}
$$

The components $\widehat{e}$ and ℓ' are redundant to ease notation below, $t_i \subseteq \mathbb{R}_{\geq 0}$, $i \in [1, n]$ are time intervals. They satisfy that for any $s_1, s_2 \in T^+_\ell(\widehat{V})$, $s_3 \in St_H$ and $\epsilon \in t_i$ with $s_1 \xrightarrow{\epsilon} s_2 \xrightarrow{e} s_3$ it holds that $s_3 \in V_i$. In other words, any state reachable from V by a time step of duration $\epsilon \in t_i$ followed by a jump via e, is contained in V_i.

Since $\bigcup_{i=1}^{n} V_i$ contains all states that are reachable from V by a time step followed by a jump via e, we also know that any state that is not reachable from V by a time step of duration $\epsilon \in \bigcup_{i=1}^{n} t_i$ does not enable the edge e. The full set of successors is

$$
D^+_\ell(T^+_\ell(\widehat{V})) := \bigcup_{e \in Edge_H(\ell)} D^+_{\widehat{e}}(T^+_\ell(\widehat{V})).
\tag{2.5}
$$

Computation Tree

The following is based on [Sch19, Sec.6.9]. The computation naturally describes a tree as seen in Figure 2.4. If there are initial states in multiple locations this naturally extends to a forest, but we will consider a single tree to avoid unnecessary complexity in notation.

Storing this tree allows us to, for example, find a set of states S for a given trajectory $(s_i, \zeta_i)_{i \geq 0}$ and $i \in \mathbb{N}$ such that $s_i \in S$. The opposite is also possible, for a

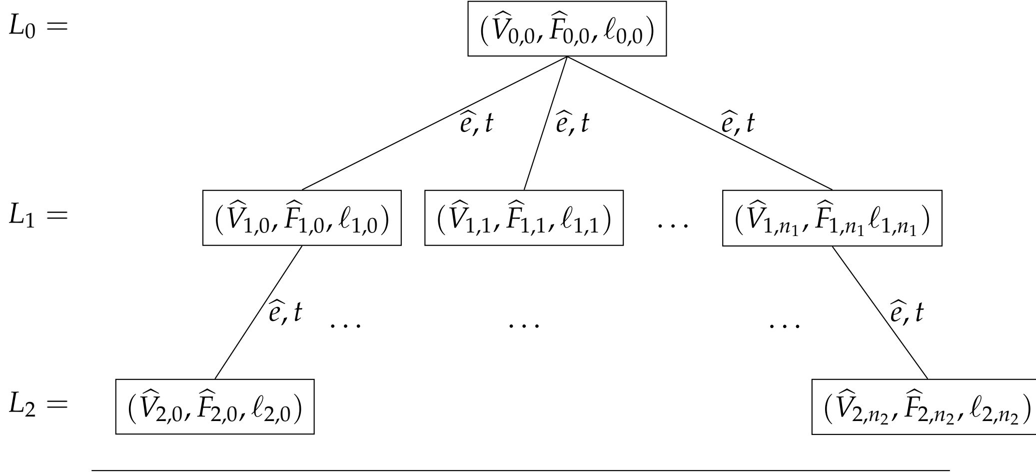


FIGURE 2.4: The general structure of a computation tree.

reachable state $s \in R_H$, we can find a set of trajectories, by one of which s is reachable. When we over approximate R_H this second use is only a heuristic.

If memory consumption is a problem, parts of the tree can be discarded. If one is only interested in whether a specific set of states is reachable, then those states can be intersected with the the states represented by a node immediately after that node is generated. A node can then be discarded once all its discrete successors, i.e. its children were generated. This effectively only maintains a list of current leaf nodes.

The tree $T_H := (\mathbb{N}, \mathbb{E})$ for a linear hybrid automaton H consists of

- nodes $(init, full, \ell, n) \in \mathbb{N}$ with $init := \widehat{V}, full := \widehat{F}$, $V, F \subseteq Val_H$, $\ell \in Loc_H$ and a numbering $n \in \mathbb{N}$,
- edges $(\mathtt{m}, \mathtt{m}', \widehat{e}, t) \in \mathbb{E}$ with $\mathtt{m}, \mathtt{m}' \in \mathbb{N}$, $e \in Edge_H$, $t := [t_l, t_u] \subseteq \mathbb{R}_{\geq 0}$.

The numbering on the nodes is necessary to model that otherwise identical nodes may exist. When we omit it, each node mentioned is assumed to have a distinct numbering.

For each pair $(\widehat{V}, \ell)$ that is constructed from $\widehat{Init}_H$, there is a root $(\widehat{V}, T^+_\ell(\widehat{V}), \ell)$. Each node $\mathtt{m} := (init, full, \ell)$ has a child $\mathtt{m}' := (\widehat{V}', T^+_{\ell'}(\widehat{V}'), \ell')$ for each $(\widehat{V}', \ell', \widehat{e}, t) \in D^+_\ell(full)$. The edge connecting them is $(\mathtt{m}, \mathtt{m}', \widehat{e}, t)$.

Let $L_i = \{(init_1, full_1, \ell_1), \ldots, (init_n, full_n, \ell_n)\}$ be the set of nodes at level $i \in \mathbb{N}$, i.e. the nodes whose shortest path to the root has length i. Then $\bigcup_{j=1}^n full_j$ corresponds to S_i in the general reachability algorithm from Section 2.2.

2.3 Successor Computations

After introducing the algorithm, we now discuss flowpipe construction, which is used to compute the continuous successors of a set of states, i.e. implement T^+_ℓ from Section 2.2.4. We will then look at the procedure to obtain the discrete successors of such a set, i.e. the implementation of D^+_ℓ. As discussed before, both operators will compute over approximations of the set of continuous and discrete successors respectively, because exact computations are not possible in general.

The focus will be on the aspects that are relevant to our algorithm. That is the representation of the result and how to use it, as well as some of the details of the computation that we can use to optimize the computation for the compositional case. For the remaining part an informal intuition will be given.

For further details omitted here, we refer the interested reader to [Fre16, Sec.2.2, 2.3]. Notably, we only consider flows of the form $\dot{x} = Ax$, as in Definition 9, which is a special case of what is described in [Fre16].

2.3.1 Continuous Successor Computation

Given a linear hybrid automaton H, $\widehat{V}$ representing $V \subseteq Val_H$ and $\ell \in Loc_H$, let X_t be the set of states reachable from V by a time step of duration $t \in \mathbb{R}_{\geq 0}$ ignoring the invariant, i.e. $X_t := \{e^{A_F t} v \mid v \in V\}$. The goal is to compute $\widehat{F}$ representing a set $F \subseteq Val_H$ with

$$X := \bigcup_{t \in \mathbb{R}_{\geq 0}} X_t \subseteq F \text{ or } X := \bigcup_{t \in [0, T_l]} X_t \subseteq F, \text{ given } T_l \in \mathbb{R}_{\geq 0}.$$

Valuation Set Representations and Operations

As already seen in Section 2.2.4 we will be working with valuation sets, so we need a representation for them. We use those available in the HyPro library [Sch19], so our algorithm can be used with boxes, support functions and convex polyhedra for example. Notably convex polyhedra directly correspond to the definition of $Init_H$, Inv_H and guards of a linear hybrid automaton H. All other representations can also be constructed from a set of linear constraints, but may introduce an over approximation. To simplify notation, we will not specially mark the over approximation introduced by this construction and use sets of linear constraints interchangeably with the valuation set representation constructed from them.

In general only over approximations are available for all operations on the representations. To reflect this, we denote these operations surrounded by $App()$. The operations that are available on representations $\widehat{V}, \widehat{V}'$ of $V, V' \subseteq \mathbb{R}^n$ are

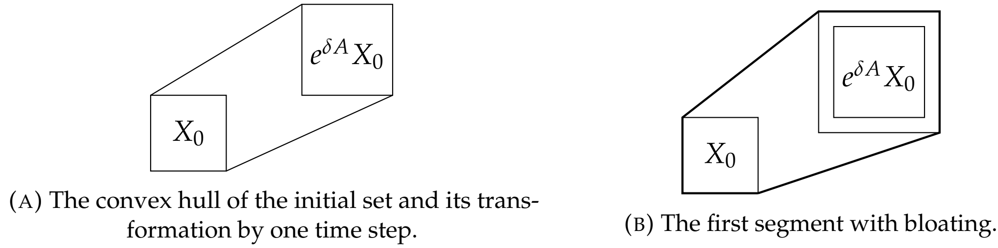
- intersection $App(\widehat{V} \cap \widehat{V}')$ representing $V_R \supseteq V \cap V'$,
- affine transformation, $App(A\widehat{V} + b)$ representing $V_R \supseteq AV + b := \{Av + b \mid v \in V\}$,
- Minkowski sum, $App(\widehat{V} \oplus \widehat{V}')$ representing $V_R \supseteq V \oplus V' := \{v + v' \mid v \in V, v' \in V'\}$,
- convex hull of the union, $App(\widehat{V} \cup \widehat{V}') := App(cHull(\widehat{V}, \widehat{V}'))$ representing $V_R \supseteq cHull(V, V')$ where $cHull(V, V')$ is the smallest convex set containing V and V',

with $n \in \mathbb{N}$, $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$.

Flowpipe Construction

The representations we use can not describe the set of states reachable by time steps exactly, especially when the flow describes non-linear functions. The discrepancy between the represented set and the set of reachable states tends to grow with the distance between the smallest and largest time steps allowed. Thus time is discretized into intervals whose width is some time step $\delta \in \mathbb{R}_{\geq 0}$. We then compute segments $\widehat{V}_i$ for $i \in \mathbb{N}_{\geq 1}$ each covering the time interval $[(i-1)\delta, i\delta]$, i.e.

$$X_{[(i-1)\delta, i\delta]} := \bigcup_{t \in [(i-1)\delta, i\delta]} X_t \subseteq V_i.$$



(A) The convex hull of the initial set and its transformation by one time step.

(B) The first segment with bloating.

FIGURE 2.5: Construction of the first flowpipe segment.

If $T_l \in \mathbb{R}_{\geq 0}$ is given, at most $n := \left\lceil \frac{T_l}{\delta} \right\rceil \in \mathbb{N}$ segments are computed.

At this point let us note two important statements. First, regarding the relationship of the sets X_t, we make essentially the same argument as in Theorem 1.

$$\begin{aligned} & X_{t+t'} \\ =& \{e^{A_F(t+t')}v \mid v \in V\} \\ =& \{e^{A_F t} e^{A_F t'} v \mid v \in V\} \\ =& e^{A_F t}\{e^{A_F t'} v \mid v \in V\} \\ =& e^{A_F t} X_{t'}. \end{aligned} \tag{2.6}$$

Second, regarding linear transformations of any two sets $S, S' \subseteq \mathbb{R}^d$ by a matrix $A \in \mathbb{R}^{d \times d}$ of some dimension d.

$$S \subseteq S' \implies AS \subseteq AS'. \tag{2.7}$$

First Segment. To compute $\widehat{V}_1$ we first obtain $\widehat{Y}_0 := \widehat{V}$ and $\widehat{Y}_\delta := App(e^{A_F\delta}\widehat{V})$, representing over approximations of X_0 and X_δ respectively. We take their convex hull $App(cHull(\widehat{Y}_0, \widehat{Y}_\delta))$. This is shown in Figure 2.5a. Intuitively this makes sense because the convex hull accommodates for all possible linear behavior, but the set still has to be extended for non-linear behavior. We will not explore how to do this in detail but instead simply assume that we obtained $\widehat{V}_B$ such that V_B over approximates the difference between the convex hull and the potentially non-linear behavior. We can use it for bloating such that $X_{[0,\delta]} \subseteq cHull(X_0, X_\delta) \oplus V_B$, as shown in Figure 2.5b. Thus $\widehat{V}_1 := App(cHull(\widehat{Y}_0, \widehat{Y}_\delta) \oplus \widehat{V}_B)$ fulfills our requirements. This type of bloating, among others, is explained in both [Fre16; Sch19].

Flowpipe. We have the first segment $\widehat{V}_1$ such that $X_t \in V_1$ for all $t \in [0, \delta]$ or simply $X_{[0,\delta]} \subseteq V_1$ holds. Based on that we can now construct $\widehat{V}_i := App(e^{A_F\delta}\widehat{V}_{i-1})$ for $i > 1$. Due to Equation 2.6 and 2.7 $X_t \in V_i$ for all $t \in [(i-1)\delta, i\delta]$ or simply $X_{[(i-1)\delta, i\delta]} \subseteq V_i$ holds again. Using this, we can keep iteratively computing further sets. We stop the computation at $\widehat{V}_n$ or when the invariant is guaranteed to be violated, as we will discuss next.

Invariant. So far we have been ignoring the invariant. Since we are computing an over approximation and the invariant only restricts the set of reachable states, there are no specific requirements on our handling of it. We simply intersect each segment with the invariant, i.e. use $\widehat{V}_i' := App(\widehat{V}_i \cap Inv_H(\ell))$ instead of $\widehat{V}_i$. If $\widehat{V}_i' = \emptyset$ we stop the iteration as all following segments V_j', $j > i$ will also be empty. Thus in total

$c := \max(\{i \mid \widehat{V}'_i \neq \emptyset\})$, or if $T_l \in \mathbb{R}_{\geq 0}$ is given, $c := \min(n, \max(\{i \mid \widehat{V}'_i \neq \emptyset\}))$ segments are computed.

As the final result we then have $T^+_\ell(\widehat{V}) := \widehat{F} := \{\widehat{V}'_1, \dots, \widehat{V}'_c\}$ representing $F = \bigcup_{i=1}^{c} V_i$.

2.3.2 Discrete Successor Computation

Let H be a linear hybrid automaton, $\widehat{F} := \{\widehat{V}_1, \dots, \widehat{V}_c\}$ a flowpipe representing $F \subseteq Val_H$, $\ell \in Loc_H$ a location and $e := (\ell, \ell', lab, cond) \in Edge_H(\ell)$ an edge originating in ℓ. Additionally let $(\widehat{G}, \widehat{R}) := \widehat{cond}$. The goal is to compute $\langle(\widehat{V}'_1, \ell', \widehat{e}, t_1), \dots\rangle$ for each such edge with the properties given in Equation 2.4.

We only have to consider flowpipes because $D^+_{\widehat{e}}$ is only applied to results of T^+_ℓ. We must first compute the set of valuations that satisfy the guard. We do this segment wise, computing the set of satisfying segments $\widehat{F}^{sat} := \{\widehat{V}_i^{sat} \mid i \in [1, c], \widehat{V}_i^{sat} = App(\widehat{V}_i \cap \widehat{G}), \widehat{V}_i^{sat} \neq \emptyset\}$. This set is then partitioned into sets $\widehat{F}_i^{sat} \subseteq \widehat{F}^{sat}$ for $i \in [1, n]$, $n \in \mathbb{N}$. Over those the union $\widehat{V}_i^{agg} := App(\bigcup_{\widehat{V} \in \widehat{F}_i^{sat}} \widehat{V})$ is computed. If $n = |\widehat{F}^{sat}|$ each segment is handled individually, if $n = 1$ all segments are aggregated into one set. Otherwise clusters of segments are created. These clusters are typically chosen to be consecutive, i.e. $\widehat{F}_i^{sat} = \{\widehat{V}_j, \dots, \widehat{V}_k\}$ with $j, k \in \mathbb{N}$ for all $i \in [1, n]$. This reduces over approximation in the union because consecutive segments are usually close and are actually guaranteed to overlap.

For each $i \in [1, n]$ a set of time points

$$t_i := \bigcup_{j \in [1,c] \wedge \widehat{V}_j \in \widehat{F}_i^{sat}} [(j-1)\delta, j\delta]$$

can be computed. If $\widehat{F}_i^{sat}$ contains consecutive segments, then that set is an interval $t_i = [t_l, t_u] \subseteq \mathbb{R}_{\geq 0}$. The successor valuations themselves are left to compute. We do this for each $i \in [1, n]$ by

1. $\widehat{V}_i^{reset} := App(A\widehat{V}_i^{agg} + b)$ where $\widehat{R} = (A, b)$ is the edge's reset,
2. $\widehat{V}_i^{result} := App(\widehat{V}_i^{reset} \cap \widehat{Inv}_H(\ell'))$.

Finally the result is $D^+_{\widehat{e}}(\widehat{F}) := \langle(\widehat{V}_i^{result}, \ell', \widehat{e}, t_i) \mid i \in [1, n], \widehat{V}_i^{result} \neq \emptyset\rangle$.

Chapter 3

Flowpipe Construction for Compositional Hybrid Automata

Our goal is to adapt the reachability algorithm from Chapter 2 to work with a hybrid automaton $H := H_1||\dots||H_n$ without constructing $H_1||\dots||H_n$ explicitly. We aim to improve scaling with the number $n \in \mathbb{N}$ of component automata. We will work with linear component automata, which implies that their parallel composition H is also a linear hybrid automaton.

Compositional Structures As a prerequisite we introduce the compositional structures that we will need hereinafter.

- Compositional edges $\widehat{e} := (\widehat{e}_1, \dots, \widehat{e}_n)$ with $e_i := (\ell_i, \ell'_i, lab_i, cond_i) \in Edge_{H_i} \cup IdEdge_{H_i}$ for all $i \in [1, n]$ and

$$e := ((\ell_1, \dots, \ell_n), (\ell'_1, \dots, \ell'_n), lab, \bigcap_{i=1}^{n} cond_i),$$

  where $lab := lab_i$ for the smallest $i \in [1, n]$ with $lab_i \neq I$. Note that such edges e are candidates for being edges of H, i.e. $e \in Edge_H$ does not necessarily hold.

- Compositional valuation sets $\widehat{\mathcal{V}} := (\widehat{V}_1, \dots, \widehat{V}_n)$ with $\mathcal{V} := \bigtimes_{i=1}^{n} V_i$.

- Compositional flowpipes $\widehat{\mathcal{F}} = (\widehat{F}_1, \dots, \widehat{F}_n)$ with $F_i = \{\widehat{V}_{i,1}, \dots, \widehat{V}_{i,c}\}$, representing $\mathcal{F} := \bigcup_{j=1}^{c}(\bigtimes_{i=1}^{n} V_{i,j})$.

Composition of Trajectories. Now we look at the trajectories of H and how they can be composed from the trajectories of the component automata H_i. Since the variable sets Var_{H_i} are pairwise disjoint, we can assume w.l.o.g. that Var_H is sorted such that for $v, v' \in Val_H$ there are $v_i, v'_i \in Val_{H_i}$ for $i \in [1, n]$ with $v = (v_1, \dots, v_n)$ and $v' = (v'_1, \dots, v'_n)$. Additionally let $\ell := (\ell_1, \dots, \ell_n) \in Loc_H$ and $\ell' := (\ell'_1, \dots, \ell'_n) \in Loc_H$.

Remark 1. The relation $s \xrightarrow{r} s'$ holds for $r \in \mathbb{R}_{\geq 0}$ if and only if $(v_i, \ell_i) \xrightarrow{r} (v'_i, \ell'_i)$ holds for all $i \in [1, n]$.

Remark 2. For $\widehat{e} := (\widehat{e}_1, \dots, \widehat{e}_n)$ with $e \in Edge_H$ the relation $(v, \ell) \xrightarrow{e} (v', \ell')$ holds if and only if $(v_i, \ell_i) \xrightarrow{e_i} (v'_i, \ell'_i)$ holds for all $i \in [1, n]$.

In the remaining chapter we adapt the reachability algorithm from Chapter 2 following the same steps as we did there. We begin with the computation tree and then move on to adapt the operators to compute continuous and discrete successors.

3.1 Compositional Computation Tree

The first step we take is extending the structure from Section 2.2.4 by inserting another level of composition. For the tree $T_H = (\mathbb{N}, \mathbb{E})$ a node $\mathtt{m} := (init, full, \ell) \in \mathbb{N}$ consists of a compositional valuation set $init := \widehat{\mathcal{V}}$, a compositional flowpipe $full := \widehat{\mathcal{F}}$ and a location $\ell := (\ell_1, \dots, \ell_n) \in Loc_H$. An edge $\mathtt{e} := (\mathtt{m}, \mathtt{m}', \widehat{e}, t) \in \mathbb{E}$ consists of the compositional parts

- $\mathtt{m}, \mathtt{m}' \in \mathbb{N}$,
- $\widehat{e} := (\widehat{e}_1, \dots, \widehat{e}_n)$ with $e \in Edge_H$,
- $t := [t_l, t_u] \subseteq \mathbb{R}_{\geq 0}$ a time interval with the same properties as in Section 2.2.4.

For the initial states we first only consider the two component automata H_1, H_2. Let $\iota_i := |\widehat{Init_{H_i}}|$ for $i \in [1, n]$, $n' := |Var_H| - |Var_{H_1}|$, $m' := |Var_H| - |Var_{H_2}|$. We assume w.l.o.g.

$$Var_{H_1} = \{x_1, \dots, x_{n'}\}, Var_{H_2} = \{x_{n'+1}, \dots, x_{n'+m'}\} \text{ and}$$
$$Var_{H_1||H_2} = \{x_1, \dots, x_{n'}, \dots, x_{m'+n'}\}$$

The initial states of $H_1||H_2$ can then be obtained by

$$\begin{aligned}
&Init_{H_1||H_2} \\
=&(Init_{H_1} \times \mathbb{R}^{m'}) \cap (\mathbb{R}^{n'} \times Init_{H_2}) && | \text{ Definitions 6, 7} \\
=&Init_{H_1} \times Init_{H_2} \\
=&(\bigcup_{j=1}^{\iota_1} Init_{H_1}(j)) \times (\bigcup_{k=1}^{\iota_2} Init_{H_2}(k)) && | \text{ Definition 9} \\
=&\bigcup_{(j,k) \in [1,n_1] \times [1,n_2]} Init_{H_1}(j) \times Init_{H_2}(k).
\end{aligned} \tag{3.1}$$

This naturally extends to all n component automata. Let $(\widehat{V}_{i,j}, \ell_{i,j})$ be chosen such that $V_{i,j_i} \times \ell_{i,j_i} = Init_{H_i}(j_i)$ for $i \in [1, n]$, $j_i \in [1, \iota_i]$. For each $(\widehat{Init_{H_1}}(j_1), \dots, \widehat{Init_{H_1}}(j_n))$ where $(j_1, \dots, j_n) \in \bigtimes_{i=1}^{n} [1, \iota_i]$ we construct a pair $(\widehat{\mathcal{V}}, \ell)$ with $\widehat{\mathcal{V}} := (\widehat{V}_{1,j_1}, \dots, \widehat{V}_{n,j_n})$, $\ell := (\ell_{1,j_1}, \dots, \ell_{n,j_n})$. Each such pair is used as the root $(\widehat{\mathcal{V}}, \mathcal{T}_\ell^+(\widehat{\mathcal{V}}), \ell)$ of a computation tree. As before, we consider only a single tree, even though there may be many.

3.2 Precision

Our goal is to represent the set of reachable states as precisely as possible. We will now justify the definitions above, look at how the precision compares to an explicitly constructed compositional hybrid automaton H and where that matters.

The initial states are given as a cartesian product. Assuming a single root $\mathtt{m}$ with $init = \widehat{\mathcal{V}} = (\widehat{V}_1, \dots, \widehat{V}_n)$, the initial states are $\mathcal{V} = \bigtimes_{i=1}^{n} V_i$. We again make the same assumptions on variable ordering as above. Now we want to find the set of states reachable by a step $\xrightarrow{r}$ for some $r \in \mathbb{R}_{\geq 0}$. That is the set

$$\begin{aligned}
&\{v' := (v'_1, \dots, v'_n) \in Val_H \mid v := (v_1, \dots, v_n) \in \mathcal{V}, v \xrightarrow{r} v'\} \\
=&\{v' := (v'_1, \dots, v'_n) \in Val_H \mid v_i \in V_i, v_i \xrightarrow{r} v'_i \text{ for } i \in [1, n]\}.
\end{aligned}$$

For some $i \in [1, n]$ and an according $j \in [1, |\widehat{F}_i|]$, the set $\{v'_i \mid v_i \in V_i, v_i \xrightarrow{r} v'_i\}$ is approximated by the j-th segment $V_{i,j}$ of $\widehat{F}_i \in \widehat{\mathcal{F}}$. We get the following approximation of the set of states reachable from $\mathcal{V}$ by $\xrightarrow{r}$.

$$\{v' := (v'_1, \ldots, v'_n) \in Val_H \mid v'_i \in V_{i,j} \text{ for } i \in [1, n]\} = \bigtimes_{i=1}^{n} V_{i,j}.$$

The construction for jumps works similarly. We over approximate the states reachable from the cartesian product over flowpipe segments $\bigtimes_{i=1}^{n} V_{i,j}$ via an edge $e \in Edge_H$ by the cartesian product $\bigtimes_{i=1}^{n} V_i$ over the initial valuation sets $init = \widehat{\mathcal{V}} = (\widehat{V}_1, \ldots, \widehat{V}_n)$ of an according child node.

This is why the reachable sets of states in the above computation tree are all represented by cartesian products. This is a considerable loss of precision if one is interested in the relationship between variables of different component automata. An important situation where this arises is jump synchronization between automata. The guard of an automaton often implies stricter bounds on the variables of other automata than we can establish by matching flowpipe segments.

Example 4 (Precision)**.** We assume that neither arithmetic nor the representation introduce over approximation for this example. Let H_1 be a linear hybrid automaton with a single location and single variable x that grows with $\dot{x} = 1$. Let H_2 be equivalent with a variable y. With a time step $\delta := 1$ our first segments would be $V_{1,1} = [0, 1]$ and $V_{2,1} = [0, 1]$ and the reachable set we obtain for $H_1 || H_2$ is the box $[0, 1] \times [0, 1]$.

Analysing the explicitly constructed automaton $H_1 || H_2$ and using a fitting representation like oriented boxes or polyhedra results in the line segment $\{(x, y) \mid x = y, x \in [0, 1]\}$ instead.

Constraints on x or y exclusively are satisfied by the box if and only if they are satisfied by the line segment, but the constraint $x - y \leq 0.5$, for example, is not satisfied by the former but satisfied by the latter. Also, intersecting the first set with the guard $x = 1$ yields $\{(x, y) \mid x = 1, y \in [0, 1]\}$ while an intersection with the second set yields the single point $(1, 1)$.

3.3 Adapting Successor Operators

Next we define operators $\mathcal{D}^+$ and $\mathcal{T}^+$ for the new compositional structures. These operators must meet the same requirements as D^+, T^+ in Section 2.2.4.

3.3.1 Continuous Successors

The continuous successors of a pair $(\widehat{\mathcal{V}}, \ell)$ as above could be computed as

$$\mathcal{T}_\ell^+(\widehat{\mathcal{V}}) := (T_{\ell_1}^+(\widehat{V}_1), \ldots, T_{\ell_n}^+(\widehat{V}_n)).$$

However, the longest time step of H from some $s \in \mathcal{V} \times \ell$ is at most as long as the shortest time step of any H_i from some $s_i \in V_i \times \ell_i$. This is a consequence of Inv_H being the intersection of the variable-extended invariants of the component automata. We build all flowpipes with the

F_1	F_2		F_n
$V_{1,1}$	$V_{2,1}$	$\cdots$	$V_{n,1}$
$V_{1,2}$	$V_{2,2}$		$V_{n,2}$
$V_{1,3}$	$V_{2,3}$		$V_{n,3}$
$V_{1,4}$	$V_{2,4}$		$V_{n,4}$
$V_{1,5}$	$V_{2,5}$		$V_{n,5}$
$V_{1,6}$	$V_{2,6}$	$\cdots$	$V_{n,6}$

FIGURE 3.1: Schematic of a compositional flowpipe and the order of construction of segments.

same time step $\delta \in \mathbb{R}_{\geq 0}$, so we can truncate them to the number of segments of the shortest flowpipe $\min\{|T^+_{\ell_i}(\widehat{V}_i)| \mid i \in [1,n]\}$. For an efficient implementation it is important to iterate $j = 1, 2, \dots$ and compute the j-th segment of each flowpipe $\widehat{V}_{1,j}, \dots, \widehat{V}_{n,j}$ in each step, as opposed to iterating $i \in [1,n]$ and computing the i-th flowpipe $T^+_{\ell_i}(\widehat{V}_i) = \{\widehat{V}_{i,1}, \widehat{V}_{i,2}, \dots\}$ in each step. This is shown in Figure 3.1. We want to reach the end condition $App(\widehat{V}_{i,j} \cap \widehat{Inv}_{H_i}(\ell_i)) = \emptyset$ as early as possible, before constructing segments that we will truncate away.

3.3.2 Discrete Successors

To compute the discrete successors of a pair $(\widehat{\mathcal{F}}, \ell) := ((\widehat{F}_1, \dots, \widehat{F}_n), (\ell_1, \dots, \ell_n))$ we must first obtain the set $Edge_H(\ell)$ of edges of H originating in ℓ.

Label Sets

In modeling of hybrid systems, it sometimes occurs that a model has many edges that are identical except for their label. We will see such an example in Section 4.1. It is useful to treat these edges as a single edge with a set of labels instead. We denote such sets of labels by $\boldsymbol{lab}$ instead of lab. Once we introduce this notion and implement its analysis efficiently, we can use it in modeling as a powerful tool to express discrete synchronization. We will show an example of that in Section 4.2.3.

Let H be a hybrid automaton as defined before. Let $\sim \subseteq Edge_H^2$ be a relation with $(\ell_1, \ell_1', lab_1, cond_1) \sim (\ell_2, \ell_2', lab_2, cond_2)$ if and only if $\ell_1 = \ell_2$, $\ell_1' = \ell_2'$, $cond_1 = cond_2$, i.e. $\sim$ is equality ignoring labels. Then the equivalent hybrid automaton to H using label sets is $Coll(H)$, which is equal to H except for

$$\begin{aligned} Edge_{Coll(H)} := \{&(\ell, \ell', \{lab_1, \dots, lab_m\}, cond) \mid m \in \mathbb{N}_{\geq 1}, \\ &\{(\ell, \ell', lab_1, cond), \dots, (\ell, \ell', lab_m, cond)\} \in Edge_{H}/_{\sim}\}. \end{aligned}$$

We extend $\sim$ to $(Edge_{Coll(H)} \cup Edge_H)^2$, as the same elementwise equality, ignoring labels and label sets, as before. We additionally define the self loops from Definition 5 for $Coll(H)$ as $e_I(Coll(H), \ell) := (\ell, \ell, \overline{\boldsymbol{Lab}}_{Coll(H)}, cond)$ where $e_I(H, \ell) = (\ell, \ell, I, cond)$. The label I is replaced by $\overline{\boldsymbol{Lab}}_{Coll(H)}$, which is the absolute complement of $Lab_{Coll(H)} = Lab_H$.

The semantics of $Coll(H)$ are defined as those of H. The semantics of any hybrid automaton H' using label sets are those of the hybrid automaton $Coll^{-1}(H')$ that is uniquely defined by the requirement $Coll(Coll^{-1}(H')) = H'$.

The notion of edges with sets of labels is useful because it simplifies modeling, makes the implementation more efficient and, as we will see below, allows us to also simplify the implementation by handling edge cases elegantly.

Computing the Set of Edges

Let H be a compositional automaton of n components. Furthermore, let $i \in [1,n]$, $\ell := (\ell_1, \dots, \ell_n) \in Loc_H$, $Edge^{\cup I}_{H_i}(\ell_i) := Edge_{H_i}(\ell_i) \cup e_I(H_i, \ell_i)$ and $\widehat{e} := (\widehat{e}_1, \dots, \widehat{e}_n)$ be a tuple of edges with $e_i \in Edge^{\cup I}_{H_i}(\ell)$. Let lab_i be the label of e_i. Applying Definition 7 yields that $e \in Edge_H(\ell)$ if and only if

- $i \in [1,n]$ with $lab_i \neq I$ exists and
- for all $i, j \in [1,n]$ with $lab_i \neq I$, $lab_j \neq I$ the labels are the same $lab := lab_i = lab_j$ and

- for all $i \in [1, n]$ with $lab_i = I$ the common label, i.e. $lab := lab_j$ for some $j \in [1, n]$, $lab_j \neq I$, does not occur in H_i, i.e. $lab \notin Lab_{H_i}$.

We can use label sets to simplify the above requirements and at the same time implement edges with multiple labels efficiently. We compute the set of edges

$$CEdge_H(\ell) := \{(\ell, \ell', \bigcap_{i=1}^{n} \boldsymbol{lab}_i, \bigcap_{i=1}^{n} cond_i) \mid (\ell_i, \ell'_i, \boldsymbol{lab}_i, cond_i) \in Edge^{\cup I}_{Coll(H_i)} \text{for } i \in [1, n], \bigcap_{i=1}^{n} \boldsymbol{lab}_i \neq \emptyset,\ \ell' = (\ell'_1, \dots, \ell'_n)\}.$$

This set is easy to compute and satisfies $CEdge_H(\ell) \sim Edge_H(\ell)$. This means that every state reachable by jumps via edges in $Edge_H(\ell)$ is also reachable by jumps via edges in $CEdge_H(\ell)$, since their elements only differ by their labels.

To ease notation and mental burden, we will allow label sets in all hybrid automata, justified by the fact that label sets are a generalization. This allows us to simply use H instead of $Coll(H)$. We will also use $Edge_H$ instead of $Edge_{Coll(H)}$ or $CEdge_H$, even though the implementation uses $CEdge_H$.

Modeling as a Tree. Let $\widehat{e} := (\widehat{e}_1, \dots, \widehat{e}_n)$ with $e_i \in Edge^{\cup I}_{H_i}$ for all $i \in [1, n]$ now. Let $P_{lab}(\widehat{e}) := \bigcap_{i=1}^{n} \boldsymbol{lab}_i$ and equivalently for smaller tuples. For efficiency one should avoid to enumerate the cartesian product $\bigtimes_{i=1}^{n} Edge^{\cup I}_{H_i}(\ell_i)$ in its entirety. The problem of enumerating all tuples $\widehat{e}$ such that $P_{lab}(\widehat{e}) \neq \emptyset$, i.e. $e \in CEdge_H(\ell)$ can be modeled as a search on a tree. Let $T^{edges}{}_H(\ell) := (N, E)$ with

- $N := \{(\widehat{e}_1, \dots, \widehat{e}_m) \mid m \in [0, n],\ e_i \in Edge^{\cup I}_{H_i}(\ell_i) \text{ for } i \in [1, m], P_{lab}((\widehat{e}_1, \dots, \widehat{e}_m)) \neq \emptyset\}$ and
- $E := \{((\widehat{e}_1, \dots, \widehat{e}_m), (\widehat{e}_1, \dots, \widehat{e}_m, \widehat{e}_{m+1})) \in N \times N\}$.

We store $P_{lab}(\widehat{e})$ with each $\widehat{e} \in N$. The children of each node can then be obtained efficiently because we can check $P_{lab}((\widehat{e}_1, \dots, \widehat{e}_m)) \neq \emptyset$, $m \in [1, n]$ by

$$P_{lab}((\widehat{e}_1, \dots, \widehat{e}_{m-1})) \cap lab_m \neq \emptyset.$$

Note that the result of the empty intersection is the neutral element with respect to intersection, which is Lab_H is this context. The leaf nodes, those at depth n, represent the edges in $CEdge_H(\ell)$. They can be enumerated, for example, by depth first search as shown in Algorithm 1. The empty tuple $()$ is the root of $T^{edges}{}_H(\ell)$, so we obtain $Edge_H(\ell)$ by $Edge_H(\ell) = \{e \mid \widehat{e} \in \texttt{DFS}((), Lab_H)\}$.

The tree may still be large, even if $|Edge_H(\ell)|$ is small. The size of the tree can vary based on the order that the edge sets are processed in. An edge set $Edge^{\cup I}_{H_i}(\ell_i)$ whose only edge synchronizes with no other edges can limit the size of the tree to $|N| = |\{(), e_I(H_i, \ell_i)\}| = 2$ if it is the very first one to be processed. If it is the last one, then there may still be up to

$$\prod_{j \in [1, n-1]} |Edge^{\cup I}_{H_j, \ell_j}| \text{ nodes.}$$

The order of the edge sets can be chosen for each compositional location individually. This is an opportunity to employ various heuristics or use knowledge about the

Algorithm 1: Pseudo-code algorithm for depth first search in ${T^{edges}}_H(\ell)$.

```
Function DFS(ê, lab):
    Edges := ∅
    i := |ê|
    if i = n then
        return ê
    forall e ∈ Edge^{∪I}_{H_i}(ℓ_i) do
        lab' := lab ∩ lab(e)
        if lab' ≠ ∅ then
            Edges := Edges ∪ DFS((ê,ê), lab')
    return Edges
```

model. One may, for example, process those edge sets first whose edges have the fewest labels combined.

Computing the Valuation Sets

We have computed $Edge_H(\ell)$ for a location $\ell := (\ell_1, \ldots, \ell_n) \in Loc_H$ and are given a compositional flowpipe $\widehat{\mathcal{F}} := (\widehat{F}_1, \ldots, \widehat{F}_n)$ with $\widehat{F}_i := (\widehat{V}_{i,1}, \ldots, \widehat{V}_{i,c})$. We need to compute $\mathcal{D}^+_{\widehat{e}}(\widehat{\mathcal{F}}) = (\widehat{\mathcal{V}}, \ell', \widehat{e}, t)$ for each $\widehat{e} := (e_1, \ldots, e_n)$ with $e := (\ell, \ell', lab, cond) \in Edge_H(\ell)$. Let $(\widehat{G}_i, \widehat{R}_i) := \widehat{cond}_i$ where $\widehat{cond}_i$ is the representation of the condition of e_i.

The sets of reachable states are $\mathcal{V}_j := \bigtimes_{i=1}^{n} V_{i,j}$ for each $j \in [1, c]$. The jump $(v, \ell) \xrightarrow{e} (R(v), \ell')$ for $v := (v_1, \ldots, v_n) \in \mathcal{V}_j$, with R being the reset of e, is possible if and only if $(v_i, \ell_i) \xrightarrow{e_i} (G_i(v_i), \ell'_i)$ is possible for all $i \in [1, n]$. So we compute all indices $j \in [1, c]$ where for all $i \in [1, n]$ the satisfying segment $\widehat{V}^{sat}_{i,j} := App(\widehat{V}_{i,j} \cap \widehat{G}_i)$ is not empty. Again there are efficiency concerns to be considered. The intersection with the guard and the check for emptiness of the result are a single operation in HyPro and and it is the only relevant operation we are performing here. To minimize the number of these intersections that we need to make, we iteratively compute $SatInds_i \subseteq [1, c]$ from $i = 0$ to $i = n$ by

$$SatInds_i := \begin{cases} [1, c], & \text{if } i = 0, \\ \{j \in SatInds_{i-1} \mid \widehat{V}^{sat}_{i,j} \neq \emptyset\}, & \text{otherwise.} \end{cases} \tag{3.2}$$

Then $SatInds := SatInds_n$ contains the indices j for which the corresponding segments $V_{i,j}$ of all flowpipes $\widehat{F}_i$ contain valuations that satisfy the guard. Note that we only need to compute the intersection with the guard for all segments for the first flowpipe. For all following flowpipes we need to check only the segments whose corresponding segments in all previous flowpipes were satisfying.

In the same way as in Section 2.3.2 we partition $SatInds$ into $SatInds_k \subseteq SatInds$ for $k \in [1, m]$ and create the aggregated valuation sets as the union

$$\widehat{V}^{agg}_{i,k} := App(\bigcup_{j \in SatInds_k} \widehat{V}^{sat}_{i,j}).$$

We assume that the indices in $SatInds_k$ are consecutive. Similarly to Section 2.3.2 the time intervals t_k are computed by

$$t_k := \bigcup_{j \in SatInds_k} [(j-1)\delta, j\delta].$$

Each set $\widehat{V}^{agg}_{i,k}$ is further processed to obtain

1. $\widehat{V}^{reset}{}_{i,k} := App(A\widehat{V}^{agg}_{i,k} + b)$ with $(A, b) := \widehat{R}_i$,
2. $\widehat{V}^{result}{}_{i,k} := App(V^{reset}{}_{i,k} \cap \widehat{Inv}_H(\ell'_i))$.

Finally let $\widehat{\mathcal{V}}_k := (\widehat{V}^{result}{}_{k,1}, \ldots, V^{result}{}_{k,n})$ to form the result

$$\mathcal{D}^+_{\widehat{e}}(\widehat{\mathcal{F}}) := \langle (\widehat{\mathcal{V}}_k, l', e, t_k) \mid k \in [1, m], \widehat{V}^{result}{}_{i,k} \neq \emptyset \text{ for all } i \in [1, n] \rangle.$$

Optimizing Edge Set Enumeration

In Chapter 4 we will see two situations where $|Edge_H(\ell)|$ grows exponentially with the number n of component automata, but only a single edge can actually be taken. We present an optimization for such cases.

Let $(N, E) := T^{edges}{}_H(\ell)$. Given $\ell \in Loc_H$, a compositional flowpipe $\widehat{\mathcal{F}}$ and $\widehat{e} := (\widehat{e}_1, \ldots, \widehat{e}_m) \in N$, let $SatInds_i$ be defined as above for $i \in [0, m]$ and $\widehat{V}^{sat}_{i,j} := App(\widehat{V}_{i,j}, \widehat{G}_i)$ where G_i is the guard of e_i. For each such $\widehat{e} \in N$ we define $SatInds(\widehat{e}) := SatInds_m$. We then use $SatInds(\widehat{e}) \neq \emptyset$ similarly to $P_{lab}(\widehat{e})$. We can also compute $SatInds(\widehat{e})$ incrementally by $SatInds(\widehat{e}) = \{j \in SatInds((\widehat{e}_1, \ldots, \widehat{e}_{m-1})) \mid \widehat{V}^{sat}_{m,j} \neq \emptyset\}$.

We consider the subtree $T^{edges}(\ell)[\widehat{\mathcal{F}}] := (N', E')$ of edges that have satisfying segments in $\widehat{\mathcal{F}}$. Its nodes and edges are $N' := \{\widehat{e} \in N \mid SatInds(\widehat{e}) \neq \emptyset\}$ and $E' := \{(\widehat{e}_1, \widehat{e}_2) \in E \mid \widehat{e}_1, \widehat{e}_2 \in N'\}$. Algorithm 2 shows the adaption of Algorithm 1 to incorporate the new constraint. The size of $T^{edges}{}_H(\ell)[\widehat{\mathcal{F}}]$ and thus the running time of Algorithm 2 varies depending on the order that the edge sets are processed in, in the same way as in Section 3.3.2. A heuristic for the order of the edge sets should now consider the guards of the edges. The fewer constraints the guards have the later the edge set should be processed.

Evaluating the added constraint $SatInds(\widehat{e}) \neq \emptyset$ may also require the computation of more satisfying segments $\widehat{V}^{sat}_{i,j}$ than Algorithm 1. Thus the new constraint represents

Algorithm 2: Adaption of Algorithm 1 for depth first search in $T^{edges}{}_H(\ell)[\widehat{\mathcal{F}}]$.

Function DFS($\widehat{e}$, $\boldsymbol{lab}$, $SatInds$):
- $Edges := \emptyset$
- $i := |\widehat{e}|$
- **if** $i = n$ **then**
  - **return** $\widehat{e}$
- **forall** $e \in Edge^{\cup I}_{H_i}(\ell_i)$ **do**
  - $\boldsymbol{lab}' := \boldsymbol{lab} \cap \boldsymbol{lab}(e)$
  - $SatInds' := \{j \in SatInds \mid \widehat{V}^{sat}_{i,j} \neq \emptyset\}$
  - **if** $\boldsymbol{lab}' \neq \emptyset \wedge SatInds' \neq \emptyset$ **then**
    - $Edges := Edges \cup$ DFS(($\widehat{e}, \widehat{e}$), $\boldsymbol{lab}'$, $SatInds'$)
- **return** $Edges$

a different trade off between the number of edges to enumerate and the number of intersections to compute to check for satisfying segments.

3.4 Urgent Edges

Before we look at some applications of our algorithm, we need to extend it to support urgent edges. Intuitively urgency of an edge means that when its guard is enabled, the edge, or another enabled edge, must immediately be taken.

Definition 11 (Urgent Edge)**.** Any hybrid automaton H may be extended by a set $UEdge_H \subseteq Edge_H$ designating its urgent edges. For $(v, \ell), (v', \ell) \in St_H$, $r \in \mathbb{R}_{\geq 0}$. The relation $(v, \ell) \xrightarrow{r} (v', \ell)$ holds if and only if the conditions from Definition 3 hold and the function f additionally satisfies $(f(\varepsilon'), v'') \notin cond$ for $\varepsilon' \in [0, r)$, any $v'' \in Val_H$ and any urgent edge $e := (\ell, \ell', lab, cond) \in Edge_H(\ell)$.

From this definition, we can see that the complements of guards of urgent edges act similarly to invariants that only constrain time steps, but not discrete jumps. There are some nuances because the interval $[0, r)$ is half open and when working with convex representations, the complement of the guard can not be formed unless it is defined by a single linear inequality. We will not explore these nuances here.

We will implement the analysis of urgent edges by adding a check to the procedure from Section 2.3.2. After intersecting each segment with the invariant, we check for the resulting set $\widehat{V}$ whether $\widehat{V} \subseteq G$ for each guard G of an urgent edge and stop the flowpipe construction if this is the case. The subset check is conservative in the sense that a positive result is guaranteed to be correct but negative result is not. This is a safe over approximation of the allowed behavior.

Chapter 4

Applications

We will now look at a few models and apply our algorithm to them. These tests will show strengths and weaknesses of the algorithm and reveal opportunities for improvement.

4.1 Robot Swarm Model

First we consider a model of a simple scenario demonstrating emergent behavior taken from [Leo+19]. Simple robots are modeled by automata $H_1, \ldots, H_n$ that each have an internal clock variable x_i and, when the clock reaches a threshold, flash an LED and reset their clock to zero. When a robot flashes its LED, all other robots multiply the value of their clock by some factor $\alpha > 1$. We are interested in the behavior of the whole swarm, i.e. $H := H_1 || \ldots || H_n$.

In [Leo+19] four different models are considered. Two using shared variables, two using exclusively labels for synchronization. Our algorithm is applicable to both models using label synchronization. We choose the model *lsync II* because it was found to lead to better scaling of the analysis.

4.1.1 The Model

In Figure 4.1 the automaton modeling the i-th robot is shown. Control initially starts in the location $wait := (wait_1, \ldots, wait_n) \in Loc_H$. Time passes and the clocks of the automata increase their values. When an automaton reaches $x_i = 1$, control jumps to the location $adapt := (adapt_1, \ldots, adapt_n) \in Loc_H$ because H_i jumps via the upper edge with label $flash_i$ and all others synchronize via their respective lower edge.

The location *adapt* is only used to differentiate between the case where a clock was set to a value less than one and may continue and the case where a clock was set to a value greater than or equal to one and must be reset to zero. No time may pass in *adapt* because the outgoing edges are urgent and the guards of the two edges with the *return* label cover the entirety of $\mathbb{R}$.

4.1.2 Analysis Parameters

We will compare the two algorithms we presented. On the one hand we use the compositional algorithm presented in Chapter 3, which works on separate component automata, without the edge set enumeration optimization of Section 3.3.2. On the other hand we employ the plain flowpipe construction algorithm outlined in Chapter 2, working on the explicitly constructed parallel composition. We perform the construction using HySt and use our implementation of the compositional algorithm and the preexisting implementation of the plain algorithm in HyPro.

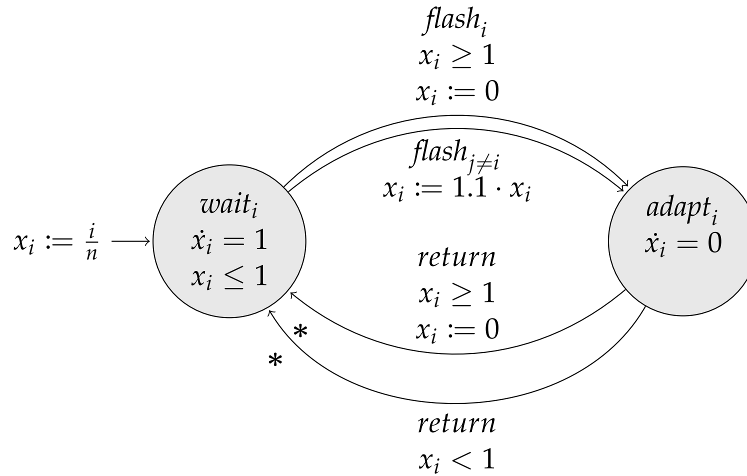


FIGURE 4.1: The i-th automaton of the *lsync II* model we are using.

We consider two parameters for this model. The number n of robots in the model and the size of the used time step δ, which is a proxy for precision. Which representation is used for the compositional algorithm is not important because the state spaces are one dimensional so any representation degenerates to intervals. More complex representations only add overhead, which is why we use boxes. For the plain algorithm, a more complex representation may improve precision, but as we will see, with a small enough δ, boxes are precise enough for all n that the plain algorithm is able to handle. For the maximum jump depth we use 20 as in [Leo+19] and the initial value for the clock of each H_i is $\frac{i-1}{n}$.

4.1.3 Initial Results

The results of our experiments are summarized in Tables 4.1 and 4.2. For the plain algorithm the number of nodes in the computation tree is not listed because was no branching so there were always the expected 21 nodes generated. We identify several limiting factors.

Precision. Initially all segments are boxes or intervals of length δ in each dimension. With every jump that does not reset x to zero precision is lost, because a segment $V := [3\delta, 4\delta]$, for example, that is used as an initial set, results in the first segment

TABLE 4.1: Run time in seconds and number of nodes in the computation tree for the plain algorithm for different numbers of robots n and time steps sizes δ. Time out is two minutes.

δ	$n =$	1	2	3	4	5	6	7	8	9	10
$2 \cdot 10^{-5}$	time	4.79	1.84	1.74	1.87	2.41	2.23	2.59	TO	3.36	5.56
10^{-5}	time	7.15	3.59	2.88	3.15	3.61	3.46	3.64	5.15	4.35	7.42

δ	$n =$	11	12	13	14	15	16	17	18	19	20
$2 \cdot 10^{-5}$	time	TO	6.77	12.4	19.6	38.2	TO	TO	TO	TO	TO
10^{-5}	time	TO	10.2	12.3	22.6	52.5	TO	TO	TO	TO	TO

TABLE 4.2: Run time in seconds and number of nodes in the computation tree for the compositional algorithm for different numbers of robots n and time steps sizes δ. Time out is two minutes.

δ	$n =$	1	2	3	4	5	6	7	8	9	10
10^{-4}	time	0.20	0.15	0.19	0.36	0.53	0.35	1.46	4.27	0.91	3.07
	nodes	21	21	21	21	21	21	36	55	27	24
10^{-5}	time	2.00	1.32	1.21	1.25	1.33	1.11	1.03	1.23	1.07	1.40
	nodes	21	21	21	21	21	21	21	21	21	21

δ	$n =$	11	12	13	14	15	16	17	18	19	20
10^{-4}	time	2.00	0.91	1.13	1.20	1.76	59.9	14.2	50.4	45.2	TO
	nodes	43	22	22	22	22	253	54	207	72	120
10^{-5}	time	2.35	1.17	1.27	1.60	2.09	3.20	5.08	9.28	16.9	33.2
	nodes	40	21	21	21	21	21	21	21	21	21

$[3\delta, 5\delta]$ in the next flowpipe, increasing the segments width by δ. This also leads to segments overlapping, as the next segment in the example would be $[4\delta, 6\delta]$. This means that multiple segments will be satisfying the $x \geq 1$ guard before the flowpipe construction can be terminated. The aggregation of the corresponding segments in other automata causes further loss of precision. In Section 5.2.1 we suggest a possible solution. The according Figure 5.1 visualizes the described situation.

Modeling can have an impact on this, as for example the reset $x_i := 0$ on the edge labeled flash_i is superfluous for the semantics of the system, but is important to collapse the segment that enables this jump to $[0, 0]$.

Urgent Edges It occurs sometimes that a reachable set in *adapt* overlaps with the guard of two urgent edges. Due to over approximations the algorithms can not determine that the reachable set is fully contained in the guards of the edges, so control remains in *adapt* indefinitely. Non termination is only avoided by a local time horizon. This phenomenon can most clearly be seen in the timings of the plain algorithm for $n \in \{8, 11\}$.

Branching. For the compositional algorithm there is a clear correlation of running times and the number of nodes in the computation tree. This correlation can also be observed for the plain algorithm with time steps larger than what is shown here. We can also see in the data that using smaller time steps can help with this problem. With the smaller time step, the compositional algorithm generates fewer branches and runs faster for $n \in \{7, 8, 10, 16, 17, 18, 19, 20\}$. The plain algorithm behaves similarly when going from $\delta = 10^{-3}$, for which the data is not shown, to the shown time step sizes. With $\delta = 10^{-3}$ it times out for all $n \geq 7$ because of branching, while there is no branching and fewer time outs for the two time step sizes shown. The smaller time steps reduce branching because they cause the calculated segments to be more precise and thus smaller. Smaller segments intersect less often with more than one guard than larger ones.

Since branching is so impactful, the maximum jump depth has a big impact on the running times. The number of nodes grows exponentially with the jump depth and the loss of precision with every jump causes even more branching.

TABLE 4.3: Run time in seconds and number of nodes in the computation tree for the compositional algorithm for different numbers of robots n. Time out is two minutes.

δ	$n =$	40	80	120	160	200	240	280	320	360	400
10^{-5}	time	1.63	2.34	3.56	13.0	34.5	11.9	17.5	58.5	44.8	TO
	nodes	21	22	26	50	77	52	60	147	114	NA

Exponential Model Growth. The size of the models generated for the plain algorithm is not listed but it is significant. Since there are two edges leading from $adapt_i$ to $wait_i$ for each $i \in [1, n]$ and they all have the *return* label, there are 2^n edges in $Edge_H(adapt)$. For $n = 16$ and above this caused the model file parser alone to time out. While the parser may be optimized, $|Edge_H(adapt)|$ still grows exponentially, so the problem would persist for n not much larger than what we tested.

The compositional algorithm avoids the problem of writing $Edge_H(adapt)$ to a file and reading it again because it takes the component automata each on its own. However, computing the edge set as described in Section 3.3.2 still enumerates all edges in $Edge_H(adapt)$. This is the reason for the exponential growth of running times of the compositional algorithm with $\delta = 10^{-5}$ from $n = 16$ to $n = 20$, despite no branching.

Summary. The issue of loss of precision and branching caused by it were mostly overcome by using a small enough δ, at least for the jump depth limit of 20. As the number of automata n increases, exponential model growth becomes the limiting factor. This makes the plain algorithm unusable for $n \geq 16$. While the effect on the compositional algorithm is less significant, it still causes the running time to grow exponentially. In the next section we will look at a solution for this.

Additionally, urgent edges sometimes cause technical problems. These could be avoided by adding a fresh variable with an invariant that prohibits time from passing in *adapt* or by adding urgent locations, i.e. locations that no time may pass in, to the formalism and model. We will however not consider this any further.

4.1.4 Optimized Edge Set Enumeration

We have determined that the limiting factor is the exponential growth of $Edge_{H_i}(adapt)$. We now apply the optimization from Section 3.3.2. The results of using the compositional algorithm with this optimization are shown in Table 4.3. Up to $n = 120$ the running time increases only very slightly, suggesting that algorithm scales well with the number of automata. Beyond $n = 120$ branching occurs again and impacts running time similarly to before. This, presumably, happens because the starting values of the clocks of the automata are closer the more automata there are. For $n = 100$ the starting values are $0, 0.01, 0.02 \ldots$ for example.

Increasing the maximal jump depth also reintroduces the problems of precision and branching again. We were not able to obtain an analysis result with a jump depth of 30. We suggest two approaches to deal with this problem in Section 5.2 of the next chapter.

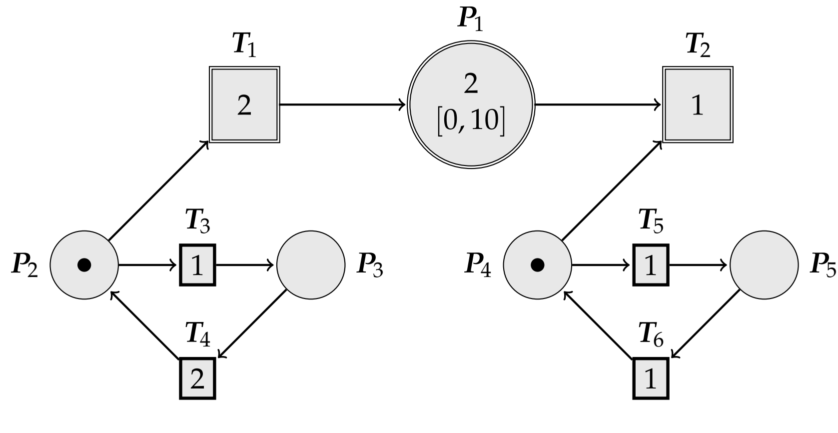


FIGURE 4.2: A simple hybrid Petri net.

4.1.5 Conclusion

This specific model, with its linear flow, may be better analysed with specialized methods that compute continuous successors in one step. These methods fail however if we replace the clocks' behavior by non-linear functions. The model also still serves as a useful example in its current form and shows some of the strengths and weaknesses of our approach.

Both algorithms suffer from branching, but this problem can be alleviated to such an extent by smaller time steps, that exponential model growth becomes the limiting factor. For the compositional algorithm, exponential model growth can be overcome entirely by the optimization from Section 3.3.2. For large n or larger maximum jump depths branching is reintroduced and becomes the limiting factor again.

The plain algorithm does not scale well with the number of automata, but the option to use more precise representations is available to analyse with a larger maximum jump depth for few automata. The compositional algorithm allows the analysis of many automata but is limited in jump depth by precision and branching. No solution to this is implemented, but there are directions for further work on this.

4.2 Hybrid Petri Nets

Hybrid Petri nets as explored for example in [DA01] can naturally be translated to compositional linear hybrid automata using only labels for synchronization. We will look at the simple example seen in Figure 4.2 and only informally discuss the semantics relevant to this example. Our translation to a hybrid automaton can be used as a formal definition of the net's semantics.

4.2.1 Petri Net Semantics

In this hybrid Petri net there are discrete places and discrete transitions between them, here P_2, P_3, P_4, P_5 and T_3, T_4, T_5, T_6. Each discrete place P_i has a number $m_i \in \mathbb{N}$ of tokens. The initial number of tokens is indicated by dots in the node in the graphical representation. Each discrete transition T_i has a delay $d_i \in \mathbb{R}_{\geq 0}$ associated with it, which is denoted by a number in the node. A transition is enabled when its source place has at least one token. If the transition T_i is enabled for d_i time, it fires. This moves one token from its source to its target place.

There are also a continuous place P_1 and continuous transitions T_1, T_2. The continuous place can be thought of as a tank with an inflow and an outflow. We denote its filling level by m_1 for consistency with discrete places. The initial level is indicated by a number in its node again. There are also bounds on its level. Here we are interested in whether and when those are violated and stop the analysis when they are. The change, i.e. the derivative, of the tank's level is the difference between the flow of its enabled input transition and its enabled output transitions T_1, T_2.

A continuous transition has a flow denoted in its node. It is enabled if its source place, which is discrete, is enabled. For example T_1 is enabled if at least one token resides in P_2.

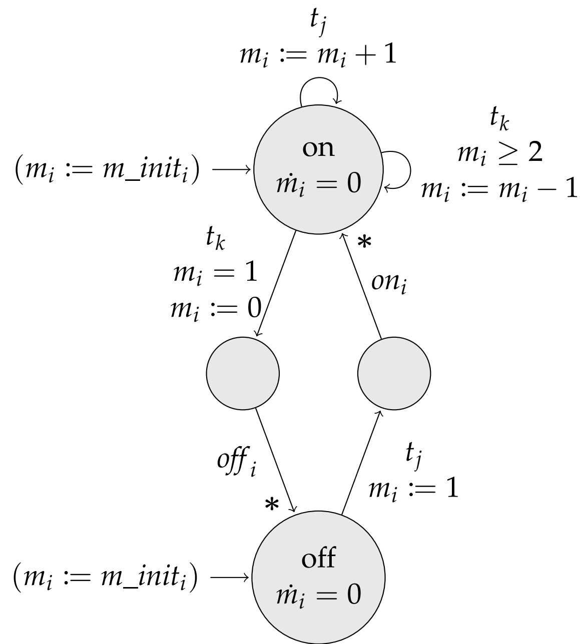


FIGURE 4.3: Translation of a discrete place P_i with an input transition T_j and an output transition T_k. Only one of "on" and "off" are initial locations. The initial location is on if $m_init_i \geq 1$, off otherwise.

4.2.2 Hybrid Automaton Translation

We will first do an intuitive translation of the Petri net to hybrid automata. We translate each place and transition into a hybrid automaton, except that continuous transitions have no internal state other than a boolean toggle between on and off, so we do not model them. The continuous place instead interacts directly with the discrete places enabling its input and output transitions.

In Figure 4.4 you can see the automaton corresponding to a continuous place P_i with the discrete place P_j enabling its input and P_k enabling its output. The tanks level, or continuous marks, are tracked by m_i and its total flow by *flow*. The requirement that its level must stay within $[0, 10]$ is translated to an invariant, so violating these bounds

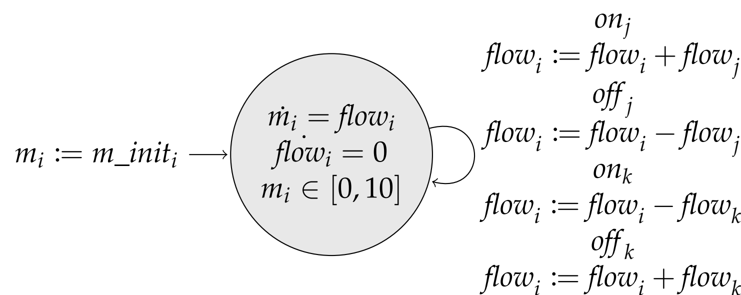


FIGURE 4.4: Translation of a continuous place P_i with an input transition enabled by P_j and an output transition enabled by P_k. The four labels and resets are to be interpreted as four distinct self loops.

causes a deadlock in the the automaton. Its level is initialized to the appropriate value and then updated by four self loops edges that correspond to each input or output turning on or off. The continuous place itself has no events to report.

We continue with discrete places. The result of the translation of a discrete place P_i with input T_j and output T_k can be seen in Figure 4.3. Again, the marks are tracked by m_i but the changes in value are discrete in this automaton. The label t_k corresponds to the output transition T_k firing and thus a mark is removed. For this to happen, the place must have had at least one mark and thus been in the "on" location. If the last mark is removed, a jump to the off location is taken. A jump back to the "on" location is taken when the input transition T_j fires. There are two intermediate states, that no time can pass in, that are jumped through on the way from "on" to "off" and the other way around. They are required because the jumps between states must synchronize with the firing of transitions and then synchronize with the on_i and off_i labels.

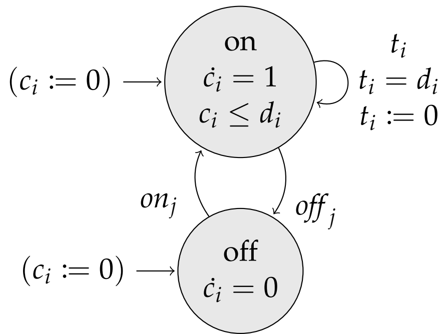


FIGURE 4.5: Translation of a discrete transition T_i with a source place P_j. Only one of "on" and "off" are initial locations. The initial location is on if $m_initial_j \geq 1$, off otherwise.

Finally, we consider the translation of discrete transitions, shown in Figure 4.5. The only variable of the automaton is a clock measuring the time that the transition is enabled. When the clock value reaches the delay d_i, the transition fires. This resets the clock and communicates the firing through the label t_i. The transition switches locations in synchronization with its source place.

Analysis. Trying to use either of the two algorithms on the Petri net from Figure 4.2 using the translation above shows disappointing results. We can not even analyse up to a global time horizon of four time units with a two-minute timeout. Before timing out over 55.000 nodes are generated, so the reason is clearly branching again, just as observed in Section 4.1. One reason for the branching is that the discrete transitions in the left and the right cycle very often fire at the exact same time. This introduces a branch because the two cases of the left or the right transition firing first are analysed in different branches of the computation tree. Since there is real non-determinism in the hybrid automaton model here, this can not be avoided with higher precision. We suggest a possible solution in Section 5.2.3.

However, the branching problem is only this egregious because of the intermediate states in the translation of discrete places. They additionally cause branching whenever a transition fires because both the source and target of the firing transition jump to intermediate states, and then it is non-deterministic which one takes the second jump first. This causes two branches to be created. When two transitions fire at the same time six branches are created, instead of the two branches one would expect.

4.2.3 Improved Translation

We can improve the model by merging the labels on_i and off_i of each discrete place P_i, with the labels t_j and t_k of their input and output transitions. We now use labels

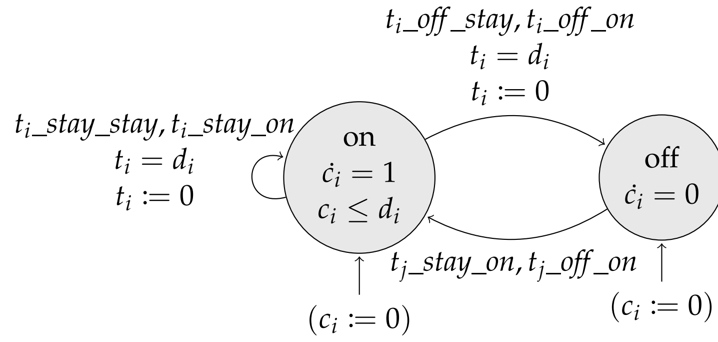


FIGURE 4.7: Translation of a discrete transition T_i whose source place has the input T_j. Only one of "on" and "off" are initial locations. The initial location is on if $m_initial_j \geq 1$, off otherwise.

$t_i \times \{stay, off\} \times \{stay, on\}$ for each discrete transition T_i. They signify that the transition has fired and that the source has either stayed in location "on" or switched to "off" and the target has similarly either stayed in "off" or switched to "on".

An edge that has the label on_i for some discrete place P_i above, can now be changed to use the labels $t_j_stay_on$, $t_j_off_on$ for each input of transition T_j of P_i. Replacing off_i works the same way with $t_j_off_stay$, $t_j_off_on$ for outputs T_k of P_i. The translation of continuous places does not change other than the replacement we just specified. The improved translations of discrete transitions and places are shown in Figure 4.7 and 4.6 respectively.

Compared to the first translation, we are reducing branching at the cost of additional labels. This tradeoff is worthwhile for this specific example. For arbitrary hybrid Petri nets a more complex translation is needed. If a similar tradeoff can be made there is an interesting question for future research. One may also use a discrete synchronization mechanism that is different from label synchronization altogether, as we touch on in Section 5.2.4.

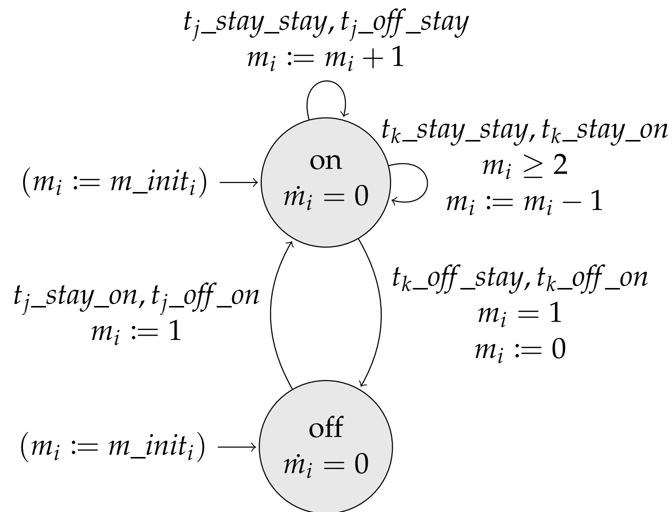


FIGURE 4.6: Translation of a discrete place P_i with an input transition T_j and an output transition T_k. Only one of "on" and "off" are initial locations. The initial location is "on" if $m_init_i \geq 1$, "off" otherwise.

Analysis. The results of running our analysis on the improved translation of the Petri net are summarized in Table 4.4. Considering that before there were over 55.000 nodes for $T_g = 4$ already, we can see that significantly fewer nodes are being generated. The compositional algorithm with the optimization from Section 3.3.2 enumerates exactly one edge per node, while the unoptimized version enumerates four edges per node. The differences in runtime are however not significant enough to draw any conclusions. The plain algorithm is about two to

TABLE 4.4: Results of running the improved Petri net translation with the compositional, optimized compositional and plain algorithms. Run time in seconds and number of nodes in computation tree for different global time horizons T, $\delta = 10^{-3}$ and unbounded jumps. Time out is two minutes.

Algo.	$T =$	2	4	6	8	10	12	14	16	18	20
comp	time	0.05	0.14	0.34	0.92	2.36	5.90	16.6	45.7	117	TO
comp*	time	0.16	0.19	0.32	0.85	2.81	5.33	14.9	34.6	87.0	TO
plain	time	0.12	0.33	0.86	2.54	7.11	16.8	MO	MO	MO	MO
all	nodes	7	31	71	167	551	1.1k	2.7k	8.8k	19k	>21k

three times slower than the compositional one. We conjecture that this ratio increases with the size of the model. The plain algorithm is also less memory efficient than the compositional one, this may however be caused by implementation details rather than begin a property of the approach. Reducing the non-determinism in the model by changing the delay of T_3 to 0.75 in Figure 4.2 drastically reduces the number of nodes generated. That allows us to analyse with $T_g = 30$ and larger.

Chapter 5

Conclusions

5.1 Summary

We built upon existing methods to implement an algorithm that scales significantly better for compositional automata with many components. We handle each component automaton separately and handle synchronization explicitly, which is why we consider only automata whose synchronization is based on labels.

Since we keep computations separate for each automaton and apply synchronization as late as possible, we were able to implement measures to reduce the number of both generated locations and edges. In both cases the numbers went from exponentially growing to feasible magnitudes, as our experiments have shown.

We also extended the label synchronization mechanism to allow sets of labels, which improves performance and allows for new ways of modeling, which we showcased on two examples.

5.2 Future Work

During the development of this work, opportunities for extensions that support further applications revealed themselves and we encountered several possible optimizations. We will elaborate on these extensions and optimizations. First, we present two ways to improve the precision of the algorithm, an optimization for stutter jumps and the possibility of introducing an explicit time dimension. To deal with branching, caused by non-determinism, we propose detecting fixed points. We end on the less concrete research directions of more advanced discrete synchronization and extending our algorithm to work with specialized reachability analysis of subclasses of hybrid automata.

5.2.1 Stutter Jump Optimization

When the discrete successors are computed for some node with $\mathit{full} = (\widehat{F}_1, \ldots, \widehat{F}_n)$ then for each edge that is enabled, i.e. *SatInds* is not empty, we potentially aggregate segments, compute the reset and do a first segment computation in the corresponding child node. We perform these steps for each of the n flowpipes, even if for some of them the edge they take is a stutter edge. Recall that a stutter edge is an edge that does not change the location, i.e. a self loop, and that does not change the variable valuation.

In addition to the time that these computations take, they also introduce over approximation. Aggregation causes segments to cover larger time intervals each time it is applied, which increases over approximation. The first segment computation also increases the width of the covered time interval by one time step and adds over approximation through bloating.

For some $r_1, r_2' \in \mathbb{R}_{\geq 0}$, $s_1, \ldots, s_4 \in St_H$ and a stutter edge e we can simplify $s_1 \xrightarrow{r_1} s_2 \xrightarrow{e} s_3 \xrightarrow{r_2} s_4$ to $s_1 \xrightarrow{r_1+r_2} s_4$ because $s_2 = s_3$. This means that we can ignore stutter jumps and compute more segments in the existing flowpipe instead. The child node that corresponds to a stutter edge could theoretically have the same *init* set as we defined so far, but an efficient implementation should not compute it because it adds no information that is not already available in the computation tree. The flowpipe in the child node would start with the interval of satisfying segments which were chosen for aggregation from the corresponding flowpipe in the parent node. Beyond those, the flowpipe would contain the segments computed from the last segment in the interval.

With this optimization, matching flowpipe segments to another becomes more difficult. Segments from different flowpipes in the same node with the same index no longer necessarily cover the same time interval. Consequently, the reasoning from Section 3.2 can no longer be applied and segments can not be matched one to one based on their indices to obtain the set of reachable states as their cartesian product. In Figure 5.1 you see an example where two segments are initially one time step wide, then for a jump, two segments are aggregated to obtain $V'_{2,1}$.

Instead we need to keep track of the time interval that each segment covers. To obtain the set of states reachable in some time interval, we must find intervals of segments for each of the flowpipes of a node, such that the segments in each interval cover the time interval of interest together. Notably, these matchings are not necessarily unique. As a side effect, this would allow us to use a different aggregation strategy for each automaton.

It is important to handle local time horizons in some way other than by an extra variable, because the clock variable for the time horizon is reset on every jump, preventing all jumps from being stutter jumps.

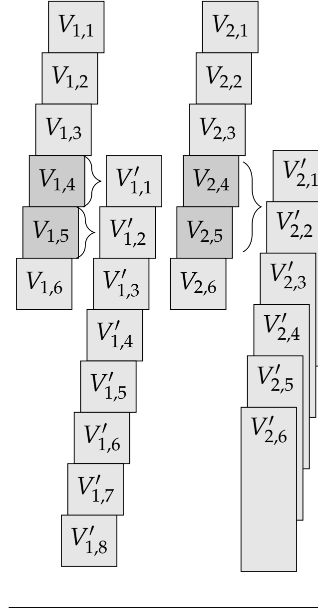


FIGURE 5.1: Schematic of flowpipes before and after a jump. The vertical position and extend of segments corresponds to the time interval that they cover. The satisfying indices are $\{4, 5\}$.

5.2.2 Explicit Time Dimension

In Section 3.2 and Example 4 we discussed the loss of precision that comes with our approach. In particular that the information about the relation of variables of different component automata is lost. This problem can potentially be addressed by introducing an explicit time variable into each component automaton. The synchronization of the automata on the time dimension is only implicitly handled in our current approach, by matching segments by index. Dealing with this synchronization explicitly should make it possible to lose no information.

During the computation of discrete successors of some node $\mathtt{n}$, the non-empty intersection of a guard with some segment $V_{1,j}$ of the first flowpipe $\widehat{F}_1$ results in a segment $V_{1,j}^{sat}$. If it has an explicit time dimension, we can project it to that dimension and we obtain an interval $[t_l, t_u] \subseteq \mathbb{R}_{\geq 0}$ that contains the lengths of all time steps that result in valuations that enable the edge in question. The segment $V_{2,j}$ of the next flowpipe should then be intersected with its corresponding guard, but also with the constraints defined by $[t_l, t_u]$. Projecting the result to the time dimension again results in a narrower interval. The process is repeated for all flowpipes.

It is also possible to compute a more precise set of reachable states. Given some index j, we used $\bigtimes_{i=1}^{n} V_{i,j}$ as an approximation of the set of states reachable by an according set of trajectories, defined by the path leading to $\mathtt{m}$. We assume a representation that uses linear constraints. The constraints of $V_{i,j}$ only use variables of the i-th component automaton and thus implicitly do not constrain any other variables. Because of this, the cartesian product $\bigtimes_{i=1}^{n} V_{i,j}$ is the conjunction of the constraints of the valuation sets $V_{i,j}$, $i \in [1, n]$.

If the component automata each have a time variable, then the constraints representing the cartesian product also include that variable. We can apply one step of the Fourier Motzkin variable elimination procedure to remove the time dimension, projecting on the remaining dimension. This finally gives us a closer approximation of the set of reachable states. Note that this can not only be applied to general convex polyhedra, which might be the first representation that comes to mind, but also more performant representations like template polyhedra, as they are implemented in HyPro, see [Tse20].

Example 5 (Explicit Time Dimension)**.** We replay the scenario from Example 4 with an explicit time dimension. Using the compositional algorithm, we obtain the line segment $\{(x,t) \mid x = t,\ x \in [0,1]\}$ as the first segment for H_1. For H_2 we similarly obtain $\{(y,t) \mid y = t,\ y \in [0,1]\}$.

Given the guard $x = 1$ for H_1 and no guard for H_2, we compute

$$\{(x,t) \mid x = t,\ x \in [0,1]\} \cap \{(x,t) \mid x = 1\} = (1,1).$$

Projecting to the t variable yields the point interval $[1,1]$. There is no guard for H_2, so we compute

$$\{(y,t) \mid y = t,\ y \in [0,1]\} \cap \{(y,t) \mid t = 1\} = (1,1).$$

Computing the reachable set as described above yields

$$\{(x,y,t) \mid x = t,\ y = t,\ x \in [0,1],\ y \in [0,1]\}.$$

Eliminating t and leaves us with $\{(x,y) \mid x = y,\ x \in [0,1],\ y \in [0,1]\}$, which is exactly the line segment we would obtain using the plain algorithm.

Whether this approach is computationally feasible or erases all performance improvements gained using our approach is unclear. Projection using the Fourier Motzkin procedure has doubly exponential complexity, but here only a single variable has to be eliminated. Projecting to the time dimension may also be feasible, because it can be achieved by solving two linear programs, one finding the smallest value of the time variable, the other the largest value. The procedure can also be applied selectively for specific edges or specific segments that intersect forbidden states.

5.2.3 Fixed Points

In Section 4.2 we observed that our Petri net model included indeterminism that caused branching. We were able to improve the model to reduce branching, but it was still the limiting factor, prohibiting the analysis with large time horizons. The branching occurs specifically because two edges are enabled by the same segments and thus there are two possible orders that they could be taken in. Semantically it does however not matter which one is taken first and the set of reachable states

should accordingly be the same for both orders. This means that fixed point detection could be used to terminate one of the two branches.

Checking newly computed segments against all previously created segments is very expensive both in time and space. Since it takes two jumps for the two branches to reach nodes that should be identical to each other, it is sufficient to check against segments of nodes up to two levels above the node being checked. In this case it also suffices to check the newly computed initial sets against existing initial sets.

5.2.4 Advanced Discrete Synchronization

We extended the discrete synchronization mechanism of labels to allow sets of labels. This is only one possible extension, which can also be simulated using ordinary hybrid automata. At the point of discrete synchronization during the analysis, it is known which edges are enabled for which time intervals. Based on this information an arbitrary synchronization algorithm can be designed. For the example of hybrid Petri nets, each event, i.e. each time point at which one or more transitions fire, could be handled at once. This also allows to resolve indeterminism.

Discrete synchronization can also be extended into an entirely different direction. Since we can project to one or more variables, we can use the resulting set in guards and resets of edges of other automata. Resets like $x := 2y$, where x is a variable of one automaton and y a variable of another, can be implemented for example.

5.2.5 Extension to Discrete and One Step Reachability

Dealing with component automata individually allows us to analyse their syntax individually as well. We can thus determine for each component automaton whether it lies within a subclass of hybrid automata for which a method for the computation of continuous successors is available which is faster or more precise than flowpipe construction. This method has to provide certain information and operations to be compatible with our approach.

The method has to compute the set of states reachable from a given valuation set in a given location. The representation of the result has to support some form of intersection with guards and resets have to be applicable. In addition to this it has to be possible to obtain an over approximation of the enabling time intervals. That is the time intervals that contain the durations of all time steps that result in a valuation that satisfies the guard.

Such a method is conceivable for various flavours of discrete automata. For hybrid automata with flows with constant derivatives there are one-step approaches available that may be adaptable to fulfill our requirements [Sch19, Sec.8.1].

Bibliography

[Alt+18] Matthias Althoff et al. “ARCH-COMP18 Category Report: Continuous and Hybrid Systems with Linear Continuous Dynamics”. In: *Proc. of ARCH'18*. Vol. 54. EPiC Series in Computing. EasyChair, 2018, pp. 23–52. DOI: 10.29007/73mb.

[Alt15] Matthias Althoff. “An Introduction to CORA 2015”. In: *Proc. of ARCH'15*. Vol. 34. EPiC Series in Computing. EasyChair, 2015, pp. 120–151. DOI: 10.29007/zbkv.

[And+17] Sidharta Andalam et al. “Hybrid Automata Model of the Heart for Formal Verification of Pacemakers”. In: *Proc. of ARCH'16*. Vol. 43. EPiC Series in Computing. EasyChair, 2017, pp. 9–17. DOI: 10.29007/822m.

[BBJ15] Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. “HYST: A Source Transformation and Translation Tool for Hybrid Automaton Models”. In: *Proc. of HSCC'15*. Association for Computing Machinery, 2015, pp. 128–133. DOI: 10.1145/2728606.2728630.

[BHS17] Sergiy Bogomolov, Christian Herrera, and Wilfried Steiner. “Verification of Fault-Tolerant Clock Synchronization Algorithms”. In: *Proc. of ARCH'16*. Vol. 43. EPiC Series in Computing. EasyChair, 2017, pp. 36–41. DOI: 10.29007/hq8s.

[DA01] René David and Hassane Alla. “On Hybrid Petri Nets”. In: *Discrete Event Dynamic Systems* 11.1–2 (2001), pp. 9–40. DOI: 10.1023/A:1008330914786.

[Fre05] Goran Frehse. “Compositional Verification of Hybrid Systems Using Simulation Relations”. Dissertation. Radboud Universiteit Nijmegen, 2005.

[Fre16] Goran Frehse. “Scalable Verification of Hybrid Systems”. Habilitation à diriger des recherches. Univ. Grenoble Alpes, 2016. URL: https://hal.archives-ouvertes.fr/tel-01714428.

[FZK04] G. Frehse, Zhi Han, and B. Krogh. “Assume-guarantee reasoning for hybrid I/O-automata by over-approximation of continuous interaction”. In: *Proc. of CDC'04*. Vol. 1. IEEE, 2004, pp. 479–484. DOI: 10.1109/CDC.2004.1428676.

[Hen96] T. A. Henzinger. “The theory of hybrid automata”. In: *Proc. of LICS'96*. IEEE, 1996, pp. 278–292. DOI: 10.1109/LICS.1996.561342.

[Leo+19] Francesco Leofante et al. “Engineering Controllers For Swarm Robotics Via Reachability Analysis In Hybrid Systems”. In: *Proc. of ECMS'19*. European Council for Modelling and Simulation, 2019, pp. 407–413. DOI: 10.7148/2019-0407.

[Sch19] Stefan Schupp. “State Set Representations and their usage in the Reachability Analysis of Hybrid Systems”. Dissertation. RWTH Aachen University, 2019. DOI: 10.18154/RWTH-2019-08875.

[SNÁ17] Stefan Schupp, Johanna Nellen, and Erika Ábrahám. “Divide and Conquer : Variable Set Separation in Hybrid Systems Reachability Analysis”. In: *Proc. of QAPL'17*. Vol. 250. EPTCS. Open Publishing Association, 2017, pp. 1–14. DOI: 10.4204/EPTCS.250.1.

[Tse20] Phillip Tse. *Efficient Polyhedral State Set Representations for Hybrid Systems Reachability Analysis*. 2020. URL: https://ths.rwth-aachen.de/wp-content/uploads/sites/4/tse_master.pdf (visited on 10/27/2020).