hybr2d Theory of Hybrid Systems Informatik 2 | RWTH AACHEN UNIVERSITY

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

Bachelor of Science Thesis

Reachability Analysis of Hybrid Systems with Label Synchronization

Haiyan Saadi

Communicated by
Prof. Dr. Erika Ábrahám

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
József Kovács

Aachen, July 10, 2024

Abstract

In this work we extend a forward reachability analysis method for rectangular automata to a concurrent synchronized analysis of multiple automata. Successor states of rectangular automata can be exactly represented by a set of polyhedra, avoiding the need for over-approximation. We introduce a synchronization technique based on projecting the local states on the time dimension and finding a common time interval for every synchronizing jump, then computing successor state sets only within this interval. The synchronization context of each state set is saved in the search trees of the individual automata, which eliminates the need for computation and comparison of paths in order to ensure correct synchronization. Our method shows great prospect in dealing with high-dimensional models and is extendable to multiple state set representations and automata types.

Contents

Notation **9**

1 Introduction **11**

2 Preliminaries **13**
2.1 Hybrid Systems 13
2.2 State Set Representations 17
2.3 Reachability Analysis 19
2.4 Flowpipe Construction 19
2.5 Variable Set Separation 23

3 Synchronized Reachability Analysis of Rectangular Automata **25**
3.1 Global Time and Synchronization Labels 25
3.2 Label Synchronization 29

4 Experimental Results **35**
4.1 Implementation 35
4.2 Benchmarks 37

5 Conclusion **43**
5.1 Summary 43
5.2 Discussion and Future Work 43

Bibliography **45**

Notation

Throughout this work, we will use the following notational conventions.

Notation (Variables). *We will use small-case letters x, y, z, ... to describe real-valued variables of hybrid automata. For a variable $x \in \mathbb{R}$ of a hybrid automaton*

- *$\dot{x}$ denotes the first time derivative of x.*
- *x' denotes the value of x after a discrete transition.*

We extend this notation to sets of variables, e.g., if $X = \{x_1, \dots, x_n\}$ is a set of variables then $\dot{X} = \{\dot{x_1}, \dots, \dot{x_n}\}$ is the set of first time derivatives and $X' = \{x'_1, \dots, x'_n\}$ is the set of variable values after a discrete transition.

Notation (Variable Valuations). *Small-case letters v, u, ... are used to describe variable valuation, i.e., $v \in \mathbb{R}^d$ is a real-valued vector of length d.*

Notation (Search Tree Nodes). *We also use small-case letters like m, n, p, s, ... to refer to nodes of the search tree (see Definition 3.1.1), r usually refers to the root node of a tree.*

Notation (Set Cardinality). *For a set X, we denote its cardinality by $|X|$.*

Notation (Power Set). *For a set X, we denote its power set by $2^X = \{A \mid A \subseteq X\}$.*

Notation (d-dimensional Space). *For a set X we denote the d-dimensional space of X by $X^d = \underbrace{X \times \cdots \times X}_{d \text{ times}}$. If not stated otherwise we assume $d \in \mathbb{N}_{>0}$.*

Notation (Logical Formluas). *We use small-case Greek letters φ, ψ, ... to describe logical formulas.*

Notation (Predicates). *For a set of variables X we use the notation $Pred_X$ to describe all predicates over X, which are quantifier-free arithmetic formulas with variables from X.*

Notation (Intervals). *For $S \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$, and $a,b \in S$ we use the following interval notation.*

- $[a,b] \coloneqq \{s \in S \mid a \leq s \leq b\}$
- $[a,b) \coloneqq \{s \in S \mid a \leq s < b\}$
- $(a,b] \coloneqq \{s \in S \mid a < s \leq b\}$
- $(a,b) \coloneqq \{s \in S \mid a < s < b\}$

We use the notation $\mathbb{I}$ for the set of all real-valued intervals.

Chapter 1

Introduction

Hybrid systems are characterized by a combination of discrete state changes and continuous evolution. They are deployed in many applications ranging from heating control units and autopilot systems in automotive and aviation to production plants and medical devices. Many of these applications are safety-critical, and ensuring the correct behaviour of these systems is essential and has been the focus of much research in the past years. Formal methods have been investigated and developed to verify and validate the safety of hybrid systems, some of which are based on theorem proving [PQ08] and others on satisfiability checking [FHT$^+$07].

One prominent safety verification approach is based on iterative forward reachability analysis, which verifies safety properties by computing the set of reachable states and checking them against a set of "bad" or undesired states. In forward reachability analysis hybrid systems are usually modelled as hybrid automata, and the set of bad states can be described by a logical formula over the automaton's variables.

Since the reachability problem is undecidable for general hybrid automata, many tools use approximative methods to compute the reachable set, e.g., *flowpipe-construction-based* methods like SpaceEx [FLGD$^+$11] and CORA [Alt15] and many more. Despite the advancement of such reachability analysis methods, scalability remains a major challenge. System models with high dimension experience complex behaviour and require improved analysis techniques to ensure their safety. Decomposition of large complex systems has been applied to some flowpipe-construction-based methods, where the system is split into multiple lower-dimensional subsystems with limited dependencies, allowing the analysis of each subsystem separately [CS16, BFF$^+$18].

Methods that use decomposition like *Variable Set Separation* [SNÁ17] introduce imprecision to the computed reachable set because they ignore the dependencies between the variables, the most important of which is time. Running the analysis on the subsystems individually assumes a different time evolution for each subsystem, whereas in the original global system time evolves unitarily over all system variables. One way of including the notion of global time into the analysis of multiple hybrid systems is by synchronizing the evolution of the subsystems. Traditionally this can be achieved by the construction of a product automaton, which is the parallel composition of the input automata. However, this approach just yields back the original automaton before decomposition.

Therefore, different "synchronization" methods have been proposed to compute the reachable set of multiple hybrid automata without the need to build the product automaton. e.g., *shallow synchronization* [BCL+10] explores the local time evolution of every automaton in a network of hybrid automata until they perform a synchronizing transition, in which case they realign their times. The tool BACH 2 [BLW+10] uses a path-oriented technique to compute the bounded reachable set of linear hybrid automata (LHA) compositions. Paths are encoded to a set of linear constraints and special constraints are added for synchronization control.

In this thesis, we introduce a synchronization approach for forward reachability analysis of multiple rectangular automata based on synchronizing transitions with common labels. The algorithm distinguishes between two types of transitions, ones that allow an automaton to evolve locally, and others that force the system to synchronize jump successor computation.

We begin by presenting the fundamental concepts and definitions in Chapter 2. Specifically Section 2.4 describes flowpipe construction for linear hybrid automata, as well as the exact computation of jump and time successors of rectangular automata. In Chapter 3 we introduce our synchronized forward reachability analysis approach. Section 3.1 explains the new data structures and their functional objectives in our approach. We redefine the search tree of the forward reachability analysis and extend it with vital synchronization information. We then present our method in Section 3.2, which involves a general forward reachability analysis algorithm in Section 3.2.1, the computation of successor states in Section 3.2.2 and the search algorithm for finding synchronizing states in Section 3.2.3. In Chapter 4 we present and evaluate our implementation in the tool HYPRO. We use three well-known benchmarks to assess runtimes, memory-usage and the complexity of our proposed approach. Finally, we conclude in Chapter 5 by summarizing our findings, discussing the contribution and limitations and giving an outlook into potential future research directions.

Chapter 2

Preliminaries

2.1 Hybrid Systems

Hybrid systems are systems that combine *discrete* and *continuous* behaviours. A discrete component exhibits instantaneous state changes, like the change in direction when a free-falling ball bounces off the ground, or a sensor signal reporting a certain binary value (e.g., *on/off* or *alarm/no alarm*). Whereas a continuous component evolves according to some dynamical law in a real-valued state space, like time or speed in a physical system. This combination of continuous and discrete aspects leads to complex system behaviours and poses the challenge of formal reasoning and analysis of such systems. An example of a hybrid system, to which we have already hinted, is a ball bouncing off the ground from a certain altitude, the speed of the ball is changing continuously with time according to the gravitational acceleration, and a bounce off the ground prompt an instantaneous change in direction. Another example is a heating unit controlled by a digital controller. The heating unit is turned on when the temperature falls below a certain lower threshold and turned off when the temperature rises above an upper threshold, in each of the two states the temperature evolves continuously.

Hybrid Automata. In order to verify properties of hybrid systems, a formal modelling language was introduced that accurately describes the system and its behaviour. A *hybrid automaton* encodes all the relevant properties of a hybrid system as well as describes its evolution. Each variable of the hybrid automaton represents a property of the system that evolves continuously through time. For a variable x, $\dot{x}$ denotes the first derivative of x during continuous change and x' denotes the value of x after a discrete change, i.e., applying a reset function to the variable x. Furthermore, $Pred_{Var}$ denotes the set of all predicates over the set Var.

Definition 2.1.1 (Hybrid Automaton [Sch19]). *A hybrid automaton is a tuple* $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ *where*

- *Loc is a finite set of locations or control modes;*
- *$Var = \{x_0, x_1, \ldots, x_{d-1}\}$ is an ordered finite set of real-valued variables, where d is the dimension of $\mathcal{H}$;*
- *Lab is a finite set of synchronization labels containing the stutter label $\tau \in Lab$;*

- *Flow* : *Loc* $\rightarrow$ *Pred* $_{Var \cup \dot{Var}}$ *defines the flow or dynamics of each location;*
- *Inv* : *Loc* $\rightarrow$ *Pred* $_{Var}$ *defines an invariant for each location;*
- *Edge* $\subseteq$ *Loc* $\times$ *Lab* $\times$ *Pred* $_{Var}$ $\times$ *Pred* $_{Var \cup Var'}$ $\times$ *Loc is a finite set of discrete transitions or jumps, where a jump* $(l_1,a,g,r,l_2) \in$ *Edge is defined by its source location* l_1*, synchronization label* a*, guard* g*, reset function* r *and target location* l_2*;*
- *Init* : *Loc* $\rightarrow$ *Pred* $_{Var}$ *defines an initial predicate for each location.*

Valuation of the variables of the hybrid automaton are represented as vectors $v \in \mathbb{R}^d$. An invariant φ of a location l is a predicate that restricts the possible variable valuation in l to those that satisfy φ. Invariants force a discrete state change of the system before the invariant is violated. Discrete state changes are achieved through *transitions* or *jumps*. A jump $e \in$ *Edge* can be taken when the current variable valuation v satisfies the guard g and the valuation v' after applying the reset r satisfies the invariant of the target location l_2

Hybrid automata can be classified into subclasses based on their expressiveness. These subclasses are usually defined by syntactical restrictions on the type of predicates describing the automaton's flows, invariants, guards, and resets. A relatively simple subclass is *timed automata* [AD94], where every variable $x \in$ *Var* has the derivative $\dot{x} = 1$. A more expressive one is *linear hybrid automata* [ACH+95], where the flows, invariants and transition relations of the system can be defined by linear expressions over the set of variables *Var*. The reachability problem is decidable for timed automata but undecidable for linear hybrid automata.

Rectangular automata is another subclass of hybrid automata that is less expressive than linear hybrid automata but more expressive than timed automata. Variables of a rectangular automaton are pairwise independent in their time evolution, i.e., the flow and reset values of a variable x_1 is not influenced by the value of another variable x_2. Specifically, the first derivative of each variable in a rectangular automaton is given from an interval $[a,b]$, where a and b are constants. The same goes for the reset function values; when a jump is taken, the value of each variable is either left unchanged or reset non-deterministically to a value from a specific interval. This property of rectangular automata results in decoupled behaviours of the variables, because the ranges of possible values and derivative values for one variable cannot depend on the value or derivative value of another variable.

To introduce the formal definition of rectangular automata, first we need to define rectangular sets.

Definition 2.1.2 (Rectangular Set [Ábr17]). *A set* $R \subset \mathbb{R}^n$ *is rectangular if it is a Cartesian product of (possibly unbounded) intervals, all of whose finite endpoints are rational. The set of all rectangular sets in* $\mathbb{R}^n$ *is denoted* $\mathscr{R}^n$*.*
We use the notation R_i *for the projection of* R *onto the* i*-th coordinate axis, so that* $R = R_1 \times \cdots \times R_n$*.*

A rectangular automaton is a hybrid automaton whose flows, invariants and transition relations are defined by rectangular sets.

Definition 2.1.3 (Rectangular Automaton [Ábr17]). *A rectangular automaton is a tuple* $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ *where*

- Loc *is a finite set of locations or control modes;*
- $Var = \{x_0, x_1, \ldots, x_{d-1}\}$ *is an ordered finite set of real-valued variables, where* d *is the dimension of* $\mathcal{H}$*;*
- Lab *is a finite set of synchronization labels;*
- $Flow : Loc \to \mathscr{R}^d$ *defines the flow or dynamics of each location;*
- $Inv : Loc \to \mathscr{R}^d$ *defines an invariant for each location;*
- $Edge \subseteq Loc \times Lab \times \mathscr{R}^d \times \mathscr{R}^d \times 2^{\{1,\ldots,d\}} \times Loc$ *is a finite set of discrete transitions or jumps;*
- $Init : Loc \to \mathscr{R}^d$ *defines an initial rectangular set for each location.*

For the jumps, an edge $e = (l, a, g, r, jump, l') \in Edge$ may move control from location l to location l' starting from a valuation in $Inv(l) \cap g$, changing the value of each variable $x_i \in jump$ non-deterministically to a value from r_i (the projection of r to the i-th dimension), and leaving the values of the other variables unchanged. The combination of a location $l \in Loc$ and a variable valuation $v \in \mathbb{R}^d$ describe the state of the hybrid automaton $\sigma = (l,v)$. The state space of a hybrid automaton $\mathcal{H}$ is denoted by $\Sigma = Loc \times \mathbb{R}^d$.

Since our main focus in this work is rectangular automata we will now present their operational semantics. Rectangular automata have two types of semantics that govern their behaviour (see Definition 2.1.4). A time step, described by the rule $\texttt{Rule}_{\texttt{flow}}$, defines how a state of the hybrid automaton can evolve with time in a certain location l, while a jump or a discrete step, described by the rule $\texttt{Rule}_{\texttt{jump}}$, defines the instantaneous change of state from one source location to a target location, under the condition that the predecessor state satisfies the guard of the jump, and the successor state satisfies the invariant of the target location.

Definition 2.1.4 (Operational Semantics of Rectangular Automata [DSÁR23]). *The one-step semantics of rectangular automaton* $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ *is defined by the following rules:*

$$\frac{l \in Loc \qquad v \in \mathbb{R}^d \qquad t \in \mathbb{R}_{\geq 0} \qquad rate \in Flow(l) \qquad v' = v + rate \cdot t \qquad v' \in Inv(l)}{(l,v) \xrightarrow{t} (l,v')} \ \textsc{Rule}_{\textsc{flow}}$$

$$\frac{e = (l,a,g,r,jump,l') \in Edge \qquad v \in g \qquad v \in Inv(l) \qquad v' \in r \qquad \forall i \notin jump \cdot v'_i = v_i \qquad v' \in Inv(l')}{(l,v) \xrightarrow{e} (l',v')} \ \textsc{Rule}_{\textsc{jump}}$$

a step (discrete or time step) according to those semantics is called an execution step

$$\to \; = \; \xrightarrow{t} \cup \xrightarrow{e}$$

A path (or run or execution) π of $\mathcal{H}$ is a sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \dots$ such that $v_0 \in \mathit{Inv}(l_0)$ and $\sigma_i \rightarrow \sigma_{i+1}$ for all $i \geq 0$.

As hybrid automata model real-life systems, and many systems are constructed from multiple subsystems, it is important to describe the combination of multiple automata running in parallel. The parallel composition of two hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$ describes the concurrent execution of the two systems described by the automata. Time evolves simultaneously in $\mathcal{H}_1$ and $\mathcal{H}_2$, and the two automata must synchronize on common labels, i.e., transitions with a synchronization label $a \in \mathit{Lab}_1 \cup \mathit{Lab}_2$ must be taken synchronously in both automata.

Definition 2.1.5 (Parallel Composition [Sch19]). *The product automaton of two hybrid automata* $\mathcal{H}_1 = (\mathit{Loc}_1, \mathit{Var}_1, \mathit{Lab}_1, \mathit{Flow}_1, \mathit{Inv}_1, \mathit{Edge}_1, \mathit{Init}_1)$ *and* $\mathcal{H}_2 = (\mathit{Loc}_2, \mathit{Var}_2, \mathit{Lab}_2, \mathit{Flow}_2, \mathit{Inv}_2, \mathit{Edge}_2, \mathit{Init}_2)$ *with* $\mathit{Var}_1 = \mathit{Var}_2$ *is the hybrid automaton*

$$\mathcal{H}_1 \parallel \mathcal{H}_2 = (\mathit{Loc}, \mathit{Var}, \mathit{Lab}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Init})$$

where

- $\mathit{Loc} = \mathit{Loc}_1 \times \mathit{Loc}_2$
- $\mathit{Var} = \mathit{Var}_1 = \mathit{Var}_2$
- $\mathit{Lab} = \mathit{Lab}_1 \cup \mathit{Lab}_2$
- $\mathit{Flow}((l_1,l_2)) = \mathit{Flow}_1(l_1) \wedge \mathit{Flow}_2(l_2)$ *for all* $(l_1,l_2) \in \mathit{Loc}$
- $\mathit{Inv}((l_1,l_2)) = \mathit{Inv}_1(l_1) \wedge \mathit{Inv}_2(l_2)$ *for all* $(l_1,l_2) \in \mathit{Loc}$
- $\mathit{Init}((l_1,l_2)) = \mathit{Init}_1(l_1) \wedge \mathit{Init}_2(l_2)$ *for all* $(l_1,l_2) \in \mathit{Loc}$
- *Edge is the smallest set that contains for each* $(l_1,a_1,g_1,r_1,l_1') \in \mathit{Edge}_1$ *and* $(l_2,a_2,g_2,r_2,l_2') \in \mathit{Edge}_2$ *the edge* $((l_1,l_2),a,g_1 \wedge g_2,r_1 \wedge r_2,(l_1',l_2'))$ *if:*
  - *either* $a = a_1 = a_2$
  - *or* $a_1 \notin \mathit{Lab}_2$ *and* $a_2 = \tau$
  - *or* $a_2 \notin \mathit{Lab}_1$ *and* $a_1 = \tau$

The parallel composition of two hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$ describes the synchronized system which consists of $\mathcal{H}_1$ and $\mathcal{H}_2$. Analysing the synchronization of multiple automata is essential for the verification of complex systems, as it allows us to reason about the behaviour of the system as a whole. However, even if we overlook the fact that constructing the product automaton is a costly operation, the analysis of the compound automaton is still challenging, as we always risk the state-space explosion problem.

In the following we present a model of the water tank system from [Lyg04], with slight modification to the flows such that the model of the system is a rectangular automaton.

Example 2.1.1 (Water Tank System). *Assume two identical water tanks, each with a valve to control the flow of water out of the tank, and a hose that refills (with water flow* w*) exactly one of the tanks at each point in time. We denote the water level in*

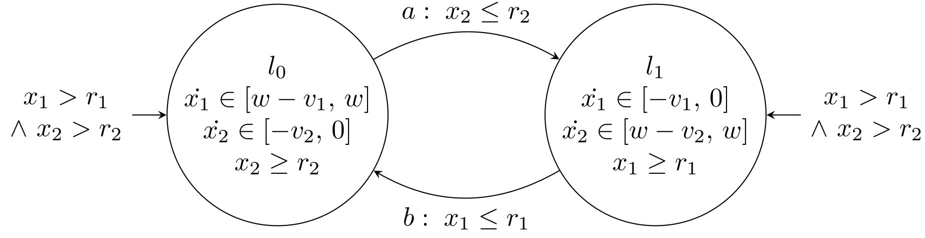


Figure 2.1: Rectangular automaton model of the water tank system.

each of the tanks by x_1 and x_2, respectively. The valve of the first tank could be closed, no water flows out of the tank, or completely open, leading to maximum water outflow and a water level decrease of v_1, or anywhere in between. Similarly, for the second tank, the state of the valve controls the water outflow between 0 and v_2. The hose fills one of the tanks causing an increase of $w \in \mathbb{R}_{>0}$ in water level. The derivative of the water level in the first tank is therefore in the interval $[w - v_1, w]$ if it's getting refilled and in the interval $[-v_1, 0]$ otherwise. The same applies for the second tank, with the derivative in the interval $[w - v_2, w]$ if it's getting refilled and in the interval $[-v_2, 0]$ otherwise. When refilling the first tank, the hose switches to the second tank when its water level x_2 reaches a given lower threshold $r_2 \in \mathbb{R}_{>0}$. The switch from the second tank to the first one works analogously when x_1 reaches some $r_1 \in \mathbb{R}_{>0}$.

The water tank system is modelled by the rectangular automaton in Figure 2.1. The automaton has two locations l_0 and l_1, where l_0 represents the state where the first tank is being refilled and the second tank is not, and l_1 represents the state where the second tank is being refilled and the first tank is not. The flow condition of each location describes the rate of change of the water level in each tank, and the invariant of each location forces the control to take the transition to the other location when the water level in the other tank reaches a certain threshold. The guards of the transitions make sure that each of the tanks get refilled until the water level in the other tank reaches the lower threshold.

2.2 State Set Representations

A state $\sigma \in \Sigma = (Loc \times \mathbb{R}^d)$ of a hybrid system is tuple (l,v) of a location $l \in Loc$ and a variable valuation $v \in \mathbb{R}^d$. Sets of valuations N which agree on the same location form a *symbolic state* $(l,N) = \{(l,v) \mid v \in N\}$ or a *state set*. To analyse hybrid systems we need to represent the state sets in a way that allows us to perform set operations like intersection, union, Minkowski sum, or emptiness test efficiently. Boxes, polytopes, zonotopes and other geometric state set representations usually offer a low computational effort for set operations, but they often introduce large over-approximations of the real state sets.

Definition 2.2.1 (Box [Sch19]). *A box representation is a vector A_B of d real-valued intervals $A_B = (A_0, \ldots, A_{d-1}) \subseteq \mathbb{I}^d$ represent a d-dimensional box.*

$$A = \left\{ x \;\middle|\; \bigwedge_{i=0}^{d-1} x_i \in A_i \right\}$$

where the d-dimensional set is spanned by the Cartesian product of the given intervals such that $Set(A_B) = A_0 \times \cdots \times A_{d-1}$.

Definition 2.2.2 (Zonotope [Sch19]). *A d-dimensional zonotope representation is a tuple* $A_Z = (c, g_0, \ldots, g_{n-1})$ *with a centre* $c \in \mathbb{R}^d$ *and a (possibly empty) sequence* $g_0, \ldots, g_{n-1}$ *of vectors from* $\mathbb{R}^d$ *called generators.* A_Z *represents the set*

$$A_Z = \left\{ c + \sum_{i=0}^{n-1} \lambda_i \cdot g_i \;\middle|\; \lambda_i \in [-1,1] \subseteq \mathbb{R}^d \right\},$$

and we use $|A_Z|$ *to denote the number of generators in* A_Z.

On the other hand, symbolic representations (e.g., support functions [BNO03] and Taylor models [Neu03]) can provide more precise results compared to geometric representations, but the computational effort for set operations can be high.

In this work, we are interested with the reachability analysis of rectangular automata, and a very important characteristic of rectangular automata is that the time successor of a state set under rectangular dynamics is always a (convex) polyhedron. Therefore, since polytopes are bounded polyhedra, we use polytopes to represent the state sets in our reachability analysis. Furthermore, polytopes are closed under most operation required for the analysis. Only the union of two convex polytopes A and B required the additional effort of computing the convex hull of the union $A \cup B$. This allows our analysis to efficiently apply operations on the state sets without having the downside of over-approximation.

Definition 2.2.3 (Polytope [Sch19]). *A d-dimensional convex polytope* P_H *in* $\mathcal{H}$*-representation is a pair* (N,c) *with* $N \in \mathbb{R}^{m \times d}$ *and* $c \in \mathbb{R}^m$ *which defines a convex set*

$$P_H = \bigcap_{i=0}^{m-1} h_i$$

as the intersection of finitely many half-spaces $\{h_0, h_1, \ldots, h_{m-1}\}$ *with* $h_i = \{x \in \mathbb{R}^d \mid n_{i,_} \cdot x \leq c_i\}$, *where* $n_{i,_}$ *is the i-th row of* N.
The same polytope can also be represented in $\mathcal{V}$*-representation as the convex hull of a finite set* $P_V = \{v_0, \ldots, v_{m-1}\}$ *of vertices* $v_i \in \mathbb{R}^d$:

$$P_V = \left\{ x \;\middle|\; x = \sum_{i=0}^{m-1} \lambda_i v_i \wedge \sum_{i=0}^{m-1} \lambda_i = 1 \wedge \lambda_i \in [0,1] \right\}$$

We note that different polytope representations lead to different computational complexities for set operations. The $\mathcal{H}$-representation is more efficient for set operations like intersection or intersecting with a hyperplane, while the $\mathcal{V}$-representation is more efficient for set operations like Minkowski sum or testing for emptiness [Sch19]. During reachability analysis, conversion method between the two representations are usually applied when the operation is too expensive or impossible in the original representation. We will see the effect of choosing the polytope representation on synchronized reachability analysis in Chapter 4.

2.3 Reachability Analysis

The importance of safe behaviour of hybrid systems e.g., autopilot systems, or pacemaker devices for the human heart highlights the need for formal verification, because faulty behaviour of such systems can have detrimental effects. Reachability analysis of hybrid systems is concerned with determining whether a system is able to reach a set of bad states P_{bad}, representing undesired or risky system behaviours. Model checking approaches to reachability analysis rely on state set exploration, where the reachable state space is explored in order to find a path from an initial state to a bad state, in other words we check if $Reach_{\mathcal{H}} \cap P_{bad} = \emptyset$, where $Reach_{\mathcal{H}}$ denotes the set of reachable states of automaton $\mathcal{H}$. In contrast to deductive verification approaches, for example, where proofs are used to verify a system instead of state set exploration.

Iterative reachability analysis methods prove safety by iteratively computing a successor (or predecessor) state from the current state. Forward reachability analysis tries to iteratively determine the successor states of all initial states, and checks if the system could reach a bad state [ACH+95]. Whereas backward reachability analysis iteratively determines the predecessor states of all bad state, and checks if the system could reach an initial state [MBT05].

Since the reachability problem for general hybrid automata is undecidable [HKPV98], we compute an over-approximation of the reachable state space. This means that we can only prove that a system is safe if we find an over-approximation of the reachable state space that does not intersect with the set of bad states. i.e., if $Reach'_{\mathcal{H}} \cap P_{bad} = \emptyset$ where $Reach'_{\mathcal{H}} \supseteq Reach_{\mathcal{H}}$ denotes an over-approximation of the reachable states of the automaton $\mathcal{H}$. On the other hand to prove unsafety, we need an under-approximation of the reachable state space to ensure that the bad state we have found is also reachable in the actual system and was not only found due to our over-approximation. These types of analysis face certain challenges, for example, computing the unbounded-time reachable set is not always possible in finite time. Therefore, it is common to resort to computing the reachable set within a specified *time horizon* T, which represents the maximum duration of uninterrupted time evolution in a certain location of the hybrid automaton. Another problem that arises is *Zeno execution*; an infinite number of discrete transitions in a finite amount of time. To deal with this problem we could introduce a bound J on the number of jumps. In unbounded reachability analysis we compute the reachable state until no new states are reached, i.e., a fixed point is reached.

2.4 Flowpipe Construction

The idea of flowpipe-construction-based reachability analysis is to perform forward reachability analysis by computing the successor states of the initial states by discretizing the time horizon T into N time steps ($\delta = \frac{T}{N}$) and over-approximating the reachable state set for each time step. In other words, the flowpipe of a reachable state of the hybrid automaton is a set of segments, that each over-approximates the actual reachable state set by time evolution for a time interval of length δ. All flowpipe segments in a location l must satisfy the invariant of l. After computing the flowpipe, we can compute jump successors by intersecting the flowpipe segments with the guard and applying the reset function on those segments that satisfy the guard, and if the

invariant of the target location is satisfied after applying the reset, we continue with the flowpipe construction in the target location and so on.

For **LHA II** each control mode specifies a dynamic system by a system of linear differential equations

$$\dot{x} = \mathcal{A}x \tag{2.1}$$

over the variables $x = (x_0, \ldots, x_{d-1})^T$ of the given hybrid automaton $\mathcal{H}$, with $\mathcal{A} \in \mathbb{R}^{d \times d}$ a being a coefficient matrix. The solution of 2.1 is given by

$$x(t) = \underbrace{e^{t\mathcal{A}}}_{\Phi} \cdot x(0) \tag{2.2}$$

and it describes the states that are reached at time point t when starting form initial state $x(0)$ at time point $t = 0$ and following the flow specified by the matrix $\mathcal{A}$. This linear transformation of initial variable valuation $x_0 = x(0)$ can directly be extended to sets of variable valuations N such that for a set N_0 of initial variable valuations

$$N_t = \Phi \cdot N_0.$$

With this computation we can compute the reachable state set at any time point t but still does not give us the reachable state set within a time interval. Several methods overcome this problem by over-approximating the reachable state set for a time interval $[0,\delta]$ [LG09, LG10, Dan00]. The idea is to over-approximate the error α between the over-approximation Ω of the reachable state set for time interval $[0,\delta]$ and the actual reachable state set

$$Reach_{[0,\delta]} = \left\{(l,v) \;\middle|\; v = e^{t\mathcal{A}} \cdot x_0,\, t \in [0,\delta],\, x_0 \in N_0\right\}$$

We do this by considering the union of line segments between the initial state and the state reached after time δ for each initial state $x_0 \in N_0$, which defines the convex hull of N_0 and $N_\delta = e^{\delta\mathcal{A}} \cdot x(0)$. For an initial state $x \in N_0$ and the state $r = e^{\delta\mathcal{A}} \cdot x$ that is reached from x at time point δ the connecting line segment between x and r is defined as

$$\left\{s_x(t) = x + \frac{t}{\delta}(r - x) \;\middle|\; t \in [0,\delta]\right\}$$

now, the error between the line segment $s_x(t)$ and the actual trajectory $\zeta_x(t) = e^{t\mathcal{A}}x$ can be quantified as

$$\|\zeta_x(t) - s_x(t)\| = \left\|e^{t\mathcal{A}}x - x - \frac{t}{\delta}(e^{\delta\mathcal{A}} - I)x\right\|$$

In [Gir05] this error was approximated using Taylor's theorem

$$\left\|e^{t\mathcal{A}}x - x - \frac{t}{\delta}(e^{\delta\mathcal{A}} - I)x\right\| \leq \underbrace{(e^{\delta\|\mathcal{A}\|} - 1 - \delta\|\mathcal{A}\|)\|x\|}_{\alpha}$$

this result allows us to bloat the convex hull with a ball $\mathcal{B}_\alpha$ of radius α and safely over-approximate the reachable state set for time interval $[0,\delta]$

$$\Omega = (N_0 \cup N_\delta) \oplus \mathcal{B}_\alpha$$

where the operator $\oplus$ denotes the Minkowski-sum of two sets.

Definition 2.4.1 (Minkowski-sum [Sch19]). *The Minkowski-sum of two sets* A, $B \subseteq \mathbb{R}^d$ *is defined as*

$$A \oplus B = \{a + b \mid a \in A \wedge b \in B\}$$

This computes the first flowpipe segment Ω_0 for the time interval $[0,\delta]$, allowing us via linear transformation to compute further flowpipe segments Ω_i for all time intervals $[i\delta,(i+1)\delta]$ until we reach the time horizon T.

$$\Omega_i = \Phi \cdot \Omega_{i-1}$$

With this method, starting from a defined set of initial states (l,N_0) we can compute the flowpipe segments Ω_i, $i = 0,\ldots,N-1$ for a given dynamics $\mathcal{A}$ and time horizon T using a fixed time step $\delta = \frac{T}{N}$. All flowpipe segments must satisfy the invariant of location l, therefore we check for each computed segment Ω_i whether the invariant is violated, and we stop the computation of further flowpipe segments $\Omega_{i+1},\Omega_{i+2},\ldots$ if $N_i \cap Inv(l) = \emptyset$. Now remains the computation of discrete successors from our current location during the analysis.

For each transition $e = (l,a,g,r,l') \in Edge$ and for each flowpipe segment $\Omega_i = (l,N_i)$, we check whether the flowpipe segment *enables* the transition, i.e., whether the jump can be taken from any state in the segment, or in mathematical terms, whether

$$N'_i = N_i \cap g \neq \emptyset$$

if the intersection N'_i is not empty then the transition e is enabled and the reset function r can update the variable valuations of every non-empty segment, which intersects the guard g, we obtain

$$\Omega''_i = (l,N''_i) = (l,r(N'_i))$$

finally we check whether any valuation from N''_i satisfy the invariant of the target location l', i.e., whether

$$N'''_i = N''_i \cap Inv(l') = \emptyset$$

and then we continue with computing the flowpipe in l' for all non-empty segments $\Omega'''_i = (l',N''')$.

In case of **Rectangular Automata** the previous approach introduces unnecessary effort, since rectangular automata have *constant derivatives*. The flowpipe construction approach uses segmentation, with the aim of reducing the over-approximation error between the linear over-approximation and non-linear evolution during the flowpipe construction. Because the evolution for rectangular automata is linear, the whole flowpipe is a linear set, and we can replace the flowpipe construction by a much simpler approach. In rectangular automata all assertion predicates encode rectangular sets, i.e., each predicate is a conjunction of constraints comparing variables to constants. Therefore, a state of a rectangular automaton can be described by a location and a conjunction of linear real-arithmetic constraints over the variables of the automaton. Consider rectangular automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ and let us represent the initial state set of $\mathcal{H}$ symbolically by (l,φ), where l is a location of $\mathcal{H}$ and φ is a conjunction of constraints. Now the set of reachable states from (l,φ) can be described by a union of similar symbolic states. We can compute the flow successors

of state (l,φ) by defining another state $T^+((l,\varphi))$, to which we introduce quantified variables t, x^{pre} representing time and time predecessor states respectively

$$\exists t.\ \exists x^{pre}.\ t \geq 0 \ \wedge\ \varphi[x^{pre}/x] \ \wedge\ Flow(l)[x^{pre},x/x,\dot{x}] \ \wedge\ Inv(l)$$

then we eliminate the quantified variables t and x^{pre} using variable elimination techniques, e.g., Gaussian elimination and Fourier-Motzkin variable elimination to get a description of the time successor states. For calculating jump successors we can take a similar approach and define a quantified variable to describe the state before the jump, with the help of which we can define a new state $J^+((l,\varphi))$ for each jump $e = (l,a,g,r,jump,l') \in Edge$, where g is the guard and r is the reset function of the jump. We represent the guard g and the reset r as a conjunction of constraints.

$$\exists x^{pre}.\ \varphi[x^{pre}/x] \ \wedge\ g[x^{pre}/x] \ \wedge\ r[x/x'] \ \wedge\ Inv(l') \ \wedge\ Inv(l)[x^{pre}/x]$$

and again we eliminate the quantified variable x^{pre} to get a description of the jump successor states. Note that there is no need for time variable when computing the jump successors, since the jump takes no time to commit.

Example 2.4.1 (Rectangular Reachability). *Consider a rectangular automaton with one location l_0 and one transition e as depicted in Figure 2.2. We write all rectangular sets as a conjunction of constraints over the variables of the automaton. The flow in location l_0 is defined by $1 \leq \dot{x} \leq 2$ and the invariant is $x \leq 5$, the guard of the transition e is $x \geq 1$ and the reset function is $x' := 0$. Now if we start in initial state $(l_0, x = 0)$ we can describe the time successors of the state as follows*

$$\exists t.\ \exists x^{pre}.\ t \geq 0 \ \wedge\ \underbrace{x^{pre} = 0}_{\varphi[x^{pre}/x]} \ \wedge\ \underbrace{x^{pre} + t \leq x \leq x^{pre} + 2t}_{Flow(l)[x^{pre},x/x,\dot{x}]} \ \wedge\ \underbrace{x \leq 5}_{Inv(l_0)}$$

using Gaussian variable elimination to eliminate x^{pre} and Fourier-Motzkin to eliminate t we get

$$\begin{aligned} &\exists t.\ \exists x^{pre}.\ t \geq 0 \ \wedge\ x^{pre} = 0 \ \wedge\ x^{pre} + t \leq x \leq x^{pre} + 2t \ \wedge\ x \leq 5 \\ \Leftrightarrow\ &\exists t.\ t \geq 0 \ \wedge\ t \leq x \leq 4t \ \wedge\ x \leq 5 \\ \Leftrightarrow\ &0 \leq x \leq 5 \end{aligned}$$

that means that after time t the system would be in a state $(l_0, 0 \leq x \leq 5)$. From this state we can compute the jump successors, which we describe as follows

$$\exists x^{pre}.\ \underbrace{0 \leq x^{pre} \leq 5}_{\varphi[x^{pre}/x]} \ \wedge\ \underbrace{1 \leq x^{pre}}_{g[x^{pre}/x]} \ \wedge\ \underbrace{x = 0}_{r[x/x']} \ \wedge\ \underbrace{x \leq 5}_{Inv(l')} \ \wedge\ \underbrace{x^{pre} \leq 5}_{Inv(l)[x^{pre}/x]}$$

and if we eliminate x^{pre} using Fourier-Motzkin variable elimination we get

$$\begin{aligned} &\exists x^{pre}.\ 0 \leq x^{pre} \leq 5 \ \wedge\ 1 \leq x^{pre} \ \wedge\ x = 0 \ \wedge\ x \leq 5 \ \wedge\ x^{pre} \leq 5 \\ \Leftrightarrow\ &x = 0 \end{aligned}$$

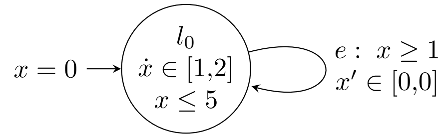


Figure 2.2: Rectangular automaton with one location and one self loop.

2.5 Variable Set Separation

A challenging aspect of reachability analysis in general, and flowpipe-construction-based approaches in particular, remains the ability to analyse large models with complex behaviours. The *variable set separation* [SNÁ17] method has approached this problem for PLC-controller plants by studying the characteristics of the variables of such models. Specifically, the method leverages the dependencies within groups of variables in the hybrid system to divide the variable set into multiple syntactically independent subsets. This allows for the analysis to be performed in the subspaces defined by the variable subsets, instead of the global space, reducing the computational effort needed to compute the reachable set.

Syntactical independence of the variable subsets means that the evolution of the variables of a subset does not directly influence the variables in other subsets. Formally, it means that all predicates $\varphi \in \mathit{Pred}_{\mathit{Var}}$ (as well as jump resets $\varphi \in \mathit{Pred}_{\mathit{Var} \cup \mathit{Var}'}$ and flows $\varphi \in \mathit{Pred}_{\mathit{Var} \cup \dot{\mathit{Var}}}$) in the hybrid automaton definition must be decomposable to a conjunction $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ of predicates $\varphi_i \in \mathit{Pred}_{\mathit{Var}_i}$ over the respective variable subsets. Each global state set $(l,v) \subseteq \mathit{Loc} \times \mathbb{R}^d$ can be represented by its projection $v{\downarrow}_{\mathit{Var}_i} = v_i \in \mathbb{R}^{|\mathit{Var}_i|}$ to the subspace. This representation, called $(l,v_1,\ldots,v_n)$ the *projective representation* of (l,v), drops the connection between subspaces, therefore over approximates the global states.

Although we are able to split global states into syntactically independent projections, the semantics of the variables in different subspaces might still be connected in some sense. One obvious connection between the variables in different subspaces is the passage of time. In rectangular automata, time is the only dependency between the subspaces, because the variables of a rectangular automaton are already pairwise independent. In Chapter 3, we will see how we can bridge this gap for rectangular automata and eliminate the over-approximation introduced by computing in multiple subspaces.

Chapter 3

Synchronized Reachability Analysis of Rectangular Automata

In variable set separation the reachable set of a hybrid system is computed after dividing the system into multiple independent *subspaces*. Each subspace representing an independent hybrid system of its own. But since the method drops the dependencies between the subspaces, it over-approximates the reachable set. In this chapter we present an algorithm to compute the reachable set of multiple hybrid systems synchronously, yet avoiding the state space explosion problem, that arises when constructing the parallel composition. The algorithm leverages global time and synchronization labels in order to synchronize jump successor computation. Every subspace is represented in our analysis by a hybrid automaton. Every automaton can evolve locally and compute the reachable sets that do not require synchronization. However, if a synchronization label is shared between two or more components, the algorithm takes into account all the automata that share the synchronization label when computing their jump successors, ensuring that synchronizing transition in different subspaces has been taken within the same global time interval. Therefore, the computation of the reachable set using synchronized reachability analysis simulates the reachability analysis of the parallel composition of the input automata, but without the effort of actually constructing the product automaton.

This approach is implemented as an extension to the rectangular reachability analysis using HyPro [Sch19]. We restricted our implementation to rectangular automata and polyhedral representation of state sets, so that we can perform exact computations of the reachable sets. In Section 3.1 we introduce the concepts and data structures that are needed to perform the synchronized reachability analysis, and in Section 3.2 we present and explain the algorithm.

3.1 Global Time and Synchronization Labels

In order to perform reachability analysis that synchronizes the evolution of multiple hybrid automata, we have introduced new data structures and extensions to exist-

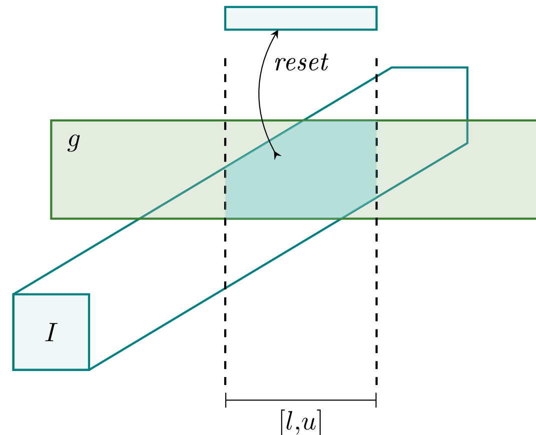


Figure 3.1: Global time: The global time interval $[l,u]$ is intersected with the state set before applying the reset function when taking a synchronizing jump.

ing data structures to track the dependencies between the automata. These data structures are crucial for ensuring proper synchronization during the analysis. The following concepts are defined for a set of hybrid automata $H = \{\mathcal{H}_1, \ldots, \mathcal{H}_n\}$.

Global Time. To compute the jump successors of multiple jumps synchronously we have to consider the global time that have passed throughout the evolution of the system, and check whether a time interval exists, within which all synchronizing jumps that are involved in a synchronization step are enabled. To achieve this each automaton is extended with a time variable t_i that is initialized with 0 and has a constant flow $\dot{t}_i = 1$ in every location of the automaton $\mathcal{H}_i$. The computation of a synchronizing jump successor required that all input automata take the jump at the same time. Consequently, all local time intervals $t_i \in [l_i,u_i]$ that enable the synchronizing jump are intersected to create a global time interval $[l,u]$, where synchronization is possible.

This global time interval is then intersected with each state set that enables the jump, resulting a "reduced" state set, i.e., the part of the state set bounded by the global time interval, on which the reset function is applied. This way we ensure that only jumps that are enabled in all the automata are taken during the analysis. Figure 3.1 shows the intersection of the interval $[l,u]$ with the flowpipe of state set I and the guard g before applying the reset function.

Synchronization Dictionary. The synchronization of the automata is done according to the synchronization labels, therefore it is not necessarily the case that all the automata need to synchronize on each jump. The synchronization dictionary is a data structure that saves for each label the set of automata that need to synchronize on that label. Given a set of hybrid automata $H = \{\mathcal{H}_1, \ldots, \mathcal{H}_n\}$ with $\mathcal{H}_i = (Loc_i, Var_i, Lab_i, Flow_i, Inv_i, Edge_i, Init_i)$, we can formally define the synchronization dictionary by the following function

$$syncDict : Lab \to 2^{\mathcal{H}}$$

where $Lab = Lab_1 \cup \cdots \cup Lab_n$ and for each label $a \in Lab$ we have the function value $syncDict(a) = \{\mathcal{H}_i \mid a \in Lab_i\,,\ 1 \leq i \leq n\}$. This way we already know at the beginning of the analysis for each label which automata are required to synchronize when taking a jump.

Search Tree Extension. During the analysis, discrete jumps can be taken non-deterministically because one location of the hybrid automaton can have multiple jumps enabled at the same time. Therefore, a search tree is generated by the analysis for each of the input automata. Nodes of the search tree represent the passage of time and the parent-child relation between nodes in the tree represent discrete jumps.

Definition 3.1.1 (Search Tree [Sch19]). *For a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ with a dimension $d = |Var|$, a state set Σ, a time horizon $T \in \mathbb{R}_{\geq 0}$, a search tree is a tuple*

$$S = (Nodes, Root, Succ, Trace, State, Completed)$$

with the following components:

- *a finite set Nodes of nodes and a root node $Root \in Nodes$;*
- *a set $Succ \subseteq Nodes \times Nodes$ of edges such that $(Nodes\,, Root\,, Succ)$ is a tree;*
- *a function $State : Nodes \to (Loc, 2^{R^d})$ that assigns to each node a symbolic state of $\mathcal{H}$ as data;*
- *a function $Trace : Succ \to (\mathbb{I} \times Edge)$ assigning to each edge of the search tree an interval and a jump of $\mathcal{H}$;*
- *a function $Completed : Nodes \to \{0, 1\}$, we say that a node s is completed if $Completed(s) = 1$;*
- *for each node $s \in Nodes$, either $Completed(s) = 0$ and s has no successors (i.e., $\forall (s', s'') \in Succ, s' \neq s$), or $Completed(s) = 1$ and for each $(s, s') \in Succ$ with $Trace((s, s')) = (I, e)$ we have that*

$$FP(State(s)) = \{State(s') \mid (s, s') \in Succ\}$$

a search tree is called complete for a jump depth $J \in \mathbb{N}_{\geq 0}$ if each node $s \in Nodes$ with depth less than J is completed.

We use $depth(s)$ for $s \in Nodes$ to denote the depth of node s in the tree, i.e., the number of edges from $Root$ to s.

In the context of synchronized reachability analysis, we have to deal with n search trees, and we need to keep track of some information that is helpful for the synchronization, e.g., if a tree node is the result of taking a synchronizing jump in automaton $\mathcal{H}_i$, we need to know which nodes from the trees of the other automata has taken the same synchronizing jump. For this purpose, we extend the search trees with a mapping $Sync$ that keeps track of exactly this information. It assigns to each node an n-tuple of nodes $(p_1, \ldots, p_n)$ that contains the last synchronization partners from automata $\mathcal{H}_1, \ldots, \mathcal{H}_n$ respectively. We define a search forest as n different search trees with a mapping that encodes the synchronization information between the automata.

Definition 3.1.2 (Search Forest). *For a set of hybrid automata $\mathcal{H} = \{\mathcal{H}_1, \ldots, \mathcal{H}_n\}$ with $\mathcal{H}_i = (Loc_i, Var_i, Lab_i, Flow_i, Inv_i, Edge_i, Init_i)$, a search forest is a set of n search trees $S = \{S_1, \ldots, S_n\}$, where*

$$S_i = (Nodes_i, Root_i, Succ_i, Trace_i, State_i, Completed_i),$$

and a mapping

$$Sync : Nodes \to (Nodes_1, \ldots, Nodes_n),$$

where $Nodes = \bigcup_{i=1}^{n} Nodes_i$ and for each node $s \in Nodes_i$ we have that $Sync(s) = (p_1, \ldots, p_n)$ with $p_j \in Nodes_j$ for all $1 \leq j \leq n$ and $p_i = s$.

The storage of this information in the search forest reduces the effort needed to find new jump successors whenever a jump with a synchronization label is taken. It also avoids redundant computations since we do not need to search the entire search tree of an automaton to find nodes that can synchronously take a jump. Instead, we only need to search the subtree whose root is the stored node. Note that this definition of a search forest is only applicable in the context of label synchronization between multiple automata, where the value of n is known before initializing the search forest. After each computation of jump successors, a new element is added to the mapping *Sync*, which allows us to access these newly created synchronization nodes in the other automata. We use the notation $Sync_i(s)$ to refer to the element of $Sync(s)$ at index i.

In the following we will explain how we maintain the correct information in *Sync* when computing new jump successors and adding them to the search forest. Initially, The mapping *Sync* contains only the root nodes of the search trees, i.e., $Sync(r_i) = (r_1, \ldots, r_n)$ for all $1 \leq i \leq n$, where r_i is the root node of search tree S_i.

For every jump successor computation $(l,v) \xrightarrow{e} (l',v')$ a node s' is created in the search forest for state set (l',v'), which requires the addition of a new mapping for s' in *Sync*. Let us consider nodes s and s' in search tree S_i, which represent states (l,v) and (l',v') respectively with (l',v') being the jump successor of (l,v) according to jump e. Further, let $Sync(s) = (p_1, \ldots, p_n)$.

If jump e is *local*, the successor node s' can inherit the values of its mapping from its parent node s with the single modification of $p_i = s'$

$$Sync(s') = (p_1, \ldots, \underbrace{s'}_{pos\ i}, \ldots, p_n)$$

If e is a synchronizing jump with label a, we need to consider the mappings of all the nodes that are involved in the synchronization. We define the set

$$syncInd(a) = \{k \in \{1, \ldots, n\} \mid \mathcal{H}_k \in syncDict(a)\}$$

as the set of indices of all the automata (or search trees) that are involved in the synchronization of transitions with label a. Then we define the mapping $Sync(s') = (p'_1, \ldots, p'_n)$ as follows

- $\forall k \in syncInd(a)$ we define p'_k as the successor of node p_k according to jump e,
- $\forall j \notin syncInd(a)$ we define $p'_j = deep(\{Sync_j(p_k) \mid k \in syncInd(a)\})$,

where $deep(\{p_1, \dots, p_n\}) = p_i$ such that $depth(p_i) \geq depth(p_j)\ \forall 1 \leq j \leq n$.

Since jump e has synchronized all automata in $SyncDict(a)$, a new node has been added to each of the search trees of these automata, namely the e-jump successor of p_k for $k \in syncInd(a)$. This is exactly the information that we save in the mapping $Sync(s')$ such that all future synchronizations from s' consider *re-synchronizing* the system from the last synchronized state. Moreover, The function *deep* returns the node with the maximum depth from a set of nodes in a search tree, and by assigning $deep(\{Sync_j(p_k) \mid k \in syncInd(a)\})$ to p'_j we make sure that the global state of the system after synchronization is consistent with all previous synchronizations.

3.2 Label Synchronization

The goal of reachability analysis is to determine whether a given state in a system can be reached from a set of initial states. In view of label synchronization of rectangular automata, reachability analysis becomes more challenging due to the need to synchronize the evolution of multiple automata. Traditional approaches involve constructing the product automaton, which can be computationally expensive and memory-intensive, especially for large-scale systems. To address this challenge, we propose an algorithm that avoids the construction of the product automaton while still accurately computing the reachable set. Our algorithm considers the synchronization labels of the jumps and the global time evolution to selectively compute only the relevant jump successors. By saving information about the dependencies between the input automata, we can significantly reduce the search space of synchronization nodes and improve the efficiency of the reachability analysis. In the following sections, we will provide a detailed explanation of our algorithm and its implementation. We will discuss how the presented data structures and extensions ensure proper synchronization in our approach.

3.2.1 Forward Reachability Analysis of n Automata

We present an extended general flowpipe-construction-based forward reachability algorithm that accommodates the parallel processing of n input automata. The algorithm, presented in Algorithm 1, utilizes a first-in-first-out working queue Q to manage the nodes that require processing. The queue is initialized with the initial states of the input automata. It is important to note that the first-in-first-out property of the working queue is essential for ensuring a breadth-first exploration of the search forest. This parallel exploration allows for the simultaneous evolution of individual search trees, thereby minimizing redundant computations in the search for synchronization partners. Implementing a first-in-last-out queue would not only mean that the individual search trees are explored depth-first, but also that the first search tree would have to be fully explored before the next search tree is considered. This would lead to a lot of redundant computations. Each element in the working queue is a pair $(node, i) \in Nodes_i \times \{1, \dots, n\}$, where *node* is a node in the search tree of the i-th automaton. In line 6, a synchronization dictionary is initialized, mapping each label $a \in Lab$ to a set of hybrid automata $\{\mathcal{H}_1, \dots, \mathcal{H}_m\}$ with $m \leq n$, such that $a \in Lab_{\mathcal{H}_i}$ for each $1 \leq i \leq m$. In Section 3.2.3 we explain how the synchronization dictionary is used to search for the correct synchronization partners. Then in lines 7-13, the algorithm processes the nodes in the working queue until either the queue is empty

or an unsafe state is reached. The processing of a node includes the computation of the time successors and the jump successors, as well as the addition of the jump successors to the working queue.

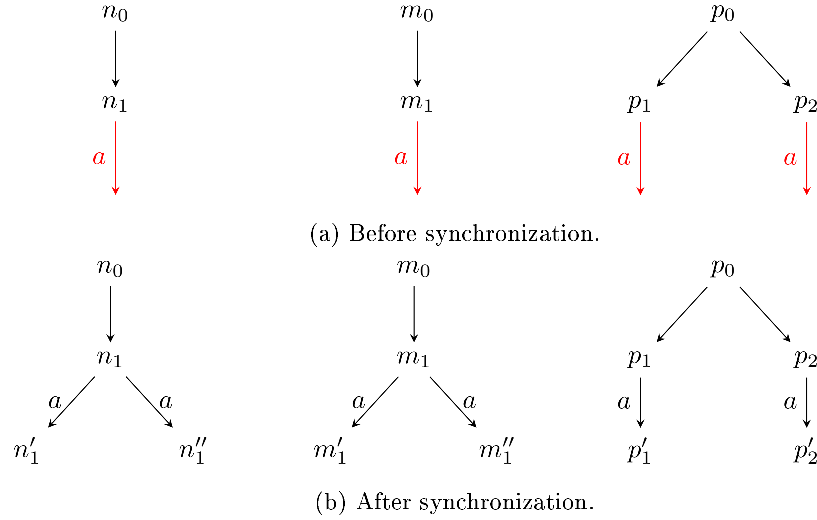
Algorithm 1 Synchronized forward reachability analysis

Input: set of rectangular automata $\mathcal{H}_1, \ldots, \mathcal{H}_n$
Output: safe/unsafe
1: $Q = \emptyset$
2: **for all** $\mathcal{H}_i$ **do**
3: $\quad R_i = \{(l, Init_i(l)) \mid l \in Loc_i\}$
4: $\quad Q = Q \cup \{(node, i) \mid node \in R_i\}$
5: **end for**
6: $syncDict = initializeSyncDictionary(\mathcal{H}_1, \ldots, \mathcal{H}_n)$
7: **while** $Q \neq \emptyset$ **do**
8: $\quad (node, i) = getElement(Q)$
9: $\quad result = processNode(node, \mathcal{H}_i, Q)$
10: $\quad$ **if** $result \neq safe$ **then**
11: $\quad\quad$ **return** *unsafe*
12: $\quad$ **end if**
13: **end while**
14: **return** *safe*

3.2.2 Successor Computation

The reachability analysis presented in Algorithm 1 requires the iterative computation of one-step time successors and jump successors of states of the hybrid automaton. For rectangular automata the exact computation of a bounded time successor can be efficiently achieved via symbolic computation based on finite linear constraints [CÁF11]. The computation of jump successors is also straightforward in the case of one rectangular automaton, but since we have to consider n input automata with label synchronization, the computation becomes a little more challenging. We can categorize jumps into two types: local jumps and synchronizing jumps. Local jumps are jumps with labels that are unique to a single automaton, and they can be computed locally. On the other hand, synchronizing jumps can only be taken in combination with jumps from other input automata that have the same synchronization label and within the same global time interval. In Algorithm 2, we describe how to process a node from the working queue Q. The parameter $\mathcal{H}_i$ is the input automaton to which the node belongs.

The processing of a node from the queue is split into two parts. First the time successor computation (see Section 2.4), and second that of all jump successors, which is also twofold. For local jumps we compute the jump successor (line 8) by applying the reset function on the state set and intersecting the result with the invariant of the target location, without any interaction with the other automata. Then we create a new node in search tree S_i for the jump successor, and we add the new node to the working queue. We can identify the local jumps by taking advantage of the synchronization dictionary that we have initialized in Algorithm 1. Note that the synchronization dictionary only considers labelled jumps, unlabelled jumps are con-



(a) Before synchronization.

(b) After synchronization.

Figure 3.2: Search tree synchronization: n_1 and m_1 can synchronize with both p_1 and p_2.

sidered local jumps by giving them a label unique to their automaton. Assuming that we are processing node s from search tree S_i, for each jump $e = (l,a,g,r,\mathit{jump},l')$ it holds

$$\mathit{syncDict}(a) = \{\mathcal{H}_i\} \;\Leftrightarrow\; \mathit{jump}\ e\ \mathit{is\ local}$$

For a synchronizing jump, i.e., $e = (l,a,g,r,\mathit{jump},l')$ with $|\mathit{syncDict}(a)| > 1$, we have to check the other automata for possible synchronization. This requires search functions that traverse the search trees of the other automata to find all possible synchronization nodes. We will look at the search method in detail in Section 3.2.3. After the successor computation of every synchronizing jump we have to update our search forest (line 18), this includes adding the successor nodes to all the search trees that are involved in the synchronization, as well as updating the mapping *Sync* with the new nodes mapped to their synchronization partners. It is possible that one synchronizing jump has multiple synchronization possibilities, i.e., more than one node in (at least) one of the search trees involved in the synchronization fulfil all the requirements to take the jump synchronously (possibly in different time intervals). In this case all synchronization possibilities need to be explored individually. We create a successor node in the search tree for each synchronization possibility that we find, and the successor state is computed with respect to the common time interval of the synchronizing states. Figure 3.2 shows an example of three search trees before and after a synchronizing jump with label a. Red edges represent enabled jumps before computing their successors, and we omit the labels of local jumps. In Figure 3.2a we have nodes n_1 and m_1, who are compatible for synchronization, and they can synchronize with two nodes p_1 and p_2 from the third search tree. Therefore, we need to create two successors of n_1 and the same for m_1 and update the mappings of the nodes $\mathit{Sync}(n'_1)$ and $\mathit{Sync}(n''_1)$ accordingly. Specifically, after the synchronization, in Figure 3.2b, we will have $\mathit{Sync}(n'_1) = (n'_1,m'_1,p'_1)$ and $\mathit{Sync}(n''_1) = (n''_1,m''_1,p'_2)$.

Algorithm 2 $processNode(node, \mathcal{H}_i, Q)$

```
1: result = computeTimeSuccessor(node)
2: if result ≠ safe then
3:     return unsafe
4: end if
5: possibleJumps = getEnabledTransitions(node)
6: for all e = (l,a,g,r, jump ,l') ∈ possibleJumps do
7:     if syncDict(a) = {H_i} then
8:         successor = computeJumpSuccessor(node, e)
9:         if successor ≠ ∅ then
10:            node.addChild(successor)
11:            Q.enqueue(successor, i)
12:        end if
13:    else
14:        time = node.getTimeProjection()
15:        visitedHA = {H_i}
16:        succTimePairs = findSyncSuccessors(i, node, e, a, time, visitedHA)
17:        for all (successor, time) ∈ succTimePairs do
18:            updateTreeWithSyncNodes(successor, i)
19:            for all 1 ≤ j ≤ n do
20:                Q.enqueue((Sync_j(successor), j))
21:            end for
22:        end for
23:    end if
24: end for
25: return safe
```

3.2.3 Search for Synchronization Nodes

Now we will explain how the method *findSyncSuccessors* works. As mentioned before we have implemented a recursive search method to find all the nodes that can synchronize with the node that is being processed. To recognize where to stop our recursion, we save the set of hybrid automata that we have visited and searched for synchronizing nodes, and once we have visited all the hybrid automata in $syncDict(a)$ we can break out of the recursion and return the results. The search method is presented in Algorithm 3. The first two parameters i and $node$ are the index of the search tree (or hybrid automaton or subspace) in which $node$ exists, and the node for which we need to find synchronization partners, $e = (l,a,g,r, jump ,l')$ is the jump that needs to be applied on $node$ in case we have found a synchronization possibility, $syncTime$ is the time interval, in which e is enabled and $visitedHA$ is the set of hybrid automata, in which we have already found synchronization partners.

The function *findSyncSuccessors* explores the search tree of each hybrid automaton that is involved in the synchronization. Here we take advantage of the function $Sync$ and the fact that for two nodes n_1 and m_1, which are the successors of a synchronizing jump, n_1 (or any of its children) can only synchronize with m_1 (or any of its children) and vice versa. When processing node $n \in S_i$ instead of searching the entire search tree $S_j, i \neq j$ for a synchronization partner for n we only need to search the

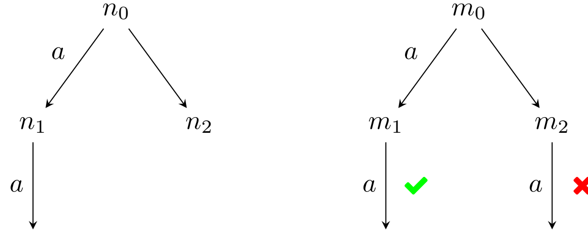


Figure 3.3: Search tree synchronization: For search trees S_1 (left) and S_2 (right), n_1 can synchronize with m_1 but not with m_2.

subtree of $Sync_j(n)$, This way we can reduce the search space and avoid redundant computations. This idea is demonstrated in Figure 3.3, where we have two search trees S_1 and S_2 of automata $\mathcal{H}_1$ and $\mathcal{H}_2$ respectively. $\mathcal{H}_1$ and $\mathcal{H}_2$ can synchronize on label a and assuming the nodes n_1, m_1 and m_2 have a common time interval in which all jumps with label a are enabled, n_1 can in this case synchronize with m_1 but not with m_2 because they do not share the same synchronization history. Again, we omit the labels of local jumps in Figure 3.3.

There are multiple conditions that need to be satisfied for the nodes to be able to take a jump synchronously. First we have to make sure that the synchronizing jumps in the different subspaces can be taken at the same global time, i.e., the intersection of the time intervals that enable the synchronizing jumps cannot be empty. We check this condition by computing the projection of the state sets onto the time dimension t_i defined in Section 3.1, and intersecting these projections (line 16) to get a common time interval, in which all synchronizing jumps are enabled. Second, we need to check the compatibility of the mapping *Sync*. This compatibility check consists of two conditions that we check for the synchronizing nodes pairwise, the first of which is that the nodes have the same synchronization history. For two nodes $n \in S_i$ and $m \in S_j$ having the same synchronization history means that the trace of n and the trace of m in their respective search trees are equal after ignoring all the steps that are not a synchronizing jump with label in $Lab_i \cap Lab_j$. This condition is ensured by the way we update our *Sync* function upon adding a new node to the search forest and the fulfilment of $c1$, in line 17 of Algorithm 3. Notice that we only check if $Sync_i(canNode)$ is a predecessor of *node* because $Sync_j(node)$ being a predecessor of *canNode* is satisfied by our choice of the candidate node since we only consider candidate nodes that are in the subtree of $Sync_j(node)$. This is ensured by the method $getCandidateNodes(j, node, label)$ in line 12, which returns the set of nodes in the subtree of $node \in S_j$ that have a jump with label *label*.

The second condition has to do with the hybrid automata that are not currently part of the synchronization. For two nodes $n \in S_i$ and $m \in S_j$ and for all $k \in \{1, \ldots, n\} \setminus \{i, j\}$, $Sync_k(n)$ and $Sync_k(m)$ cannot be siblings in different branches of the search tree S_k. This condition is formalized in condition $c2$ (line 18) and it makes sure that the two synchronizing nodes n and m have not previously synchronized with different nodes from automaton $\mathcal{H}_k$ in different branches of the search tree S_k.

Once we reach the last function call in the recursion we enter the if-branch in line 1 and

compute the jump successor of *node* using $computeJumpSuccessors(node, e, syncTime)$ (line 2), which first reduces the state of node to the state bounded by $syncTime$ before intersecting with the guard and applying the reset. Since we represent the states of the automata as polytopes, we can compute the reduced state set by intersecting the state set with the hyperplanes defined by the lower and upper bounds of the time interval.

Algorithm 3 $findSyncSuccessors(i, node, e = (l,a,g,r, jump\ ,l'), syncTime, visitedHA)$

1: **if** $visitedHA = syncDict(a)$ **then**
2: $jumpSuccessors = computeJumpSuccessors(node, e, syncTime)$
3: $resultSet = \emptyset$
4: **for all** $newNode \in jumpSuccessors$ **do**
5: $node.addChild(newNode)$
6: $resultSet.insert((newNode, syncTime))$
7: **end for**
8: **return** *resultSet*
9: **else**
10: $\mathcal{H}_j = getElementFrom(syncDict(a) \setminus visitedHA)$
11: $visitedHA = visitedHA \cup \{\mathcal{H}_j\}$
12: $candidateNodes = getCandidateNodes(j, Sync_j(node), a)$
13: $resultSet = \emptyset$
14: **for all** $canNode \in candidateNodes$ **do**
15: $computeTimeSuccessor(canNode)$
16: $timeIntersection = syncTime \cap canNode.getTimeProjection()$
17: $c1 = Sync_i(canNode).isPredecessorOrEquals(node)$
18: $c2 = \bigwedge_{k \in \{1,\dots,n\} \setminus \{i,j\}} Sync_k(canNode).hasAncestorRelation(Sync_k(node))$
19: **if** $timeIntersection \neq \emptyset \wedge c1 \wedge c2$ **then**
20: $nodeTimePairs = findSyncSuccessors(j, canNode,$
21: $e, timeIntersection, visitedHA)$
22: **for all** $(syncNode, time) \in nodeTimePairs$ **do**
23: $jumpSuccessors = computeJumpSuccessors(node, e, time)$
24: **for all** $newNode \in jumpSuccessors$ **do**
25: $Sync_j(newNode) = syncNode$
26: $node.addChild((newNode))$
27: $resultSet.insert((newNode, time))$
28: **end for**
29: **end for**
30: **end if**
31: **end for**
32: **return** *resultSet*
33: **end if**

Chapter 4

Experimental Results

In this chapter, we present the experimental results of our implementation. We begin by introducing the implementation details and the tool we have extended, HyPro [SÁMK17], which provides state set representations and performs reachability analysis on hybrid automata. Our extension adds support for concurrent analysis of multiple rectangular automata, which involves a modularized approach with multiple worker processes. We also define the concept of a *task* in the context of synchronized reachability analysis, and highlight the differences in our approach compared to previous work, such as the inclusion of synchronization possibilities and the use of a garbage collector to remove unreachable nodes. Finally, we present some benchmark evaluations and tests that summarize the performance of our method.

4.1 Implementation

We have implemented our method as an extension to the tool HyPro [SÁMK17]. HyPro provides a range of state set representations and performs reachability analysis on various types of hybrid automata. Our extension adds support for performing concurrent analysis of multiple hybrid automata and checking for synchronization whenever a jump successor is computed. The tool is implemented in a modularized manner. The reachability analysis is broken down into *tasks*, which are processed by a *worker*. A task is a node of the search tree, which contains all the information needed for the successor computation. We have extended the definition of a task from [Sch19] to include the index of the automaton (i.e., index of the worker) that the task belongs to. This allows the main process to know which worker instance is responsible for processing this task.

Definition 4.1.1 (Task). *Assume a search forest generated by synchronized reachability analysis of a given set of hybrid automata* $\mathcal{H} = \{\mathcal{H}_1, \ldots, \mathcal{H}_n\}$*. A task*

$$t = (s,i)$$

is specified by a node s *in the search tree* S_i *for* $i \in \{1, \ldots, n\}$ *(see Definition 3.1.2).*

The worker process is responsible for computing the time and jump successors of a task t. Our method implements n workers, one for each input automaton, which are managed by one main process. Workers are able to communicate with each other

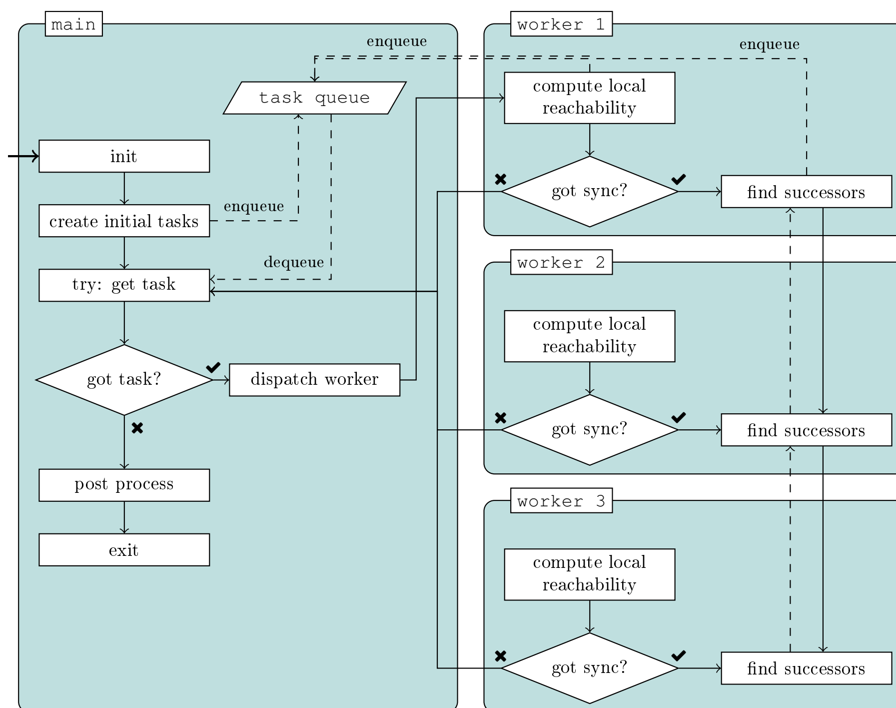


Figure 4.1: Modularized synchronized reachability analysis process.

when synchronization is needed. Any worker which is processing a synchronizing jump initiates a recursive function call that looks for synchronization possibilities in each of the other workers and returns those possibilities after having computed a synchronized jump successor (in the corresponding automaton) and added it to the search tree and the working queue. As an illustration the structure of the analysis process for three hybrid automata is shown in Figure 4.1.

Just like in [Sch19], we use a task queue to manage the tasks and the main process initializes the task queue and dispatches the workers. The difference however, is that it is also responsible for getting the tasks from the queue and choosing which worker to start. The worker now computes the local reachability of the task, this steps includes computing the time successors and the jump successors of local (i.e., non-synchronizing) jumps. If the task has a synchronizing jump, the worker starts a recursive function through all the workers, that are involved in this synchronization. Upon returning the synchronization possibilities, and adding them to the search tree and the task queue, the worker gives the control back to the main process to check again for tasks. In Figure 4.1 we omitted the arrows from the methods *find successors* and *compute local reachability* of workers 2 and 3 to `task queue` for readability. Also, the main process can start any worker and not necessarily `worker 1` as shown in Figure 4.1, this was also omitted for readability.

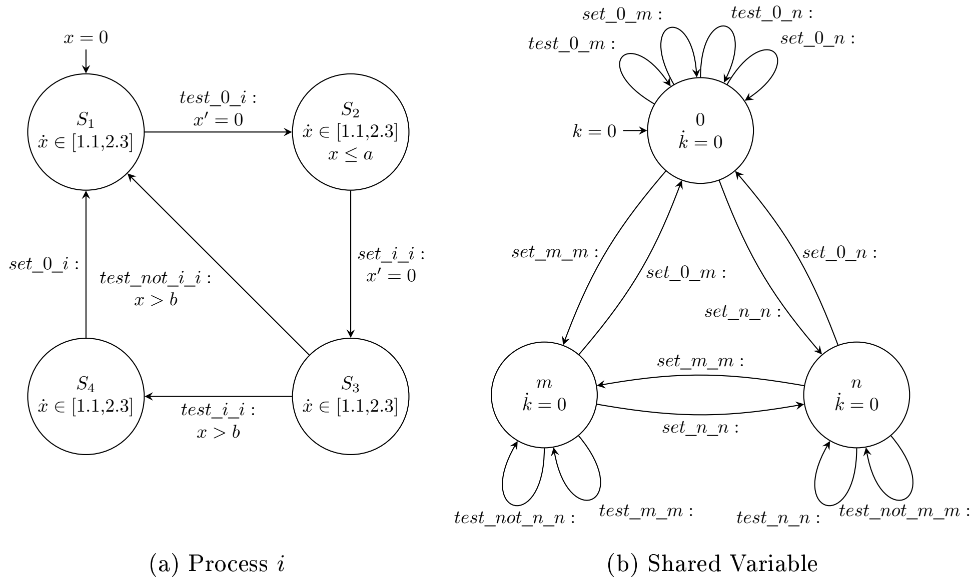
During the recursive function call, the global time interval, within which a synchronization is possible can only be determined at the time of popping out of the recursion. Therefore, a *find successors* method can only determine whether a jump is enabled

and has non-empty successors after all underlying recursions has exited. If at that point the method determines that synchronization is not possible—because the jump has empty successors in one of the workers, jump successors in the underlying workers would have already been computed and saved in the search tree. We deploy a garbage collector at the end of every recursion to remove the nodes that have been added to the search trees by the *find successors* method but are unreachable because synchronization was not possible. Furthermore, our implementation can only handle models with a single initial state, because it assigns one search tree (with a single root node) to each model. This limitation can be overcome be extending models with multiple initial states, by adding a new "dummy" initial state that transitions to the original initial state without any time evolution.

4.2 Benchmarks

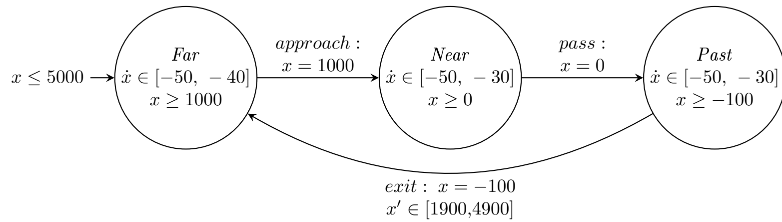
We use three well-known benchmarks to evaluate our proposed method. The Train-Gate-Controller system (TGC) [Hen00], Fischer's protocol (Fischer) [Lam87] and Nuclear Reactor System (NRS) [Wan05]. We ran our tests on a Lenovo Intel Core i7 machine with 4 cores and 16 GB of RAM running Windows Subsystem for Linux.

Fischer's Protocol is a mutual exclusion algorithm, where n processes compete to enter a critical section. The benchmark has two static variables a and b, that describe the period of time a process can maximally wait between initiating a request and entering the critical section, and the minimum period of time a process can stay in the critical section respectively. To test our synchronization method we use the shared variable automaton from [BLW+10], which is represented in Figure 4.2b.

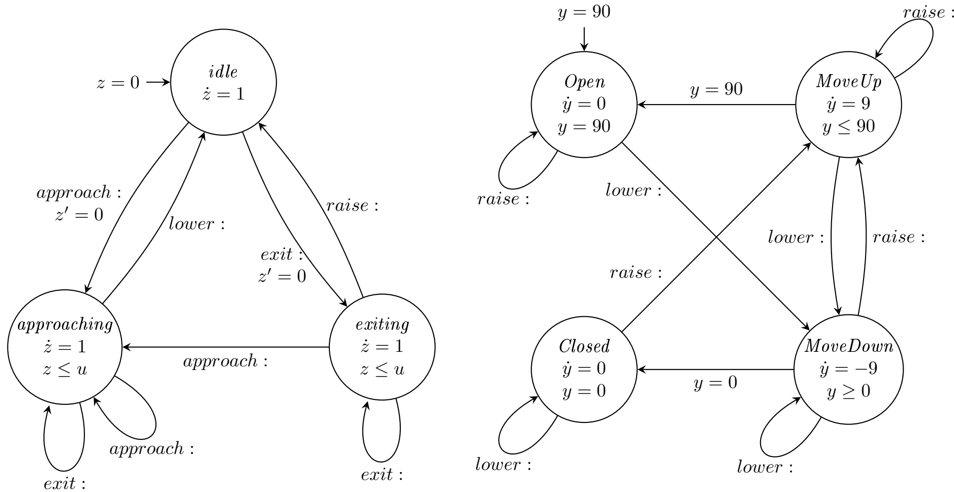


(a) Process i (b) Shared Variable

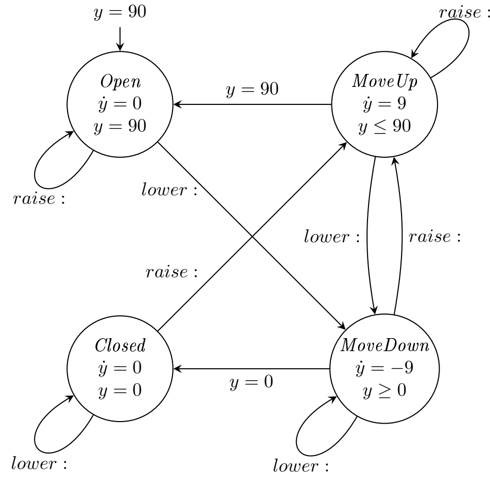
Figure 4.2: Fischer's protocol.



(a) Train automaton.



(b) Controller automaton.



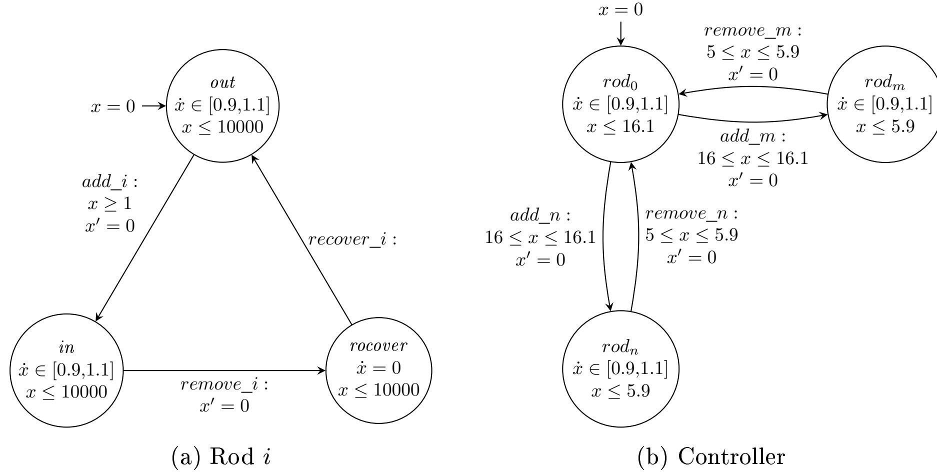
(c) Gate automaton.

Figure 4.3: Train-Gate-Controller system.

Each state of the shared variable automaton Figure 4.2b encodes which process is in the critical section (state 0 means no process is in the critical section), and the synchronizing transitions force the mutual exclusion property on the precesses.

Train-Gate-Controller is a system consisting of three hybrid automata modelling a train on a circular railway and an automated gate that closes whenever the train passes the position of the gate. The train automaton is described in Figure 4.3a, where variable x denotes the position of the train on the railway. The gate is at position 0 and the train moves at speed between 40 and 50 when it is not near the gate, upon approaching the gate it might slow down to 30. The controller, described in Figure 4.3b, is responsible for sending *lower* and *raise* signals to the gate. The controller receives an *approach* signal from the train automaton when it passes position 1000, to which the controller must react within u time units (u is a symbolic constant that represents the reaction delay of the controller) by sending a *lower* signal. And finally the gate automaton, in Figure 4.3c, reacts to the *lower* signal by closing the gate before the train passes. The gate opens again when the train moves away from the gate.

Nuclear Reactor System models a nuclear reactor with n rods. Each rod that has



(a) Rod i (b) Controller

Figure 4.4: Nuclear reactor system.

been moved out of the heavy water must stay out of the water for a recovery period of at least $T_{rec} \in [16,16.1]$ time units. Figure 4.4a shows the rod-process for the i-th rod. Each rod-process synchronizes with the controller shown in Figure 4.4b. The synchronization between the controller and the rod-processes ensures that each rod stays out of the water for at least T_{rec} time units before it is moved back in, and that no two rods are in the heavy water at the same time.

We have run our implementation on the train-gate-controller benchmark with a time horizon of $T = 25$ and a jump depth of $J = 3$. The value of the constant u in our experiments was set to 2. The reachable state sets of the benchmark are shown in Figure 4.5. We can see that the system has computed the successors of the first synchronizing jump at global time $t \in [18,25]$ which is the time interval within which the train has reached position 1000 and has issued an *approach* signal. This forces the controller to change into the state *approaching*. In Figure 4.5 we can see that the controller can stay in that state for a maximum of two time units before it must send a *lower* signal to the gate, allowing it to start closing at rate 9 degrees per time unit. The gate then needs 10 time units to move into the state *closed* at the latest by time point $t = 35$ and before the train has reached the gate. After the train passes the gate it reaches position -100 and can issue an *exit* signal at time $t_{train} \in [40,50.3]$, which must synchronize with the controller's *exit* transition from the *idle* state to *existing*. Since the *idle* state evolves only until time point $t_{controller} = 50$ (because of the time horizon $T = 25$) the system can execute the *exit* transition within global time interval $t \in [40,50]$ moving the controller into the *exiting* state and allowing it to signal to the gate that it can start opening again.

The controller automaton cannot take the jump *approach* before the train has reached position 1000, because the jump is blocked by the guard $x = 1000$ in the train automaton. The same goes for the jump *exit*, because of label synchronization the controller cannot move into the *exiting* state and allow the gate to start opening before the train has passed the gate and the jump *exit* is enabled in the train automaton.

Table 4.1: Runtimes in seconds.

Representation	Mean	Median	Min	Max	Std
Train Gate Controller					
$\mathcal{V}$-Polytope	2.24	1.41	1.56	3.07	0.4
$\mathcal{H}$-Polytope	1.56	1.29	1.2	3.25	0.54
Fischer's Protocol					
$\mathcal{V}$-Polytope	2.9	2.67	2.51	4.06	0.49
$\mathcal{H}$-Polytope	4.09	3.84	3.4	5.67	0.64
Nuclear Reactor					
$\mathcal{V}$-Polytope	4.64	4.71	3.71	6.29	0.7
$\mathcal{H}$-Polytope	2.21	2.18	2.08	2.59	0.13

Now we proceed to evaluate the computational performance of our proposed method. To ensure statistical significance, we conducted the analysis 20 times on each benchmark. The local time horizon was set to $T = 30$, and the maximum jump depth was bounded by $J = 10$. For the Fischer's protocol benchmark, we used the parameter values $a = 8$ and $b = 12$ for three process-automata and a shared variable automaton. In the case of the nuclear reactor benchmark, we considered a system with 10 rods for this evaluation. The resulting runtimes are presented in Table 4.1.

Our algorithm has performed better using the $\mathcal{H}$-Polytope representation in the train-gate-controller and nuclear reactor benchmarks compared to the calculations using the $\mathcal{V}$-Polytope representation. This can be attributed to the added intersection operation with time intervals while checking for synchronization possibilities. For each synchronizing jump $e = (l,a,g,r,jump,l')$ the algorithm needs to perform $|syncDict(a)|$ intersection operations of the time interval (represented as a polytope) with the state set. The $\mathcal{H}$-Polytope representation is more efficient in this case because intersecting $\mathcal{H}$-Polytopes can be done in polynomial time, while intersecting $\mathcal{V}$-Polytopes requires conversion methods. Furthermore, computing jump successors of a state set in a particular time interval requires the construction of two halfspaces from the lower and upper bound of the time interval, and intersecting them with the state set. This operation is less costly when using $\mathcal{H}$-Polytopes.

The Fischer's protocol benchmark has shown the opposite behaviour, where the $\mathcal{V}$-Polytope representation performed better than the $\mathcal{H}$-Polytope representation. A possible explanation for this result is that the number of jumps in the Fischer instance we used is considerably bigger than the number of jumps in the nuclear reactor model. This has led to the size of the search forest being more than twice as big in Fischer. At this size of the search forest, the cost of reducing the $\mathcal{H}$-Polytopes outweighs the gain we get from the efficient intersection operation. In the next experiment, we try to capture the scalability of our method by increasing the number of automata in the Fischer's protocol and nuclear reactor benchmarks. We have set a local time horizon $T = 20$ and a jump bound $J = 10$, and we have represented the state sets as $\mathcal{V}$-Polytopes. The results are shown in Table 4.2.

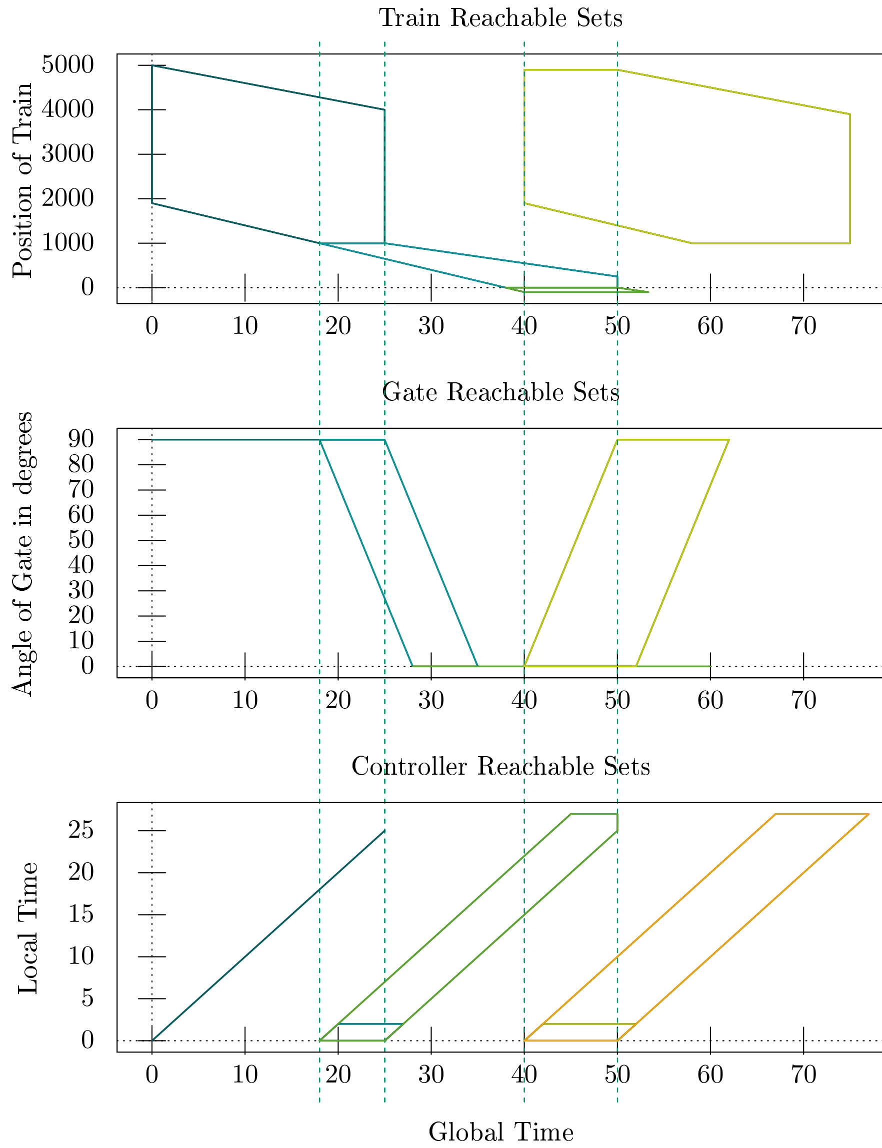


Figure 4.5: The reachable state sets of TGC benchmark with $\mathcal{H}$-Polytope representation, time horizon $T = 25$ and a jump depth $J = 3$.

Table 4.2: Scalability metrics of Fischer's protocol and nuclear reactor benchmarks, timeout (TO) was set to 15 minutes.

	#rods/ processes	$\|Var\|$	*time (sec)*	*mem (KB)*	*Search Forest size*	*avg #Sync Candidates*
Fischer	3	7	2.14	22044	522	1.12
	6	13	241.51	136336	36421	1.36
	8	17	TO	513156	-	-
NRS	10	22	2.73	21852	201	1.15
	15	32	5.81	23500	301	1.11
	20	42	16.18	27840	401	1.08
	40	82	192	161048	801	1.04

The results show that our method performed significantly better on the nuclear reactor model than on Fischer, where the algorithm timed out after 15 minutes upon raising the number of process automata to 8. Whereas in the nuclear reactor benchmark, the algorithm was able to compute the reachable state set for 40 rods in 3.2 minutes. This can be explained by the blow-up in the search forest size in Fischer's protocol, which is caused by the higher number of synchronizing jumps.

As expected the memory consumption correlated positively with the number of variables and the size of the search forest. The overall number of variables did not strongly affect the runtime of the analysis, especially because each automaton (in both benchmarks) had the dimension 2, where one dimension was reserved for the additional local time variable, except for the shared variable automaton which only had the local time variable.

Our extension of the search forest with the *Sync* function has proven to be a promising approach in both benchmarks. The function has been able to reduce the number of candidates for synchronization (i.e., nodes that need to be checked if they're a viable synchronization partner) to almost only one node per search tree. However, this extension needs further testing on models that involve more local jumps, which would lead to bigger subtrees when searching for synchronization partners.

Chapter 5

Conclusion

5.1 Summary

In this work, we have developed a novel synchronization technique for the concurrent reachability analysis of multiple rectangular automata. Our approach took advantage of local time variables to realign the evolution of the input automata whenever a synchronizing jump is taken. By extending forward reachability methods that use decomposition, our method enables efficient and accurate analysis of large high-dimensional system models. Our research demonstrates that synchronization on selected jumps is more efficient than strict synchronization using parallel composition. We applied a new approach for saving dependency information between the analysed systems, which allowed for including time dependencies during the computation of jump successors with little added effort.

5.2 Discussion and Future Work

The proposed method was developed with the goal of addressing the scalability issue of reachability analysis to high-dimensional hybrid systems. We have shown that our approach achieved synchronized reachability analysis of rectangular automata with little memory overhead. A combination of the variable set separation method [SNÁ17] and our synchronization technique could provide a more accurate successor computation. Our current implementation requires that the input automata have disjoint variable sets. Further work might include extending the method to handle shared variables, which would involve the projection of the state sets on multiple dimensions (not just the time dimension) and the intersection of state sets to find a common state that satisfies the synchronization requirements. However, our search tree extension is not effected by shared variables and can still be used to reduce the search space for synchronizing states even if the variable sets are not disjoint.

As mentioned in Section 4.1 our method is limited to single-rooted search trees, i.e., models that have a single initial state. This could be bypassed by modifying the models to have a new single initial state with trivial local transitions to the original multiple initial states. But, multiple initial states could also be handled by implementing an updated initial step in the analysis that creates the search trees with multiple roots. One effect of such a modification would be that we have to consider

synchronization partners for each of the initial states, which might lead to duplicating some search tree nodes.

Performing the synchronized analysis on rectangular automata with state sets represented as polytopes has shown promising results. Our method has outperformed the shallow synchronization method [BCL+10] on benchmarks like Fischer's protocol and the nuclear reactor system. Moreover, the synchronization method is not dependent on the state set representation and can be extended to other representations like zonotopes or ellipsoids. An extension to linear hybrid automata could also be examined in the future, because the search tree structure in HyPro is the same. However, the over-approximation of the reachable sets in linear hybrid automata might lead to flawed synchronization because the projection of the state sets on the time dimension would not be exact. Further research is needed in that regard

Bibliography

[Ábr17] Ábrahám, Erika: *Modeling and analysis of hybrid systems*. RWTH Aachen University, Lecture Notes, 2017.

[ACH+95] Alur, R., C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine: *The algorithmic analysis of hybrid systems*. Theoretical Computer Science, 138(1):3–34, 1995. `https://doi.org/10.1016/0304-3975(94)00202-T`.

[AD94] Alur, Rajeev and David L. Dill: *A theory of timed automata*. Theoretical Computer Science, 126(2):183–235, 1994. `https://doi.org/10.1016/0304-3975(94)90010-8`.

[Alt15] Althoff, Matthias: *An introduction to CORA 2015*. In *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 120–151. EasyChair, 2015. `https://doi.org/10.29007/zbkv`.

[BCL+10] Bu, Lei, Alessandro Cimatti, Xuandong Li, Sergio Mover, and Stefano Tonetta: *Model checking of hybrid systems using shallow synchronization*. In *Formal Techniques for Distributed Systems*, pages 155–169. Springer Berlin Heidelberg, 2010, ISBN 978-3-642-13464-7. `https://doi.org/10.1007/978-3-642-13464-7_13`.

[BFF+18] Bogomolov, Sergiy, Marcelo Forets, Goran Frehse, Frédéric Viry, Andreas Podelski, and Christian Schilling: *Reach set approximation through decomposition with low-dimensional sets and high-dimensional matrices*. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, pages 41–50, New York, NY, USA, 2018. Association for Computing Machinery. `https://doi.org/10.1145/3178126.3178128`.

[BLW+10] Bu, Lei, You Li, Linzhang Wang, Xin Chen, and Xuandong Li: *Bach 2: Bounded reachability checker for compositional linear hybrid systems*. In *2010 Design, Automation & Test in Europe Conference & Exhibition*, pages 1512–1517, 2010. `https://doi.org/10.1109/DATE.2010.5457051`.

[BNO03] Bertsekas, D., A. Nedic, and A. Ozdaglar: *Convex Analysis and Optimization*. Athena Scientific optimization and computation series. Athena Scientific, 2003. `http://www.athenasc.com/convexity.html`.

[CÁF11] Chen, Xin, Erika Ábrahám, and Goran Frehse: *Efficient bounded reachability computation for rectangular automata.* In *Reachability Problems*, pages 139–152. Springer Berlin Heidelberg, 2011. https://doi.org/10.1007/978-3-642-24288-5_13.

[CS16] Chen, Xin and Sriram Sankaranarayanan: *Decomposed reachability analysis for nonlinear systems.* In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 13–24, 2016. https://doi.org/10.1109/RTSS.2016.011.

[Dan00] Dang, Thi Xuan Thao: *Verification and Synthesis of Hybrid Systems.* Theses, Institut National Polytechnique de Grenoble - INPG, October 2000. https://theses.hal.science/tel-00006738.

[DSÁR23] Delicaris, Joanna, Stefan Schupp, Erika Ábrahám, and Anne Remke: *Maximizing reachability probabilities in rectangular automata with random clocks.* In *International Symposium on Theoretical Aspects of Software Engineering*, pages 164–182. Springer, 2023. https://doi.org/10.1007/978-3-031-35257-7_10.

[FHT+07] Fränzle, Martin, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert: *Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure.* JSAT, 1:209–236, May 2007. https://doi.org/10.3233/SAT190012.

[FLGD+11] Frehse, Goran, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler: *SpaceEx: Scalable verification of hybrid systems.* In *Computer Aided Verification*, pages 379–395. Springer Berlin Heidelberg, 2011. https://doi.org/10.1007/978-3-642-22110-1_30.

[Gir05] Girard, Antoine: *Reachability of uncertain linear systems using zonotopes.* In *Hybrid Systems: Computation and Control*, pages 291–305. Springer Berlin Heidelberg, 2005. https://doi.org/10.1007/978-3-540-31954-2_19.

[Hen00] Henzinger, Thomas A.: *The Theory of Hybrid Automata.* Springer Berlin Heidelberg, 2000. https://doi.org/10.1007/978-3-642-59615-5_13.

[HKPV98] Henzinger, Thomas A., Peter W. Kopke, Anuj Puri, and Pravin Varaiya: *What's decidable about hybrid automata?* Journal of Computer and System Sciences, 57(1):94–124, 1998. https://doi.org/10.1006/jcss.1998.1581.

[Lam87] Lamport, Leslie: *A fast mutual exclusion algorithm.* ACM Trans. Comput. Syst., 5(1):1–11, jan 1987. https://doi.org/10.1145/7351.7352.

[LG09] Le Guernic, Colas: *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics.* Theses, Université Joseph-Fourier - Grenoble I, October 2009. https://theses.hal.science/tel-00422569.

[LG10] Le Guernic, Colas and Antoine Girard: *Reachability analysis of linear systems using support functions.* Nonlinear Analysis: Hybrid Systems, 4(2):250–262, 2010. `https://doi.org/10.1016/j.nahs.2009.03.002`.

[Lyg04] Lygeros, John: *Lecture notes on hybrid systems.* In *Notes for an ENSIETA workshop*, 2004. `https://api.semanticscholar.org/CorpusID:2793544`.

[MBT05] Mitchell, I.M., A.M. Bayen, and C.J. Tomlin: *A time-dependent hamilton-jacobi formulation of reachable sets for continuous dynamic games.* IEEE Transactions on Automatic Control, 50(7):947–957, 2005. `https://doi.org/10.1109/TAC.2005.851439`.

[Neu03] Neumaier, Arnold: *Taylor forms—use and limits.* Reliable Computing, 9:43–79, 2003. `https://doi.org/10.1023/A:1023061927787`.

[PQ08] Platzer, André and Jan David Quesel: *KeYmaera: A hybrid theorem prover for hybrid systems (system description).* In *Automated Reasoning*, pages 171–178. Springer Berlin Heidelberg, 2008. `https://doi.org/10.1007/978-3-540-71070-7_15`.

[SÁMK17] Schupp, Stefan, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski: *Hypro: A C++ library of state set representations for hybrid systems reachability analysis.* In *NASA Formal Methods*, pages 288–294. Springer International Publishing, 2017. `https://doi.org/10.1007/978-3-319-57288-8_20`.

[Sch19] Schupp, Stefan: *State Set Representations and their Usage in the Reachability Analysis of Hybrid Systems.* Dissertation, RWTH Aachen University, Aachen, 2019. `https://doi.org/10.18154/RWTH-2019-08875`.

[SNÁ17] Schupp, Stefan, Johanna Nellen, and Erika Ábrahám: *Divide and conquer: Variable set separation in hybrid systems reachability analysis.* Electronic Proceedings in Theoretical Computer Science, 250:1–14, July 2017. `http://dx.doi.org/10.4204/EPTCS.250.1`.

[Wan05] Wang, Farn: *Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures.* IEEE Transactions on Software Engineering, 31(1):38–51, 2005. `https://doi.org/10.1109/TSE.2005.13`.