State Set Representations and their Usage in the Reachability Analysis of Hybrid Systems

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Stefan Alexander Schupp, Master of Science
aus Haan, Deutschland

Berichter: Universitätsprofessorin Dr. Erika Ábrahám
Universitätsprofessor Dr. Goran Frehse

Tag der mündlichen Prüfung: 25. September 2019

Abstract

Hybrid systems in computer science are systems with combined discrete-continuous behavior. This work presents results obtained in the field of safety verification for linear hybrid systems whose continuous behavior can be described by linear differential equations. We focus on a special technique named flowpipe-construction-based reachability analysis, which over-approximates the reachable states of a given hybrid system as a finite union of state sets. In these computations we can use different geometric and symbolic representations for state sets as datatypes.

The choice of the state set representation has a strong impact on the precision of the approximation and on the running time of the analysis method. Additionally, numerous further parameters and heuristics influence the analysis outcome.

In this work we investigate on the influence and optimal usage of these parameters. Our results are collected in a publicly available open-source `C++` programming library named HyPro. The major contributions of this work are threefold:

- We present our HyPro library offering implementations for several state set representations that are commonly used in flowpipe-construction-based reachability analysis. A unified interface in combination with reduction and conversion methods supports the fast implementation of versatile analysis methods for linear hybrid systems.
- We put our library to practice and show its applicability by embedding a flowpipe-construction-based reachability analysis method in a CEGAR-based abstraction refinement framework. The parallelization of this approach further increases its performance.
- We introduce methods to decompose the search space and replace high-dimensional computations by computations in lower-dimensional subspaces. This method is applicable under certain conditions. An automated check of these conditions, an automated decomposition, and the integration of dedicated analysis methods for subspace computations extend our approach.

Zusammenfassung

Unter hybriden Systemen in der Informatik versteht man Systeme, welche diskretes als auch kontinuierliches Verhalten vereinen. In dieser Arbeit werden Ergebnisse aus dem Bereich der Verifikation linearer hybrider Systeme vorgestellt. Das kontinuierliche Verhalten der betrachteten Systeme kann dabei durch lineare Differentialgleichungen beschrieben werden. Diese Arbeit beschäftigt sich im Besonderen mit der über-approximierenden Erreichbarkeitsanalyse, in der die Menge der erreichbaren Zustände durch eine endliche Vereinigung von Zustandsmengen approximiert wird. Zustandsmengen während der Berechnung werden dabei durch verschiedene geometrische als auch symbolische Repräsentierungen dargestellt.

Die Wahl der Zustandsmengenrepräsentierung hat einen starken Einfluss auf die Präzision der Approximation als auch auf die Laufzeit der Analyse. Zusätzlich wird das Ergebniss der Analyse durch weitere Parameter und Heuristiken beeinflusst.

In dieser Arbeit erforschen wir den Einfluss und die optimale Einstellung dieser Parameter. Unsere Ergebnisse sind in der öffentlichen `C++` Bibliothek HyPro zur Verfügung gestellt. Die Beiträge dieser Arbeit lassen sich in drei Teile gliedern:

- Wir präsentieren unsere HyPro Programmierbibliothek. Diese beinhaltet Implementierungen verschiedener Datentypen, die in Algorithmen für die Erreichbarkeitsanalyse hybrider Systeme verwendet werden können um Zustandsmengen hybrider Systeme zu repräsentieren. Eine vereinheitlichte Schnittstelle zusammen mit Reduktions- und Konvertierungsmethoden erlauben die schnelle Implementierung von flexiblen Analysemethoden für lineare hybride Systeme.
- Wir zeigen die Anwendbarkeit der Methoden und Datenstrukturen in HyPro anhand der Einbettung eines üblichen Erreichbarkeitsanalyseansatzes in ein Framework, in welchem eine schnelle aber grobe Analyse iterativ durch die Verwendung von Gegenbeispielen verfeinert wird. Eine Parallelisierung des Ansatzes ist ebenfalls gegeben, welche die Laufzeiten weiter verbessert.
- Die Einführung von Methoden, um teure Berechnungen in hochdimensionalen Zustandsräumen durch weniger aufwendigen Berechnungen in niedrig-dimensionalen Zustandsräumen zu ersetzen. Die vorgestellte Methode ist nur unter bestimmten Bedingungen verwendbar. Wir präsentieren ein automatisiertes Verfahren, um diese Bedinungen zu überprüfen und wenn möglich solche niedrigdimensionale Räume zu identifizieren. Die Verwendung dedizierter Verfahren für die Analyse bestimmter Unterklassen von hybriden Systemen in Kombination mit der Entkopplung des Zustandsraumes vervollständigen unseren Ansatz.

Declaration of Authorship

I, Stefan Alexander Schupp declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research. I do solemnely swear that:

1. This work was done wholly or mainly while in candidature for the doctoral degree at this faculty and university;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this university or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others or myself, this is always clearly attributed;
4. Where I have quoted from the work of others or myself, the source is always given. This thesis is entirely my own work, with the exception of such quotations;
5. I have acknowledged all major sources of assistance;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published before. A detailed list can be found on the next page.

Aachen, September 29, 2019

Published Work

This work is partly based on the following peer-reviewed papers that have already been published. A detailed attribution of contributions can be found in Section 1.2 on page 9.

[CSB+15] Xin Chen, Stefan Schupp, Ibtissem Ben Makhlouf, Erika Ábrahám, Goran Frehse, and Stefan Kowalewski. "A Benchmark Suite for Hybrid Systems Reachability Analysis". In: *Proc. of NFM'15*. Vol. 9058. LNCS. Springer, 2015, pp. 408–414. DOI: 10.1007/978-3-319-17524-9_29.

[SÁC+15] Stefan Schupp, Erika Ábrahám, Xin Chen, Ibtissem Ben Makhlouf, Goran Frehse, Sriram Sankaranarayanan, and Stefan Kowalewski. "Current Challenges in the Verification of Hybrid Systems". In: *Proc. of CyPhy'15*. Vol. 9361. Information Systems and Applications, incl. Internet/Web, and HCI. Springer, 2015, pp. 8–24. DOI: 10.1007/978-3-319-25141-7_2.

[SÁB+17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. "HyPro: A C++ Library for State Set Representations for Hybrid Systems Reachability Analysis". In: *Proc. of NFM'17*. Vol. 10227. LNCS. Springer, 2017, pp. 288–294. DOI: 10.1007/978-3-319-57288-8_20.

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

[SÁ18a] Stefan Schupp and Erika Ábrahám. "Efficient Dynamic Error Reduction for Hybrid Systems Reachability Analysis". In: *Proc. of TACAS'18*. Vol. 10806. LNCS. Springer, 2018, pp. 287–302. DOI: 10.1007/978-3-319-89963-3_17.

[SÁ18b] Stefan Schupp and Erika Ábrahám. "Spread the Work: Multi-threaded Safety Analysis for Hybrid Systems". In: *Proc. of SEFM'18*. Vol. 10886. LNCS. Springer, 2018, pp. 89–104. DOI: 10.1007/978-3-319-92970-5_6.

[SÁ18c] Stefan Schupp and Erika Ábrahám. "The HyDRA Tool – a Playground for the Development of Hybrid Systems Reachability Analysis Methods". In: *Proc. of the PhD Symposium at iFM18 (PhD-iFM18)*. University of Oslo, 2018, pp. 22–23.

[SWÁ18] Stefan Schupp, Justin Winkens, and Erika Ábrahám. "Context-dependent Reachability Analysis for Hybrid Systems". In: *Proc. of FMI'18*. IEEE Computer Society Press, 2018, pp. 518–525. DOI: 10.1109/IRI.2018.00082.

[SÁ19] Stefan Schupp and Erika Ábrahám. "Context-dependent Reachability Analysis for Hybrid Systems". In: *Reuse in Intelligent Systems* (2019). to appear.

Acknowledgements

The past few years at the group Theory of Hybrid Systems were filled with lots of memorable hours which involved many people who took part in this work one way or another. At this point, I want to express my deepest appreciation for my supervisor Erika Ábrahám, who gave me the opportunity to work in this interesting field of research and who leads this amazing group with its friendly air. Thank you for the patience and effort spent when guiding and teaching me during the whole time and for the many valuable discussions we had. I want to express my sincere gratitude to my second supervisor, Goran Frehse for his support and the interesting discussions we had whenever we met either during visits or at conferences.

My daily work in the group would not have been the same without my colleagues Xin Chen, Florian Corzilius, Rebecca Haehn, Nils Jansen, Gereon Kremer, Francesco Leofante, Ulrich Loup and Johanna Nellen—for the past years you were resposible for the nice atmosphere at the chair, for some unforgettable evenings, and some nice conference visits spent together. Additionally, I want to thank my student assistants Marta Grobelna, Felix Seidl, and Phillip Tse for their effort and time spent on helping with the development of HyPro. Special thanks goes to Ibtissem Ben Makhlouf and Stefan Kowalewski who were part in the HyPro team for their contributions to the project and valuable discussions we had.

Furthermore, I want to thank Jannik Hülls, Anne Remke, Felix Freiberger and Holger Hermanns who are some of the first users of HyPro and with patience helped a lot to bring the library to its current state. I am grateful for the support and the helpful tips from Werner Seiler. At this point I want thank my colleagues from the hybrid systems community, especially the ARCH-COMP organization team who spent so much effort on putting together this competition with its friendly and supportive character. This thesis would not have been possible without the help from Nina Kusch and Philipp Wacker, who spent hours on proofreading. Additionally, I want to thank Torsten Zimmermann for the great advice on how to present the obtained results properly in the final presentation.

Last but not least I want to thank my family and friends for their encouragement and support they provided. During the last years, I have spent a lot of time at the chair or traveling to conferences. At this point I want to thank Vasilena for the support and patience thoughout the whole time.

Contents

1

Introduction

Technical and scientific development in the past decades has enabled us to use more and more technical systems employing automation for our comfort in our everyday life. The increase in the computational capabilities of digital processors, as well as their low price, has led to a significant increase in their usage in many ways.

Nowadays, integrated digital systems are used all around us for controlling in various areas. Modern cars contain hundreds of digital controllers performing all kinds of tasks needed for the correct operation of the vehicle. Recent development in the field of autonomous driving has even increased the bond between digital systems and continuous environments, establishing the usage of digital systems in safety-critical applications. Not even considering fully autonomous cars but already simple systems for public transport like autonomous monorail trains at airports involve safety-critical components—the train should not only brake when arriving at a certain gate but also be able to detect obstacles on the track and react in time. Going further, digital controllers in planes or more recently in crewless autonomous rockets perform highly critical tasks in which a failure can lead to severe incidents. Another field where digital controllers are involved are production plants, e.g., automated chemical plants or general production facilities where digital (embedded) systems supervise and control the behavior of the plant. Applications range from simple supervision using sensors to controllers interacting with the system using actuators such as valves and switches based on their sensing of the environment. An example could be a digital controller which is supposed to keep the temperature of a liquid in a chemical plant within a certain range by adjusting a heating device (see Figure 1.1). Using temperature sensors, the controller observes the state of its environment, i.e., the temperature and influences the system's behavior using its actuator in the form of a switch to operate the heating device. What the aforementioned systems all have in common is the interaction of a digital system with a continuous environment, thus being part of the large group of cyber-physical system (CPS).

For every designed system we require that it operates correctly with respect to some specification; this is even more important for safety-critical applications in which human life is at stake or significant financial damage may be caused by

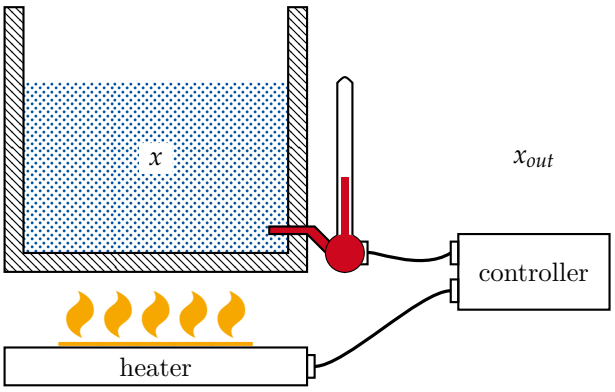


Figure 1.1: Schematic of the liquid heating system consisting of a tank filled with a liquid, a heating device and a controller to keep the liquid in the tank at a certain temperature x.

erroneous behavior. Typical safety specifications exclude malicious behavior of a system described as a set of states over the system's quantities. Put differently, in the absence of a specific behavior characterized by a set of system states we can declare the system safe with respect to this set of states. Reconsidering our chemical plant, a controller would try to exclude behavior where the temperature of a liquid in a tank rises above a particular value—the system is considered safe if this case never happens.

Testing as a method for estimating the correctness of a system's behavior is commonly used in various areas during the development to get an impression about the system's behavior. In those cases, the system is tested against concrete test situations which may lead to hazardous situations and system failure. While this method is usually a cheap approach to finding potential errors in the system design, testing is not suitable for the verification of CPS or more general for systems with an infinite state space. As testing considers a potentially large but finite set of system executions, guarantees for the safety of a system cannot be given for systems with an infinite state space; continuous behavior as observed in CPS induces an infinite state space for system quantities evolving continuously.

Formal verification as a method to guarantee the safety of hybrid systems with respect to certain specifications has been in the focus of research in the past decades. Being able to provide conclusive proofs in case of success, formal verification is especially suited for the analysis and verification of safety-critical systems. To be able to apply formal verification to a system, we need an *abstraction* of said system, a model that reflects the relevant properties but abstracts away unnecessary details. In our example of the chemical plant, it does not matter whether the tank of liquid is blue or red. In contrast to that, its volume or the amount of liquid instead do affect the way the temperature inside the tank evolves.

Continuous behavior, in the following referred to as *dynamics* of (physical) quantities, has been studied for a long time in engineering. The term *dynamical system* which was mainly coined by Henri Poincaré [Poi92] and George David

Birkhoff [Bir27] characterizes the continuous behavior considered in this work. In our example of the liquid-filled tank, we can describe the dynamics of the temperature x without any heating influence as a linear ordinary differential equation (ODE)

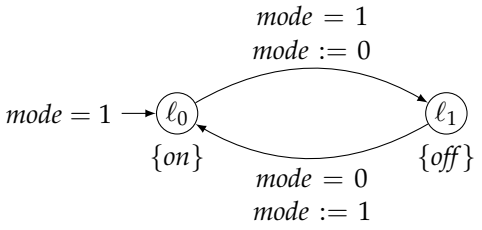
$$\begin{aligned} \dot{x} &= K \cdot (x_{out} - x) \\ \dot{x}_{out} &= 0 \end{aligned}$$

where $\dot{x}, \dot{x}_{out}$ describe unknown time-dependent functions $x(t), x_{out}(t)$ by their time derivatives $\frac{\partial x}{\partial t}$ respectively $\frac{\partial x_{out}}{\partial t}$. In our simplified model, we may use the constant K to encode the influence of the surface of the tank and the thermal conduction of the liquid. The evolution of the temperature of our liquid thus is proportional to the difference between its temperature and the temperature x_{out} outside the tank. The *initial value problem* specifies a system of ODEs and initial values for their variables and asks for time-dependent functions that are solutions of the ODE system and that take at time point zero the given initial values. For our example ODE system above and the concrete initial values $(x_0, x_{out,0}) \in \mathbb{R}^2$, the solution for the fluid temperature is given as

$$x(t) = e^{tA} \cdot x_0$$

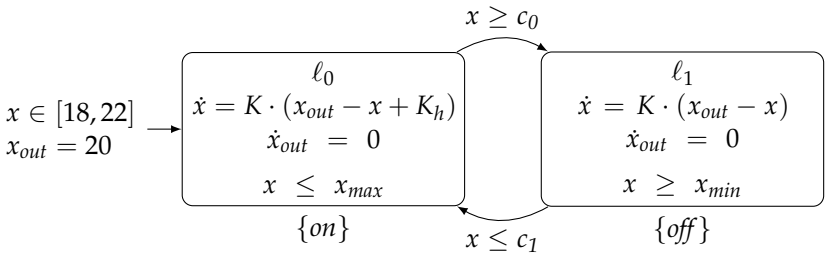
where the matrix exponential A depends on the constant K and $x_{out,0}$.

While dynamical systems are mainly considered in engineering sciences, the underlying models in computer science are mostly discrete, which is rooted in the discrete operation of digital processors which discretely execute instructions and sequences thereof (programs). The most simplistic models for programs are state transition systems in which locations (depicted as circles) encode program states and transitions (jumps) between them (depicted as arrows) indicate the program flow. An arrow denotes the entry point of a program with no source location—an execution of the program starts at this arrow's target location and step by step traverses the transition system. The observable behavior of the program is reflected by a sequence of locations (which encode states) according to the traversal of the transition system. Formal methods employ temporal logic to validate properties of the program such as "it can never happen that the program visits a certain location". A transition system generalizes this concept by adding a finite set of variables Var, and guards and variable resets on discrete jumps. We can model the control-program of digital controller for the exemplary chemical plant as the following transition system, where the variable *mode* stores the system's current control mode ($mode = 1$ denotes that the heater is on, whereas $mode = 0$ encodes that the heater is off):



The state of a program in this model is reflected by a combination of the location in which the control stays in and the current variable valuation.

For both model types, various techniques have been developed in the past to analyze the properties of the respective system by analyzing the corresponding model. The usage of digital controllers in a continuous environment demanded a new modeling language—it is no longer sufficient to analyze continuous and discrete parts of a combined system individually, as they may interact. Modeling paradigms such as hybrid Petri nets [AD98; DA01] and hybrid automata [Hen96] were introduced, which allow to model mixed discrete-continuous behavior. Our plant example can be modeled by the following hybrid automaton, where K, K_h, c_0, and c_1 are constants quantifying the thermal conduction, the influence of the heater when turned on, and thresholds for the controller to switch between modes (we have removed the variable *mode* storing the control mode):



The model unifies both discrete and continuous behaviors—the transition system structure is used to model the discrete part, whereas in each location *ordinary differential equations* model the evolution of time. The interaction between the discrete and the dynamic behavior is modeled by the guards and variable resets attached to the jumps, and by *invariant conditions* attached to the locations to put boundaries on the dynamic behavior. States are now represented as a tuple of a location (the discrete part) and a variable valuation (the dynamic part). Similarly, we have to adapt specifications we want to verify by involving aspects coming from the dynamic behavior, i.e., variable valuations. For instance, we can now attempt to validate statements such as "it never happens that within 20 s the temperature rises above 65 °C".

Computer scientists use model checking [BK08] as a formal approach towards the safety verification of models such as LTS and variants thereof. Control engineers, on the other hand, are interested in stability and robustness criteria of dynamic systems. For hybrid systems new approaches were needed which can prove safety not only based on the discrete structure but also considering the continuous dynamic part of the model.

Hybrid systems safety verification can be done through reachability analysis which tries to answer the question "is a certain set of bad states reachable in a given hybrid system model starting from a defined set of initial states?". As the reachability problem for general hybrid automata is undecidable [HKP+98], different approaches for semi-decision procedures have been developed. *Bounded model checking* approaches encode bounded executions of a given hybrid automaton into logical formulas and utilize satisfiability modulo theories (SMT) techniques to determine (bounded) safety [RS05; EFH08; GKC13]. Closely

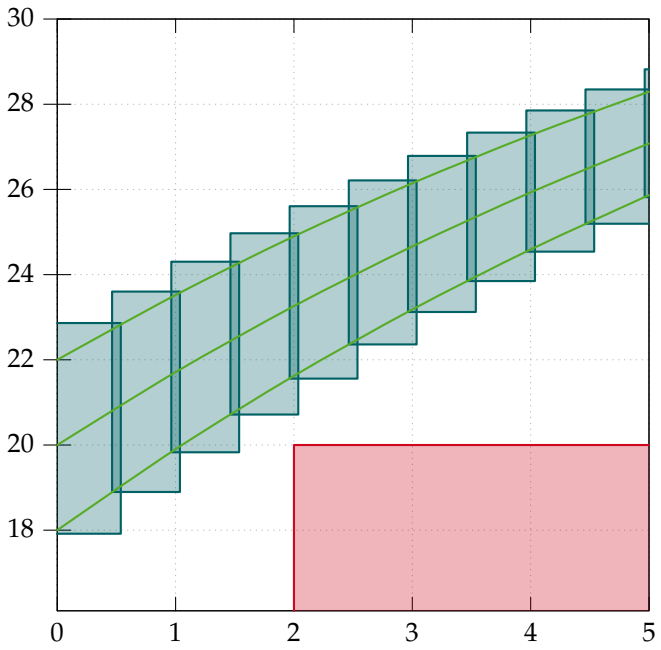


Figure 1.2: Over-approximation of the set of reachable states by boxes for the tank-example (petrol), exemplary trajectories (green) and the set of undesired behavior (red). The plot shows the temperature x (vertical axis) over time (horizontal axis).

related, approaches based on *theorem proving* axiomatize a theory for hybrid systems to formally specify their behavior as hybrid programs and use semi-automated theorem provers to derive their safety [PQ08]. Approaches using *rigorous simulation* attempt to validate safety by simulating a finite set of traces for a given system together with rigorous extrapolation to the full set of traces. In this work, we focus on *flowpipe-construction*-based reachability analysis for linear hybrid systems. In this method, the set of reachable states is over-approximated by a set of sets represented by geometric shapes [FLD+11; CÁS13; Alt15]. Based on approaches like orthogonal polyhedra [Dan00], oriented rectangular hulls [SK03], or zonotopes [Gir05] a general reachability analysis algorithm for flowpipe-construction-based methods was established and continuously improved to reduce over-approximation errors and computational effort [Le 09]. Improvements towards the scalability by using different state set representations, for instance *support functions* [LG10; FR12; FKL13], using decomposition and reduction methods [BD17a; BFF+18], or by employing parallelized approaches [GRB+18] were proposed. Another branch of research aimed at improving the precision of the over-approximation [FKL13; BFG+17].

Considering our tank example, we can over-approximate the set of dynamic evolutions starting from the initial state using constants $K = 0.1$ and $K_h = 18$ and boxes to represent state sets (see Figure 1.2). The over-approximation of the set of reachable states by a sequence of boxes guarantees that all trajectories up to a time bound (example trajectories in green) are fully contained in the

union of the boxes. This approach enables us to use the set of boxes to verify safety properties—instead of arguing about the continuous functions describing the system's dynamics we can argue about a set of convex sets. For instance, we are now able to validate the statement "After 2 s the temperature is invariantly above 20 °C" by computing an intersection of the set of malicious behavior (red) with the approximation of the set of reachable states. As the intersection is empty, the statement holds for all executions starting in the set of initial state up to a time bound of 5 s.

Mode changes, i.e., the discrete behavior can be handled using the approximation of the set of reachable states as well; in our example in those parts of boxes that intersect the half-space $x \geq c_0$ the discrete jump to location ℓ_{off} might be enabled. We also observe that boxes might not be the best set to approximate our trajectory; instead, we might want to consider a more tight approximation using a different state set representation to over-approximate the set of trajectories.

Using state of the art flowpipe-construction-based reachability analysis methods as a basis for our development, in this work we aim at improving existing approaches by investigating the following questions:

- How can we efficiently represent state sets? How can different state set representations be incorporated into a flowpipe-construction-based reachability analysis method?
- How can we extend existing approaches to obtain a scalable method for flowpipe-construction-based reachability analysis?
- How can we exploit system features and domain-specific knowledge to our advance?

Our answers and results to these research questions are presented in the course of this work. After having introduced preliminary information and notation in Chapter 2, we will provide a more thorough introduction to hybrid systems, hybrid automata and flowpipe-construction-based reachability analysis in Chapter 3. Our work starts with a collection of examples for hybrid systems and an overview on challenges that tool developers have to face when creating a flowpipe-construction-based reachability analysis tool for hybrid systems. Chapter 6 will present our work on the first research question describing our collection of state set representations. In Chapters 7 and 8 we present our improvements which are based on the previous chapters. In the following, we discuss our contributions in more detail, followed by a list of publications this work is based on.

1.1 Contributions

The main contributions of this thesis are presented in the Chapters 4 to 8. In the following, we describe our main contributions along with pointers to the respective sections.

Hybrid Systems: Examples and Challenges. Our first publications relevant for this dissertation focus on the state of the art in hybrid systems safety verification to establish a foundation for our research. Our collection of benchmarks [CSB+15] for hybrid systems reachability analysis as a basis for future experimental evaluations provides details about the type and characteristics of individual hybrid systems in industry and academia. During this phase, we obtained an overview of the landscape of flowpipe-construction-based reachability analysis for hybrid systems and were able to gain insights into open problems that need to be tackled by future tool developers [SÁC+15]. Our main results from this phase are as follows:

- We provide a collection of benchmarks along with a classification of the considered models which complements existing collections of benchmarks.
- We identify challenges relevant for tool developers that need to be addressed to improve the applicability of hybrid systems safety verification.

A Programming Library for State Set Representations. In the context of the HyPro-project[1] funded by the German research council we put our previously gained knowledge into practice by developing our `C++`-library HyPro for state set representations for hybrid systems reachability analysis [SÁB+17]. With a diverse collection of state set representations at hand, we enhanced our library by algorithms and data structures required to set up a state of the art reachability analysis method which allowed us to get promising experimental results on the collected benchmarks. We achieved the following goals during this phase:

- We created and published a `C++`-library for state set representation for flowpipe-construction-based reachability analysis for linear hybrid systems.
- We extended our library with data structures and algorithms that are typically needed for many reachability analysis approaches to further assist the tool development in the area of hybrid systems safety verification.

CEGAR with Partial Path Refinement. On top of the development of HyPro, we addressed some of the challenges identified in the earlier phase of this work by implementing a CEGAR method using partial path refinement [SÁ18a]. CEGAR stands for *counterexample-guided abstraction refinement*. Its idea is to use an over-approximative abstraction of the system and if spurious counterexamples are detected in the abstraction then to refine it to exclude the found and other "similar" spurious counterexamples. The refinement step in our approach is done by recomputing reachability along found counterexample paths using search configurations that lead to more precise computations. For the refinement steps, we utilize *strategies* that define a sequence of analysis parameter configurations which typically come with increasing computational effort but reduced approximation error. During the refinement we reuse as much of the previous computations as possible, which leads to what we call *partial path refinement*. The usage of high-precision configurations only when

[1] `https://ths.rwth-aachen.de/research/projects/hypro/` (checked July, 30^{th}, 2019)

required resulted in promising experimental results. Following this thread, we also worked on the parallelization of our approach [SÁ18b] as an extension to partial path refinement. Achievements of this phase include:

- The development of a generalized CEGAR-based approach for partial path refinement in hybrid systems reachability analysis.
- The successful implementation of this approach in a tool prototype based on HyPro. The author did most implementation work; parts of the presented refinement approach have been implemented during the Master's thesis of Dustin Hütter [Hüt16].
- The extension of our approach towards the application in a multi-threaded environment.

Subspace Decomposition. In the context of a collaboration with my colleague Johanna Nellen we extended the HyPro-library to set up a dedicated reachability analysis method for PLC-controlled plants based on subspace decomposition of hybrid automata [SNÁ17]. The main idea is to divide the variables into subsets and to compute the evolution independently for each subset, instead of computing it globally in a high-dimensional space for all variables at the same time. Additionally, if the variables in a subset have commonalities in their dynamics, then dedicated reachability analysis techniques might be applicable to reduce the computational costs and to increase the precision. Exploiting the modular nature of the HyPro-library together with the diverse collection of state set representations, we were able to obtain interesting experimental results that gave ideas for further research. While our first approach was static in the sense that a subspace decomposition had to be provided by the user, in the following year we published the extension of this approach for an automated on-the-fly decomposition [SWÁ18]. In this work, we also extended HyPro with state of the art approaches for timed automata, a subclass of general hybrid automata. An additional extension for the analysis of rectangular automata, a subclass of general hybrid automata and a super-class of timed automata, was recently added to our library [SÁ19]. Essential contributions of this phase include:

- The development of a state space decomposition approach.
- The implementation of this approach extended with a graph-based method to automatically synthesize state space decompositions during analysis.
- An extension of the HyPro library with specialized reachability analysis methods for subclasses of hybrid systems which is integrated into our decomposition method.

HyPro and HyDRA Tools. All implementations mentioned earlier have been made publicly available; HyPro as an open-source library, HyDRA as an executable. We have participated with our implementation in the ARCH

Competition[2] each year since its launch in 2017. There are not so many tools available in this area, and those that are available support the analysis of different hybrid system types and offer specialized analysis techniques. Therefore it is not easy to fairly compare their results. Still, in a friendly competition, the tool developers are encouraged to participate and exchange ideas. The results of this friendly competition from 2017 until 2019 have been published in [ABC+17; ABC+18; ABF+19].

For repeatability all benchmarks and configurations used for the experimental evaluations are available online[3].

1.2 Publications

In the following section, we will present publications relevant for this dissertation together with a summary of the content of each publication and a paragraph about my contributions to this publication. All publications mentioned were written together with my supervisor Erika Ábrahám.

Relevant Publications

[SÁC+15] Stefan Schupp, Erika Ábrahám, Xin Chen, Ibtissem Ben Makhlouf, Goran Frehse, Sriram Sankaranarayanan, and Stefan Kowalewski. "Current Challenges in the Verification of Hybrid Systems". In: *Proc. of CyPhy'15*. Vol. 9361. Information Systems and Applications, incl. Internet/Web, and HCI. Springer, 2015, pp. 8–24. DOI: 10.1007/978-3-319-25141-7_2

In this survey paper, we discuss state-of-the-art reachability analysis methods for hybrid systems with a focus on flowpipe-construction-based approaches. After an introduction to the topic, we describe current tools along with their characteristics regarding the type of approach they implement, the type of dynamics they can handle, and special distinguishing features. In an experimental evaluation, we apply some of the tools to a selection of benchmarks and identify strengths and limitations of tools. In conclusion to this, we present a collection of open problems that need to be addressed in the future to increase the usability of hybrid systems safety verification methods for industrial and academic applications.

In this paper, I worked on the description of tools along with their features and carried out the experimental evaluation. Furthermore, I contributed to the identification of open problems and challenges in hybrid systems reachability analysis.

[CSB+15] Xin Chen, Stefan Schupp, Ibtissem Ben Makhlouf, Erika Ábrahám, Goran Frehse, and Stefan Kowalewski. "A Benchmark Suite for Hybrid Systems Reachability Analysis". In: *Proc. of NFM'15*. Vol. 9058. LNCS. Springer, 2015, pp. 408–414. DOI: 10.1007/978-3-319-17524-9_29

[2] https://cps-vo.org/group/ARCH/FriendlyCompetition (checked July, 30th, 2019)
[3] https://ths.rwth-aachen.de/people/stefan-schupp/repeatability/

In this paper, we present a collection of benchmarks for flowpipe-construction-based reachability analysis for hybrid systems. Our collection contains a variety of linear and non-linear hybrid systems along with safety specifications. For each benchmark we give a description of the underlying system, a classification concerning syntactic properties, identify specific challenges, and provide model files for SpaceEx and Flow*. We evaluate the tools SpaceEx, Flow*, and DReach on the presented benchmarks.

I worked on the experimental evaluation of the linear benchmarks and the translation between different input languages. Furthermore, I contributed to the collection and presentation of the benchmarks.

[SÁB+17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. "HyPro: A C++ Library for State Set Representations for Hybrid Systems Reachability Analysis". In: *Proc. of NFM'17*. Vol. 10227. LNCS. Springer, 2017, pp. 288–294. DOI: 10.1007/978-3-319-57288-8_20

In this work, we publish HyPro, our C++-library for state set representations for the flowpipe-construction-based reachability analysis of linear hybrid systems. We present the range of features implemented in the library with a focus on the potential usage of different implementations of number types during computation. In an experimental evaluation, we compare the running times of a state-of-the-art flowpipe-construction-based reachability analysis algorithm on a selection of benchmarks by using multiple different number types, various state set representations and multiple backends for linear optimization performed during the analysis. We compare the running times of those configurations with results from SpaceEx.

My contribution to this work was the setup and implementation of the library HyPro. I provided its implementation except for the implementation of zonotopes and Taylor models which were provided by Ibtissem Ben Makhlouf respectively Xin Chen. Parts of the library were implemented in the course of the theses from Christoph Kugler [Kug14], Simon Froitzheim [Fro16], Phillip Florian [Flo16], Igor Bongartz [Bon16], and Sabrina Kielmann [Kie18], which I supervised. I obtained the experimental results by using my implementation of a flowpipe-construction-based reachability analysis method, which is also contained in the HyPro distribution, and ran the experiments for SpaceEx.

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

In this collaboration with my colleague Johanna Nellen, we aimed at improving reachability analysis methods for plants controlled by programmable logic controllers (PLCs). We model a closed-loop control of a PLC-controller and a plant as a hybrid automaton with urgent transitions to reflect the control flow of the PLC program. We extend our

flowpipe-construction-based reachability analysis method by exploiting domain-specific knowledge about clocks and about program variables that do not evolve continuously. Our experimental results indicate that subdividing a model into syntactically independent subspaces is promising.

My part in this work was the adaption of the reachability analysis method towards the analysis of syntactically independent subspaces and a method to handle urgent transitions in HyPro. The first adaption allows handling variables that do not evolve continuously as well as clocks separately to reduce the state space dimension for the analysis of the remaining continuously evolving variables. I am the main author of the paper and wrote major parts of it.

[SÁ18a] Stefan Schupp and Erika Ábrahám. "Efficient Dynamic Error Reduction for Hybrid Systems Reachability Analysis". In: *Proc. of TACAS'18*. Vol. 10806. LNCS. Springer, 2018, pp. 287–302. DOI: 10.1007/978-3-319-89963-3_17

In this work, we present a counterexample-guided abstraction refinement (CEGAR) approach for flowpipe-construction-based reachability analysis methods. This approach aims to increase the efficiency of current reachability analysis methods by selectively increasing the precision of the analysis on potential counterexample paths. This is realized by utilizing multiple analysis parameter configurations with different levels of precision during the analysis. The change of parameters is triggered whenever a potential counterexample path is discovered and this specific path is refined using a different analysis parameter configuration, leading to a more precise approximation of the set of reachable states on this path and with the aim to declare a counterexample spurious.

My contribution to this work lies in the development of the method, its implementation in HyPro and its evaluation, assisted by the thesis students Dustin Hütter [Hüt16] and Johannes Neuhaus [Neu16]. I am the primary author and I wrote major parts of this paper.

[SÁ18b] Stefan Schupp and Erika Ábrahám. "Spread the Work: Multi-threaded Safety Analysis for Hybrid Systems". In: *Proc. of SEFM'18*. Vol. 10886. LNCS. Springer, 2018, pp. 89–104. DOI: 10.1007/978-3-319-92970-5_6

In this work, we present a parallelized version of our flowpipe-construction-based reachability analysis, which is compliant with our counterexample-guided abstraction refinement (CEGAR)-approach of partial path refinement. Based on previous design concepts for flowpipe-construction-based reachability analysis methods [FR09], we modularize our approach and switch to a worker-based method. We discuss aspects specific to the parallelization, i.e., synchronization and locking in data structures during the analysis, as well as methods for work balancing during running time.

With the assistance of the thesis student Johannes Neuhaus [Neu16], I have set up a dedicated tool HyDRA which is based on HyPro and implements our CEGAR-approach in a parallelized way. Apart from

working on the implementation, I evaluated the approach. I am the main author and I wrote major parts of the paper.

[SWÁ18] Stefan Schupp, Justin Winkens, and Erika Ábrahám. "Context-dependent Reachability Analysis for Hybrid Systems". In: *Proc. of FMI'18*. IEEE Computer Society Press, 2018, pp. 518–525. DOI: 10.1109/IRI.2018.00082

In this paper, we extend our approach of analyzing syntactically independent sets of variables. We present an automated technique that can detect decompositions of a given hybrid automaton. Furthermore, we extend our portfolio of available approaches by adding a state-of-the-art analysis method for timed automata. In combination with syntactic decompositions, this allows analyzing hybrid automata containing clocks more efficiently by using decomposition and tailored approaches for the analysis of subspaces.

Together with the thesis student Justin Winkens [Win18], we implemented the analysis method for timed automata in HyPro along with an automated state space decomposition method. Furthermore, we created a modularized version of HyDRA in which workers are composed of individual handlers. We evaluated our approach on a selection of benchmarks. I am the main author and wrote most parts of this paper.

[SÁ18c] Stefan Schupp and Erika Ábrahám. "The HyDRA Tool – a Playground for the Development of Hybrid Systems Reachability Analysis Methods". In: *Proc. of the PhD Symposium at iFM18 (PhD-iFM18)*. University of Oslo, 2018, pp. 22–23

In this extended abstract, we present the HyDRA tool, in which all previously presented approaches are combined into one flowpipe-construction-based reachability analysis tool for linear hybrid systems.

I wrote major parts of this paper summing up our previous work.

Further Publications

- Jó Ágila Bitsch Link, Christoph Wollgarten, Stefan Schupp, and Klaus Wehrle. "Perfect Difference Sets for Neighbor Discovery: Energy Efficient and Fair". In: *Proc. of ExtremeCom'11*. ACM Press, 2011, 5:1–5:6. DOI: 10.1145/2414393.2414398
- Stefan Schupp. "Interval Constraint Propagation in SMT-compliant Decision Procedures". MA thesis. RWTH Aachen University, 2013
- Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving". In: *Proc. of SAT'15*. Vol. 9340. LNCS. Springer, 2015, pp. 360–368. DOI: 10.1007/978-3-319-24318-4_26
- Jannik Hüls, Stefan Schupp, Anne Remke, and Erika Ábrahám. "Analyzing Hybrid Petri Nets with Multiple Stochastic Firings Using Hypro".

In: *Proc. of VALUETOOLS'17*. ACM Press, 2017, pp. 178–185. DOI: 10.1145/3150928.3150938

- Matthias Althoff, Stanley Bak, Dario Cattaruzza, Xin Chen, Goran Frehse, Rajarshi Ray, and Stefan Schupp. "ARCH-COMP17 Category Report: Continuous and Hybrid Systems with Linear Continuous Dynamics". In: *Proc. of ARCH'17*. Vol. 48. EPiC Series in Computing. EasyChair, 2017, pp. 143–159. DOI: 10.29007/4dcn

- Matthias Althoff, Stanley Bak, Xin Chen, Chuchu Fan, Marcelo Forets, Goran Frehse, Niklas Kochdumper, Yangge Li, Sayan Mitra, Rajarshi Ray, Christian Schilling, and Stefan Schupp. "ARCH-COMP18 Category Report: Continuous and Hybrid Systems with Linear Continuous Dynamics". In: *Proc. of ARCH'18*. Vol. 54. EPiC Series in Computing. EasyChair, 2018, pp. 23–52. DOI: 10.29007/73mb

- Stefan Schupp, Francesco Leofante, Erika Ábrahám, and Armando Tacchella. "Robot Swarms as Hybrid Systems". In: *Proc. of SNR'18*. to appear. EPTCS, 2018

- Francesco Leofante, Stefan Schupp, Erika Abraham, and Armando Tacchella. "Engineeering Controllers for Swarm Robotics Via Reachability Analysis in Hybrid Systems". In: *Proc. of ECMS'19*. to appear. 2019

2

Preliminaries

In this chapter, we present some preliminaries and basic notations which will be used throughout this work. We begin with presenting relevant information about sets in Section 2.1 and linear algebra in Section 2.2. Section 2.3 provides further information about closed convex sets, their representation and general operations on sets. In Section 2.4 we discuss intervals, interval arithmetic and operations on intervals.

2.1 Sets

The *cardinality* $|A| \in \mathbb{N} \cup \{\infty\}$ of a set A is the number of elements in A. We call a set *empty* (notation $A = \emptyset$) if $|A| = 0$, *finite* if $|A| \in \mathbb{N}$, and *infinite* if $|A| = \infty$ holds. We typically use uppercase letters (e.g., I, S) for sets and lowercase letters for set elements (e.g., $s \in S$ or $i \in I$). This applies also to variables: lowercase letters denote variables and uppercase letters denote sets of variables (e.g., $x \in V$). We use the standard set operators $A \cap B$, $A \cup B$, $A \times B$ and predicates $A \subset B$, $A \subseteq B$ to represent *intersection*, *union*, *cartesianProduct*, *properContainedness* and *containedness* of two sets A and B. We use 2^A to denote the set of all subsets of A.

We use $\mathbb{R}$ to denote the set of all real numbers, $\mathbb{N}$ for the set of natural numbers including zero, and $\mathbb{Q}$ for the set of rational numbers. We may use a subscript to limit those sets, i.e., $\mathbb{R}_{>0}$ refers to the strictly positive real numbers. We use the usual arithmetic operators $+, -, \cdot, \div$ and relational operators $\leq, <, =, >, \geq$ on numbers with their standard semantics; we sometimes use standard simplifications in the notation, e.g., we write xy for $x \cdot y$. We use $\mathbb{R}^d$, $d \in \mathbb{N}_{>0}$, to refer to the d-dimensional (Euclidean) real vector space, which is the d-ary Cartesian product of $\mathbb{R}$, i.e., the set of all *d-tuples* $(a_0, \ldots, a_{d-1})$ of real numbers $a_i \in \mathbb{R}$, $i = 0, \ldots, d-1$; we use similar notations for other number sets. We use an underscore as a wildcard character to denote that an element of a tuple may take any value from its domain, i.e., the expression $A = (\_, \_, 1) \in \mathbb{R}^3$ denotes the set of all 3-tuples of real numbers where the third element equals one.

2.2 Vectors and Matrices

This section recapitulates basic concepts and notations from linear algebra used throughout this work. We assume the reader is familiar with linear algebra over the real numbers and refer to works such as [Art91] for further details. An element $x \in \mathbb{R}^d$ in the d-dimensional real space is called a vector.

Definition 2.1: Vector

A d-dimensional (real) vector $x \in \mathbb{R}^d$ ($d \in \mathbb{N}_{>0}$) is an ordered sequence of d real values $x_0, \ldots, x_{d-1} \in \mathbb{R}$ called *coordinates*. A column vector is ordered vertically

$$x = \begin{pmatrix} x_0 \\ \vdots \\ x_{d-1} \end{pmatrix} .$$

whereas a row vector is ordered horizontally $(x_0, \ldots, x_{d-1})$. *Transposition* x^T transformes a column vector x into a row vector of the same dimension with the same entries and vice versa, the transpose of a row vector is a column vector.

We use small letters for vectors and always assume vectors to be *column vectors* unless stated differently (in this case we refer to it as a *row vector*). In this work, we sometimes use the notion vector and *point* in the d-dimensional vector space $\mathbb{R}^d$ interchangeably. We use the notation $\|x\|_p$ ($p \in \mathbb{N}_{>0}$) to denote the p-norm for a vector $x = (x_0, \ldots, x_{d-1})$ following the classical definition

$$\|x\|_p = \left(\sum_{i=0}^{d-1} |x_i|^p \right)^{1/p} .$$

where $|\cdot|$ is the absolute value of the given argument. Arithmetic operations on vectors are defined in a component-wise manner.

Definition 2.2: Vector Arithmetic

The addition of two d-dimensional vectors $x, y \in \mathbb{R}^d$ is defined as follows:

$$x + y = (x_0 + y_0, \ldots, x_{d-1} + y_{d-1}) ,$$

multiplication of a vector $x \in \mathbb{R}^d$ with a scalar $\lambda \in \mathbb{R}$ is also defined component-wise as $\cdot : (\mathbb{R} \times \mathbb{R}^d) \to \mathbb{R}^d$ with

$$\lambda \cdot x = (\lambda \cdot x_0, \ldots, \lambda \cdot x_{d-1}) .$$

Vectors from $\mathbb{R}^d$ with vector addition and scalar multiplication build a vector space over $\mathbb{R}$. The zero vector $\mathbf{0} = (0, \ldots, 0) \in \mathbb{R}^d$, referred to as the *origin*, is the additive identity. The multiplicative identity is $1 \in \mathbb{R}$. For each vector x, the additive inverse is $(-x) = (-1) \cdot x$, such that $x + (-x) = \mathbf{0}$.

Apart from component-wise operations, the dot product between two vectors is of interest, as it defines a notion of angles between the two vectors.

Definition 2.3: Dot-product

The scalar product (sometimes dot product) of two d-dimensional vectors $x, y \in \mathbb{R}^d$ is defined as

$$\langle x, y \rangle = \sum_{i=0}^{d-1} x_i \cdot y_i \ .$$

Note that scalar product is not a group operation, as the result of the product of two vectors is a scalar. Two vectors $x, y \in \mathbb{R}^d$ are *orthogonal* to each other, if their scalar product equals zero:

$$x \perp y \Leftrightarrow \langle x, y \rangle = \sum_{i=0}^{d-1} x_i \cdot y_i = 0 \ .$$

We call the (column) vectors $x_0, \ldots, x_{n-1} \in \mathbb{R}^d$ ($n \in \mathbb{N}_{>1}$) *linearly dependent* if

$$\exists \lambda_0, \ldots, \lambda_{n-1} \in \mathbb{R}. \ \left(\bigvee_{i=0}^{n-1} \lambda_i \neq 0 \right) \wedge \sum_{i=0}^{n-1} \lambda_i \cdot x_i = \mathbf{0}$$

holds, otherwise x and y are called linearly independent. Row vectors are linearly dependent if their transpose are linearly dependent, and linearly independent otherwise.

A matrix is a collection of elements, rectangularly ordered in rows and columns. Here we consider real values as matrix elements.

Definition 2.4: Matrix

A (real-valued) matrix $\mathcal{A}$ of dimension $n \times m$ is a collection of $n \cdot m$ real numbers arranged in a rectangular array with n rows and m columns:

$$\mathcal{A} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,m-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,m-1} \end{pmatrix} \ .$$

The set of all matrices of dimension $n \times m$ is denoted by $\mathbb{R}^{n \times m}$. The matrix entry at row i and column j in matrix $\mathcal{A}$ is referenced by $a_{i,j}$, while we use $a_{i\_}$ to reference the i-th row and consequently $a_{\_j}$ to refer to the j-th column of $\mathcal{A}$.

A matrix $\mathcal{A} \in \mathbb{R}^{n \times m}$ can also be seen as a set of m column vectors in $\mathbb{R}^n$, or analogously as a set of n row vectors in $\mathbb{R}^m$. Consequently, an n-dimensional (column-)vector $x \in \mathbb{R}^n$ can be seen as a matrix with a single column, i.e., $x \in \mathbb{R}^{n \times 1}$. A matrix $\mathcal{A} \in \mathbb{R}^{n \times m}$ is called *quadratic* or *square*, if $n = m$ holds. The *transpose* of a matrix $\mathcal{A} \in \mathbb{R}^{n \times m}$ is the matrix $\mathcal{A}^T = B \in \mathbb{R}^{m \times n}$ with $b_{i,j} = a_{j,i}$.

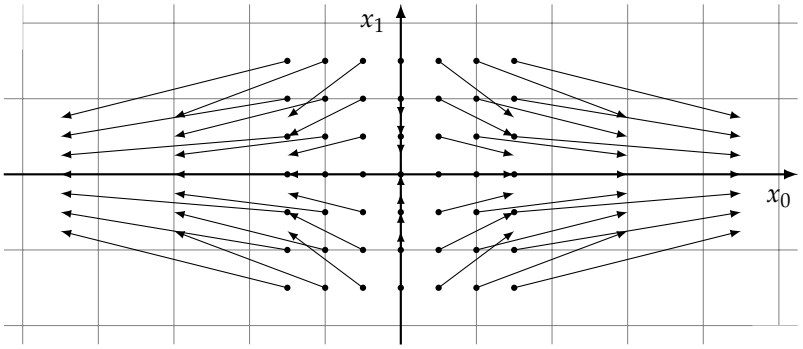


Figure 2.1: A vector field in R^2 induced by the mapping $\mathcal{A}x = (2x_0, -0.5x_1)$.

Arithmetic operations on matrices are used with the usual semantics, i.e., matrix addition $+ : (\mathbb{R}^{n\times m} \times \mathbb{R}^{n\times m}) \to \mathbb{R}^{n\times m}$ and multiplication $\cdot : (\mathbb{R}^{n\times p} \times \mathbb{R}^{p\times m}) \to \mathbb{R}^{n\times m}$ are defined as $\mathcal{A} + \mathcal{B} = \mathcal{C}$ and $\mathcal{D} \cdot \mathcal{E} = \mathcal{F}$ with

$$c_{i,j} = a_{i,j} + b_{i,j} \qquad \textit{and} \qquad f_{i,j} = \langle d^T_{i\_}, e_{\_j} \rangle$$

for all $n, m, p \in \mathbb{N}_{>0}$, $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{F} \in \mathbb{R}^{n\times m}$, $\mathcal{D} \in \mathbb{R}^{n\times p}$, $\mathcal{E} \in \mathbb{R}^{p\times m}$, $i = 0, \ldots, n-1$ and $j = 0, \ldots, m-1$. Furthermore, multiplication of a matrix with a scalar is defined as $\lambda \cdot \mathcal{A} = \mathcal{A} \cdot \lambda = \mathcal{B}$ with

$$b_{i,j} = \lambda \cdot a_{i,j}$$

for all $n, m \in \mathbb{N}_{>0}$, $\mathcal{A} \in \mathbb{R}^{n\times m}$, $\lambda \in \mathbb{R}$, $i = 0, \ldots, n-1$ and $j = 0, \ldots, m-1$.

Matrices from $\mathbb{R}^{n\times m}$ with matrix addition and scalar multiplication build a vector space over $\mathbb{R}$. The matrix additive identity is the *zero matrix* $\mathbf{0} \in \mathbb{R}^{n\times m}$ whose entries are all zero. Additive inverse of $\mathcal{A}$ is $(-1) \cdot \mathcal{A}$. The scalar multiplicative identity is $1 \in \mathbb{R}$. Furthermore, for square matrices, additionally considering matrix multiplication yields an associative algebra; the matrix multiplicative identity is the *identity matrix* $Id \in \mathbb{R}^{d\times d}$ with diagonal entries $a_{i,i} = 1$, $i = 0, \ldots, d-1$, and all other entries being 0.

The *rank* of a matrix $\mathcal{A}$, written $\mathsf{rank}\,\mathcal{A}$, refers to the dimension spanned by its column vectors, which is equivalent to the maximal number of linearly independent columns in $\mathcal{A}$. This is equivalent to the dimension spanned by its row vectors, i.e., the maximal number of linearly independent rows in $\mathcal{A}$. A matrix $\mathcal{A} \in \mathbb{R}^{n\times m}$ is said to be of *full rank* if $\mathsf{rank}\,\mathcal{A} = \mathsf{min}(n, m)$ holds.

A square matrix $\mathcal{A} \in \mathbb{R}^{d\times d}$ is called *invertible* if it has a multiplicative *inverse* $\mathcal{A}^{-1}$ with $\mathcal{A} \cdot \mathcal{A}^{-1} = \mathcal{A}^{-1} \cdot \mathcal{A} = Id$. This is the case if and only if $\mathsf{rank}\,\mathcal{A} = d$. Matrices are referred to as being *non-singular* or *non-degenerate* if they are invertible and consequently as *singular* or *degenerate* if they cannot be inverted.

For vectors $x \in \mathbb{R}^d$, we interpret (left) multiplication by a square matrix $\mathcal{A} \in \mathbb{R}^{d\times d}$ as a *linear transformation* of x by $\mathcal{A}$:

$$\mathcal{A} \cdot x = \left(\langle a^T_{0\_}, x \rangle, \ldots, \langle a^T_{d\_}, x \rangle \right)^T .$$

Using this intuition, later we will interpret a square matrix $\mathcal{A} \in \mathbb{R}^{d\times d}$ as a linear mapping $\mathcal{A} : \mathbb{R}^d \to \mathbb{R}^d$ that assigns to each $x \in \mathbb{R}^d$ its translate $\mathcal{A} \cdot x \in \mathbb{R}^d$ (see Figure 2.1) to describe the dynamic behavior of certain hybrid system models.

We can use linear mappings $\mathcal{A} : \mathbb{R}^m \to \mathbb{R}^n$ also for projections. In the scope of this work, we will restrict ourselves to projections along coordinate axes. For general projections on subspaces of $\mathbb{R}^d$, we refer to [Zie95].

Definition 2.5: Projection on Orthant Planes

Let $S \subseteq \mathbb{R}^d$ and $I \subseteq \{0,\dots,d-1\}$. Let $f : |I| \to I$ be a bijective monotone function. The *projection* $S\downarrow_I$ of S to I is the set

$$\left\{(y_0,\dots,y_{|I|-1}) \;\middle|\; (x_0,\dots,x_{d-1}) \in S \wedge (\forall i \in I. y_{f(i)} = x_i)\right\} .$$

We can use a linear mapping $P : \mathbb{R}^d \to \mathbb{R}^{|I|}$ to project a set to I, using the matrix $\mathcal{A} \in \mathbb{R}^{|I|\times d}$ with entries

$$a_{ij} = \begin{cases} 1 & j \in I \wedge i = f(j) \\ 0 & \text{otherwise.} \end{cases} .$$

Example 2.1: Projection

For projecting subsets of $\mathbb{R}^3$ to $I = \{0,2\}$, the projection matrix is defined as follows:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} .$$

Applying this projection to a point $(4,5,6)^T \in \mathbb{R}^3$ yields:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 4 \\ 6 \end{pmatrix} .$$

Linear Real Arithmetic

As we use Cartesian coordinates to represent elements in $\mathbb{R}^d$, we can use linear equations and linear inequations to describe partitions of $\mathbb{R}^d$.

Definition 2.6: Linear Polynomial

A linear polynomial p over a finite ordered set of variables $Var = \{x_0,\dots,x_{d-1}\}$ is defined as

$$a_d + \sum_{i=0}^{d-1} a_i \cdot x_i,$$

$a_0,\dots,a_d \in \mathbb{R}$. We define $Var(p) = \{x_i \in Var \mid a_i \neq 0\}$ and call p *univariate* if $|Var(p)| = 1$ holds.

Using this definition we can formalize linear constraints as a comparison of a linear polynomial to a constant.

Definition 2.7: Linear Constraint

A linear real-arithmetic *constraint* c over a finite ordered set of variables Var has the (normal) form

$$c : p \sim 0$$

where p is a linear polynomial over Var and $\sim \in \{<, \leq, =, >, \geq\}$ is a comparison predicate. The constraint c is called *univariate* if p is univariate.

Besides the above *normal forms* (e.g., $2 + 2x > 0$) for linear polynomials and constraints, we will also use equivalent representations that we gain by standard transformations (e.g., $3x - x > -2$); note that all linear polynomials resp. constraints can be easily brought to the above normal forms. The *solution set* of a linear constraint $c : p \sim 0$ over $Var = \{x_0, \ldots, x_{d-1}\}$ is defined as $Sat_c = \{m \in \mathbb{R}^d \mid m \models c\}$; it contains all variable assignments m that satisfy c, i.e., that evaluate c to true when we substitute m_i for x_i for $i = 0, \ldots, d-1$ (see Example 2.2).

Example 2.2: Solution Set

Let $p : x_0 - x_1$ be a linear polynomial over variables $Var = \{x_0, x_1\}$. The solution set Sat_c for $c : p = 0$ contains all models $m \in \mathbb{R}^2$ with assignments for x_0, x_1 where $x_0 = x_1$ holds, i.e.,

$$Sat_c = \left\{ (m_0, m_1) \in \mathbb{R}^2 \;\middle|\; m_0 = m_1 \right\}$$

such that the set of points in Sat_c defines a line in $\mathbb{R}^2$. Changing the relation to $c' : p \leq 0$ yields

$$Sat_{c'} = \left\{ (m_0, m_1) \in \mathbb{R}^2 \;\middle|\; m_0 \leq m_1 \right\}$$

which defines a half-space in $\mathbb{R}^2$ as defined below.

Definition 2.8: Half-space

A d-dimensional *half-space* h is a set

$$h = \left\{ x \in \mathbb{R}^d \;\middle|\; n^T \cdot x \leq c \right\}$$

for some $n \in \mathbb{R}^d$ and $c \in \mathbb{R}$. We refer to n as the *normal vector* of h and to c as the *offset* of h. The set $\bar{h} = \{x \in \mathbb{R}^d \mid n^T \cdot x = c\}$ is called the *bounding hyperplane* of the half-space h.

We can represent a half-space h as a tuple (n, c), where $n \in \mathbb{R}^d$ and $c \in \mathbb{R}$, which represents the set $\{x \in \mathbb{R}^d \mid n^T \cdot x \leq c\}$.

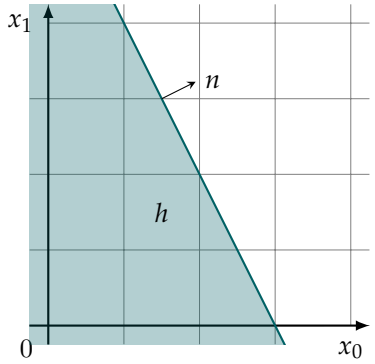


Figure 2.2: Geometric representation of a half-space h (petrol) in $\mathbb{R}^2$. The bounding hyperplane is given by the equation $2x_0 + x_1 = 1.5$.

Variable Elimination

For a Boolean conjunction $\varphi = \bigwedge_{j=0}^{n-1} c_j$ of linear constraints over variables $Var = \{x_0, \ldots, x_{d-1}\}$, variable elimination techniques such as Fourier-Motzkin variable elimination [Fou27; Mot36] and Gaussian elimination for equational constraints can be applied to eliminate a variable x_k and obtain a *satisfiability*-equivalent formula $\varphi' = \bigwedge_{j=0}^{m-1} c'_j$ over the remaining variables $Var = \{x_0, \ldots, x_{k-1}, x_{k+1}, \ldots, x_{d-1}\}$.

Gaussian Variable Elimination. It can be used to eliminate a variable $x_k \in Var$ from $\bigwedge_{j=0}^{n-1} c_j$ if there is an $l \in \{0, \ldots, n-1\}$ such that

$$c_l : a_{l,d} + \sum_{i=0}^{d-1} a_{l,i} x_i = 0$$

with $a_{l,k} \neq 0$. The idea is to transform c_l to the form $x_k = p'_l$ with

$$p'_l = \frac{-a_{l,d}}{a_{l,k}} + \sum_{i=0,\ldots,k-1,k+1,\ldots,d-1} \frac{-a_{l,i}}{a_{l,k}} x_i,$$

replace in each constraint c_j the variable x_k by p'_l yielding c'_j, and return $\bigwedge_{j=0}^{n-1} c'_j$.

Fourier-Motzkin Variable Elimination. *Fourier-Motzkin variable elimination* can be used to eliminate a variable x_k from a conjunction of linear constraints over variables Var, and achieve a *satisfiability*-equivalent conjunction of linear constraints over $Var \setminus \{x_k\}$. This method can also be applied to equational constraints; however, using Gaussian elimination for a system of linear equalities is more efficient, which is why in the following we only consider systems of inequalities. Assume a Boolean conjunction $\varphi = \bigwedge_{j=0}^{n-1} p_j \sim_j 0$, $\sim_j \in \{<, \leq, \geq, >\}$ of linear inequalities over the variable set $Var = \{x_0, \ldots, x_{d-1}\}$.

Let I be the set of all incides $j \in \{0,\dots,n-1\}$ with $x_k \notin Var(p_j)$. For all other indices $j \in \{0,\dots,n-1\} \setminus I$, we transform the constraint

$$c_j : a_{j,d} + \sum_{i=0}^{d-1} a_{j,i} x_i \sim_j 0$$

with $a_{j,k} \neq 0$ to the form $x_k \sim'_j p'_j$ with

$$p'_j = \frac{-a_{j,d}}{a_{j,k}} + \sum_{i=0, i\neq k}^{d-1} \frac{-a_{j,i}}{a_{j,k}} x_i,$$

and with $\sim'_j$ being $\sim_j$ for $a_{j,k} > 0$ and otherwise $\sim'_j$ is $\geq, >, <, \leq$ for $\sim_j$ being $\leq, <, >, \geq$ respectively. Let I_l contain those such indices l for which $\sim'_l \in \{\geq, >\}$, I_u those indices u with $\sim'_u \in \{\leq, <\}$, and let $\sim_{l,u}$ be $\leq$ if $\sim'_l, \sim'_u \in \{\leq, \geq\}$ and $<$ otherwise. The resulting formula is $\varphi' = (\wedge_{j\in I} c_j) \wedge (\wedge_{l\in I_l} \wedge_{u\in I_u} p'_l \sim_{l,u} p'_u)$.

Example 2.3: Fourier-Motzkin Variable Elimination

Consider the formula

$$\begin{aligned}\varphi = x + \frac{5}{2}y \leq \frac{23}{2} \wedge -x + \frac{1}{4}y \leq -\frac{1}{2} \wedge \\ x - \frac{1}{2}y \leq \frac{5}{2} \wedge -x - 2y \leq -5.\end{aligned}$$

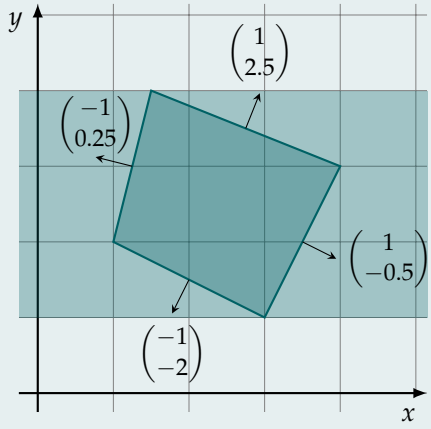
To eliminate x, we rearrange the constraints and identify lower (underlined) and upper (overlined) bounds for x:

$$\begin{aligned}\Leftrightarrow \overline{x \leq -\frac{5}{2}y + \frac{23}{2}} \wedge \underline{\frac{1}{4}y + \frac{1}{2} \leq x} \wedge \\ \overline{x \leq \frac{1}{2}y + \frac{5}{2}} \wedge \underline{-2y + 5 \leq x}\ .\end{aligned}$$

After combining lower and upper bounds for x, we obtain a satisfiability-equivalent formula only in y:

$$\begin{aligned}&\Leftrightarrow \frac{1}{4}y + \frac{1}{2} \leq -\frac{5}{2}y + \frac{23}{2} \wedge -2y + 5 \leq -\frac{5}{2}y + \frac{23}{2} \wedge \\ &\quad \frac{1}{4}y + \frac{1}{2} \leq \frac{1}{2}y + \frac{5}{2} \wedge -2y + 5 \leq \frac{1}{2}y + \frac{5}{2} \\ &\Leftrightarrow 1 \leq y \wedge y \leq 4\ .\end{aligned}$$

The graphical representation of the original constraint set and the result of the variable elimination is shown below.



Intuitively, the elimination of x in this example corresponds to a projection of the set specified by φ along the x-axis (see Definition 2.5).

As Fourier-Motzkin variable elimination combines lower and upper bounds to create new constraints, the resulting formula φ' may contain at most n^2 constraints where n denotes the number of constraints in the original formula φ. To improve this behavior, heuristics exist which allow to discard some combinations of lower and upper bounds a priori [Imb90]. Furthermore, in the case of φ containing equations in x_k, Gaussian elimination should be favored, as it does not increase the number of constraints. Consequently, equations of the form $p = x_k$ should not be replaced by two inequations $p \leq x_k \wedge p \geq x_k$ during pre-processing of the formula.

2.3 Geometric Sets

In this section, we will introduce basic concepts of convex sets in $\mathbb{R}^d$ and their representation required for the remainder of this dissertation. In the previous section, we have already introduced the notion of a *point* in $\mathbb{R}^d$ which represents the 0-dimensional *affine subspace* in $\mathbb{R}^d$. In geometry, concepts such as points, *lines*, *planes* and *hyperplanes* play an important role—they represent affine subspaces of dimension 0, 1, 2, and $d-1$ of the vector space $\mathbb{R}^d$, and are sometimes also referred to as *flats*. Thus, non-empty affine subspaces are the translates of linear vector spaces [Zie95]. The dimension of an affine subspace (sometimes *affine dimension*) is the dimension of the corresponding vector space. We use the operator $\dim A$ to refer to the affine dimension of A. As indicated in the previous section, flats are the geometric representation of solution sets *Sat* of systems of linear equations.

Convex Sets

This section is based on [Zie95] and provides basic information about convex sets, which will be needed to understand the concepts of convex polytopes.

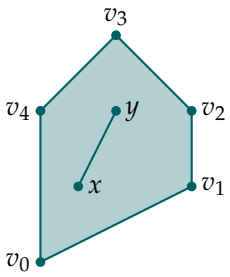


Figure 2.3: A compact convex set represented as the convex hull of its vertices $v_0, \ldots, v_4$.

A set $S \subseteq \mathbb{R}^d$ is called *convex*, if for any two points $x, y \in S$ all points of the line segment $\overline{xy}$ between x and y are also in S, i.e., if

$$\forall x, y \in S.\ \forall \lambda \in [0,1] \subseteq \mathbb{R}.\ (\lambda x + (1-\lambda)y) \in S\ . \tag{2.1}$$

A set $S \subseteq \mathbb{R}^d$ is *open* if

$$\forall x \in S.\ \exists \epsilon \in \mathbb{R}_{>0}.\ \forall y \in \mathbb{R}^d.\ \|x-y\|_2 \leq \epsilon \Rightarrow y \in S\ .$$

The set $S \subseteq \mathbb{R}^d$ is *closed* if its complement $\mathbb{R}^d \setminus S$ is open. A set $S \subseteq \mathbb{R}^d$ is *bounded* if

$$\exists \epsilon \in \mathbb{R}_{>0}.\ \forall x, y \in S.\ \|x-y\|_2 \leq \epsilon\ .$$

A set $S \subseteq \mathbb{R}^d$ is *compact* if it is closed and bounded.

The *convex hull* of a set $V \subseteq \mathbb{R}^d$ of d-dimensional points is defined as $cHull(V) = \emptyset$ if $V = \emptyset$ and

$$cHull(V) = \left\{ \sum_{i=0}^{n-1} \lambda_i v_i \;\middle|\; n \in \mathbb{N}_{>0} \wedge \left(\bigwedge_{i=0}^{n-1} v_i \in V \right) \wedge \left(\bigwedge_{i=0}^{n-1} \lambda_i \in \mathbb{R} \wedge 0 \leq \lambda_i \leq 1 \right) \wedge \sum_{i=0}^{n-1} \lambda_i = 1 \right\}$$

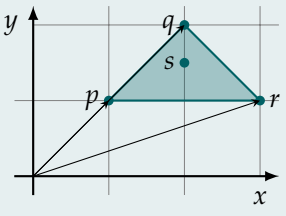
otherwise (see Figure 2.3 for an example).

Intuitively, each point s in a compact convex set S can be represented as the convex combination of the generating points of S. We refer to this representation of the set as the $\mathcal{V}$-representation and call the generating points *vertices* (see Definition 2.9).

Example 2.4: Convex Combination

Consider the triangle defined by the three points

$$p = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, q = \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \text{ and } r = \begin{pmatrix} 3 \\ 1 \end{pmatrix}\ .$$



The point $s = (2, 1.5)^T$ can be represented as the convex combination

$$\begin{pmatrix} 2 \\ 1.5 \end{pmatrix} = \lambda_0 \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \lambda_1 \begin{pmatrix} 2 \\ 2 \end{pmatrix} + \lambda_2 \begin{pmatrix} 3 \\ 1 \end{pmatrix},$$

where $\lambda_0 = \frac{1}{4}, \lambda_1 = \frac{1}{2}$, and $\lambda_2 = \frac{1}{4}$.

We use Equation (2.1) to define vertices/extreme points of a convex set.

Definition 2.9: Vertex

A point v of a convex set S is a *vertex* of S, often also called *extreme point*, if for all pair of points $x, y \in S$ and all $0 < \lambda < 1$ we have that

$$v \neq \lambda x + (1 - \lambda) y$$

holds. We write $\mathit{vertices}(S)$ to denote the set of vertices of S.

More general, a vertex of a convex set S is a 0-dimensional face of S. A *face* of a compact convex set S and a given $c \in \mathbb{R}^d$ is the set

$$\mathit{face}(S, c) = \{x \in S \mid c_0 \in \mathbb{R} \wedge (\forall y \in S.\ cy \leq c_0) \wedge cx = c_0\}$$

A *face* of S is a set $F \subseteq \mathbb{R}^d$ for which there exists a $c \in \mathbb{R}^d$ such that $F = \mathit{face}(S, c)$. The *normal vectors* of a face F of S is the set of all $c \in \mathbb{R}^d$ for which $F = \mathit{face}(S, c)$.

The faces of dimension 0, 1, $d-2$, $d-1$ of a compact convex set $P \subseteq \mathbb{R}^d$ are called vertices, *edges*, *ridges*, and *facets*. Using Definition 2.9, a compact set S can be represented by the *convex hull* of its vertices

$$S = \mathit{cHull}\,\{v \mid v \in \mathit{vertices}(S)\}\ .$$

Note that we need at least $d+1$ vertices to define a set S in $\mathbb{R}^d$ where $\dim S = d$. We refer to the set spanned by exactly $d+1$ vertices as a *simplex* in $\mathbb{R}^d$.

A property of unbounded convex sets in the Euclidean space is that they contain a d-dimensional ray $r = \{t \cdot x \mid t \in \mathbb{R}_{\geq 0}\}$ for a fixed $x \in \mathbb{R}^d$. Note that $\{\mathbf{0}\}$ is the only finite ray. A set composed from rays is referred to as a cone.

Definition 2.10: Cone

A set $C \subseteq \mathbb{R}^d$ is a *d-dimensional* (convex) *cone* if $C \neq \emptyset$ and

$$\forall x, y \in C.\ \forall \lambda_x, \lambda_y \in \mathbb{R}_{\geq 0}.\ \lambda_x x + \lambda_y y \in C\ .$$

The *conical hull* of a set $V \subseteq \mathbb{R}^d$ is $\mathit{cone}(V) = \{\mathbf{0}\}$ if $V = \emptyset$ and

$$\mathit{cone}(V) = \left\{ \sum_{i=0}^{n-1} \lambda_i x_i \;\middle|\; n \in \mathbb{N}_{>0} \wedge \left(\bigwedge_{i=0}^{n-1} x_i \in V \wedge \lambda_i \in \mathbb{R}_{\geq 0} \right) \right\}$$

otherwise.

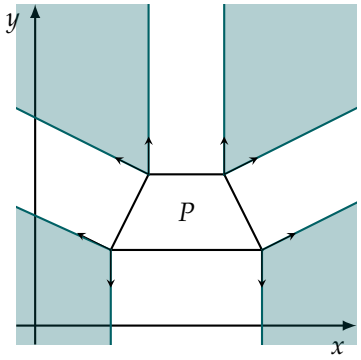


Figure 2.4: Normal fan (light petrol) of a compact convex set P.

Note that $\{\mathbf{0}\}$ is the only finite convex cone. Note furthermore that conical hulls are convex cones, thus $\{\mathbf{0}\}$ is also the only finite conical hull.

Any (potentially unbounded) closed convex set S thus can be represented as the Minkowski sum (see Definition 3.7) of the convex hull of a set of points V and a convex cone C:

$$S = cHull(V) \oplus C \ .$$

Following [Zie95], a set $\mathcal{F}$ of d-dimensional cones is called a *d-dimensional fan* if it has the following properties:

- Every non-empty face of a cone in $\mathcal{F}$ is also a cone in $\mathcal{F}$.
- The intersection of any two cones in $\mathcal{F}$ is a face of both.

In this work, we only need to consider *complete* fans—a d-dimensional fan $\mathcal{F}$ is complete if the union of all cones in $\mathcal{F}$ is $\mathbb{R}^d$. In the context of linear programming, we are interested in the *normal fan*.

Definition 2.11: Normal Fan

Let $S \subseteq \mathbb{R}^d$ be a compact convex set with faces $\mathcal{F}$. The *normal fan* $\mathcal{N}(S)$ of S is the set

$$\mathcal{N}(S) = \left\{ \left\{ x \in \mathbb{R}^d \;\middle|\; F \subseteq face(S, x) \right\} \;\middle|\; F \in \mathcal{F} \right\}$$

Intuitively, the normal fan of S is the set of cones, each of which contain the normal vectors of a face of S (see Figure 2.4). The solution to the linear program $\max_{x \in S} c \cdot x$ for given compact convex $S \subseteq \mathbb{R}^d$ and $c \in \mathbb{R}^d$ answers the question "Which cone of $\mathcal{N}(S)$ does c lie in?" [Zie95]. We will come back to the normal fan in combination with linear programming (LP) when computing the support of a box (see Section 6.2).

Support of a Set. For a given set $S \subseteq \mathbb{R}^d$, its support $\rho_S : \mathbb{R}^d \to (\mathbb{R} \cup \{\infty\})$ is a function which maps to each direction $l \in \mathbb{R}^d$ a scalar $\rho_S(l) \in (\mathbb{R} \cup \{\infty\})$ such that

$$\rho_S(l) = \sup_{s \in S} \langle s, l \rangle$$

holds. For a closed set ρ_S computes for each direction $l \in \mathbb{R}^d$ the smallest offset for a half-space with normal vector l which fully contains S.

Polyhedra and Polytopes

Later, in Section 6.3 we will define and use polytopes for the representation of state sets. Here we would like to point to a duality regarding the definition of a more general type of sets called polyhedra.

On the one hand, a polyhedron $P \subseteq \mathbb{R}^d$ can be defined to consist of the common solutions of finitely many linear constraints in d variables. On the other hand, bounded polyhedra, which we call polytopes, can be defined as the convex hull of finitely many points, and polyhedra as the Minkowski sum of such a convex hull and a cone. Both definitions are equivalent in the sense that they cover the same sets. Note that while this equivalency seems intuitive, it is non-trivial to formally prove. For the interested reader, we refer to [Zie95].

These two views, the arithmetic view from Section 2.2 and the geometric prespective from Section 2.3, will be reflected by two different datatypes for polytopes, one based on the defining linear constraints and one based on generating points of the convex hull.

2.4 Intervals and Interval Arithmetic

Intervals provide a convenient way to represent sets of values and are used in many applications. In general, an interval I defines a bounded range of values, i.e., the set of all values between the lower bound of I and its upper bound. Intervals may be defined over various domains; in this work however, only intervals over real or rational numbers are considered. For a comprehensive introduction of intervals and interval arithmetic, we refer to [MKC09] and only present selected parts of this work required for the comprehension of this thesis. Our implementation of intervals for HyPro is based on an implementation of interval arithmetic in the context of an ICP-based approach for satisfiability modulo theories (SMT)-solving [Sch13] and is integrated into the Computer Arithmetic Library (CArL)[1].

Definition 2.12: Interval

An *interval* I is defined by a *lower bound* $\underline{I} \in \mathbb{R} \cup \{-\infty\}$, an *upper bound* $\overline{I} \in \mathbb{R} \cup \{\infty\}$ and two relation symbols $\sim_l, \sim_u \in \{<, \leq\}$ as the set

$$I = \left\{ x \in \mathbb{R} \;\middle|\; \underline{I} \sim_l x \wedge x \sim_u \overline{I} \right\},$$

where we require that $\sim_l$ ($\sim_u$) is $<$ if $\underline{I} = -\infty$ ($\overline{I} = \infty$).

[1] `github.com/smtrat/carl` (checked July, 30th, 2019)

We use $\mathbb{I}$ for the set of all real-valued intervals. Bound types are represented by either square or round brackets, where a square bracket represents a weak ($\leq$) and a round bracket represents a strict ($<$) bound. Combinations of both are allowed. Intervals with only strict bounds are called *open*, with only weak bounds *closed*, otherwise *half-open* and *half-closed*. An interval is unbounded if at least one of the bounds is infinite ($\pm\infty$). The *width* of an interval I is defined as $|\overline{I} - \underline{I}|$. Note that the width of an unbounded interval is ∞, the width of a *point-interval* is zero, and the width of an empty interval is non-positive. In this work, we will mostly deal with *closed and bounded* intervals, and restrict the following descriptions to them.

Inclusion and *intersection* for non-empty intervals A, B can be computed easily by using

$$\begin{aligned} A \subseteq B &\Leftrightarrow \underline{B} \leq \underline{A} \wedge \overline{A} \leq \overline{B} \\ A \cap B &= [\max(\underline{A}, \underline{B}), \min(\overline{A}, \overline{B})] \ . \end{aligned}$$

Intervals are not closed under *union*, therefore we consider the convex hull of their union as the smallest closure. Thus for non-empty intervals A, B we have:

$$cl(A \cup B) = [\min(\underline{A}, \underline{B}), \max(\overline{A}, \overline{B})]$$

Arithmetic over the reals can be extended to interval arithmetic as follows (again, restricted to closed and bounded intervals).

Definition 2.13: Interval Arithmetic

For non-empty intervals A, B and scalars $c \in \mathbb{R}$ we define:

$$\begin{aligned} A + B &= [\underline{A} + \underline{B}, \overline{A} + \overline{B}] \\ A - B &= [\underline{A} - \overline{B}, \overline{A} - \underline{B}] \\ A \cdot B &= [\min\{\underline{A} \cdot \underline{B}, \underline{A} \cdot \overline{B}, \overline{A} \cdot \underline{B}, \overline{A} \cdot \overline{B}\}, \max\{\underline{A} \cdot \underline{B}, \underline{A} \cdot \overline{B}, \overline{A} \cdot \underline{B}, \overline{A} \cdot \overline{B}\}] \\ c \cdot A &= \begin{cases} [c \cdot \underline{A}, c \cdot \overline{A}] & \text{if } c \in \mathbb{R}_{\geq 0} \\ [c \cdot \overline{A}, c \cdot \underline{A}] & \text{otherwise} \end{cases} \end{aligned}$$

For empty intervals, the operations are straightforward.

Note that multiplication can be implemented more efficiently as a case distinction in which eight of nine cases only require to compute two products [MKC09]. For further arithmetic operations like interval division and predicates like comparison, which are more involved and not required for the general comprehension of this work, we refer to [MKC09; Rat96].

Example 2.5: Interval Arithmetic

Addition:

$$[2,3] + [1,4] = [3,7]$$

Multiplication:

$$[2,3] \cdot [-2,4] = [-6,12]$$

Subtraction:

$$[2,3] - [1,4] = [-2,2]$$

Scalar multiplication:

$$[2,3] \cdot \frac{-1}{4} = \left[\frac{-3}{4}, \frac{-2}{4}\right]$$

Interval arithmetic is defined such that for all intervals A, B, elements $a \in A$ and $b \in B$ and arithmetic operations $\odot \in \{+, -, \cdot, \div\}$ it holds that $a \odot b \in A \odot B$. This naturally implies *inclusion isotonicity*, i.e., $A \odot B \subseteq C \odot D$ for all intervals $A \subseteq C$ and $B \subseteq D$.

Let $e[\alpha_0, \ldots, \alpha_{n-1} / \beta_0, \ldots, \beta_{n-1}]$ denote the simultaneous syntactical replacement of each α_i by β_i in e. We talk about *interval-valued variables* if we allow variables $x_0, \ldots, x_{d-1}$ to take any values from some interval domains $I_0, \ldots, I_{d-1}$. Assume a linear polynomial p over the variable set $\{x_0, \ldots, x_{d-1}\}$ in normal form. It is easy to see that all possible evaluations of p under the given variable intervals form the interval that we get when we replace in p each variable x_i by its interval I_i and compute the result using interval arithmetic:

$$\left\{ p[x_0, \ldots, x_{d-1} / v_0, \ldots, v_{d-1}] \;\middle|\; \bigwedge_{i=0}^{d-1} v_i \in I_i \right\} = p[x_0, \ldots, x_{d-1} / I_0, \ldots, I_{d-1}] \,.$$

This property is not assured, e.g., if p is not in normal form or if p is not linear, but for all arithmetic expressions p we can state the over-approximative property:

$$\left\{ p[x_0, \ldots, x_{d-1} / v_0, \ldots, v_{d-1}] \;\middle|\; \bigwedge_{i=0}^{d-1} v_i \in I_i \right\} \subseteq p[x_0, \ldots, x_{d-1} / I_0, \ldots, I_{d-1}] \,.$$

Consequently, vectors and matrices can be extended to interval-valued vectors and matrices. Operations such as matrix-matrix and matrix-vector multiplication are defined analogous using interval-arithmetic operations; for a real-valued matrix $\mathcal{A} \in \mathbb{R}^{d \times d}$ and a vector $I = (I_0, \ldots, I_{d-1})^T$ of intervals $B_i \in \mathbb{I}$ we define

$$\mathcal{A} \cdot I = \begin{pmatrix} a_{0\_} \cdot I \\ \vdots \\ a_{d-1\_} \cdot I \end{pmatrix}$$

where $a_{r\_} \cdot I = \mathcal{A}_{r,0} \cdot I_0 + \cdots + \mathcal{A}_{r,d-1} \cdot I_{d-1}$.

We illustrate this with an example:

Example 2.6: Interval-valued matrices and vectors

Matrix-vector multiplication of a real-valued matrix $\mathcal{A} \in \mathbb{R}^{2\times 2}$ and an interval-valued vector $x \in \mathbb{I}^2$:

$$\begin{pmatrix} 2 & 3 \\ 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} [5,6] \\ [0,2] \end{pmatrix} = \begin{pmatrix} 2 \cdot [5,6] + 3 \cdot [0,2] \\ 1 \cdot [5,6] + 4 \cdot [0,2] \end{pmatrix} = \begin{pmatrix} [10,18] \\ [5,14] \end{pmatrix} .$$

3

Formal Verification of Hybrid Systems

3.1 Hybrid Systems

In Computer Science we often consider *discrete* systems whose state changes can be seen as instantaneous. Typical examples are program executions: though they are physical and thus continuous processes, we can often model their executions as a sequence of atomic, discrete steps. However, if we are interested in the duration of the computations or the behavior of a dynamic environment that is influenced by the computations, we need to extend our discrete view to include also continuously changing quantities.

Systems from mechanical or electrical engineering, or physical systems in general exhibit a continuous behavior in which quantities evolve continuously over time (referred to as *dynamic systems*). Over the years, engineers have developed various approaches to analyzing dynamic systems with regard to their needs. Techniques and theoretical approaches towards stability analysis of controllers have been developed and are still of interest.

Hybrid systems in computer science are systems with mixed discrete-continuous behavior which unify both notions in one system—the discrete (switching) behavior of a digital system in combination with the continuous nature of a dynamical system. Apart from physical processes that are inherently hybrid, e.g., the bouncing of a ball from the ground, hybrid systems also comprise systems in which a digital controller interacts with a continuous environment. Examples include automotive or aviation systems, e.g., the model of a cruise control or a vehicle platoon, systems from electrical engineering, e.g., models of DC-DC converters, systems from biology, e.g., the model of a pacemaker for the human heart. Additionally, academia has come up with many artificial systems, e.g., a simplified model of a reactor cooling system [NOS+92], a model of a point mass moving on a plane with varying dynamics or a model of moving heaters in a fixed room-setup [FI04]. Combining both a model for a plant as well as a model for a digital controller into one model results in more expressive models and allows for the design and analysis of closed-loop models.

In general, hybrid systems are considered a subclass of cyber-physical systems (CPSs) as they incorporate both digital as well as physical components. As CPS are often safety-critical, much effort has been put into the safety-verification

of hybrid systems by means of reachability analysis. The reachability problem for a hybrid system $\mathcal{H}$ tries to determine whether a specific set of states P_{bad} is reachable in $\mathcal{H}$ via an execution starting from a set of initial states *Init* (see Section 3.3).

3.2 Hybrid Automata

As previously mentioned, the behavior of hybrid systems exhibits both discrete and continuous components. For digitally controlled physical systems, the discrete part of the state space is usually determined by the states/modes of the controller. While the controller is in a given state, the values of the observed physical quantities such as temperature, pressure, etc. evolve continuously over time according to the dynamics defined for that state.

To be able to make statements about specific properties of a given system, formal methods require an abstraction of the given system in the form of a model that holds all relevant information required but abstracts away unnecessary details. Among other representations, e.g., hybrid Petri nets [DA01], *hybrid automata* [Hen96] have become a popular modeling paradigm for mixed discrete-continuous systems.

Hybrid automata can be seen as extensions of discrete transition systems, which consist of a set of locations *Loc* (*control modes*), a set of *variables Var*, and a set of guarded *discrete transitions Edge* (*jumps*) between locations. The current location ℓ together with the current variable valuation ν specifies the current *state* of the system. Discrete state changes can happen if the enabling condition of a jump (*guard*) is satisfied by the current state, i.e., if the current variable assignment in ℓ satisfies the guard predicate. Upon taking a jump, apart from changing the location, the variable valuation can also be updated to new values according to the *reset* for this jump.

Hybrid automata extend these models with continuous dynamic behavior. While the control stays in a location ℓ, *time transitions* (*flows*) let the values of the variables evolve continuously according to the *dynamics* in ℓ, which can for instance be specified by ordinary differential equations (ODEs). As long as the control stays in a particular mode, the variable valuations ν need to satisfy the *invariant* of that mode. Simultaneously enabled time passage and jumps, or several simultaneously enabled jumps might introduce non-determinism. *Urgent* jumps might reduce this non-determinism by forcing a discrete step if any urgent jump is enabled.

For a set X of real-valued variables, let in the following $Pred_X$ denote the set of all *predicates over* X, which are quantifier-free arithmetic formulas with variables from X. We do not fix the functions in the arithmetic theory, such that the general definition allows besides addition and multiplication e.g., trigonometric functions in predicates. Later we will consider models where all predicates are conjunctions of linear real arithmetic constraints, comparing linear expressions to constants. We will also use a specific fragment $Pred^{ass}_{X_1,X_2}$ for *assignment predicates*, which are predicates that are conjunctions of constraints of the form $x \sim e$, where $x \in X_2$, e is an expression over variables from X_1, and $\sim \in \{<, \leq, =, \geq, >\}$. For an assignment predicate $\varphi = \bigwedge_{i=0,\ldots,n-1}(x_i \sim_i c_i) \in$

$Pred^{ass}_{X_1,X_2}$ and some $x \in X_2$ we define $\varphi[x] = \bigwedge_{i=0,\dots n-1, x_i = x}(x \sim_i c_i)$ to be the conjunction of all constraints in φ with left-hand-side x.

We write $\nu \models \varphi$ to denote that the valuation $\nu \in \mathbb{R}^{|X|}$ for the variables X is a model of the predicate $\varphi \in Pred_X$, and for a valuation set N we write $N \models \varphi$ if $\nu \models \varphi$ for all $\nu \in N$. We write $\varphi_1 \equiv \varphi_2$ to denote semantical equivalence (i.e., $\nu \models \varphi_1 \Leftrightarrow \nu \models \varphi_2$ for all valuations ν) between predicates.

Definition 3.1: Hybrid Automata: Syntax

A *hybrid automaton* is a tuple

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

consisting of the following components:

- A finite set Loc of *locations* or *control modes*.
- A finite ordered set $Var = \{x_0, \dots, x_{d-1}\}$ of real-valued *variables*; sometimes we use the vector notation $x = (x_0, \dots, x_{d-1})$. The number d is called the *dimension* of $\mathcal{H}$. We denote the set $\{\dot{x}_0, \dots, \dot{x}_{d-1}\}$ of dotted variables by $\dot{Var}$ (which represent first derivatives during continuous change) and the set $\{x'_0, \dots, x'_{d-1}\}$ of primed variables as Var' (which represent values directly after a discrete change).
- A finite set Lab of *synchronization labels* containing the *stutter label* $\tau \in Lab$.
- $Flow : Loc \to Pred^{ass}_{Var,\dot{Var}}$ specifies the *flow* or *dynamics* for each location.
- $Inv : Loc \to Pred_{Var}$ assigns an *invariant* predicate to each location.
- $Edge \subseteq Loc \times Lab \times Pred_{Var} \times Pred^{ass}_{Var,Var'} \times Loc$ is a finite set of *discrete transitions* or *jumps*. For a jump $(\ell, a, g, r, \ell') \in Edge$, ℓ is its *source* location, a is its *synchronization label*, ℓ' is its *target* location, g specifies the jump's *guard*, and r its *reset*, where primed variables represent the state directly after the discrete jump.
- $Init : Loc \to Pred_{Var}$ assigns an *initial* predicate to each location.

We sometimes use subscript notation to associate components with a concrete hybrid automaton, i.e., $Init_{\mathcal{H}}$ refers to the set of initial states of the hybrid automaton $\mathcal{H}$. We sometimes skip the label component of a hybrid automaton and its edges if it is not relevant in the given context.

The behavior of a hybrid automaton is defined by an operational semantics as follows (see Definition 3.2 for the formal definition). A *state* $\sigma \in \Sigma = (Loc \times \mathbb{R}^d)$ of a given hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ is a tuple $\sigma = (\ell, \nu)$ of a location $\ell \in Loc$ and a variable valuation $\nu \in \mathbb{R}^d$. We collect (symbolically represented) sets of valuations N which agree on the same location in *state sets* $(\ell, N) = \{(\ell, \nu) \mid \nu \in N\}$, which we also call a *symbolic state*. In

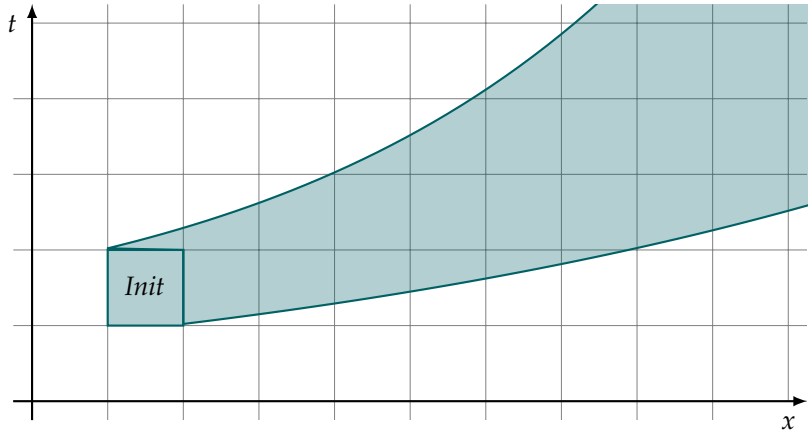


Figure 3.1: A sketch of a flowpipe. The shaded area (petrol) contains all reachable states starting from the set *Init* by letting time pass. The shape of the flowpipe is determined by the initial set and the flow specification.

the following, we overload operations f over valuations sets to be applicable also to state sets (ℓ, N) as defined by $f((\ell, \mathrm{N})) = (\ell, f(\mathrm{N}))$. In fact, we use the terms state set and *set of valuations* synonymously throughout the rest of this work.

Initial states (ℓ, ν) satisfy both the initial and the invariant conditions of location ℓ. States change according to discrete and continuous/time transitions as follows. A time transition (*flow*) models the passage of time t: while control stays in a location ℓ, the values of the state variables evolve continuously according to a function that is a solution to a system of ODEs given by the flow condition of the current location. Furthermore, the invariant predicate $Inv(\ell)$ of the location must not be violated by any variable valuation during time evolution.

Given a set of initial states, the states which are reachable via time transitions according to the flow in the given location form a *flowpipe* (see Figure 3.1). When flows are described by linear predicates, i.e., linear differential equations, we refer to this as *linear dynamics*; otherwise we use the term *non-linear dynamics*.

Discrete transitions (*jumps*) model a change of mode, given that the guard of a jump is satisfied in the predecessor state. The successor state of a discrete jump, which is the result of applying the reset function on the predecessor state, must satisfy the invariant of the target location.

Definition 3.2: Hybrid Automata: Operational Semantics

The one-step semantics of a hybrid automaton

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

is specified by the following rules:

$$\frac{\begin{array}{c}\ell \in Loc \quad \nu, \nu' \in \mathbb{R}^d \quad f : [0,\tau] \to \mathbb{R}^d \\ \partial f/\partial t = \dot{f} : (0,\tau) \to \mathbb{R}^d \quad f(0) = \nu \quad f(\tau) = \nu' \\ \forall \epsilon \in (0,\tau).\ f(\epsilon), \dot{f}(\epsilon) \models Flow(\ell) \\ \forall \epsilon \in [0,\tau].\ f(\epsilon) \models Inv(\ell)\end{array}}{(\ell,\nu) \xrightarrow{\tau} (\ell,\nu')} \quad \texttt{Rule}_{\texttt{flow}}$$

$$\frac{\begin{array}{c}e = (\ell, g, r, \ell') \in Edge \quad \ell, \ell' \in Loc \quad \nu, \nu' \in \mathbb{R}^d \\ \nu \models g \quad \nu, \nu' \models r \quad \nu' \models Inv(\ell')\end{array}}{(\ell,\nu) \xrightarrow{e} (\ell',\nu')} \quad \texttt{Rule}_{\texttt{jump}}$$

Executions (*runs*, *paths*) π of a hybrid automaton are ordered sequences of states that are connected by time and discrete steps as specified by the operational semantics rules $\texttt{Rule}_{\texttt{flow}}$ and $\texttt{Rule}_{\texttt{jump}}$.

Definition 3.3: Path

A path of $\mathcal{H}$ is a (finite or infinite) sequence of states of $\mathcal{H}$ that are connected through alternating time and discrete steps:

$$\pi = \sigma_0 \xrightarrow{\tau_0} \sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{\tau_2} \sigma_3 \xrightarrow{e_3} \sigma_4 \xrightarrow{\tau_4} \cdots$$

with $\sigma_i = (\ell_i, \nu_i) \in Loc_{\mathcal{H}} \times \mathbb{R}^d$ being states of $\mathcal{H}$, $\tau_i \in \mathbb{R}_{\geq 0}$, $e_i \in Edge_{\mathcal{H}}$, and $\nu_0 \models Inv_{\mathcal{H}}(\ell_0)$.
We call a path *initial* if additionally $\nu_0 \models Init_{\mathcal{H}}(\ell_0)$. A state σ is *reachable* in $\mathcal{H}$ if there is an initial path in $\mathcal{H}$ leading to it.

We assume finite paths $\sigma_0 \xrightarrow{\tau_0} \cdots \xrightarrow{\tau_{2k}} \sigma_{2k+1}$ to end with a time step (possibly of duration 0). The *length* $|\pi| \in \mathbb{N} \cup \{\infty\}$ of a path π is the number of jumps in it. The *duration* $dur(\pi) \in \mathbb{R} \cup \{\infty\}$ of a path π is the time which passes on π, i.e.,

$$dur(\pi) = \sum_{i=0}^{\infty} \tau_{2i} \ .$$

A state σ is *visited* by a (possibly infinite) path $\pi = \sigma_0 \xrightarrow{\tau_0} \sigma_1 \xrightarrow{e_1} \sigma_2 \cdots$ of length $k \in \mathbb{N} \cup \{\infty\}$ if there exist $i \in \mathbb{N}_{\leq k}$ and $\tau \in \mathbb{R}$ with $0 \leq \tau \leq \tau_{2i}$ such that $\sigma_{2i} \xrightarrow{\tau} \sigma$ and $\sigma \xrightarrow{\tau_{2i}-\tau} \sigma_{2i+1}$; we write $Reach_\pi$ for the set of all states visited by π.

In contrast to the continuous notion of visiting a state, we use the pointwise notion for *prefix*, i.e., a finite path $\pi_1 = \sigma_0 \xrightarrow{\tau_0} \cdots \xrightarrow{\tau_{2k}} \sigma_{2k+1}$ is a *prefix* of all other paths $\pi_2 = \sigma_0 \xrightarrow{\tau_0} \cdots \xrightarrow{\tau_{2k}} \sigma_{2k+1} \cdots$.

For simplicity, unless stated otherwise, in the following we mean by path a finite initial path.

An infinite path π in a hybrid system is called *time divergent*, if $dur(\pi) = \infty$ and *time convergent* otherwise.

We extend the notion of paths π over states (ℓ, ν) to *symbolic paths* Π over symbolic states (ℓ, N):

$$\Pi = (\ell_0, \mathrm{N}_0) \xrightarrow{[\tau_{l,0}, \tau_{u,0}]} (\ell_1, \mathrm{N}_1) \xrightarrow{e_1} (\ell_2, \mathrm{N}_2) \xrightarrow{[\tau_{l,2}, \tau_{u,2}]} (\ell_3, \mathrm{N}_3) \xrightarrow{e_3} \cdots$$

Such a *symbolic path* Π includes all paths

$$\pi = (\ell_0, \nu_0) \xrightarrow{\tau_0} (\ell_1, \nu_1) \xrightarrow{e_1} (\ell_2, \nu_2) \xrightarrow{\tau_2} (\ell_3, \nu_3) \xrightarrow{e_3} \cdots$$

for which $\nu_i \in \mathrm{N}_i$ and $\tau_{2i} \in [\tau_{l,2i}, \tau_{u,2i}]$ for all i; we write $\pi \in \Pi$ to denote this inclusion.

For symbolic paths Π we define the set of *visited* states as $\mathit{Reach}_\Pi = \cup_{\pi \in \Pi} \mathit{Reach}_\pi$.

A symbolic path is *contained* in another symbolic path, written $\Pi_1 \subseteq \Pi_2$, if for all $\pi \in \Pi_1$ it holds that $\pi \in \Pi_2$. A symbolic path Π_1 is a *prefix* of a symbolic path Π_2 if Π_2 starts with Π_1, i.e., either they are equal or Π_2 has the form $\Pi_1 \xrightarrow{e} \sigma \xrightarrow{\tau} \cdots$.

The longest common prefix of two symbolic paths Π_1, Π_2 is denoted as $\mathit{pref}(\Pi, \Pi')$, which is ϵ if no common prefix exists.

A *counterexample* for a d-dimensional hybrid automaton $\mathcal{H}$ and an unsafe state set $P_{bad} \subseteq \mathbb{R}^d$ is an initial path π^* of $\mathcal{H}$ that visits at least one state from P_{bad}, i.e.,

$$\mathit{Reach}_{\pi^*} \cap P_{bad} \neq \emptyset \ .$$

We can also model different system components separately and use shared variable concurrency and jump label synchronization to build their parallel composition.

Intuitively, hybrid automata which are composed in parallel execute concurrently: time evolves simultaneously in all components, whereas for jumps the components may synchronize with each other or execute independently. For the latter case, in order to express (and possibly restrict) discrete changes caused by the environment, components that do not synchronize on a jump execute a local jump with the stutter label τ.

Definition 3.4: Parallel Composition

Consider 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 $\mathit{Var}_1 = \mathit{Var}_2$. The *product automaton*

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

is the hybrid automaton with

- $\mathit{Loc} = \mathit{Loc}_1 \times \mathit{Loc}_2$,
- $\mathit{Var} = \mathit{Var}_1 (= \mathit{Var}_2)$,

- $Lab = Lab_1 \cup Lab_2$,
- $Flow(\ell_1, \ell_2) = Flow_1(\ell_1) \wedge Flow_2(\ell_2)$ for all $(\ell_1, \ell_2) \in Loc$,
- $Inv(\ell_1, \ell_2) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$ for all $(\ell_1, \ell_2) \in Loc$,
- $Init(\ell_1, \ell_2) = Init_1(\ell_1) \wedge Init_2(\ell_2)$ for all $(\ell_1, \ell_2) \in Loc$,
- $Edge$ is the smallest set that contains for each $e_1 = (\ell_1, a_1, g_1, r_1, \ell_1') \in Edge_1$, $e_2 = (\ell_2, a_2, g_2, r_2, \ell_2') \in Edge_2$ the edge $(\ell_1, \ell_2), a, g_1 \wedge g_2, r_1 \wedge r_2, (\ell_1', \ell_2')$ if:
  - either $a = a_1 = a_2$,
  - or $a_1 \notin Lab_2$ and $a_2 = \tau$,
  - or $a_2 \notin Lab_1$ and $a_1 = \tau$.

Note that in practice the construction of the product automaton is tedious, as the resulting automaton size is typically large.

Classes of Hybrid Automata

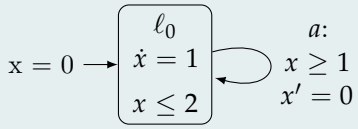
There are several subclasses of hybrid automata that are interesting for some theoretical reason or practical aspect. These subclasses are typically defined as syntactical fragments, restricting the type of predicates allowed to specify flows, invariants, guards, and resets. In the following, we will shortly present the subclasses relevant for this work and indicate how reachability can be computed for those subclasses. A detailed presentation of the approaches for the reachability analysis of each subclass can be found in Section 8.1.

Timed Automata. *Timed automata (TA)* are a relatively simple subclass of hybrid automata in which each variable $x \in Var$ is a *clock* measuring the time with derivative $\dot{x} = 1$ [AD94]. In this fragment, predicates $Pred_{Var}$ are conjunctions of constraints $x \sim c$ with $c \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$ comparing clocks to constants, i.e., invariants and guards encode *rectangular sets*. Discrete jumps in TA either leave the value of a clock unchanged or reset it to zero. The syntactic restrictions on TA render their reachability problem decidable and allow for sound and complete verification algorithms for safety properties [ACD90]. We can create a finite abstract model of a TA, the so-called *region transition systems*, to which we can apply standard (explicit or symbolic) model checking approaches to answer the TA reachability problem. Zone-based abstractions improve this approach using difference bound matrices (DBMs) as a dedicated datatype for states sets (see [BK08] and Section 8.1).

TA are often used to model and verify digital real-time systems or processes in which timing is critical but no further dynamic behavior is present. Examples include network communication protocols [ACH+95] or production chains.

Example 3.1: Timed Automaton

Consider the following timed automaton $\mathcal{T}$:

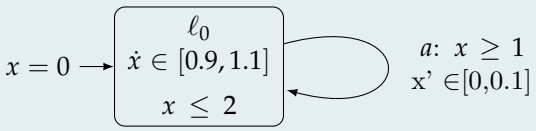


The automaton has a single clock x, which is reset periodically in time intervals of length $[1, 2]$. While this automaton itself is not very exciting, a parallel composition of $\mathcal{T}$ with another, more expressive hybrid automaton might be more interesting. Using the synchronization label a on the discrete jump, the automaton $\mathcal{T}$ can be seen as a non-deterministic event generator embedded into a larger context.

Rectangular Automata. A more expressive subclass is *rectangular automata (RA)* in which not only invariants and guards but also all assertion predicates encode *rectangular sets*, i.e., each predicate is a conjunction of constraints comparing variables to constants. Instead of $a \leq x \wedge x \leq b$ we also write $x \in [a, b]$. Using this notation, flows are composed from constraints $\dot{x} \in [a, b]$, and variables can be reset on jumps using $x' \in [a, b]$ with $x \in Var$ and $a, b \in \mathbb{N}$.

Example 3.2: Rectangular Automaton

The following RA is similar to the timed automaton model above, but its clock is subject to drift and cannot be precisely reset.



In general, the reachability problem for RA is undecidable. However, reachability via paths having a *bounded* number of jumps can be still decided, e.g., using the same method as described for LHA I below. Furthermore, the unbounded reachability problem for RA is decidable under the condition of being *initialized*, meaning that if two locations specify different flows for a variable, then each jump between them resets that variable to a constant value from an interval.

Definition 3.5: Initializedness

A hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init))$ is called *initialized* if the following holds:

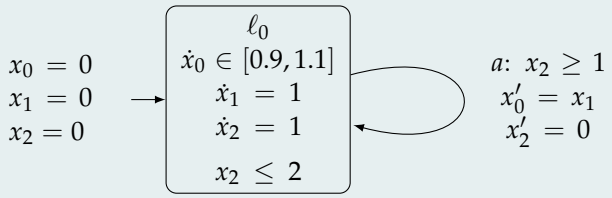
$$\forall(\ell, g, r, \ell') \in Edge.\ \forall x \in Var.\ Flow(\ell)[x] \not\equiv Flow(\ell')[x] \Rightarrow$$
$$\exists a, b \in \mathbb{N}.\ r[x] \equiv (x' \in [a, b])$$

Intuitively, *initializedness* requires that whenever the dynamics of a variable changes when taking a discrete jump, the variable has to occur non-trivially in the reset function.

Linear Hybrid Automata I. Further increasing the expressivity, we arrive at the class of *linear hybrid automata I (LHA I)*, whose flows are defined by rectangular sets as for RA, but whose other predicates may also contain linear expressions over the model variables. That means, invariants and guards may contain constraints of the form $\mathcal{A}x \sim b$ and resets may use *affine mappings* $x' = \mathcal{A}x + b$, where $\mathcal{A} \in \mathbb{R}^{d\times d}$ and $b \in \mathbb{R}^d$. Similarly to RA, the jump-bounded reachability problem for LHA I is decidable, but the general reachability problem is not.

Example 3.3: Linear Hybrid Automaton I

Besides a skewed clock x_0, the LHA I below has also a precise clock x_1, which is used to periodically set the skewed clock to the precise time. Another precise clock x_2 is used to assure that this synchronization happens regularly at least one and at most two time units after the last reset.

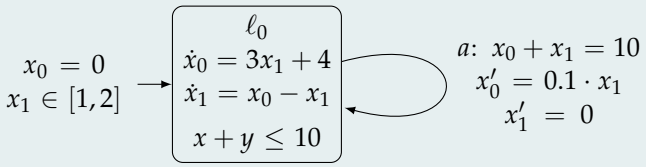


One possible approach to compute reachability for LHA I is based on logical characterizations of state sets and formula transformations along with quantifier elimination to compute successors [Hen96]. A detailed description for LHA I-reachability analysis is outlined in Section 8.1.

Linear Hybrid Automata II. The main focus of this work is on this model class. *Linear hybrid automata II (LHA II)* extends the expressivity of LHA I by allowing linear expressions also in the flows, i.e., allowing to specify dynamics by systems of linear ODEs. For this class, even the problem of bounded reachability is undecidable.

Example 3.4: Linear Hybrid Automaton II

In contrast to all the above model classes, LHA II allow variable dynamics to be *coupled*, e.g., the dynamics of a variable might depend on the value of another variable, as in the following example LHA II automaton:



Many mechanical and physical systems such as moving point masses can be described by LHA II automata—for more examples see Chapter 4. Furthermore, higher-order linear differential equations can be described as a system of first-order linear differential equations by adding intermediate variables.

Table 3.1 sums up the decidability results of the presented subclasses of hybrid automata along with their specific characterization.

3.3 Reachability Analysis for Hybrid Systems

At the heart of hybrid systems safety verification is the computation of the set of reachable states. Basic safety properties specify a set of bad states P_{bad}, whose reachability is safety-critical. *Reachability analysis* as a method for safety verification of hybrid systems tries to determine, whether the set of *bad states* P_{bad} is reachable in a given hybrid automaton $\mathcal{H}$ from any of its initial states. Formally, the method tries to answer whether $Reach_{\mathcal{H}} \cap P_{bad} = \emptyset$, where $Reach_{\mathcal{H}}$ denotes the set of reachable states in $\mathcal{H}$.

Current reachability analysis tools for hybrid systems use different approaches to verify safety via reachability analysis. Some approaches are based on *theorem proving* and combine deductive, real algebraic, and computer-algebraic proving technologies. On the one hand, these techniques are compelling and can handle (at least in theory) a wide range of models by using deduction to provide mathematical proof for the safety of a given system. On the other hand, these approaches are semi-interactive and need experienced users to guide the proof. Some tools provide predefined strategies and allow to create user-defined ones which can be of great help to increase the level of automation and reduce the need for interaction to a minimal level. The most prominent tool implementing this approach is KeYmaera [Pla08; PQ08].

Other approaches use logical characterizations of the reachability problem and employ bounded model checking and satisfiability modulo theories (SMT) solving technologies to check for safety [BHM+09; Gao12]. The idea is to formulate the one-step reachability relation as a mixed integer-real arithmetic formula. In case the solution of the ordinary differential equation is known (for instance if the derivatives are constant), fast SMT-solvers can be employed. Otherwise, the logic and the corresponding decision procedures need to be extended with a theory for differential equations. Theoretically, even if incomplete, these techniques might succeed in proving safety as well as unsafety. However, running times are hard to predict and incomplete methods might return inconclusive answers, even for decidable problems. Tools using this approach are for

Table 3.1: Decidability results for subclasses of hybrid automata (TA= timed automata, IRA = initialized rectangular automata, RA = rectangular automata, LHA I = hybrid automata with constant derivatives, LHA II = hybrid automata with linear ODEs, HA = general hybrid automata).

subcl.	derivatives	conditions	bnd. reach.	unbnd. reach.
TA	$\dot{x} = 1$	$x \sim c$	✔	✔
IRA	$\dot{x} \in [c_1, c_2]$	$x \in [c_1, c_2]$ jump resets x when $\dot{x}$ changes	✔	✔
RA	$\dot{x} \in [c_1, c_2]$	$x \in [c_1, c_2]$	✔	✖
LHA I	$\dot{x} = c$	$x \sim g_{linear}$	✔	✖
LHA II	$\dot{x} = f_{linear}$	$x \sim g_{linear}$	✖	✖
HA	$\dot{x} = f$	$x \sim g$	✖	✖

instance DReach [Gao12; GKC13], HSolver [RS05], hydlogic [IUH11] or ISatODE [EFH08; Egg14].

Another category is that of *iterative* reachability analysis methods. *Forward* reachability analysis starts with the set of initial states and iteratively computes successor states until either the set of bad states is reached, or a fixed-point has been found. In contrast to that, *backward* analysis starts from the set of bad states and iteratively computes predecessor states until either the set of initial states is reached, or a fixed-point has been found. If we use exact computations, then reaching the bad/initial state is a proof of unsafety, whereas a fixed-point that does not contain any bad/initial state proves safety.

Since the reachability problem for general hybrid automata is undecidable [HKP+98], also methods to *over-approximate* $Reach_{\mathcal{H}}$ by $Reach'_{\mathcal{H}} \supseteq Reach_{\mathcal{H}}$ have been developed. Over-approximative computations can be used to implement semi-decision procedures that can still prove safety. However, if over-approximative successor/predecessor computations detect a bad/initial state, then no conclusive answer can be given.

To ensure termination, the problem is furthermore typically restricted to reachability via paths with bounded time duration and a limited number of jumps. Instead of putting an upper bound on the total time duration of paths, in this work, we will use a *time horizon* T as an upper bound on uninterrupted time evolution without jumps. Furthermore, we will use a *jump depth* J to limit the number of discrete transitions along execution paths.

A general forward reachability analysis algorithm is depicted in Algorithm 1.

In this work, we focus on a certain kind of iterative forward reachability computation methods that are called *flowpipe-construction-based* approaches. To compute time successors, tools implementing this approach divide a bounded time horizon into smaller segments of size δ, which we refer to as *time step size*. The flowpipe, i.e., the set of trajectories of a given system is over-approximated for each segment by a single state set. The state sets are usually

Algorithm 1: General forward reachability analysis algorithm.

Input: Set of initial states I
Output: Set of reachable states R

$R := I$
$R_{new} := R$
while $R_{new} \neq \emptyset$ **do**
  $R_{new} :=$`computeReach(`R_{new}`)`$\setminus R$
  $R := R \cup R_{new}$

over-approximated by geometric or symbolic state set representations, for instance boxes (hyper-rectangles), convex polytopes, zonotopes, ellipsoids, support functions, or Taylor models (see Figure 3.2). Starting from a set of initial states, the time- and jump-successor states are computed by applying certain operations on those state set representations. The ratio between computational effort and precision can be influenced using various analysis parameters such as the time step size or the type of state set representation, which usually are provided by the user. The advantage of these techniques lies in the high level of automation and in the possibility to increase efficiency or to improve the precision according to the needs of the user. On the other hand, due to the over-approximative representation of state sets, only safety can be proven with this technique—proving unsafety would require under-approximative computations. Tools in this category are for instance CoRA [Alt15], Flow* [CÁS13], SpaceEx [FLD+11], Xspeed [GRB+18], ariadne [CBG+12], and our HyPro tool on which we will report in detail later.

3.4 Flowpipe Construction

In the following we explain *flowpipe-construction-based reachability analysis* in more detail. The method is formalized in Algorithm 2 at the end of this section. For further reading we refer to e.g., [Le 09].

In this work, we focus on *linear hybrid systems* specified by linear hybrid automata (see Section 3.2). Locally, each control mode of an autonomous LHA II specifies a dynamic system by a system of linear differential equations

$$\dot{x} = \mathcal{A}x \tag{3.1}$$

over the variables $x = (x_0, \ldots, x_{d-1})^T$ of the given hybrid automaton $\mathcal{H}$, where $\mathcal{A} \in \mathbb{R}^{d \times d}$ is a coefficient matrix. Note that we can transform systems of the form $\dot{x} = \mathcal{A}x + b$ to the form presented above by adding an artificial dimension with zero flow for the constant vector b.

Non-autonomous systems extend the dynamics by a time-dependent function $u(t)$. Usually, it is assumed that $u(t)$ is from a bounded domain U, which is

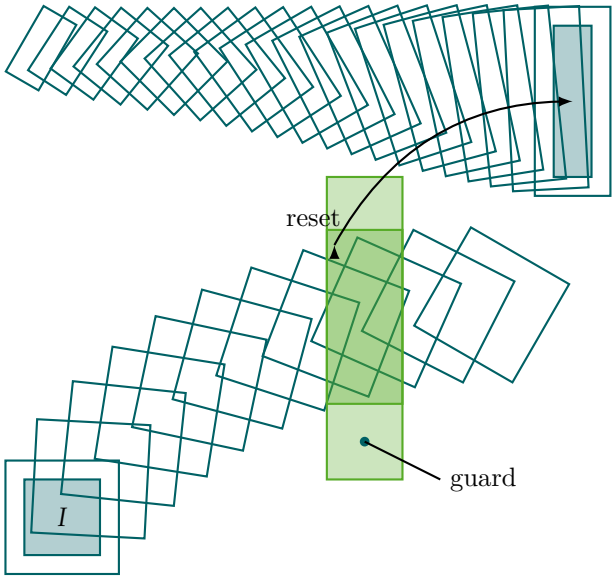


Figure 3.2: Sketch of flowpipe-construction-based reachability analysis for hybrid systems.

used to represent external inputs influencing the system evolution and results in dynamics of the form

$$\dot{x} = \mathcal{A} \cdot x + \mathcal{B} \cdot u$$

where $\mathcal{A}$ and $\mathcal{B}$ are matrices of appropriate dimensions and $u \in U$. In this work, we only consider autonomous systems, for details on flowpipe-construction-based reachability analysis methods for non-autonomous systems we refer to [Le 09] in which a state-of-the-art reachability analysis method for non-autonomous systems is presented and to [FKL13] for a refined approach.

Solutions to Equation (3.1) are of the form

$$x(t) = \underbrace{e^{t\mathcal{A}}}_{\Phi} \cdot x(0) \tag{3.2}$$

i.e., $x(t)$ is the state that is reached at time point t when starting from the initial state $x(0)$ at time point $t = 0$ and following the flow specified by the matrix $\mathcal{A}$. Consequently, Φ depends on the current location ℓ for which time successors need to be computed and also on the chosen t. The solution (Equation (3.2)) represents a linear transformation of the initial variable valuation $x_0 = x(0)$ and can directly be extended to sets of variable valuations N such that

$$\mathrm{N}_t = \Phi \cdot \mathrm{N}_0 \ .$$

While Equation (3.2) allows to compute the set of reachable states N_t at a specific point in time t, it does not yet allow to reason about the set of reachable states over a time interval. To overcome this, methods which approximate the

set of reachable states for a time interval $[0, \delta]$ have been developed [Dan00; Gir04; Gir05; Le 09; LG10].

The general idea of those methods is to over-approximate the error α between an approximation Ω of the set of reachable states for the time interval $[0, \delta]$ and the actual set of reachable states

$$Reach_{[0,\delta]} = \left\{ (\ell, \nu) \;\middle|\; \nu = e^{t\mathcal{A}} \cdot x_0, t \in [0, \delta], x_0 \in \mathrm{N}_0 \right\} .$$

One of the earliest approaches [Gir05] provides an over-approximation of α by approximating the Hausdorff-distance (see Definition 3.6) between $\Omega' = cHull(\mathrm{N}_0 \cap \mathrm{N}_\delta)$ and the actual reachable set of states.

Definition 3.6: Hausdorff Distance

The *Hausdorff distance* between two sets $A, B \subseteq \mathbb{R}^d$ is defined as

$$d_{A,B} = \max \left\{ \sup_{a \in A} \inf_{b \in B} d(a, b), \sup_{b \in B} \inf_{a \in A} d(a, b) \right\}$$

for some distance metric $d()$.

For an initial state $x \in \mathrm{N}_0$ and the state $r = e^{\delta\mathcal{A}} \cdot x$ that is reached from x at time point δ, we consider their connecting line segment

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

The union of all such line segments for all $x \in \mathrm{N}_0$ defines the convex hull of N_0 and N_δ. The error between $s_x(t)$ and the actual trajectory $\zeta_x(t) = e^{t\mathcal{A}}x$ then can be quantified as

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

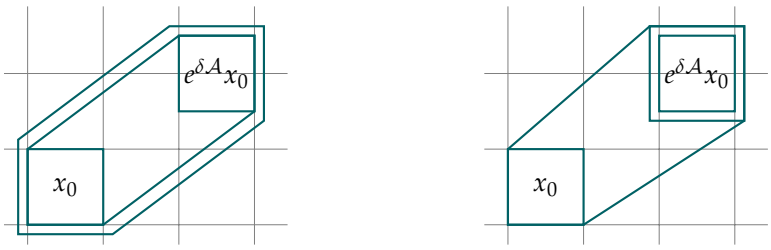
In [Gir05], this error is approximated using Taylor's theorem (the Taylor-expansion is cut off after degree two and the remainder is over-approximated) such that

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

With this result it is possible to safely over-approximate the set of reachable states $Reach_{[0,\delta]}$ for the time interval $[0, \delta]$ by bloating the convex hull with a ball $\mathcal{B}_\alpha$ of radius α

$$\Omega = (\mathrm{N}_0 \cup \mathrm{N}_\delta) \oplus \mathcal{B}_\alpha,$$

where the operator $\oplus$ denotes the Minkowski-sum (see Definition 3.7). This approach results in a uniform bloating (see Figure 3.3a)—in later works [Le 09], a non-uniform bloating method has been developed (see Figure 3.3b).



a) Uniform bloating. b) Non-uniform bloating.

Figure 3.3: Construction of the first segment of a flowpipe applying different bloating techniques (sketch).

Definition 3.7: Minkowski Sum

The Minkowski sum $A \oplus B$ of two sets $A, B \subseteq \mathbb{R}^d$ is defined as the set

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

and represents the set-theoretic equivalent to addition.

Once the first flowpipe segment Ω_0 safely over-approximating the time interval $[0, \delta]$ is computed, we can compute further segments Ω_i using the same time step size δ, thereby discretizing the time horizon T for the analysis. The recurrence relation to obtain a sequence of segments Ω_i for one location ℓ of an autonomous linear hybrid system is given as

$$\Omega_{i+1} = \Phi \cdot \Omega_i \ .$$

The repeated application of Φ results in a sequence of segments Ω_i where each segment by construction safely over-approximates a time interval $[i\delta, (i+1)\delta]$ For non-autonomous hybrid systems a second bloating step needs to be performed during the analysis which is required for each computed set Ω_i. For the interested reader, we refer to [Le 09] which contains a more detailed description of how to efficiently handle this second bloating step.

With this method at hand, we are able to compute flowpipe segments $\Omega_i, i = 0, \ldots, k-1$ for a given dynamics $\mathcal{A}$ up to a certain time horizon T using a fixed time step size $\delta = \frac{T}{k}$, starting from a defined set of initial states (ℓ, N_0). If we consider the semantic rules for letting time pass ($\texttt{Rule}_{\texttt{flow}}$) for hybrid automata, we notice that until now we have not considered the invariant of the current location. Time can pass in a location only as long as the location's invariant is satisfied. Since $\Omega_i = (\ell, \mathrm{N}_i)$ is an over-approximation of the actual set of reachable states for its assigned time interval $[i\delta, (i+1)\delta]$, we can stop the computation of further time successors in the form of further flowpipe segments $\Omega_{i+1}, \Omega_{i+2}, \ldots$ if $\mathrm{N}_i \cap \mathit{Inv}(\ell) = \emptyset$.

To compute discrete successors, for each outgoing transition $e = (\ell, g, r, \ell') \in \mathit{Edge}$ of the current location ℓ in $\mathcal{H}$ and for each flowpipe segment $\Omega_i = (\ell, \mathrm{N}_i)$

we need to verify whether the jump can be taken from any state in the segment, i.e., whether

$$\mathrm{N}_i' = \mathrm{N}_i \cap g \neq \emptyset$$

holds. Each flowpipe segment for which this property holds is said to enable the transition. All non-empty segments $\Omega_i' = (\ell, \mathrm{N}_i')$ with $\mathrm{N}_i' \neq \emptyset$ are passed to the reset function r to discretely update variable valuations and obtain

$$\Omega_i'' = (\ell, \mathrm{N}_i'') = (\ell, r(\mathrm{N}_i')) \ .$$

For linear hybrid systems, this is usually realized by an affine transformation such that $\mathrm{N}_i'' = \mathcal{A} \cdot (\mathrm{N}_i') + b$ where $\mathcal{A}$ is used to linearly transform N_i' and b is a translation vector, both of appropriate dimension.

As a last step, we check whether any valuations from N_i'' fulfills the invariant of the target location by computing $\mathrm{N}_i''' = \mathrm{N}_i'' \cap \mathit{Inv}(\ell')$ and checking whether $\mathrm{N}_i''' = \emptyset$. We continue with computing the flowpipe in ℓ' for all non-empty $\Omega_i''' = (\ell', \mathrm{N}''')$.

Algorithm 2: Pseudo-code algorithm for bounded flowpipe-construction-based reachability analysis.

Input: Hybrid automaton $\mathcal{H} = (\mathit{Loc}, \mathit{Var}, \mathit{Lab}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Init})$
Output: Over-approximation of reachable states in $\mathcal{H}$

```
Q := Init
R := ∅
while Q ≠ ∅ do
    Ω := computeFirstSegment(getElement(Q))
    while not timeBoundReached() do
        Ω := Ω ∩ Inv(ℓ)
        R := R ∪ Ω
        if not jumpBoundReached() then
            for (ℓ, g, r, ℓ') ∈ Edge do
                addElement(Q, r(Ω ∩ g) ∩ Inv(ℓ'))
        Ω := letTimePass(Ω)
return R
```

During the analysis, we employ several operations such as intersection, union, Minkowski sum, or test for emptiness on state sets, but more explicitly on sets of variable valuations. We will address this issue in a later part of this dissertation in which we present our collection of state set representations under a unified interface which is suitable to implement flowpipe-construction-based reachability analysis methods (see Chapter 6). Note that the presented approach is designed for the reachability analysis of linear hybrid automata II (LHA II). As indicated before, dedicated methods exist for some of the subclasses of LHA II, which will be considered in a later part of this work (see Chapter 8) and explained in detail there.

Analysis Parameters

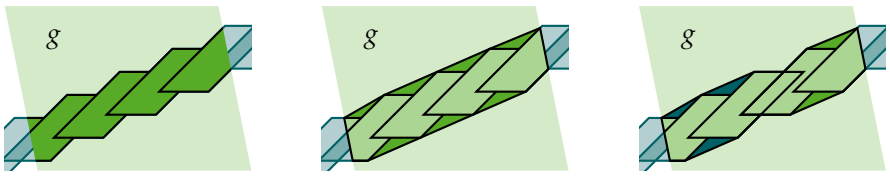
The verification of hybrid systems brings several challenges for the user. The first challenge lies in the creation of a formal model of the hybrid system at the right level of abstraction, which reflects the behavior of the system under analysis. Once we have a formal model of a hybrid system, we can employ flowpipe-construction-based reachability analysis methods to analyze the safety properties of said system for all bounded executions starting from the set of initial states.

The application of flowpipe-construction-based approaches for reachability analysis as presented before poses another challenge related to the choice of the analysis parameters such as the time step size δ. These parameters affect the introduced over-approximation errors and the computational effort and thus significantly influence the verification success. Choosing suitable analysis parameters is a tedious task that requires detailed knowledge about the implemented approaches as well as the model of the system which is to be analyzed [SÁC+15]. In this section, we provide an overview of potential parameters of interest and illustrate their influence on the analysis in terms of precision and running time. Throughout the rest of this work, we refer to valuations for the set Par of relevant analysis parameters which may be used by a reachability analysis method as a *parameter configuration*. We use $FP((\ell, \mathrm{N}), Par)$ to denote the set of symbolic states that results from computing the flowpipe for the symbolic state (ℓ, N) and all possible jump successors using the parameter configuration Par.

Time Step Size. As already pointed out, the time step size δ influences the introduced over-approximation errors, which are added when computing the first segment of the flowpipe (see Section 3.4). First approaches for flowpipe-construction-based reachability analysis [Dan00; Gir04] additionally suffered from introducing additional errors caused by wrapping effects. Later improvements [Le 09] reduced this error for non-autonomous hybrid systems by decoupling the bloating introduced by external inputs from the computation of the autonomous parts of the system. The authors of [FKL13] presented a method to dynamically adapt the time step size during computation to be able to meet a predefined error bound that was provided by the user a priori.

Discrete Jump-successor Computation. Even though it might not be obvious, the way discrete jump successors are computed has a crucial influence on the running times of a reachability analysis algorithm. Usually, more than one flowpipe segment satisfies a guard condition of a discrete transition $e = (\ell, g, r, \ell') \in Edge$ which can be processed in several ways.

Naturally, all flowpipe segments Ω_i that enable a jump can be handled individually and a new flowpipe in the target location can be computed rooted in Ω_i''' (see Figure 3.4a). The drawback of this approach is the strong branching of the search tree (see Section 6.9) as for each set satisfying a guard constraint a new discrete jump successor is created. Furthermore, the number of discrete jump successors for one transition is not known a priori and might even change when using different time step sizes, as the number of segments over-approximating



a) No aggregation. b) Aggregation. c) Clustering.

Figure 3.4: Different approaches in the discrete jump successor computation of state sets satisfying the guard g predicate (green).

a specified time interval changes. Nonetheless, this approach should not be discarded per se, as in contrast to other approaches (see below) it does not introduce additional over-approximation errors, which can have a considerable influence, depending on how state sets are represented.

Aggregation instead provides an easy-to-use approach in which all segments or parts of segments Ω_i satisfying a certain guard condition g are unified to one set (see Figure 3.4b). This approach allows for maximal control regarding the branching of the induced search tree but also introduces additional over-approximation errors, which depend on the used state set representation. Furthermore, aggregation can only be performed when all sets satisfying g are known.

A middle ground between the two approaches is *clustering* (see Figure 3.4c). In this approach, an upper bound c of discrete successors is provided a priori. Jump successors of flowpipe segments are distributed among c groups and the segments in each group are combined to one set using set-union. A discrete successor in the search tree is added for each group. This method is less restrictive than aggregation but still allows to control the branching factor of the search tree.

State Set Representation. The choice of a state set representation is crucial to the outcome of the analysis of a given system. We differentiate between symbolic and geometric state set representations to describe a set of states of a given hybrid system. Geometric representations used for reachability analysis include boxes [RS05; MKC09], convex polytopes [Zie95; Fre05], ellipsoids [KV00; KV07], oriented rectangular hulls [SK03], orthogonal polyhedra [Dan00], template polyhedra [SDI08], and zonotopes [Gir05] among others. Taylor models [MB09; Che15] and support functions [LG10] have been used to represent state sets symbolically. Each state set representation comes with its advantages and disadvantages making a universal statement difficult.

Zeno Test. *Zeno behavior* describes unrealistic behavior in which a time convergent path is executed. In general, we differentiate between two types of Zeno behavior—*chattering Zeno behavior* and *classical Zeno behavior*. Chattering Zeno behavior occurs whenever a system may execute infinitely many discrete transitions in finite or even zero time. A simple example exhibiting chattering Zeno behavior is presented in Figure 3.5, where the only transition is a constantly enabled self-loop. Classical Zeno behavior, as a super-class

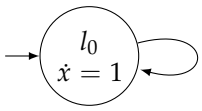


Figure 3.5: Zeno behavior: the discrete transition is always enabled and may be taken infinitely often without letting time pass.

of chattering Zeno, includes all execution paths of a system which are time convergent. An infinite path π in a hybrid automaton is called time convergent if the sum of the execution times of the time transitions is finite, and time divergent otherwise. An execution of the automaton presented in Figure 3.5 where the time step sizes are converging, e.g., a sequence of time transitions $\sigma_0 \xrightarrow{1} \sigma_1 \xrightarrow{1/2} \sigma_2 \xrightarrow{1/4} \cdots$ exhibits Zeno behavior in the classical sense. Note that paths containing sequences of time transitions as the previously mentioned one are excluded in the semantics of paths defined in this thesis as we enforce alternation of discrete jumps and time transitions. While this semantics excludes classical Zeno behavior, it does not allow for infinite paths, which instead is possible when not enforcing alternation of time transitions and discrete jumps.

In reachability analysis tools time convergent paths such as the previously mentioned one are not considered for analysis, i.e., consecutive time transitions are collapsed to a single time transition. Nonetheless, the detection of chattering Zeno behavior is more challenging.

Fixed-point Test. Tests whether a fixed-point has been reached have several consequences on the performance of a reachability analysis algorithm. On the one hand, detecting a fixed-point is always beneficial in the sense that redundant computations can be avoided. On the other hand, checking the criteria for a fixed-point is time-consuming as we have to compute set-containment to be able to detect a fixed-point during the computation.

Part I

Hybrid Systems

4

Examples of Hybrid Systems

Recent advances in algorithms have turned reachability analysis into a powerful method for the safety verification of continuous and hybrid systems. Techniques are available that can compute approximations of the reachable states for systems with linear dynamics and more than 200 variables [LG10; FLD+11; BD17a; BFF+18], and for systems with complex non-linear dynamics [Gao12; CÁS13; BCD13].

As in general the reachability problem is undecidable for hybrid systems [HKP+98] and even the one-step successors can only be computed approximately, experimental results are essential for validating algorithms, detecting their shortcomings, and identifying where further research is necessary.

Experiments in reachability require not only algorithms but also models of systems and specifications that are to be verified. Such benchmarks are not easy to come by, in particular when investigating high-dimensional systems. Research papers typically include a small number of proprietary benchmarks or modified versions of benchmarks published in other papers. A notable exception is a small collection of benchmarks in [FI04], and the benchmark collection of the ARCH workshop series [AF14], which is tailored to industrial applications. Using just a small number of benchmarks for test and evaluation entails the risk to tune tools to be efficient for specific application types only.

The results presented in the next two chapters are based on [CSB+15; SÁC+15] and may contain excerpts that are not explicitly cited in the text again. In the following, we present our collection of models for hybrid systems [CSB+15] that is also publicly available[1]. It consists of system models along with property specifications and detailed descriptions, references to prior work, input files, and exemplary results for the tools Flow* [CÁS13] and SpaceEx [FLD+11]. Additionally, we present benchmarks taken from other collections [ACH+95; FI04; AF14] which we will use for our experiments throughout this work. Apart from making the benchmarks readily available in a unified form, our benchmark collection makes the following contributions:

[1] `https://ths.rwth-aachen.de/research/projects/hypro/benchmarks-of-continuous-and-hybrid-systems/` (checked July, 30^{th}, 2019)

Classification: The benchmarks originate from a variety of domains and serve a variety of purposes, e.g., testing scalability with respect to the number of variables or the number of locations. Identifying a benchmark that suits a particular tool and helps to evaluate an individual property is non-trivial. The collection is organized by the model type (continuous/hybrid, linear/non-linear), which roughly corresponds to the kind of tool to which it is applicable. Within each class, benchmarks are listed by complexity (scalability, number of variables, locations, transitions).

Specification: To ensure comparability of results between different tools, the model and property specifications need to be unambiguous and formal. We provide such formal specifications for all included benchmarks. Note that not all benchmarks easily lend themselves to specifications in the typical form of a given set of "bad states". For example, some benchmarks for testing the accuracy of approximations give quantitative results. Finding a unified form for specifying systems as well as their properties is one of the long-term goals of this collection.

Identifying challenges: Though state-of-the-art hybrid systems reachability analysis tools are impressively successful and can solve a wide range of interesting problems in industry [AD14; ABC+17; BD17a; ABC+18; IAC+18], they are still rarely applied outside the research community. Driving research directions towards the needs of other scientific areas and application domains would increase the usage of formal approaches for the verification of hybrid systems in those areas as well. Therefore, one of our long-term goals is to identify benchmarks suitable for this purpose, even if current tools do not exhibit sufficient functionalities yet. Current challenges, some of which can also observed in the presented benchmarks are also presented and discussed in detail in Chapter 5.

4.1 The Benchmark Suite

Benchmarks are an essential asset during the process of developing a tool. The advancement of reachability analysis methods for the verification of hybrid systems is not an exception. Additionally to existing benchmark collections, for instance, the collection of the ARCH workshop [AF14], we have collected and categorized a further set of benchmarks. While the collection of ARCH contains a comprehensive collection of benchmark submissions, most of the challenging instances are not solved yet (6 out of 33 benchmarks are reported to be solved[2]).

Our benchmark suite currently covers 29 benchmarks. The included benchmarks are selected to cover different levels of expressiveness in their components.

- We provide both pure *continuous* benchmarks as well as *hybrid* models.
- The continuous dynamics is described by either *linear* or *non-linear* ordinary differential equations.

[2]State: July, 30^{th} 2019

- Further classification is provided with respect to the *number of variables* and, for hybrid behavior, the *number of locations* and the *number of discrete transitions*. One of the benchmarks is *scalable*, allowing the generation of high-dimensional models.

- The hybrid models specify *transition guards* varying in their form from half-spaces or hyperplanes over linear conditions up to non-linear ones.

- *Reset conditions* can be absent or described by linear terms.

- *Invariants* are boxes in some benchmarks and general polyhedra in others.

- Reachability analysis is hindered by *Zeno behavior*, which is present in some of the models.

Our collection of linear benchmarks includes well-known smaller academic models, such as the *bouncing ball* or the *two tank system*, as well as lesser-known benchmarks, such as the *vehicle platoon* [BDK12]. For the sake of completeness and testing purposes, we have decided to include small but frequently referenced benchmarks additionally.

Our benchmark collection also features a selection of non-linear benchmarks with both purely continuous and hybrid systems. Currently, there are 13 continuous and five non-linear hybrid models available ranging from chaotic systems such as the Van der Pol oscillator and the famous Lorentz system over systems from mechanics such as a spring pendulum to various systems taken from biology such as the spiking neurons [Izh07] or the glycemic control [Fis91]. The later two benchmarks along with the non-holonomic integrator [HM99] or the non-linear transmission line circuits [RW03] were extracted from external sources, thus enhancing the collection by relevant, non-artificial benchmarks which are now open to the formal methods community. Such non-artificial models are important for driving tool development towards being capable of solving a wide range of real-world problems.

My colleague Xin Chen contributed all of those models during his dissertation [Che15]. As this thesis is concerned with the analysis of linear hybrid systems, we will not present non-linear benchmarks in detail, but refer the interested reader to [CSB+15] for further details.

The web page presentation lists all benchmarks along with their property specifications, classified into linear continuous, non-linear continuous, linear hybrid, and non-linear hybrid models. For each model, we also list measures regarding their size. We explain each of the benchmarks in our collection on a separate web page, reference originating literature, provide a model description for downloading in SpaceEx and Flow* input format, and show example plots of the reachable state set generated by those tools.

In the following, we present a selected subset of our benchmark collection which is relevant for this thesis in detail along with further benchmarks not (yet) part of the collection that will be used for experimental evaluation later.

4.2 Linear Benchmarks

Our collection of linear hybrid systems (see Section 3.2) currently holds 11 different benchmarks, with one instance being purely continuous and the other ones being hybrid. In this section, we will present a selected subset of those benchmarks in detail; note that the remainder of the collection is available online. We have selected benchmarks, we think are interesting for tool developers because they are simple in structure, but exhibit particular properties which already might pose challenges when it comes to their verification. The presented systems here are given without safety specifications, but we will give indications on possible sets of bad states. Additionally, we present benchmarks from further collections, which will be used for the evaluation of our methods in later parts of this thesis. To aid identification, we provide a unique label for each benchmark written in true type fonts (e.g., `Bld`) which will be used during experimental evaluation to reference the respective benchmark. Properties of the used benchmarks such as the state space dimension or the number of locations are collected in Table 4.1.

Bouncing Ball

The bouncing ball (`Ball`) is probably one of the most well-known benchmarks often used as an entry-level example of a hybrid automaton [ACH+95]. Consisting of only one location and one looping transition (see Figure 4.1), its structure is relatively simple. As the name suggests, this system models a ball that is dropped from an initial altitude and bounces off the ground. The system models the ball's vertical position only, although there exist non-linear versions of this benchmark, where the ball bounces off wave-shaped surfaces and not only its vertical but also its horizontal position is modeled. In the simplified version, the ball's position x changes according to its velocity whose constant change is subject to earth's gravity. The ball bounces whenever it reaches the ground, i.e., when $x = 0$ holds. The ball itself is not rigid, i.e., the velocity v is not only inverted when bouncing but also dampened by a factor $0 < c < 1$ which varies from instance to instance (in the experiments we use $c = 0.75$).

Even though this benchmark seems to be simple, it already exhibits interesting properties. First of all, the guard condition is a hyperplane. Guard intersections with even lower-dimensional sets might be a problem, depending on the numerical properties and the state set representation used. Additionally, the system in itself exhibits Zeno behavior, as c is a factor such that v and x theoretically never reach zero at the same time—the ball bounces infinitely often in finite time.

Throughout this thesis, we will frequently use the bouncing ball for evaluation and apply the same bounds on the local time horizon $T = 3$ and maximal jump depth of three.

Thermostat

Also considered an entry-level benchmark, the thermostat system [ACH+95] (`Thmo`) has been used as a motivating example in many publications and lectures.

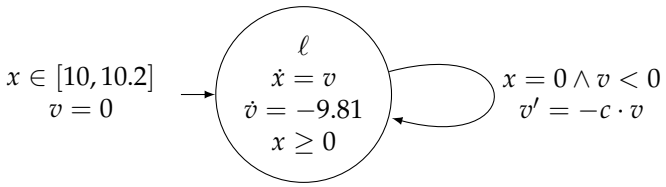


Figure 4.1: The hybrid automaton for the bouncing ball system. For experiments we chose $c = 0.75$.

Consisting of two control modes which reflect a digital controller's states, this system represents a simple cyber-physical system (CPS) and thus is often picked as an introductory example. The system contains a room heating device that can either be turned on or off. The room temperature, which changes according to the influence of the room heating and a constant behavior accounting for the room's volume, is modeled in this system. Based on the specified room temperature bounds (invariants in the modes) the controller switches the heating on or off.

Safety specifications may be the room temperature being within certain bounds, although the system's long-term behavior will allow any room temperature within the bounds specified as invariants in the control modes. Thus, only extended variants in which the controller only monitors the system at specific points in time are of interest—we will use an extended version in the later parts of this thesis (see Chapter 8). A simplified version of this extended thermostat system can be found in Figure 4.2.

For our experiments we use a local time horizon $T = 10$ and bound the global execution time to 10 s using an additional clock.

Two Tank System & Leaking Tank System

Similar to the bouncing ball, the two tank system (`2Tnk`), as well as the leaking tank (`LTnk`) are commonly used benchmarks for hybrid systems safety verification [ACH+95]. Both systems model the filling level of liquid-filled tanks. In both systems the tanks are usually leaking, i.e., the amount of liquid in the tanks is reduced with a constant outflow. Additionally, some instances are equipped with constant inflow. To avoid complete drainage of the tanks, it is possible to refill the tanks in both systems. Furthermore, sometimes valves allow to control the outflow of the tanks.

The two tank benchmark, as the name suggests, models two tanks where only the first one can be refilled. The two tanks are connected via a pipe with an attached valve, i.e., the outflow of the first tank equals the inflow of the second one. Additionally to the constant outflow of the second tank, an electric valve can be opened to increase the outflow of the second tank. The classical version of this benchmark, which is used in Section 6.4 uses two variables and four modes connected by seven discrete jumps (label `2Tnk`').

In contrast to this, the leaking tank benchmark models only one tank. The variant considered in this work models a system in which the tank has a constant inflow of liquid. A constant outflow, which is smaller than the inflow models

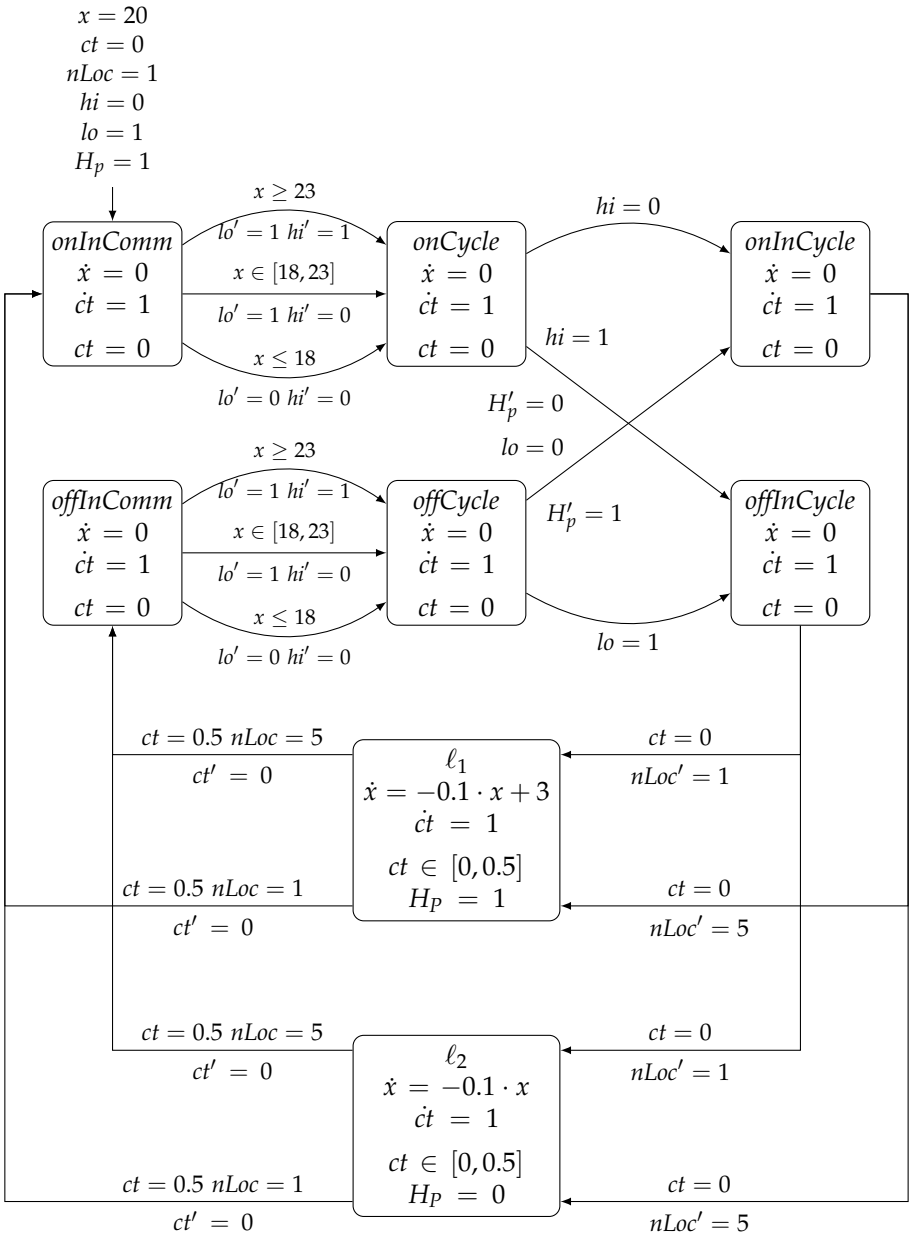


Figure 4.2: The hybrid automaton for the PLC-extended thermostat. In the image we have omitted the variables $high, low, H_{PLC}, nLoc$, and H, which have a flow of zero and represent the internal state for the PLC-controller. Furthermore we omit the additional clock which is used to bound the total execution time of the system.

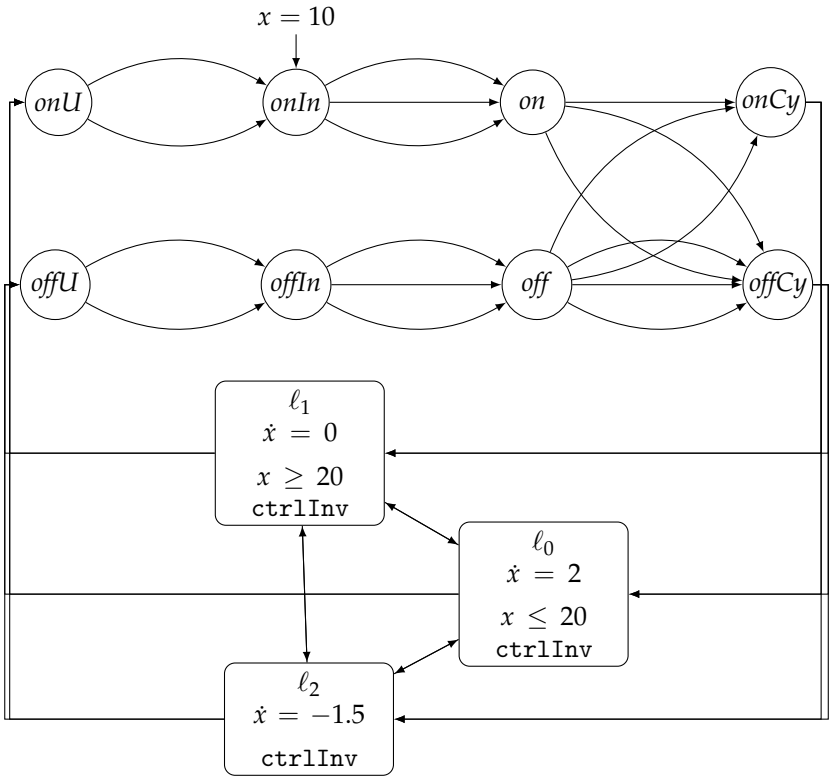


Figure 4.3: Simplified automaton for the leaking tank benchmark equipped with an explicit controller. We have omitted state variables of the controller and simplified its internal structure. Arbitrary switching in the plant (locations ℓ_0, ℓ_1, ℓ_2) is prevented by controller variables (`ctrlInv`).

the leakage of the tank. Additionally, a valve can be opened to increase the outflow and prevent the tank from overflowing.

Possible safety specifications usually argue about the filling level of the modeled tanks, which should be kept within certain bounds (the tanks should neither overflow nor run empty). In the later part of this work, we will consider extended versions [SNÁ17], where a controller has been added, which only checks the filling levels periodically (see Chapter 8). A simplified automaton of the extended versions with explicit controllers for the leaking tank benchmark can be found in Figure 4.3, however, the automaton of the two tanks benchmark with a controller is too large to be depicted here (34 locations, 296 discrete jumps). Model files for both extended systems can be found on our webpage.

In our experiments, we use a local time horizon of 20 s and bound the global execution time to 20 s via an additional clock.

Rod Reactor

The rod reactor system (`Rods`) is a simplified model of the reactor core of a nuclear power plant depicted in Figure 4.4 [ACH+95]. Such reactors feature

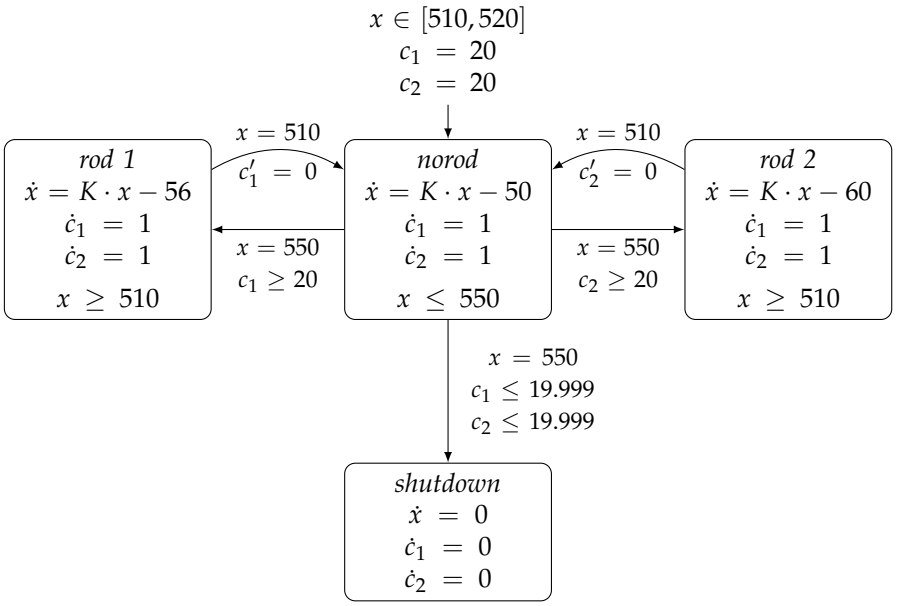


Figure 4.4: The hybrid automaton for the rod reactor system. For experiments we chose $K = 0.1$.

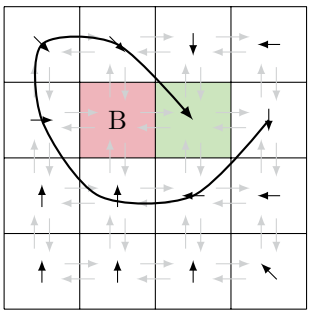
fuel rods that heat the surrounding water to power turbines, and cooling rods made from a material that inhibits the radioactive reaction, which are inserted in between the fuel rods to be able to regulate the temperature. The system models the reactor temperature depending on the state of two different cooling rods that can be extended or retracted. The cooling rods are made from different materials, i.e., they have a different influence on the temperature dynamics when used. The controller can decide which cooling rod to use whenever the temperature exceeds a fixed boundary, but cannot use the same rod for a fixed amount of time after it has been used.

This benchmark is interesting for tool developers, as it is among the smallest benchmarks (with respect to the number of locations and the state space dimension) to introduce non-deterministic switching between different control modes, i.e., there exist several paths with a different discrete fragment. Furthermore, depending on the initial set, the time duration in which a guard is enabled after the first time elapse is relatively long. This makes the benchmark interesting as it raises the demand for approaches that manage to cluster state sets satisfying the guard of a discrete jump.

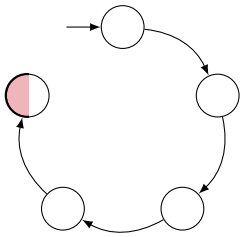
The time horizon of this benchmark is set to $T = 20$ and a maximum of five jumps is analyzed.

Navigation

The navigation benchmarks [FI04] (`Nav`) are a family of 30 instances of different sizes. The systems model a point-mass moving on a two-dimensional plane. The x- and y-position of the mass, as well as the respective accelerations $\dot{x}$ and



a) Schematic structure of a navigation benchmark instance.



b) Schematic structure of the five-dimensional switching system (`Sw5`).

Figure 4.5: Schematic overview of the navigation- and the switching-system.

$\dot{y}$ are modeled as system variables—all instances have a state space dimension of four. The plane itself is subdivided into cells, each of which is associated with a specified acceleration in x- and y-direction. Neighboring cells are connected via discrete jumps with the cell-boundary representing both the guard and the invariant condition, which forces control to switch mode for trajectories leaving the cell. Instances vary from planes subdivided into 3×3 cells up to instances with 25×25 cells. A schematic illustration is given in Figure 4.5a.

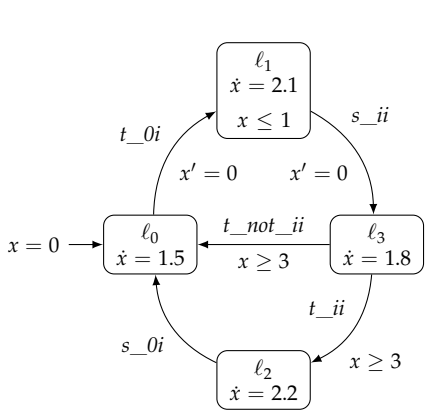
As the boundary between two neighboring cells is part of both cells, this allows for infinite switching between the cells as the cell-boundaries also represent the guard sets for the discrete jumps between the cells. Consequently, the model exhibits Zeno-behavior. Also, the initial sets specified for the original instances are usually all states contained in a particular cell. This renders the benchmark very difficult, as all outgoing discrete jumps are enabled directly and the search tree structure displays extensive branching during the analysis for later control modes as well.

All instances are equipped with two special cells: one cell representing the set of bad states and another cell representing a set of so-called "good states". While the semantics of the former is clear, the latter can be modeled as a location with no flow where control may stay infinitely long. The original idea of a set of good states is to not only validate the absence of unwanted behavior through avoiding bad states but also to be able to ensure that all trajectories end up in a particular good, absorbing location, i.e., to validate that a specific property eventually holds.

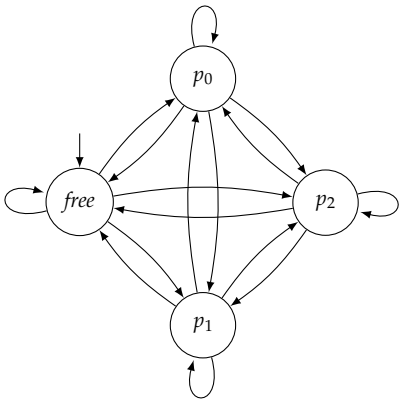
For our experiments, we use a local time horizon $T = 3$ and a maximal jump depth of eight jumps.

Switching System

The artificial switching system (`Sw5`) consists of five control modes ℓ_i, $i = 0,\ldots,4$, arranged in a linear setup, i.e., transitions $\ell_i \xrightarrow{e} \ell_{i+1}$ are featured in the model. The system models five variables, and the flow specification for each variable x_i depends on all system variables. The dynamics in each control mode are created randomly using MATLAB's `rss()` function, thus the



a) Automaton for process i in `Fish`.



b) Automaton modeling the critical section access in `Fish`. All edges are label-synchronized with the processes.

Figure 4.6: Components of Fisher's mutual exclusion protocol benchmark (`Fish`).The resulting system is a parallel composition of the automata of several processes and the critical section automaton

system is entirely artificial. The system is stabilized with an LQR controller to guarantee convergence to a stable region in the last mode. However, the system shows little robustness such that even small over-approximation errors induce diverging behavior—this can, for instance, be observed when using boxes as a state set representation for this benchmark. A schematic illustration of the system's hybrid automaton is given in Figure 4.5b.

For the safety verification, we define a set of bad states as a half-space in one variable in the last location. Note that the number of jumps is naturally bounded by the structure of the system such that at most four jumps can be observed. A local time horizon of $T = 1$ is used during the analysis.

Fisher's protocol

This benchmark (`Fish`) taken from [ACH+95] describes a system in which processes attempt to access a shared resource mutually exclusively. The access to this resource is managed by a variant of Fisher's mutual exclusion protocol [Lam87]. Each process is modeled by four modes (see Figure 4.6a), an additional component models the access to the critical section via label synchronization with the process-automata (see Figure 4.6b). We consider three processes, but theoretically, this benchmark is scalable in the number of processes. To make the model more appealing, internal clocks for the processes are modeled with drifts, which means they operate at different rates. The resulting model is a parallel composition of a fixed number of processes. While variants of this benchmark involve different numbers of processes, in this work we consider an instance with three processes similar to the one used in [BRS17; BRS18].

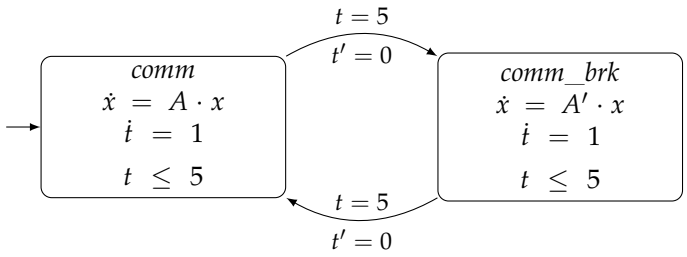


Figure 4.7: Structure of the hybrid automaton for the vehicle platoon benchmark.

The system should guarantee mutual exclusion at all times, i.e., the set of bad states is defined by any state in which more than one process accesses the critical section at the same time.

For our experiments, we use a maximal jump depth of eight jumps and a local time horizon $T = 12$.

Vehicle Platoon

Also included in the provided benchmark collection, the vehicle platoon system (`Pltn`) was originally proposed in [BKG+09]. In this benchmark, a platoon of several vehicles following a human-controlled leader vehicle is modeled. The successive vehicles attempt to keep a certain distance between each other based on the information of the surrounding vehicles obtained by radio communication. In the original version of the benchmark, communication failure is modeled, which can happen at arbitrary points in time for an arbitrary time period. Several variants of this benchmark exist that differ in the number of vehicles (three, five, and ten vehicles) and several models of communication failure. In this thesis, we consider a platoon of three vehicles where communication breaks down deterministically as used in [ABC+17; ABC+18]. Furthermore, as we do not consider non-autonomous hybrid systems in this thesis, external inputs that are present in the original model have been removed. A simplified version of the corresponding hybrid automaton is depicted in Figure 4.7.

The used benchmark has only two modes but requires nine variables to model the dynamics, which makes it more challenging than low-dimensional models. A range of several safety specifications exist, which vary the minimal distance threshold between the vehicles.

In our experiments, all executions are bounded to a length of 20 s, realized by an additional clock. To analyze all behavior within this time horizon, we use a maximal jump depth of 1000 jumps and a local time horizon $T = 20$.

Building

The building system (`Bld`) presented in [TNJ16] is a large-scale dynamical system. This particular system models the static model of the Los Angeles University hospital building with eight floors as a beam model and focuses on the forces acting on the building and its resulting movements.

Table 4.1: Properties of the benchmarks used for the evaluation of the methods and approaches presented in this thesis.

model	dimension	# locations	# jumps
`Ball`	2	1	1
`Thmo`	8	8	18
`LTnk`	12	11	34
`2Tnk`	22	34	296
`Rods`	3	4	5
`Nav09`	4	16	48
`Nav11`	4	25	80
`Sw5`	5	5	4
`Fish`	4	152	324
`Pltn`	9	2	2
`Bld`	48	1	0

With 48 state variables, it can be considered a medium-sized benchmark suited to test the scalability of an approach towards higher-dimensional state spaces. Variants with fewer variables exist, i.e., the number of modeled floors is reduced. This system only has one mode, i.e., it is a purely continuous system, i.e., no handling of discrete jumps is required. However, the high dimensionality makes it difficult to visualize graphically, therefore we refer to the webpage for the SpaceEx model specification[3].

Safety criteria put bounds on certain variables—we use the same safety specifications as used in [ABC+17; ABC+18]. Similarly, we use the same time horizon $T = 40$ in our experiments.

[3] `https://cps-vo.org/node/34059` (checked July, 30^{th}, 2019)

5

Challenges in Hybrid Systems Safety Verification

Having the theoretical foundations at hand, we can focus on practical limitations when it comes to the applicability and implementation of a general hybrid systems safety verification method. In this chapter, we address some of the challenges one has to face during the creation of a hybrid systems safety verification method and its implementation in a tool. While some of the presented challenges are more of a general nature, we want to focus on approaches implementing flowpipe-construction-based reachability analysis for linear hybrid systems and potential pitfalls relevant to this method. Most parts of this chapter are based on results from [SÁC+15] and may contain direct excerpts without explicit quotation.

5.1 State Set Representation

Approaches based on flowpipe-construction-based reachability analysis raise the demand for an efficient state set representation during computation (see Section 3.4). As mentioned earlier in this work, the choice of the state set representation is always a trade-off between computational complexity and precision. In current approaches mostly geometric but also few symbolic representations have been used. The most commonly used ones are boxes [MKC09], convex polyhedra [Zie95; CÁF11], ellipsoids [KV00], oriented rectangular hulls [SK03], orthogonal polyhedra [BMP99; Dan00], support functions [LG10], Taylor models [MB09; CÁS12], template polyhedra [SDI08; BFG+17], and zonotopes [Gir05]. Boxes and polytopes are frequently used; also support functions and zonotopes are prominent for models with linear ordinary differential equations (ODEs), whereas Taylor models can also be used for non-linear ODEs. However, none of the representations offer an optimal solution, as they have individual strengths and weaknesses, mainly in the representation size and in the efficiency of certain operations needed during the reachability analysis for linear hybrid systems. Although several tools use conversions between representations for certain computations, context-sensitive approaches have been still missing. In this thesis, we address this issue in Chapter 8.

As an example, an adaptive choice of the state set representation based on the system's dynamics in different locations can be a promising approach. Also,

automated dynamic conversion to reach an optimal trade-off between precision and efficiency during computation using an iterative refinement technique is not yet supported—approaches to guide the search [BDF+13; BDF+16] which refine a single representation exist but to our knowledge switching between representations is not considered in current approaches. Also here we propose some improvements related to a CEGAR-method (see Section 5.10)

Furthermore, there is rare support for non-convex representations. Last but not least, most representations are over-approximative, and therefore applicable for safety verification. However, for proving unsafety, novel under-approximative computations would be of help.

In Chapter 6, we will introduce our `C++` library for state set representations where we implement the most prominent state set representations currently used in flowpipe-construction-based reachability analysis along with details on the implementation of required operations. Furthermore, we will address some of those ideas for future development in Chapters 7 and 8.

5.2 Precision

For systems where the distance between the reachable and the unsafe states is small, the used precision can be crucial for the outcome of the reachability analysis. If the outcome is inconclusive (the over-approximation intersects with the unsafe state set), the only solution has been to restart the analysis from scratch with new parameters which lead to an error reduction (e.g., reduction of the time-step size in the flowpipe construction, see Section 3.4). However, as higher precision comes with longer running times, the new parameters must be chosen carefully by the user. An automatic adaptation of the analysis parameters would be not only more user-friendly but could also be applied dynamically to refine the search along only those paths which led to an intersection with the unsafe state set, instead of executing the complete analysis with a high-precision configuration. We will address this topic in a later part of this dissertation (see Chapter 7) in the context of a counterexample-guided abstraction refinement (CEGAR)-based refinement approach.

5.3 Fixed-point Recognition

Recognizing fixed-points in the reachability analysis, i.e., when the whole reachable state set of a hybrid system is already checked for safety, enables the solution of the unbounded reachability problem. However, to detect fixed-points, a considerable number of state sets need to be stored, and successor sets must be tested for inclusion. As this comes at high costs, current tools use only heuristic checks for fixed-points, i.e., checking whether discrete jump successor states are already contained in the computed approximation of the set of reachable states. A more systematic check would require highly efficient storage of state sets and fast operations on them—a possible approach could use memory-efficient under-approximations in a representation with fast inclusion and intersection computations (e.g., boxes).

5.4 Large Uncertainties

Uncertainties can be included in the models when e.g., some coefficients of the dynamics cannot be fixed precisely, or in the presence of time-varying external inputs like natural forces or users. Though systems with bounded uncertainties can be verified, models with large uncertainties are one more challenge in the verification of hybrid systems. Each uncertainty introduces a bloating factor that is carried onward and even aggregated during the computation of the reachable set. Although a few approaches were proposed to overcome these limitations (see, e.g., [RMC09]), most tools have problems to find conclusive answers for models with large uncertainties.

5.5 Zeno Behavior

Whenever it is possible to execute an infinite number of jumps in a finite amount of time, we observe Zeno behavior. Naturally, no real system exhibits Zeno behavior. However, it is hard to avoid Zeno paths in modeling, and sometimes in large systems it is even hard to detect. In [AS05] the authors distinguish between chattering Zeno (infinite jump sequences with zero dwell time) and genuine Zeno (infinite jump sequences with nonzero dwell time in-between converging to zero) behavior.

Examples for chattering Zeno behaviors can be found in switching systems, where the state space is divided into grids, each grid having its own dynamics, modeled by an own location (see for instance the navigation benchmark example in Section 4.2). Switching between different cells of a grid does not modify the continuous state and is always possible whenever the current state lies at the boundary between two cells. Therefore, infinite back-and-forth switching on boundaries can happen in such models, causing a problem for reachability analysis if the reach-set approximation is not idempotent: even if no new states are reached, successor states in a sequence of jumps may grow and even diverge as the approximation errors accumulate. If the reach-set computation is exact (such as in HyTech or Phaver), chattering Zeno has no particularly adverse effect (it may increase the number of image computations necessary to reach a fixed-point).

Genuine Zeno can be problematic for any computation that follows the execution of the system, because any finite number of successor computations may not be able to cover all reachable states. Using over-approximations may resolve the problem if they cover the limit points of the sequence. This can be achieved automatically with widening operators [CH78]; here, the difficulty lies in keeping the over-approximation reasonably small [MFK09].

5.6 Non-convex Invariants

Most tools require that the invariants of the locations are convex sets, mainly for representation reasons. However, similarly to programs that might have disjunctions in loop conditions, non-convex invariants also appear in hybrid system applications. Though one can apply model transformation to eliminate

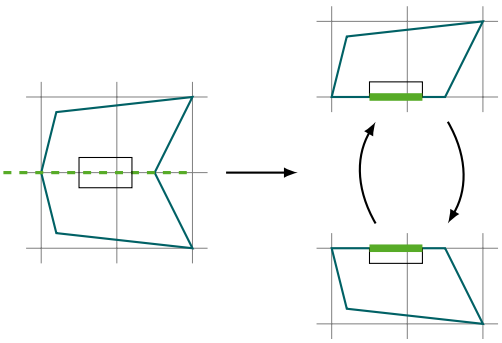


Figure 5.1: The split of a location with a non-convex invariant (left) into two locations with convex invariants (right) might introduce Zeno behavior.

non-convex invariants by splitting the non-convex set into convex subsets and introducing a new location for each convex subset, with this approach the models are extended with Zeno behavior, hardening their analysis (see Figure Figure 5.1). An efficient analysis without such model transformations could be enabled, for example, by non-convex state set representation techniques such as orthogonal polyhedra [Dan00].

5.7 Urgent Transitions

Urgency enforces a certain behavior as soon as a condition is satisfied—in this context, urgent jumps need to be taken as soon as the guard condition is satisfied. On the other hand, an urgent location ℓ enforces that no time passes while control stays in ℓ.

Invariants are one possibility in modeling to force the control to move from one mode to another (see Example 5.1) to realize an urgent location. Another possibility are urgent transitions, which prohibit time elapse as soon as they are enabled. Urgent transitions have the advantage that they make the reason for the mode change more visible (observable), and therefore they are sometimes preferred instead of the usage of invariants[1]. However, most tools do not support urgent transitions, though their analysis would even reduce the computation effort: both the expensive computations of intersections with invariants as well as the computation of flowpipes from those state sets which are included in the guard of an outgoing urgent transition become superfluous. On the other hand, as shown in [MF14] urgent transitions may complicate the analysis: in case the guard of an urgent transition is only enabled by a subset of the system's trajectories, analysis methods need to support (over-approximative) set difference computations (see Figure 5.2) to account for the shadow-like cutoff of trajectories satisfying the guard of the urgent transition.

[1] We are not aware of any trivial conversion between models using invariants and models using urgent transitions, mostly because an invariant can force the control to leave a location at its boundary only if further time elapse would violate it.

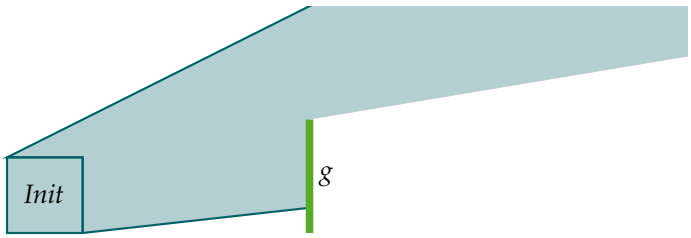
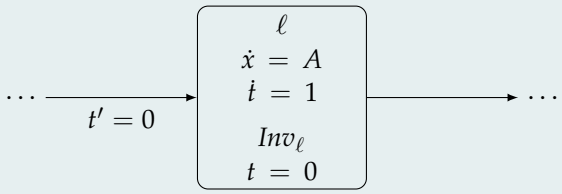


Figure 5.2: An urgent transition with a guard g which is only enabled by parts of the system's trajectories (petrol) starting from *Init* [MF14].

Example 5.1: Urgent Locations

We can enforce control to leave a certain location ℓ after a specific amount of time, even zero time using an additional clock t.



5.8 Compositionality

Large systems are usually modeled compositionally as a set of modules running concurrently. Most available tools build the parallel composition of the modules to get a non-compositional model, which can be subsequently analyzed. However, the composition results in high-dimensional systems, which pose challenges for the analysis (see Definition 3.4). For instance, in [SLÁ+18] we have worked on the analysis of robot swarms in which every robot is modeled by a hybrid automaton and the swarm is represented as the parallel-composition of all robots being part of it. Our results from this work indicate that direct, static parallel composition results in intractably large systems requiring lots of memory and a significant amount of time to compute the resulting product automaton.

Compositional analysis techniques would be advantageous, but there is no straightforward way to extend the available techniques to support compositionality. The tool SpaceEx allows to model systems in a compositional, hierarchical way via *hybrid I/O automata* [DF13]. As assume-guarantee methods proved to be useful in program verification, it might also be a promising option in hybrid systems reachability analysis. However, when we aim for push-button approaches, suitable assumption-commitment specifications should be derived automatically. An approach towards this direction has been proposed in [BDF+13] in which an abstraction of a system is obtained by location merging

in the tool SpaceEx. Another possibility could be to analyze the concurrent modules simultaneously and communicate between the concurrent analyses on synchronization-relevant computations using, e.g., partial order reduction techniques.

5.9 Counterexamples

Although a few tools, for example KeYmaera [PQ08], can provide counterexamples for unsafe models, most tools do not have this functionality. However, counterexamples are extremely important and provide valuable information for system developers to correct unsafe designs. Using simulation-based approaches, in [NÁC+13] the authors present ideas on heuristics for using simulation to obtain counterexample traces for hybrid systems.

Furthermore, counterexamples play an essential role in CEGAR where they are used for further refinement and recovery (see below).

5.10 CEGAR

Frequently used in various other research areas, counterexample-guided abstraction refinement (CEGAR) is not yet established in the field of hybrid systems. Using a relaxed version of the problem can introduce a significant speed-up in verification. In case the verification fails, a counterexample path is used to refine relevant components of the model. First attempts similar to CEGAR have been used to guide the analysis [BFG+12; BDF+16], or to refine the representation of state sets in case of hybrid systems with constant derivatives (LHA I) using template polyhedra as a state set representation [BFG+17]. In this dissertation, we are presenting a more general approach towards CEGAR-based refinement during the analysis which combines some of the features published in earlier works but also extends and generalizes the method for hybrid systems reachability analysis (see Chapter 7).

5.11 Parallelization

Regarding the efficiency of the reachability analysis of hybrid systems, the current main focus lies on improving the efficiency of sequential algorithms. Approaches for parallelization are rare and not yet well understood. However, the exploitation of multi-core hardware systems could help to improve the scalability and the applicability of available technologies to large-scale systems. The tool Xspeed [GRB+18] is to our knowledge the first to implement a parallelized reachability analysis based on flowpipe construction. In their approach, the authors parallelize in two ways: (i) different flowpipes are analyzed in parallel and (ii) via hierarchical time discretization a method for parallelization of the computation of a single flowpipe is presented. Our previously mentioned CEGAR-approach supports the parallelization of the first kind (see Section 7.5).

5.12 Modeling Language Expressiveness

To make hybrid automata as a modeling language more attractive and usable for a broader range of applications, further extensions regarding expressiveness should be considered. For example, the more general class of *cyber-physical systems (CPSs)* includes distributed hybrid systems where additionally to discrete and dynamic aspects, communication also plays an important role. Spatio-temporal hybrid automata [SL13] are a possible extension in this direction, supporting the modeling of communication and other spatial aspects.

Another relevant aspect is *randomized* behavior, which can affect either the dynamics of a system via stochastic differential equations [BL06] or the discrete behavior via probabilistic transitions [Spr00]. The later can involve probabilistic properties regarding the choice between enabled transitions. A pioneer tool in this area has been PROHVER [ZSR+10], which implements analysis algorithms using a transformation of probabilistic hybrid automata to hybrid automata without probabilistic components. Though a few further approaches have been proposed since then, this thread of research is still in its infancy.

Part II

State Set Representations

6

A Library for State Set Representations

"Im großen Garten der Geometrie kann sich jeder nach seinem Geschmack einen Strauß pflücken."

— David Hilbert

In the previous sections, the general algorithm for flowpipe-construction-based reachability analysis has been introduced in which sets of geometric shapes over-approximate state sets. The presented approach relies on a fixed set of operations on state sets which is invariant to the representation of state sets. This raises two questions:

1. How can we efficiently and precisely represent state sets in a way suitable for flowpipe-construction-based reachability analysis?
2. How can we perform the required operations on said state sets efficiently?

In the following chapter, we present our approach towards answering both of these questions. We will use our insights from the previous chapters and experience of flowpipe-construction-based reachability analysis to derive a finite set of *operations* on state sets, which enables us to implement a state-of-the-art reachability analysis method for linear hybrid systems. Additionally, we present a collection of the most commonly used *state set representations* in flowpipe-construction-based reachability analysis for linear hybrid systems and highlight their specific properties. Furthermore, we present technical details on how to *implement* the presented state set representations along with the required set of operations and improvements thereof specific to the respective representation.

All findings presented in this chapter are implemented in the `C++` library HyPro which is one of the main contributions of this thesis and which is publicly available[1]. This chapter provides an overview of the landscape of state set representations in hybrid systems reachability and is based on the work of many people. Parts of this chapter are taken from [SÁB+17] in which we initially presented HyPro and will not be explicitly indicated. Instead, for each presented state set representation, we will indicate our contribution in a separate section.

[1] `https://github.com/hypro/hypro` (checked July, 30^{th}, 2019)

Related Work. Different state set representations have been considered for the usage in flowpipe-construction-based reachability analysis in the research community, amongst them boxes [RS05; MKC09], convex polytopes [Zie95; Fre05], ellipsoids [KV00; KV07], oriented rectangular hulls [SK03], orthogonal polyhedra [Dan00], template polyhedra [SDI08], and zonotopes [Gir05]. The Matlab-toolbox CoRA [Alt15] provides implementations of boxes, convex polytopes, and zonotopes for the reachability analysis of hybrid systems. Apart from this collection, all previously mentioned representations usually are implemented and provided individually which on the one hand allows for a specialized, more efficient implementation of reachability analysis methods, but on the other hand does not allow to combine and compare different state set representations.

This chapter is organized as follows: After a brief overview of the structure of HyPro, we will isolate a small set of required operations on state sets and give an intuition on how these operations are applied in the context of state-of-the-art flowpipe-construction-based reachability analysis methods. Starting from Section 6.2 on, we will present our collection of state set representations in HyPro and provide insights on how the previously deduced operations can be implemented efficiently for the different representation types. To be able to use various representations during analysis, we present over-approximative conversion methods between the different state set representations in Section 6.8. Additionally, to the collection of state set representations, HyPro also provides various utility functions and data structures, which will be briefly discussed at the end of this chapter in Section 6.9. The state set representations in HyPro are based on some standard but also on novel algorithms for the computation of different operations.

6.1 HyPro

To analyze the strengths and weaknesses of different state set representations in hybrid systems reachability analysis, we implemented the `C++` library HyPro. The central element of this library is the collection of the most commonly used state set representations, unified by a common interface of operations on those representations. In combination with over-approximative conversion methods between all representations (see Section 6.8), we enable the user to facilitate switching between different state set representations even during running time of a flowpipe-construction-based reachability analysis method. Along with implementations of the most common state set representations, HyPro provides additional tooling to enable users to implement their own reachability analysis method quickly. This includes utility such as a parser, a plotting framework, data structures for hybrid automata, logging, and many more (see Section 6.9). A classical flowpipe-construction-based reachability analysis method as presented in Section 3.4 as well as advanced data structures and algorithms (see Chapter 7 and Chapter 8) are provided as well. The general setup and interplay of components in HyPro is illustrated in Figure 6.4. In the following, after deducing required operations on state sets, we will present the provided state set representations in detail, followed by a more general presentation of the utility shipped with HyPro.

a) Convex closure of set-union.

b) Set intersection.

Figure 6.1: Sketch of the set operations union and intersection.

Operations

The synthesis of a common set of operations on state sets allows to unify their implementation under a common interface (see Figure 6.4, *GeometricObject*). Here we shortly review required operations on state sets and highlight their appearance in flowpipe-construction-based reachability analysis for linear hybrid systems. Let in the following $A, B \subseteq \mathbb{R}^d$.

Union. The union $A \cup B$ is formally defined as the set

$$\{x \mid x \in A \vee x \in B\} \ .$$

As convex sets are not closed under union, we compute the *convex closure* cl of the union of two sets $cl(A \cup B)$, which is the smallest convex set containing $A \cup B$ (see Figure 6.1a). In the following, we imply closure when we refer to set-union operations.

With exceptions, the computation of set union usually is less-frequently used during analysis such that improvements in its implementation usually have little influence on the running time of a flowpipe-construction-based reachability analysis method. Computing the union of two state sets is used during the creation of the first flowpipe segment (see Section 3.4). Furthermore, set-union is required in case state sets are aggregated when taking a discrete jump (see Section 3.4).

Intersection. The intersection $A \cap B$ (see Figure 6.1b) is formally defined as the set

$$\{x \mid x \in A \wedge x \in B\} \ .$$

In general, convex sets are closed under this operation; however, depending on the representation, computing the closure for this representation might be necessary (e.g., for zonotopes, see Section 6.5). The intersection between two state sets is required when verifying guard- or invariant predicates which can be represented as state sets. Furthermore, validating safety specifications is performed via the intersection of the set of reachable states with the set of bad states. This indicates that reducing the computational effort of set intersection, in general, has a considerable influence on the running time as this operation is

a) Set intersection with a half-space.

b) Affine transformation.

Figure 6.2: Set operations intersection with a half-space and affine transformation.

frequently used. This influence is reduced, when purely continuous systems are analyzed, as potential guard intersections for discrete jumps do not occur.

Note that in many cases, guards, invariants, and bad states are represented by a conjunction of half-spaces which may be treated differently and more efficiently. Fixed-point detection methods require computing the intersection between a state set (ℓ, N') and the previously computed sets of reachable states of a hybrid system $\mathcal{H}$ to detect a fixed-point. If we know that (ℓ, N) is safe and $(\ell, \mathrm{N}') \subseteq (\ell, \mathrm{N})$ holds then no new information can be derived from computing successor states from (ℓ, N'). This renders fixed-point detection a use-case for set-set intersection computation to check containment.

Intersection with Half-spaces. In many cases, guards, invariant conditions, and bad states are specified as a conjunction of half-spaces, i.e., a convex polytope in $\mathcal{H}$-representation $H \subseteq \mathbb{R}^d$ (see Figure 6.2a). One way of computing the intersection between H and a given d-dimensional state set A can be done via conversion, i.e., either converting A to a $\mathcal{H}$-representation or converting H to the representation type of A. After conversion, the intersection can be performed using set-set intersection as presented before.

However, computing the intersection of H and A directly may be faster in some cases, when a costly conversion can be avoided and the state set representation allows for efficient intersection with half-spaces. Additionally, tests for emptiness of the result may be performed during computation of $A \cap H$; in HyPro this operation is implemented by the meta-operation `satisfiesHalfspaces(H)` returning whether $A \cap H \neq \emptyset$ holds. During this operation, some of the presented state set representations may even detect, whether $A \cap H = A$ holds. In Section 7.3 we show how this information can be exploited to improve flowpipe-construction-based reachability analysis during running time. As indicated before, improvements in state set intersection with half-spaces usually have a significant influence on running times during the analysis of hybrid systems, as this operation is required frequently in the presence of guarded transitions and invariant constraints.

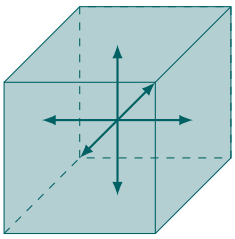


Figure 6.3: The exponential nature of the Minkowski sum: the repeated application on sets influences the complexity of the representation for the resulting set. Here: the addition of three orthogonal line segments (arrows) results in a set which has 2^3 vertices and $2 \cdot 3$ facets.

Minkowski Sum. The Minkowski sum is sometimes also known as *dilation* (computer graphics) and represents the set-theoretic equivalent to addition. The Minkowski sum of A and B is defined as

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

Convex sets are closed under the Minkowski sum. However, due to its exponential nature, repeated application has a strong influence on the complexity of the resulting output. For instance, the Minkowski sum of k pairwise orthogonal line segments results in a k-dimensional box with 2^k vertices and $2 \cdot k$ facets (see Figure 6.3).

The inverse operation, the *Minkowski difference*, also known as *erosion* (computer graphics) is defined analogously as

$$A \ominus B = \{a - b \mid a \in A \wedge b \in B\}$$

but plays a minor role in this work and thus is not implemented for most state set representations (in HyPro only boxes implement this feature).

Affine Transformation. Affine transformations are frequently applied during flowpipe construction. For a d-dimensional set S, its affine transformation $\mathcal{A} \cdot S + b, \mathcal{A} \in \mathbb{R}^{d \times d}, b \in \mathbb{R}^d$ is defined as the set

$$\{\mathcal{A} \cdot s + b \mid s \in S\} \ .$$

Affine transformations can be described as a combination of rotations, scalings, and skewings by a matrix $\mathcal{A} \in \mathbb{R}^{d \times d}$ and a translation whose parameters are given by the vector $b \in \mathbb{R}^d$. A sketch of a transformation implementing a rotation and a translation is depicted in Figure 6.2b. Affine- and linear transformations are used during the analysis to compute time successor sets for a fixed time step size (see Section 3.4). Furthermore, affine and linear transformations are often used to realize reset functions, which are assigned to discrete jumps and allow for updating state sets upon discrete mode changes of the modeled system.

A Note on Closure. The formal specification of different operations on state sets neglects any closure properties concerning the utilized state set representation. We do not only need state sets to be convex and thus require convex closure, but we also enforce each implemented operation to provide closure with respect to the underlying state set representation. Ensuring closure with respect to the representation simplifies the usage of a particular state set representation but also may imply additional effort and introduce further over-approximation errors. Furthermore, to maintain the soundness of the verification approach, we have to ensure that every inexact implementation of an operation on state sets guarantees *over-approximation* of the exact results. In the following sections, we will provide details on the implementation of the most common operations with respect to these properties. If not stated otherwise, we assume that the operands of all operations which involve more than one state set are of the same state set representation.

In the following, we describe the state set representations implemented in HyPro, including both, the data type for storing state sets as well as the operations on these data types. To simplify notation, we overload the meaning of the symbols for operations on sets and use the same syntax for the operations on the representations. This also holds for the closure operator which is defined to be the convex hull for sets, but we use it in the context of a representation type as an over-approximating representation containing the convex hull. Note that though we use the same syntax, the result of an operation on state sets does not need to be equal to the result of the same operation on even exact representations of the same sets. In this work, we discuss only over-approximative operations on representations that means that the resulting representations encode a larger set containing the exact result. We use subscript notation, i.e., A_B to indicate that we refer to the representation of a set A; in this case A_B refers to the box representation of A (see Section 6.2). Furthermore, we use $\mathcal{Set}(A_B)$ to refer to the set of states contained in A_B and the operator "=" to denote set-equivalence, for instance we use $\mathcal{Set}(A_B) = A$ to state that the state set represented by the box-representation A_B of a set A is equivalent to A (this is generally not the case).

6.2 Boxes

The concept of a box is widely used in different contexts throughout computer science. In the context of satisfiability solving, interval constraint propagation (ICP), for instance, uses boxes to represent a possible solution space, which is narrowed during the computation [VMK97; MKC09; GKC13; Sch13]. In this context, a box represents an ordered sequence of intervals without geometric interpretation, assigning each variable of the input problem a separate (thus orthogonal to the other variables) solution space (see Figure 6.5a). In hybrid systems reachability analysis, a box can be used in the same way, thus implicitly removing any dependencies between the single state space dimensions. Consequently, the box is one of the most simplistic state set representations usable in hybrid systems reachability analysis.

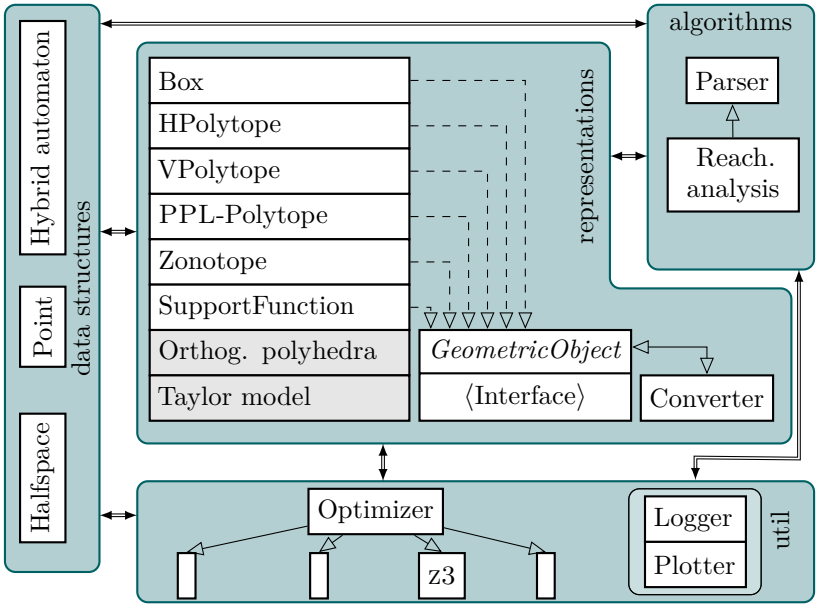


Figure 6.4: The organizational structure of HyPro. Additional to state set representations, further data structures, additional utility functionality and a state-of-the-art reachability analysis method are included.

Definition 6.1: Box

A vector A_B of d real-valued intervals $A_B = (A_0, \ldots, A_{d-1}) \subseteq \mathbb{I}^d$ represents a d-dimensional box

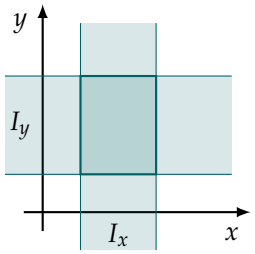
$$A = \left\{ x \;\middle|\; \bigwedge_{i=0}^{d-1} x_i \in A_i \right\}$$

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

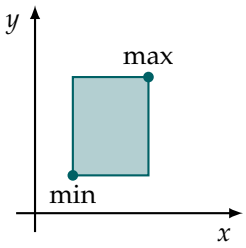
We use the operator $Box(A)$ where $A \subseteq \mathbb{R}^d$ to obtain the box representation A_B of A. Note that in general $Set(A_B) \supseteq A$ holds.

In the following, we assume that the bound types of all intervals are weak bounds. This assumption is rooted in the technical representation of an interval.

As an alternative to storing a vector of intervals, a box can be represented by two points, which contain the lower respectively upper bounds for each dimension (see Figure 6.5b). For two box representations $A_B = (A_0, \ldots, A_{d-1}), B_B = (B_0, \ldots, B_{d-1})$, where $A_i, B_i \in \mathbb{I}$, the relation $A_B \subseteq B_B$ holds whenever for all $x \in A_B$ also $x \in B_B$ holds, or equivalently, if for every A_i, B_i the relation $A_i \subseteq B_i$ holds.



a) A box represented by a vector of intervals.



b) A box represented by two points.

Figure 6.5: A box represented either by a vector of intervals or by its minimal and maximal point.

Operations

In the following sections, we will highlight details of our implementation for boxes with a focus on the operations *affine transformation* and *intersection with a half-space* as they are non-trivial to implement efficiently and provide opportunities for improved algorithms. Assume in the following two d-dimensional box representations $A_B = (A_0, \ldots, A_{d-1})$ and $B_B = (B_0, \ldots, B_{d-1})$ where $A_i, B_i \in \mathbb{I}$.

Test for Emptiness

Testing whether a box representation A_B is empty can be done for each dimension, i.e., each interval individually linearly in the dimension of the ambient state space:

$$A_B = \emptyset \Leftrightarrow \exists i \in \{0, \ldots, d-1\}.\ A_i = \emptyset.$$

Analogously, detecting whether a box is unbounded can be done individually on the intervals as well.

Union

The closure of the union of A_B and B_B can be computed in a component-wise fashion, as per definition the single state space dimensions are independent. The closure of the union of the box representations A_B, B_B represents a valid over-approximation of the convex closure of the union of the sets represented by A_B and B_B:

$$cl(A_B \cup B_B) = (cl(A_0 \cup B_0), \ldots, cl(A_{d-1} \cup B_{d-1}))\,,$$

where $cl(A_i \cup B_i) = cl([\underline{A_i}, \overline{A_i}] \cup [\underline{B_i}, \overline{B_i}]) = [\min(\underline{A_i}, \underline{B_i}), \max(\overline{A_i}, \overline{B_i})]$ (see Section 2.4). Therefore $Set(cl(A_B \cup B_B)) \supseteq cl(A_B \cup B_B)$ holds. This definition naturally extends to computing the convex closure of sets of boxes.

Intersection

Analogously to the union, the intersection of A_B and B_B can be computed in a component-wise fashion:

$$A_B \cap B_B = (A_0 \cap B_0, \ldots, A_{d-1} \cap B_{d-1})\ .$$

Special care has to be taken in case the intersection is empty, which corresponds to at least one interval $A_i \cap B_i$ being empty. In this case, the computation may preemptively be stopped, and an empty box representation returned, as a box is empty as soon as one of its intervals is empty. Note that for box-intersection $Set(A_B \cap B_B) = Set(A_B) \cap Set(B_B)$ holds.

Intersection with a Half-space

Boxes are not closed under intersection with a half-space as the exact result in general is a convex polytope. We worked on several approaches on how to efficiently intersect a box A with a half-space h and over-approximate the result by the smallest box representation A'_B such that $A'_B \supseteq A \cap h$ holds. In the following, we present two approaches that we have implemented in HyPro to obtain $Box(A_B \cap h)$ and discuss their efficiency.

Via Conversion. As HyPro already provides an interface to a linear optimization framework (see Section 6.7), a simple, yet inefficient implementation can be based on conversion. The constraints defining the box along with the constraint defining the half-space implicitly represent a convex polytope in $\mathcal{H}$-representation. We can use the conversion-methods implemented in HyPro to obtain the result:

$$A_B, h \xrightarrow{\text{to Hpoly}} A_H, h_H \xrightarrow{\cap} A'_H \xrightarrow{\text{to Box}} A'_B.$$

The computational effort of this approach highly depends on the implementation of the conversion method (see Section 6.8). While the conversion from box-representation to a convex polytope in $\mathcal{H}$-representation can be done efficiently, the converse requires solving $2d$ linear programs to obtain the interval bounds for each variable in the resulting box.

Via Interval Arithmetic. Conversion-based approaches for most operations on state set representations provide an intuitive way to compute the result. Nonetheless, those approaches neglect the structure of the representation as they generalize the problem statement. Especially for boxes, we aim at providing tailored approaches that exploit the properties of the state set representation.

We propose to compute box-halfspace intersection in the box representation by using *interval constraint propagation* (ICP) (see Section 2.4). The idea of ICP is to take a d-dimensional box domain for d variables and a set of (in)equalities over the variables. According to some heuristics, ICP iteratively selects one (in)equality and one variable appearing in it, transforms the inequality to have the selected variable alone on the left-hand-side, uses interval arithmetic to evaluate the right-hand-side, and contracts the interval domain of the selected variable (left-hand-side) according to the relation symbol of the (in)equality. For example, assuming the box domain $x \in [0,10]$, $y \in [2,7]$ and $z \in [2,3]$, from $x + y \leq z$ we can conclude

$$\begin{aligned} x + y \leq z \quad &\Rightarrow \quad x \leq z - y \in [2,3] - [2,7] = [-5,1] \\ &\Rightarrow \quad x \in [0,10] \cap (-\infty,1] = [0,1] \end{aligned}$$

Note that since $z - y$ is an upper bound on x, only the upper bound on $z - y$ (more precisely, the upper bound on z with positive coefficient 1 and the lower bound on y with negative coefficient -1) can be used to contract the upper bound of x.

ICP is in general incomplete and might not terminate. However, in the particular case of a single linear inequality $n^T \cdot x \leq c$, we have the following nice property.

Lemma 1. *ICP using a single linear inequality for contractions terminates after applying one contraction for each involved variable. Furthermore, the final contracted box domain is independent of the contraction order.*

Proof. Assume a mapping $x_i \to B_i$ for $x_i \in Var$, $i \in \{0, \ldots, d-1\}$ represented by the box $B = (B_0, \ldots, B_{d-1})$, and assume a linear inequality constraint $n^T \cdot x \leq c$, i.e.,

$$n_0 x_0 + n_1 x_1 + \cdots + n_{d-1} x_{d-1} \leq c. \tag{6.1}$$

Let $I = \{0, \ldots, d-1\}$, $I^+ = \{i^+ \in I \mid n_{i^+} > 0\}$, $I^- = \{i^- \in I \mid n_{i^-} < 0\}$, and $I^0 = \{i \in I \mid n_i = 0\}$. We can transform the above inequality for each $i^+ \in I^+$ to

$$x_{i^+} \leq c - \sum_{j^+ \in I^+ \setminus \{i^+\}} \left| \frac{n_{j^+}}{n_{i^+}} \right| + \sum_{j^- \in I^-} \left| \frac{n_{j^-}}{n_{i^+}} \right|. \tag{6.2}$$

and for each $i^- \in I^-$ to

$$x_{i^-} \geq -c + \sum_{j^+ \in I^+} \left| \frac{n_{j^+}}{n_{i^-}} \right| - \sum_{j^- \in I^- \setminus \{i^-\}} \left| \frac{n_{j^-}}{n_{i^-}} \right|. \tag{6.3}$$

ICP can now use all the above inequalities (Equations (6.2) and (6.3)) to evaluate the right-hand-sides using interval arithmetic and use the relation symbols to contract the interval domains of the left-hand-side variables.

Equation (6.2) can be used to contract the *upper* bound of each $i^+ \in I^+$, whereas Equation (6.3) can lead to contracted *lower* bounds for x_{i^-}, $i^- \in I^-$. Thus the set of all bounds that can be contracted is

$$lhs = \left\{ \underline{B}_i \;\middle|\; i \in I^- \right\} \cup \left\{ \overline{B}_i \;\middle|\; i \in I^+ \right\} .$$

In Equation (6.2), only the upper bound of the right-hand-side is used to determine the contraction for the (upper bound of the) left-hand-side, i.e., only the lower respectively upper bounds for variables with positive respectively negative coefficients. The case for Equation (6.3) is analogous: the lower bound of the left-hand-side is contracted using lower respectively upper bounds for variables with positive respectively negative coefficients. That means, the sets of all bounds that influence the contracting intervals are

$$rhs = \left\{ \overline{B}_i \;\middle|\; i \in I^- \right\} \cup \left\{ \underline{B}_i \;\middle|\; i \in I^+ \right\} .$$

We can observe that the two above sets, the influencing bounds *rhs* and the influenced bounds *lhs*, are disjoint. Therefore, the contraction step for every single variable is independent of all other contraction steps. This implies that the contraction order is irrelevant, and also that a second contraction step for the same variable has no additional effect. □

Consequently, we can compute the intersection of a half-space and a box by using interval constraint propagation to update the variable intervals once for each variable in arbitrary order, with an effort that is quadratic in the state space dimension. Example 6.1 provides an example for this approach.

Example 6.1: Half-space Intersection

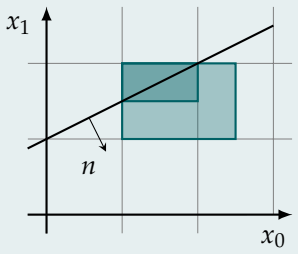
The result of the intersection of the 2-dimensional box $B = ([1, 2.5], [1, 2])$ and the half-space $h = \{x \mid x_0 - 2x_1 \leq -2\}$ is obtained using ICP as described before. The resulting box is depicted below.

Solving for x_0:

$$x_0 \leq 2x_1 - 2 \in [2, 4] - [2, 2] = [0, 2]$$
$$x_0 \in [1, 2.5] \cap (-\infty, 2] = [1, 2]$$

Solving for x_1:

$$x_1 \geq 0.5x_0 + 1 \in [0.5, 1.25] + [1, 2] = [1.5, 3.25]$$
$$x_1 \in [1, 2] \cap [1.5, \infty) = [1.5, 2]$$



Minkowski Sum

The Minkowski sum of A_B and B_B can be computed component-wise as

$$\begin{aligned} A_B \oplus B_B &= ([\underline{A}_0 + \underline{B}_0, \overline{A}_0 + \overline{B}_0], \ldots, [\underline{A}_{d-1} + \underline{B}_{d-1}, \overline{A}_{d-1} + \overline{B}_{d-1}]) \\ &= (A_0 + B_0, \ldots, A_{d-1} + B_{d-1}) \ . \end{aligned}$$

Box representations are closed under the Minkowski sum, i.e., $Set(A_B \oplus B_B) = Set(A_B) \oplus Set(B_B)$. In contrast to most other set representations, for box representations computing the Minkowski difference being the inverse operation can be done efficiently using interval arithmetic:

$$\begin{aligned} A_B \ominus B_B &= [\underline{A}_0 - \overline{B}_0, \overline{A}_0 - \underline{B}_0] \times \cdots \times [\underline{A}_{d-1} - \overline{B}_{d-1}, \overline{A}_{d-1} - \underline{B}_{d-1}] \\ &= (A_0 - B_0, \ldots, A_{d-1} - B_{d-1}) \ . \end{aligned}$$

Affine Transformation

Boxes are not closed under linear transformation with arbitrary matrices $\mathcal{A} \in \mathbb{R}^{d \times d}$. Applying an affine transformation $x' = \mathcal{A}x + b$ to a box representation

B_B, where $\mathcal{A} \neq Id$ thus requires additional effort to over-approximate the result to obtain the smallest box B'_B for which $Set(B'_B) \supseteq Set(B_B) \cdot \mathcal{A} + b$ holds. In the following, we present two approaches we developed in HyPro.

Via Conversion. One way to obtain a correct over-approximation of the affine transformation of B_B of a box B is similar to the approach for convex polytopes in $\mathcal{V}$-representation. The result is obtained by applying the operation on each vertex v_i of B_B and afterward over-approximating the result by a box. The set of transformed vertices

$$V' = \{\mathcal{A}v + b \mid v \in vertices(B_B)\}$$

can be used to obtain the resulting box representation by collecting the minimal and maximal coordinates over all vertices per dimension:

$$\begin{aligned} B'_B &= \left(B'_0, \ldots, B'_{d-1}\right) \text{ where} \\ \overline{B}'_i &= \max\left\{v_i \mid v \in V'\right\} \text{and} \\ \underline{B}'_i &= \min\left\{v_i \mid v \in V'\right\} \ . \end{aligned}$$

A d-dimensional box representation $B_B = (B_0, \ldots, B_{d-1})$ has exactly 2^d vertices which can be computed by enumerating all combinations of lower and upper bounds of the intervals B_i for the respective dimension. The linear transformation of a single vertex requires d^2 multiplications. In total this approach has an exponential running time $\mathcal{O}(2^d)$.

Via Interval Arithmetic. As each dimension in a box representation B_B is represented by an interval, it is natural to employ interval arithmetic for all operations. The second algorithm for computing an affine transformation of a box can be found in Algorithm 3. The idea is to compute the new lower and upper bounds for each dimension iteratively, where the contribution of every lower, respectively upper bound to the final result is updated continuously using interval arithmetic, which results in a quadratic running time.

After having computed the result of the linear transformation of B_B by $\mathcal{A}$, the translation by the vector b can be added separately on the boundaries to realize an affine transformation $B'_B = \mathcal{A} \cdot (B_0, \ldots, B_{d-1})^T + b$.

Algorithm 3: Iterative affine transformation for boxes.

Input: $B_B = (B_0, \ldots, B_{d-1})$, a matrix $\mathcal{A} \in \mathbb{R}^{d \times d}$, a vector $b \in \mathbb{R}^d$
Output: $B'_B = \left(B'_0, \ldots, B'_{d-1}\right) = \mathcal{A} \cdot B_B + b$

```
for i ∈ {0, ..., d − 1} do
    B'_i := b_i
    for j ∈ {0, ..., d − 1} do
        B'_i := B'_i + A_ij · B_j
```

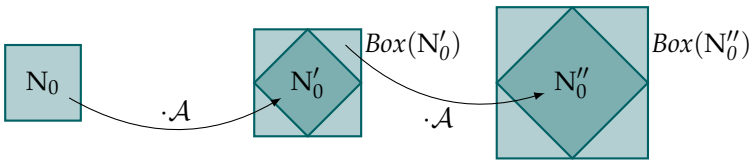


Figure 6.6: Accumulated wrapping effect of two rotations of a set of variable valuations N_0 (dark petrol) represented by a box representation (petrol).

Wrapping Effects. Especially boxes as a state set representation are known to suffer from the so-called "wrapping effect", which refers to the over-approximation errors introduced during computation [Le 09]. In HyPro, we require closure of our state set representation for all operations. To achieve this, additional over-approximation is introduced during the computation with boxes, as the result of each operation is approximated by its bounding box to achieve closure with respect to the state set representation. Especially linear or affine transformations might introduce large errors (see Figure 6.6) depending on the transformation matrix $\mathcal{A}$. As a consequence, the implicit assumption that a shorter time step size δ results in a more precise approximation (see Section 3.4) does not necessarily hold for boxes. We illustrate this behavior with the following example.

Example 6.2: Wrapping Effects

Consider the two-dimensional box representation $B_B = ([2,3],[1,4])$ and the matrix $\mathcal{A}$

$$\mathcal{A} = \begin{pmatrix} \cos(45) & -\sin(45) \\ \sin(45) & \cos(45) \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

which represents a rotation of 45 degrees in the mathematically positive sense (counter-clockwise) around the origin. Sequential application of two linear transformations of B_B by $\mathcal{A}$ results in the following:

$$\begin{aligned} B_B' =& \mathcal{A} \cdot (\mathcal{A} \cdot B_B) \\ =& \mathcal{A} \cdot \left(\left[\frac{-2}{\sqrt{2}}, \frac{2}{\sqrt{2}} \right], \left[\frac{3}{\sqrt{2}}, \frac{7}{\sqrt{2}} \right] \right)^T = \left(\left[\frac{-9}{2}, \frac{-1}{2} \right], \left[\frac{1}{2}, \frac{9}{2} \right] \right)^T . \end{aligned}$$

In contrast to this, applying a transformation $B_B'' = \mathcal{A}^2 \cdot B_B$ leads to the more precise resulting box

$$B_B'' = \mathcal{A}^2 \cdot B_B = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \cdot B_B = ([-4,-1],[2,3])^T$$

for which $B_B'' \subseteq B_B'$ naturally holds.

Computing the Support

Boxes are a subclass of convex polytopes, i.e., a conversion to a polytope for instance in $\mathcal{H}$-representation allows using linear optimization methods to compute the support of a box. However, as this is computationally expensive and we know that a box bounds the single state space dimensions independently, we use a more efficient approach in HyPro. To compute the support

$$\rho_{B_B}(l) = \sup_{b \in Set(B_B)} (l^T \cdot b)$$

in direction $l \in \mathbb{R}^d$ (see Section 6.4), we again employ interval arithmetic:

$$\begin{aligned}\rho_{B_B}(l) &= \sup_{b \in Set(B_B)} l^T \cdot b \\ &\subseteq \sup l^T \cdot B_B \\ &= \sup(l_0 B_0 + \cdots + l_{d-1} B_{d-1}) \\ &= \sup(l_0 \cdot B_0) + \cdots + \sup(l_{d-1} \cdot B_{d-1}).\end{aligned}$$

The supremum point $p \in B$ for a given direction l, i.e., the point where for all $b' \in B$ the inequation $l^T \cdot b' \leq l^T \cdot p$ holds, is a furthest point contained in B towards l. If B is a closed, convex set, there is always a vertex of B which satisfies this property.

A geometric intuition behind finding a supremum point p for a box representation B_B is given by considering the normal fan of B_B (see Definition 2.11). The normal vectors n_i of the half-spaces defining the box representation B_B form the normal fan $\mathcal{N}(B_B)$ of B_B.

Recall that each vertex v of a d-dimensional convex set S, specified by an intersection of a finite set of linear in-equations $h_i = \{x \mid n_i \cdot x \leq c_i\}$ (i.e., a convex polytope in $\mathcal{H}$-representation), is determined by the intersection of at least d hyperplanes $h'_i = \{x \mid n_i \cdot x = c_i\}$. Furthermore, boxes are a subclass of convex polytopes in $\mathcal{H}$-representation where all half-spaces are axis-aligned.

The resulting supremum point p for a given direction l is defined by the faces of the smallest d-dimensional cone C in $\mathcal{N}(B_B)$ containing l (see Figure 6.7). As the d-dimensional cones in the normal fan of a box define the orthants of the ambient space, the hyperplanes defining p are the faces bounding the orthant l lies in, i.e., the coordinate planes.

From knowing the orthant o which contains l, we can deduce the d hyperplanes defining p component-wise for a box representation B_B. For each dimension i, the sign of l_i determines, whether either the upper or the lower bound of the corresponding interval B_i is a hyperplane defining p—each coordinate p_i of p is defined by either the upper or lower bound of B_i. The support for l in B_B can now be computed by $l^T \cdot p$. This approach provides a method to compute the support for B_B which is linear in the state space dimension.

Experimental Results

We have run a selection of the operations for boxes to experimentally validate our implementation. All benchmarks were run on a machine with $4 \times 4\,\text{GHz}$

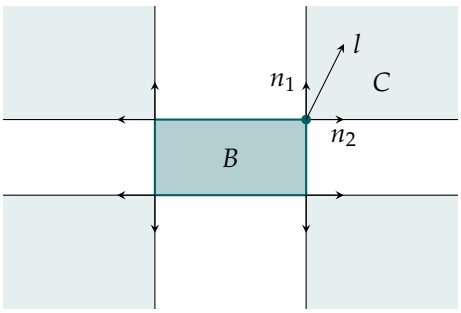
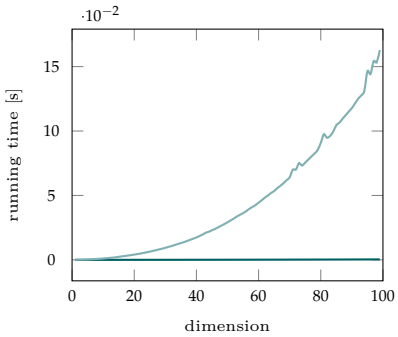


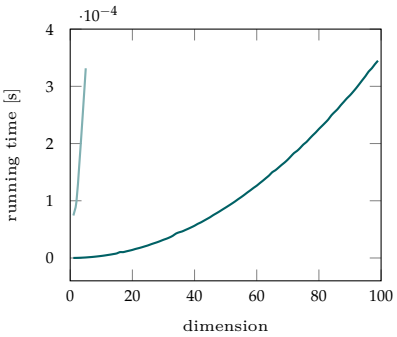
Figure 6.7: The vertex of a box B (petrol) defining $\sup_{b \in B} l^T \cdot b$ is defined by the faces of the smallest cone C containing l in the normal fan $\mathcal{N}(B)$ (light petrol).

Intel Core i7 CPUs and a memory limit of 8 GiB. We provide plots of the running times for the operations intersection with a half-space (see Figure 6.8a), affine transformation (see Figure 6.8c), and computing the support of a box (see Figure 6.8e). For each operation, we present an additional plot (Figures 6.8b, 6.8d and 6.8f) which shows an excerpt with a scaled y-axis to get more insights into the running times of the interval arithmetic-based approaches. All recorded running times represent averages over 10 000 operations performed except for the naive implementation of affine transformation where an average over 10 runs was used to keep running times within reasonable bounds. All experiments were performed on unit boxes, i.e., box representations where each $B_i = [-1, 1]$. We used randomly created matrices and vectors for the evaluation of the affine transformation. To evaluate intersection with a half-space, we used half-spaces h_i with randomly created normal vectors and an offset of zero to ensure a guaranteed intersection with the unit box, as $\mathbf{0} \in \bar{h}_i$ holds. Computing the support on unit boxes was evaluated using randomly created support directions.

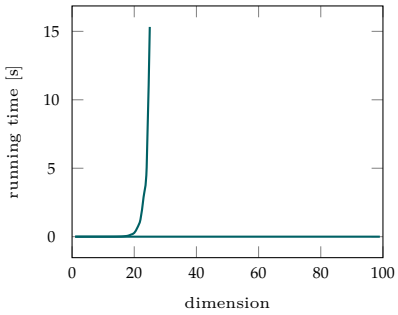
The running times indicate that the approach using interval arithmetic for all operations on boxes pays off in comparison to conversion-based approaches. Intersections with half-spaces, using conversion approaches to convex polytopes in $\mathcal{H}$-representation require solving $2 \cdot d$ linear programs when converting back to boxes, making up the major part of the running time. Similarly, computing the support of a box by using its $\mathcal{H}$-representation requires solving one linear program, while the proposed improvement requires only linearly many comparisons and one vector multiplication. In HyPro, we use an external library for linear programming (GLPK), which adds additional overhead for calling the solver and constructing the problem instance. The naive implementation of an affine transformation of a box using conversion to a polytope in $\mathcal{V}$-representation requires computing all 2^d vertices. Afterward, the affine transformation is applied on each vertex individually and the conversion back to a box can be done linearly in the number of vertices. In contrast to that, implementing affine transformations via interval arithmetic pays off in terms of computational effort, as only d^2 matrix multiplications plus $2 \cdot d$ vector multiplications are required, also observable in the running times. Furthermore, we ran into memory limits for affine transformations with the naive approach for more than 24 dimensions.



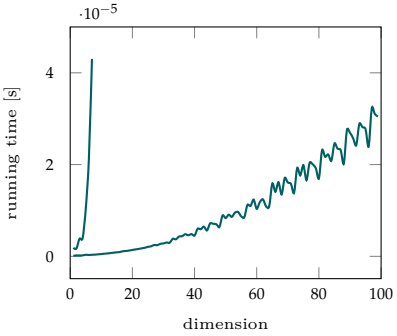
a) Intersection with a half-space.



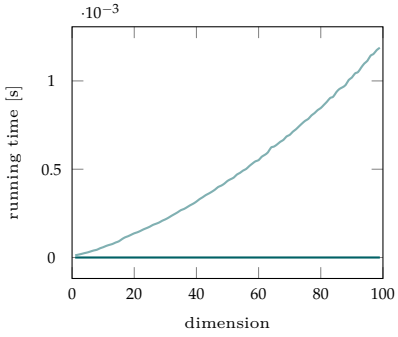
b) Intersection with a half-space (cut off).



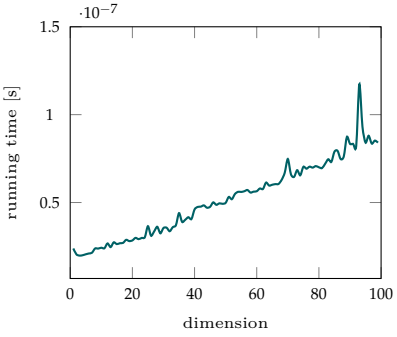
c) Affine transformation of a box.



d) Affine transformation of a box (cut off).



e) Computing the support of a box.



f) Computing the support of a box (cut off).

Figure 6.8: Benchmark results for single operations on boxes. The running times are averaged over 10 000 runs (except affine transformation), running times of the current implementation are depicted in petrol, naive implementations are depicted in light petrol.

6.3 Convex Polytopes

Convex polytopes find many applications in computer science, for instance in program verification, i.e., abstract interpretation to represent sets of variable valuations, as solution sets to linear programs, or naturally in computer graphics.

In the field of flowpipe-construction-based reachability analysis for linear hybrid systems, convex polytopes can be considered among the most precise of the commonly used state set representations. This is rooted in the fact that polytopes are closed under most operations required during the analysis. In fact, only for the union of two convex polytopes A, B additional effort is required to obtain the smallest convex polytope $C \supseteq A \cup B$, which can be computed as the convex hull of $A \cup B$. In hybrid automata, predicates such as guards and invariants are commonly described as polytopal sets and thus need no further conversion during computation. Additionally, following approaches as presented in [Le 09], the first segment of a flowpipe (see Section 3.4) can be most precisely described as a convex polytope.

Note that state set representations such as boxes (see Section 6.2) and zonotopes (see Section 6.5) are a subclass of convex polytopes; state set operations on them can be implemented using conversion methods. However, as polytope operations and conversion methods are in general comparatively expensive, specialized methods are usually applied.

As expected, the increased precision of convex polytopes comes at the cost of increased computational effort required during the analysis. In the following sections, we provide general information about convex polytopes based on [Zie95] and present our own ideas on the implementation of convex polytopes for flowpipe-construction-based reachability analysis in HyPro.

Definition 6.2: Polytope

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 $\{h_0, \dots, h_{m-1}\}$ with $h_i = \{x \in \mathbb{R}^d \mid n_{i,\_} \cdot x \leq c_i\}$ (see Definition 2.8). Recall that $n_{i,\_}$ refers to the i-th row of N.
The same polytope can also be represented as the convex hull (see Section 2.3) of a finite set $P_V = \{v_0, \dots, v_{m-1}\}$ of vertices $v_i \in \mathbb{R}^d$ (*$\mathcal{V}$-representation*):

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

In the following, we use the function $cHull(V)$ over a finite set of vertices $V = \{v_0, \dots, v_{m-1}\}$ to compute the convex hull of V. Furthermore, we use $|(N, c)|$ respectively $|V|$ to refer to the number m of half-spaces respectively

Table 6.1: Computational effort for different operations depending on the underlying polytope representation. A "-" indicates that for the operation no polynomial algorithm is known and the operation usually is implemented via conversion to the other representation while a "+" indicates the operation can be done with polynomial computational effort.

	$\cdot \cup \cdot$	$\cdot \cap \cdot$	$\cdot \oplus \cdot$	$A\cdot$	$\rho_{(\cdot)}$	$\cdot \stackrel{?}{=} \emptyset$	$\cdot \cap h$
$\mathcal{H}$-representation	-	+	-	-	+	-	+
$\mathcal{V}$-representation	+	-	+	+	-	+	-

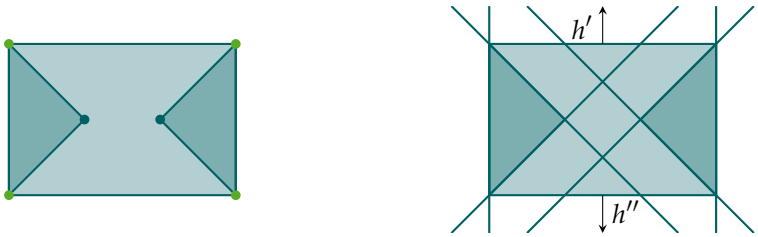
vertices required to represent P as a notion of its representation complexity of P_H respectively P_V. Note that the representation of a polytope is not unique: a $\mathcal{H}$-representation of a polytope contains all defining half-spaces but perhaps also some further implied inequalities. Analogously, a $\mathcal{V}$-representation might contain not only all vertices but any point from the convex hull of the vertices. We call a minimized representation *reduced*. We will discuss methods on how to reduce the representation of polytopes in Section 6.3. The operations presented in the following do not rely on the operands being reduced but in general profit from a reduced representation to save computational effort.

For reasons of simplicity, we use the term polytope and convex polytope interchangeably in the rest of this work. Furthermore, we always consider bounded sets, i.e., convex polyhedra which do not contain any ray or cone (see Definition 2.10). For details about convex polyhedra, i.e., possibly unbounded polytopes, we refer to [Zie95]. In this section, we present details on our own implementation of convex polytopes including several reduction techniques.

Currently, there are several implementations of convex polytopes available, e.g., the Parma Polyhedra Library (PPL) [BHZ08] or Polymake [GJ00]. Additional to our implementation, HyPro provides a wrapper class for PPL [BHZ08], which provides an efficient implementation for convex polytopes along with the most operations on state sets required for flowpipe-construction-based reachability analysis. For the future, we plan to add further wrapper classes to other implementations such as Polymake [GJ00].

Operations

In the following, we present the general ideas for the most common operations on convex polytopes required during flowpipe-construction-based reachability analysis. From Table 6.1, we can already see that most operations on convex polytopes behave complementary concerning running times and complexity, depending on the internal representation. Therefore, many operations rely on conversion between the $\mathcal{H}$-representation and the $\mathcal{V}$-representation. Assume in the following two d-dimensional polytopes $A_V = V_A = \{a_0, \ldots, a_{m-1}\}$ and $B_V = V_B = \{b_0, \ldots, b_{n-1}\}$ in $\mathcal{V}$-representation. Also assume two d-dimensional polytopes $A_H = (M, c)$ and $B_H = (N, e)$ in $\mathcal{H}$-representation represented as the intersection of m respectively n half-spaces.



a) Two former vertices (dots in petrol) are no vertices in the closure of the union of A_V and B_V while the other vertices of A_V and B_V (dots in green) are.

b) The closure of the union of A_H and B_H requires to compute new half-space normal vectors for h' and h'' which cannot be derived from original half-spaces.

Figure 6.9: The closure of the convex union of two polytopes (dark petrol) in $\mathcal{V}$-representation and $\mathcal{H}$-representation.

Test for Emptiness

Testing whether $A_H = \emptyset$ can be done using linear programming (LP) and checking whether $\{Mx \leq c \mid x \in \mathbb{R}^d\} = \emptyset$ using an LP- or SMT-solver, for instance GLPK [Mak18], SMT-RAT [CKJ+15], SOPLEX [Wun96], or Z3 [MB08]. Testing whether B_V, a convex polytope in $\mathcal{V}$-representation is empty is less involved, as we only consider bounded sets. As a consequence, checking whether a polytope is empty is equivalent to checking whether the set of vertices defining it is empty, i.e., testing whether $|V_B| = 0$.

Union

The convex closure of the union of A_V and B_V in the $\mathcal{V}$-representation can be represented by the union of the two sets of vertices

$$cl(A_V \cup B_V) = V_A \cup V_B$$

such that $cHull(V_A \cup V_B) = cl(A_V \cup B_V) = cl(Set(A_V) \cup Set(B_V))$ holds. Note that the set of points $V_A \cup V_B$ might contain redundant points (see Figure 6.9a). In Section 6.3, we will present methods on how to reduce representations.

The convex closure of the union of two d-dimensional polytopes A_H and B_H in the $\mathcal{H}$-representation is more involved. From the definition we obtain

$$cl(A \cup B) = \{\lambda a + (1-\lambda)b \mid \lambda \in [0,1] \wedge a \in A \wedge b \in B\} \ .$$

However, this does not directly allow to derive a new set of half-spaces exactly defining the $\mathcal{H}$-representation of the polytope $cl(A_H \cup B_H)$ as computing the closure may introduce new half-spaces (see Figure 6.9b).

In HYPRO, the union of two polytopes in $\mathcal{H}$-representation is computed using conversion to $\mathcal{V}$-representation using vertex enumeration methods.

Intersection

The intersection of A_H and B_H in $\mathcal{H}$-representation specified by the intersection of n, respectively m half-spaces can be obtained directly as

$$\begin{aligned} A_H \cap B_H &= \{x \mid x \in A_H \wedge x \in B_H\} \\ &= \left\{ x \;\middle|\; \bigwedge_{i=0}^{m-1} m_{i\_} \cdot x \leq c_i \wedge \bigwedge_{j=0}^{n-1} n_{j\_} \cdot x \leq e_j \right\} \\ &= \left(\begin{pmatrix} M \\ N \end{pmatrix}, \begin{pmatrix} c \\ e \end{pmatrix} \right) . \end{aligned}$$

Thus, intersection for polytopes in $\mathcal{H}$-representation can efficiently be computed by unifying the sets of defining half-spaces to compute $A_H \cap B_H$, where $A_H \cap B_H = \mathit{Set}(A_H) \cap \mathit{Set}(B_H)$ holds. Note that this approach does not necessarily result in a reduced representation, as redundant constraints may be contained (see Section 6.3 for information on how to handle redundancy). While the intersection in $\mathcal{H}$-representation allows obtaining the resulting polytope directly, this is not the case for the $\mathcal{V}$-representation. Using the formal definition, we obtain

$$\begin{aligned} A \cap B &= \{x \mid x \in A \wedge x \in B\} \\ &= \left\{ x \;\middle|\; x = \sum_{i=0}^{m-1} \lambda_i \cdot v_i \wedge \sum_{i=0}^{m-1} \lambda_i = 1 \wedge \lambda_i \in [0,1] \wedge v_i \in V_A \wedge \right. \\ &\qquad \left. x = \sum_{j=0}^{n-1} \lambda_j \cdot v_j \wedge \sum_{j=0}^{n-1} \lambda_j = 1 \wedge \lambda_j \in [0,1] \wedge v_j \in V_B \right\} . \end{aligned}$$

From this description, the new set of vertices defining $A_V \cap B_V$ is not directly accessible—in HyPro intersection between two polytopes in $\mathcal{V}$-representation is handled via conversion to $\mathcal{H}$-representation.

Intersection with a Half-space

The intersection of a convex polytope A and a half-space h is usually accompanied by a test for emptiness afterward (see Section 6.1). Detailed information about the result of an intersection with a half-space may be used during computation and can be obtained with little effort. Considering a polytope A_H in $\mathcal{H}$-representation, the intersection of a half-space h with A_H is computed by adding h as a constraint to A_H following the definition of a $\mathcal{H}$-representation as a conjunction of half-spaces. Testing $A_H \cap h = \emptyset$ is performed afterwards and can be accompanied by redundancy checks (see Section 6.3). Note that this approach naturally extends to sets of half-spaces, as we can interpret h as a convex polytope in $\mathcal{H}$-representation with only one constraint.

For a polytope B_V in $\mathcal{V}$-representation, computing the intersection with h cannot be performed directly. We may always convert B_V to $\mathcal{H}$-representation and use the above method, however in HyPro, we perform a set of computationally less expensive tests, before using the computationally costly conversion approach. By verifying the coordinates of the vertices in V_B against

the constraint defining the half-space, we can check whether the respective vertex lies inside the half-space, i.e., satisfies the inequation. In case all vertices satisfy h, we know that B_V is fully contained in h; if no vertex satisfies h, the result will be empty. In both cases, we have avoided the costly transformation to $\mathcal{H}$-representation, which is required whenever h partially intersects with B_V. As we test each point in V_B against h, it helps if $V_B = \mathit{vertices}(B_V)$ or at least if $|V_B|/|\mathit{vertices}(B_V)|$ is small.

Minkowski Sum

The definition of the Minkowski sum of two sets directly provides an approach towards computing the Minkowski sum of A_V and B_V

$$A_V \oplus B_V = \{a + b \mid a \in V_A \wedge b \in V_B\}$$

where $\mathit{Set}(A_V \oplus B_V) = \mathit{Set}(A_V) \oplus \mathit{Set}(B_V)$ holds. In general, it holds that $A_V \oplus B_V \supseteq \mathit{vertices}(A_V \oplus B_V)$ while $A_V \oplus B_V = \mathit{vertices}(A_V \oplus B_V)$ only holds in special cases, for instance if $|V_A| = 1$ or $|V_B| = 1$.

Algorithms based on zonotope construction [Fuk04; WF05] to compute the Minkowski sum of two polytopes have been developed to overcome this. Additionally, some of these algorithms allow extracting the resulting facets during computation. A variant of the algorithm presented in [Fuk04] which has been implemented in the course of the Master thesis of Christopher Kugler [Kug14] is integrated in HyPro. To be fully applicable, the method requires full knowledge of the neighborhood relations between the vertices of A_V and B_V to compute $A_V \oplus B_V$. In combination with convex hull algorithms which allow to determine the neighborhood relation of the vertices, such as the one presented in [AF92], the combined method should result in good performance as reduction is not necessary afterward using this approach. However, the current version of HyPro does not provide an implementation of this method.

Computing the Minkowski sum of two polytopes in $\mathcal{H}$-representation is more involved and currently implemented via conversion to $\mathcal{V}$-representation in HyPro.

Affine Transformation

Computing the result A'_V of the affine transformation $A'_V = \mathcal{A} \cdot A_V + b$ of A_V by a matrix $\mathcal{A} \in \mathbb{R}^{d \times d}$ and a vector $b \in \mathbb{R}^d$ can be directly performed on the set of vertices of A_V

$$A'_V = \left\{ A \cdot v_i + b \;\middle|\; \mathcal{A} \in \mathbb{R}^{d \times d} \wedge b \in \mathbb{R}^d \wedge v_i \in V_A \right\} .$$

In general, for polytopes in $\mathcal{H}$-representation, conversion to $\mathcal{V}$-representation can be used. However, for the linear transformation of $B_H = (N, e)$ in case $\mathsf{rank}(\mathcal{A}) = d$ ($\mathcal{A}$ is invertible) holds, we can use the following transformation with $y = \mathcal{A}x$:

$$\begin{aligned} \mathcal{A} \cdot B_H &= \left\{ \mathcal{A}x \;\middle|\; n_i^T \cdot x \leq e_i \wedge x \in \mathbb{R}^d \right\} \\ &= \left\{ y \;\middle|\; n_i^T \cdot \mathcal{A}^{-1} \cdot y \leq e_i \right\} \end{aligned}$$

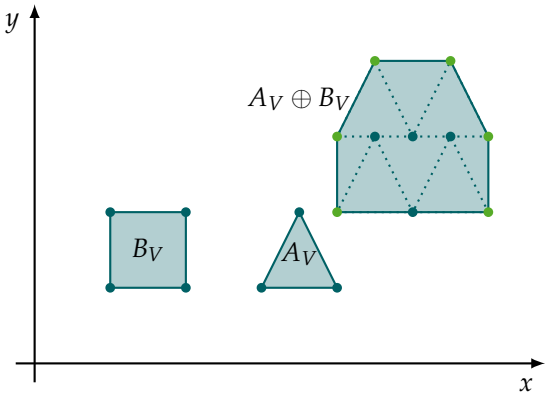


Figure 6.10: Minkowski sum of two polytopes A_V and B_V in $\mathcal{V}$-representation. The additional points inside the resulting set (petrol) are points in the non-reduced point set when computing the Minkowski sum.

which allows to compute the linear transformation on polytopes in $\mathcal{H}$-representation directly by modifying the single half-spaces $n_i^T \cdot x \leq e_i$ of B_H. Using the same principle, we can reconsider b and $y = Ax + b$:

$$\begin{aligned} \mathcal{A} \cdot B_H + b &= \left\{ \mathcal{A}x + b \;\middle|\; n_i^T \cdot x \leq e_i \right\} && \mid x = \mathcal{A}^{-1}(y - b) \\ &= \left\{ y \;\middle|\; n_i^T \cdot \mathcal{A}^{-1} \cdot (y - b) \leq e_i \right\} \\ &= \left\{ y \;\middle|\; n_i^T \cdot \mathcal{A}^{-1} \cdot y - n_i^T \cdot \mathcal{A}^{-1} \cdot b \leq e_i \right\} \\ &= \left\{ y \;\middle|\; n_i^T \cdot \mathcal{A}^{-1} \cdot y \leq n_i \cdot \mathcal{A}^{-1} \cdot b + e_i \right\} \end{aligned}$$

to compute the affine transformation $B'_H = \mathcal{A} \cdot B_H + b$. Thus, in case $\mathcal{A}$ is invertible P' can be computed as a simple transformation of the normal vectors and the offsets of the half-spaces (N, e) defining B_H. Note that for the application in flowpipe-construction-based reachability analysis this case often occurs, as the matrix exponential of a matrix defining the flow and the matrix approximating it are in most cases invertible (see Section 3.4). Consequently, only reset functions on discrete transitions which are not invertible will trigger the expensive conversion-based approach.

Computing the Support

The support $\rho_P(l)$ of a P_H for direction $l \in \mathbb{R}^d$ can be computed using standard LP techniques such as the simplex method [Dan63] implemented in state-of-the-art LP-solvers. The vector l is used as the coefficients of the cost function which is to be maximized

$$\rho_{P_H}(l) = \sup_{p \in Set(P_H)} \langle l, p \rangle \ .$$

HyPro provides a generalized interface for linear optimization which allows to exchange the optimization backend and choose between different solvers or employing several solvers incrementally (see Section 6.7).

Reduction Techniques

Irregardless of the representation used, certain operations, for instance the Minkowski sum or union increase the representation complexity (see Figure 6.10). Furthermore, the underlying number representation plays an important role in the size of the representation. For instance, when using arbitrary precision rational types `mpq_class` as implemented in the GMP library, repeated linear transformations have a strong influence on the representation size of e.g., coefficients for normal vectors of half-spaces.

In general, more complex representations do not only increase the required storage space but more importantly have a strong influence on the running time of an operation. In the following paragraphs, we present possible reduction techniques implemented in HyPro, which allow us to keep the number of half-spaces respectively vertices to represent a set small.

Redundancy Removal. As presented before, some of the operations on polytopes return results, which are non-reduced. The running time of single operations on convex polytopes is usually sensitive to the number of vertices respectively half-spaces. Consequently, repeated application of such operations leads to a drastic increase in complexity rendering flowpipe-construction-based reachability analysis using polytopes as a state set representation in practice infeasible.

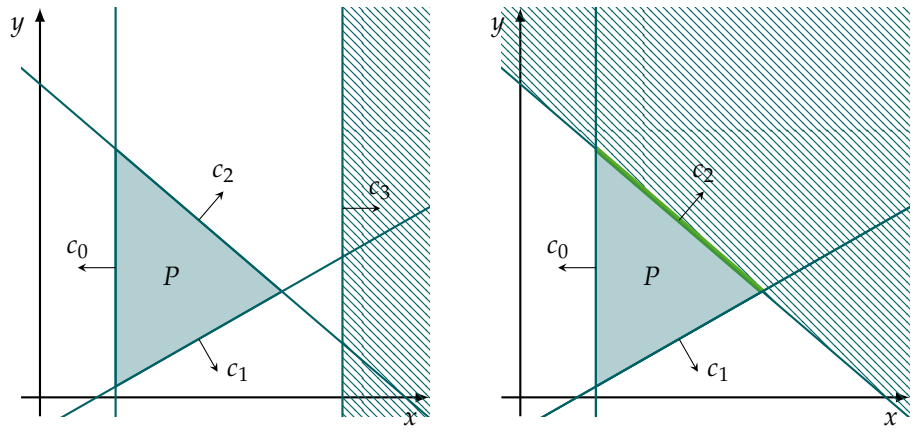
Consider the intersection of a convex polytope A_H with a half-space h. The result of $A_H \cap h$ can be obtained by adding h to the set of constraints defining A_H. However, if h does not add any new information, i.e., h does not define a face of A_H (see Figure 6.11a), adding h to A_H will only increase its complexity. This might slow down later operations which are sensitive to the number of half-spaces defining A_H. To overcome this, HyPro features a simple reduction mechanism that can be used to detect and remove redundant constraints for polytopes in $\mathcal{H}$-representation.

Definition 6.3: Redundant Constraint

Assume a predicate $P \in Pred_{Var}$ defined over a finite set of variables Var being a Boolean combination of constraints p_i.
We call a single constraint p_i redundant, if $Sat(P \setminus p_i) = Sat(P)$.

Convex polytopes in HyPro in $\mathcal{H}$-representation are represented as a conjunction of weak linear inequalities. Based on these properties, we provide a simple method to detect the redundancy of a constraint (see Algorithm 4).

The method described in Algorithm 4 iterates over the constraints $n_{i\_} \cdot x \leq c_i$ in $A_H = (N, c)$ and checks for each inequation $n_{i\_} \cdot x \leq c_i$ individually, whether the intersection of A_H and the hyperplane $n_{i\_} \cdot x = c_i$ is empty.



a) The constraint c_3 is redundant in P_H as it does not intersect any facet of P_H. The hatched area (petrol) shows the half-space h defined by the inverse of c_3 which is used to determine redundancy if $P_H \cap h = \emptyset$.

b) The constraint c_2 is not redundant as it defines a facet of P_H. The hatched area (petrol) shows the half-space defined by the inverse of c_2, the non-empty intersection which gives the facet defined by c_2 is highlighted (green).

Figure 6.11: Illustration of the redundancy-detection method implemented in HyPro to determine whether single bounding hyperplanes of constraints in P_H define a facet in P_H.

Algorithm 4: Detecting redundant constraints of convex polytopes in HyPro.

Input: A convex polytope $A_H = (N, c)$.
Output: A convex polytope A'_H where $|A'_H| \leq |A_H|$ and $Set(A'_H) = Set(A_H)$ holds.

$A'_H := A_H$
if $Sat(A_H) \neq \emptyset$ **then**
  for $i \in |A_H|$ **do**
    $h_i = (n_{i\_}, c_i)$
    $A'_H := A'_H \setminus h_i$
    if $Sat(A'_H \wedge (-n_{i\_}, -c_i)) = \emptyset$ **then**
      $A'_H := A'_H \setminus h_i$
return A'_H

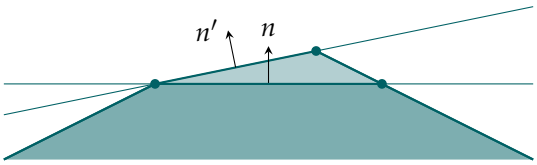


Figure 6.12: Reducing the number representation of the coefficients in the normal vector n of a half-space to obtain a new normal vector n' while ensuring over-approximation (light petrol). The resulting half-space is shifted to ensure all original vertices are contained.

As we know that all constraints are defined by weak inequations, we can realize a check for the redundancy of a single constraint $n_{i\_} \cdot x \leq c_i$ by inverting it, i.e., by negating its normal vector and offset (see Figure 6.11).

In case A_H has a non-empty intersection with $-n_{i\_} \cdot x \leq -c_i$, this indicates that the corresponding original constraint defines a face of P_H. The geometric interpretation of the algorithm takes each half-space and inverts it to obtain A'_H—as constraints are defined by a weak inequation the bounding hyperplane of the respective constraint $n_{i\_} \cdot x \leq c_i$ is still in A'_H if it is non-redundant (see Figure 6.11b). Note that to reduce computational effort, polytopes in HyPro use a cache to store whether redundancy has been removed.

Similar to redundancy for polytopes in $\mathcal{H}$-representation, we can define redundancy for polytopes in $\mathcal{V}$-representation. The set of points V_B used to define a polytope B_V in $\mathcal{V}$-representation contains redundant points whenever $V_B \setminus \mathit{vertices}(B_V) \neq \emptyset$. To find redundant points $p_i \in V_B$, we make use of the definition of a polytope in $\mathcal{V}$-representation. For each $p_i \in V_B$, we create a linear program

$$L : p_i = \sum_{j \neq i} \lambda_j \cdot p_j \wedge 1 = \sum_{j \neq i} \lambda_j \wedge \bigwedge_{j \neq i} \lambda_j \in [0,1]$$

which has a solution if p_i can be represented as a convex combination of the other points in V_B and consequently is not an extreme point of $\mathit{Set}(P_V)$. Using the LP backend in HyPro, we can determine that p_i is redundant if L has a solution and remove it from V_B.

Number Reduction. To keep the representation of rational number types low, we require a dedicated rounding mechanism which reduces the number representation but at the same time guarantees over-approximation. In the following, we present an approach for polytopes in $\mathcal{H}$-representation, which allows reducing the representation size of single half-spaces.

A half-space $h = \left\{ x \;\middle|\; n^T \cdot x \leq c \right\}$ is defined by its normal vector n and the offset c. While rounding c in a way that ensures over-approximation is straight forward (upward rounding), rounding n is more involved as simple rounding of the single coefficients of n does not necessarily ensure over-approximation.

To reduce the number representation of a single half-space h, the set of vertices V which lie on this plane is required. In a two step-process first the normal vector n is rounded according to a rounding policy, e.g.,

$$n' = \left\lfloor \frac{n}{lvalue \cdot limit} \right\rfloor$$

where *lvalue* denotes the largest coefficient in n and *limit* is a predefined upper bound for the size of the number values. The resulting normal vector n' will most probably point in a slightly different direction than the original vector (see Figure 6.12) unless n and n' are linearly dependent which is hard to predict or even to aim for during construction. To ensure over-approximation, the offset c can be adjusted such that all original vertices are contained in the resulting polytope:

$$c' = \max_{v \in V} \left(\lceil n' \cdot v \rceil \right).$$

The new half-space $h' = \{x \mid n' \cdot x^T \leq c'\}$ now has a smaller number representation but also contains all original vertices.

To reduce the number representation of a polytope B_V in $\mathcal{V}$-representation we use a bounding-box-based approach. To determine the rounding direction of each component for each vertex $v_i \in vertices(B_V)$ we compute the *barycenter* $\overline{b}$ of B_V

$$\overline{b} = \frac{\sum_i v_i}{|vertices(B_V)|}.$$

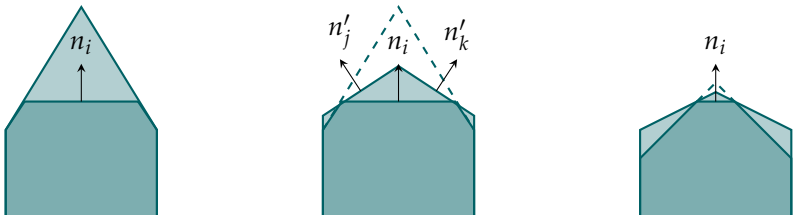
The components of the offset vector $v_i - \overline{b}$ determine the rounding directions for each component of v_i. Intuitively, this approach pushes each vertex v_i further towards the half-spaces defining the bounding box of B_V as the components in v_i are handled independently and the offset vector ensures outward rounding.

Shape Reduction. This paragraph presents ideas to reduce the representation P_H of a polytope P in $\mathcal{H}$-representation while maintaining over-approximation. These approaches have been implemented in HyPro for convex polytopes during the course of the Bachelor thesis of Igor Bongartz [Bon16], but are yet to be automated in the sense that they have to be called manually in the current state. As such, this paragraph suits as a pointer towards future development for reduction methods for convex polytopes.

We have synthesized three families of possible reduction approaches: *drop-reduction*, *union-reduction*, and *template-based reduction*. While the first two families are dedicated to $\mathcal{V}$- and $\mathcal{H}$-representations of polytopes, template-based reduction provides a more general approach which is also applicable to other state set representations. In the following, we give more insights into the approaches.

Drop-reduction techniques refer to approaches, which remove selected half-spaces h_i of $P_H = \bigcap_{j=0}^{m-1} h_j$ to reduce the representation while maintaining over-approximation to obtain

$$P'_H = \bigcap_{j=0, j \neq i}^{m-1} h_j$$



a) Reduction by constraint removal.

b) Smoothing after constraint removal.

c) Harmful smoothing introduces additional over-approximation error.

Figure 6.13: Different effects of shape reduction of a polytope in $\mathcal{H}$-representation via removal of constraints. Introduced over-approximation errors are depicted in lighter petrol.

such that $Set(P_H) \subseteq Set(P'_H)$ holds. Removing constraints may potentially add huge over-approximation errors (see Figure 6.13a). To overcome this, we can improve the result by adjusting the neighboring constraints around the constraint that is to be dropped. Note that this approach requires knowledge of the topological structure of the constraints as well as the vertices of P_H. After removal of the half-space h_i, all neighboring half-spaces $H_{neigh}(h_i)$ need to be adjusted, where

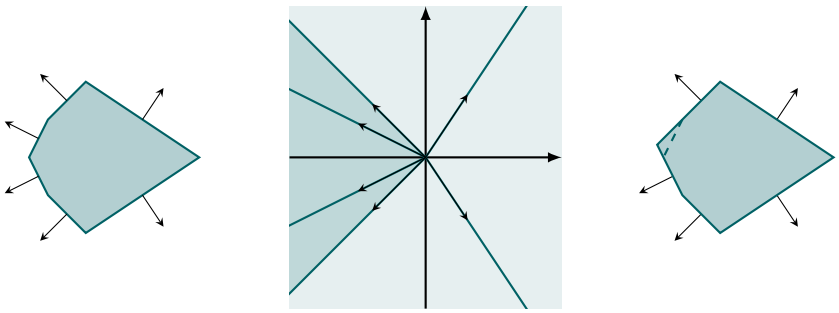
$$H_{neigh}(h_i) = \left\{ h_j \;\middle|\; \exists v \in vertices(P).\ v \in \overline{h_i} \wedge v \in \overline{h}_j \wedge i \neq j \right\} .$$

Intuitively, a half-space h_j is a neighbor of h_i if at least one of the vertices v of P_H is the solution to the linear equation system $\mathcal{A} \cdot v = b$ where the rows of $\mathcal{A}$ contain the normal vectors of h_i and h_j, i.e., v is a point in both bounding hyperplanes $\bar{h}_i : n_i \cdot v = c_i$ and $\bar{h}_j : n_j \cdot v = c_j$. To reduce over-approximation errors, *smoothing* may be applied in which the normal vectors of the neighboring half-spaces $h_j \in H_{neigh}$ are updated to

$$h'_j = \frac{n_j + n_i}{2} \cdot x \leq c'_j$$

such that the new normal vector n'_j is averaged over n_i and n_j. To ensure correct over-approximation of $Set(P_H)$, the offset of the new half-space h'_j is chosen such that all original vertices of P_H are still contained in P'_H (see Figure 6.13b).

While this method is relatively simple, the selection of the constraints which will be removed is more involved, as the introduced over-approximation error should be kept as low as possible. One idea for a heuristics is based on the normal fan $\mathcal{N}(P_H)$ of P_H (see Figure 6.14a). For now, let us assume all normal vectors of half-spaces defining P_H are normalized, i.e., $\|n\|_2 = 1$ and P_H is non-redundant. In this setup, the selection of a half-space to remove can be made by only considering the normal vectors n of the half-spaces and neglecting the offsets. The analysis of $\mathcal{N}(P)$, more specifically the distribution of normal vectors of the half-spaces defining P can serve as a heuristic for selecting planes for removal (see Figure 6.14b and Figure 6.14c). Informally, a suitable normal vector n for removal can be chosen based on its similarity to its neighbors n',



a) The polytope P. b) The normal fan $\mathcal{N}(P)$. c) Reduction.

Figure 6.14: Idea of a selection heuristics for the removal of half-spaces defining a polytope P_H to reduce its representation complexity. The shaded area in Figure 6.14b (light petrol) highlights normal vectors of half-spaces suitable for removal.

i.e., by finding a normal vector n such that $\langle n, n' \rangle$ is large. Note that smoothing the resulting object is not always beneficial, as it might introduce additional over-approximation errors instead (see Figure 6.13c).

A different way of reducing the representation complexity of a polytope is realized by *union-reduction*. Assume a convex polytope P_H in $\mathcal{H}$-representation defined by the intersection of finitely many half-spaces $h_i, i = 0, \ldots, n-1$. Union-reduction attempts to replace a finite subset $h_j, \ldots, h_k, 0 \leq j \leq k \leq n-1$ of the constraints defining P_H by a single new constraint h' such that

$$P'_H = \left(\bigcap_{0 \leq i < j \cup k < i \leq n-1} h_i \right) \supseteq P$$

holds (see Figure 6.15a). The normal vector n' of h' can be computed in several ways. Taking the average of the normalized normal vectors of the half-spaces in H is one option to obtain a suitable n'. Computing the offset c' of h' requires knowledge of the vertices of P_H—again we need to ensure over-approximation. As the vertices of P_H need to be known to compute c', we can construct n' in a different way. The set V_{hor} defines the set of vertices v_i which define the boundary/horizon of the set of half-spaces H considered for reduction

$$V_{hor} = \left\{ v \;\middle|\; \exists h \in H.\ \exists h' \notin H.\ v \in \bar{h} \wedge v \in \bar{h}' \right\} .$$

Intuitively, V_{hor} contains all vertices of P which are defined by the intersection of at least one bounding hyperplane of a half-space in H and one bounding hyperplane of a half-space h' defining P, which is not to be removed. Each d-permutation of V_{hor}, i.e., each subset of size d of V_{hor} defines a hyperplane $\bar{h}$ spanned by those d vertices of V_{hor}. We can use the average of either all or a reasonably-sized subset of normal vectors of those hyperplanes to obtain the normal vector n' of the new hyperplane. Again, the offset c' is computed to contain all original vertices of P_H.

a) Reduction by over-approximating two half-spaces by a single one. The new normal is created from the vertices v_i and v_j.

b) Reduction by using a fixed uniform hexagonal template.

Figure 6.15: Different effects of shape reduction of a polytope in $\mathcal{H}$-representation via union-reduction of half-spaces or template-based reduction. The introduced over-approximation errors are depicted in lighter petrol.

The last method, *template-based reduction* uses a template in combination with linear programming to reduce P_H. The general idea relies on having a predefined template, i.e., a set $D = \{d_0, \ldots, d_{n-1}\}, d_i \in \mathbb{R}^d$ of vectors. An over-approximation $P' \supseteq P$ is computed using linear programming

$$P'_H = \left\{ x \;\middle|\; \forall d_i \in D.\ d_i \cdot x \leq \rho_{P_H}(d_i) \right\} \ .$$

Intuitively, an over-approximation for P is obtained by computing the supporting hyperplanes of P based on a fixed set D or sampling directions (see Figure 6.15b).

While the first two approaches aim at a local reduction of a given polytope P, template-based reduction provides a more general approach. Local modifications require heuristics to choose half-spaces which will be subject to reduction. In contrast to this, the advantage of template-based reduction lies in its generality, i.e., not requiring any heuristics for reduction. Furthermore, the result for the first two approaches is not known a priori, and the choice of a suitable approach again requires heuristics. During the analysis, we use template-based reduction methods for convex polytopes.

As a final remark, all reduction techniques need to be triggered explicitly, as we did not yet implement any metrics on when reduction should be applied. One idea of a possible metric is based on the idea of zonotope order (see Section 6.5), in which the complexity of an object is given as a ratio between state space dimension d and the number of generators. The generators of a zonotope Z are given as vectors $g_i \in \mathbb{R}^d$ which allow to define Z. We can use a similar idea for a metric o for the order of a convex polytope P_H:

$$o = \frac{|P_H|}{d}$$

i.e., the ratio of half-spaces to the state space dimension. Similarly, to estimate the complexity of polytopes in $\mathcal{V}$-representation, we can choose

$$o = \frac{|V_P|}{d} \ .$$

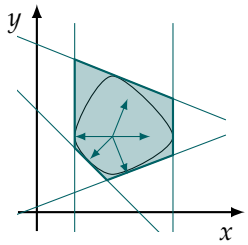


Figure 6.16: Polytopal approximation of a support function.

6.4 Support Functions

Support functions have recently been used extensively in hybrid systems reachability analysis [LG10; FLD+11; FKL13] and thus underwent an extensive development process. Support functions belong to the symbolic state set representations where the support $\rho_S(l)$ of a compact convex set S in the direction vector $l \in \mathbb{R}^d$ is defined as

$$\rho_S(l) = \sup_{s \in S}(l^T \cdot s).$$

Intuitively, the support $\rho_S(l)$ of S allows to define a half-space h with normal vector $n = l$ and the offset $c = \rho_S(l)$, where c is chosen such that $\bar{h} = \left\{x \in \mathbb{R}^d \mid n^T \cdot x = c\right\}$ is a supporting hyperplane of S in direction l, i.e., contains a face of S (see Figure 6.16).

In HyPro, the state set representation for the underlying set S may be for instance

- a ball B of radius $r : \rho_B(l) = \frac{|l|}{r}$,
- a box $B = (I_0, \ldots, I_{d-1})$ (see Section 6.2), or
- a polytope in either $\mathcal{H}$-representation or $\mathcal{V}$-representation (see Section 6.3), or
- an ellipsoid E where $\rho_E(l) = \langle l, q \rangle + \sqrt{l^T \mathcal{Q} l}$ (see Section 6.6).

The effort of computing the support of a set thus heavily depends on its representation—while the support for a ball can be computed in constant time, the support for a box requires linear effort (see Section 6.2) and for a convex polytope a linear program has to be solved, which can (theoretically) be done in polynomial time.

Operations

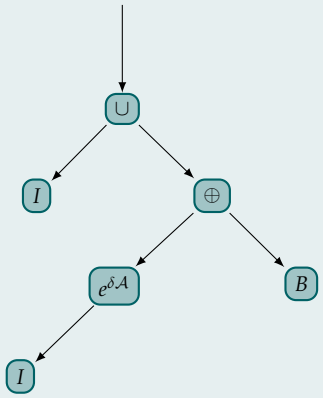
As noted for instance in [LG10] the required operations for flowpipe-construction-based reachability analysis on a given support function ρ_A of a set A can be computed efficiently:

$$\begin{aligned} \text{union} \quad & \rho_{A\cup B}(l) = && \max(\rho_A(l), \rho_B(l)) \\ \text{intersection} \quad & \rho_{A\cap B}(l) = && \min(\rho_A(l), \rho_B(l)) \\ \text{linear transformation} \quad & \rho_{\mathcal{A}\cdot A}(l) = && \rho_A(\mathcal{A}^T \cdot l) \\ \text{Minkowski sum} \quad & \rho_{A\oplus B}(l) = && \rho_A(l) + \rho_B(l) \end{aligned}$$

Once a state set $A \subseteq \mathbb{R}^d$ has been instantiated, in HyPro all operations on it are effectively stored in a tree-like structure, the *operation tree* where each concrete set A_i is a leaf node. During the analysis, this tree will grow and branch depending on the operations performed. Predecessor nodes are added to the root node, which reflects the result of the applied operation. The support of the current set represented by the root node of the operation tree for a direction l can be computed recursively by traversing said operation tree: operations such as linear transformations modify the requested direction l and forward the request to their child node while binary operations such as intersection or Minkowski sum will return an aggregation of the results obtained by recursive calls (see Figure 6.20).

Example 6.3: Support Function Operations

Consider the method of computing the first segment, as presented in Section 3.4. The operation tree of a support function for this first segment looks as follows:



where I denotes the passed initial set and $\mathcal{A}$ is the matrix describing the flow in the current location.

Note that in theory, we consider recursive operation-tree traversal while in practice we traverse the operation tree in an iterative fashion to avoid the overhead on the stack introduced by recursive calls as the operation tree can quickly outgrow hundreds of nodes during analysis.

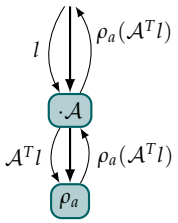


Figure 6.17: Linear transformation of a support function.

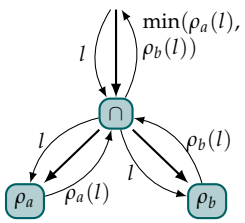


Figure 6.18: Intersection of two support functions.

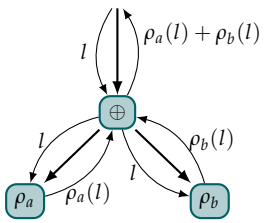


Figure 6.19: Minkowski sum of two support functions.

Figure 6.20: Operations on support functions can be stored in a tree-like data structure. Note that union of support function works analogously to the Minkowski sum but result aggregation from the sub-trees works differently (see Section 6.4).

Union

As stated above, the union of two support functions can be modeled as a binary operation combining two operation trees. Sampling the support for the unified state set results in sampling both sub-trees individually and taking the maximum support value, i.e., computing the supporting hyperplane including both sets (see Figure 6.21). As the trees are sampled sequentially, further sampling can be avoided in case it can be determined that the result in one subtree is unbounded.

Intersection

A simple implementation of the intersection of two support functions ρ_A and ρ_B can be computed similarly to the union operation by sampling all subtrees and collecting the results $\rho_A(l)$ and $\rho_B(l)$ for a specific direction l. A valid over-approximation of $\rho_{A\cap B}$ for l is given as

$$\rho_{A\cap B}(l) = \min\{\rho_A(l), \rho_B(l)\} \ .$$

This approach is implemented in HyPro, however there are more sophisticated approaches [LG10; FR12], as this method may result in coarse over-approximations.

Intersection with a Half-space

The intersection of a support function ρ_S of a set $S \subseteq \mathbb{R}^d$ and a half-space $h = \{x \mid n^T \cdot x \leq c\}$ may be modeled by considering h as a polytope with only one constraint and using a fall-back to the regular implementation of support function intersection and the corresponding methods to compute the support for the resulting set. Note that computing the support of a single half-space in direction l results in $+\infty$ as a support value for all choices of l unless $l = \lambda \cdot n, \lambda \in \mathbb{R}_{>0}$ which can be exploited to speed up operation tree traversal.

a) Computing the support of two sets in direction l by returning the supporting hyperplane with the largest support value.

b) Computing the support of the union of two support functions in infinitely many directions will return the supporting hyperplanes of the convex hull of the two sets.

Figure 6.21: Computing the support of the union of two support functions returns the maximal support value under all sets that are unified.

Affine Transformation

Computing an affine transformation $\rho'_S = \mathcal{A} \cdot \rho_S + b$ of a support function ρ_S of a d-dimensional set S adds a single node as a parent to the current node in the operation tree as an unary operation. In a basic setup, the node stores the transformation parameters, i.e., the matrix $\mathcal{A} \in \mathbb{R}^{d \times d}$ and the vector $b \in \mathbb{R}^d$. During the sampling of ρ_S, i.e., computing the support for a given direction l, l is transformed to $l' = \mathcal{A}^T \cdot l$ (see Figure 6.17). Intuitively, instead of computing the support for a set $\rho'_S(l) = (\mathcal{A} \cdot \rho_S)(l) = \rho_S(\mathcal{A}^T \cdot l)$, the direction l is transformed to return the same result but without modifying the original set. An intuition is depicted in Figure 6.22, where instead of transforming a set S and then computing the support towards a direction l, the support of S for a modified direction l' is computed which returns the same result.

Computing the Support

While all operations on support functions correspond to insertions in the operation tree, in essence just performing those operations can be done in constant time. Computing the support of a set, which is the central operation required for support functions is the most costly operation. In general, computing the support of a given support function requires to traverse its operation tree recursively. Thus, the computational effort required depends on the number of nodes in the search tree and especially on the number of leaf-nodes. While all non-leaf-nodes in the search tree do not require much computational effort during the tree traversal, in all leaf-nodes the support of a particular state set in its specific representation has to be computed. Consequently, the state set representation for leaf-nodes strongly influences the running time of a flow-pipe-construction-based reachability analysis method in which computing the support of a set is an essential operation.

Even though in theory the traversal of the operation tree can be done recursively, during experiments the increasing depth of the search tree and thus the increased number of recursive function calls uses a large amount of memory on the stack. To overcome this, we have implemented a generalized scheme

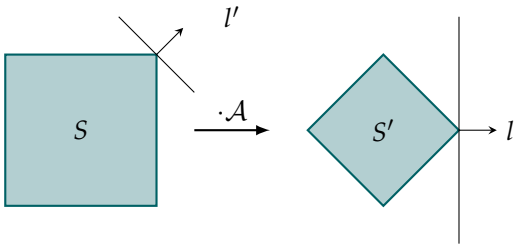


Figure 6.22: Intuition behind computing the support for a linearly transformed set: instead of transforming S and computing its support towards l, the support of S in direction l' is computed which allows to avoid potentially expensive transformations to S'.

that turns the recursive sampling calls for each node into an iterative approach. The basic idea of this algorithm follows a classical approach to mimic stacks of recursive calls. We maintain three stacks, one for the parameters; one to keep track of the recursive function calls; one for the return values. Depending on the type of node and the traversal direction, parameters are transformed and recursive calls simulated, or results are aggregated and passed to the calling frame. Transformation of parameters and aggregation of the results depends on the called function—a generalized implementation which we are working on uses placeholders for those functions which are instantiated depending on the type of operation.

Operation Tree Reduction

As the size of the operation tree stored for a set S represented by a support function increases with every operation, methods to reduce the search tree have been developed. In the following, we will present our ideas on operation tree reduction as implemented in HyPro.

Linear Transformation Reduction. The results presented in this section are based on results obtained in the Master's thesis of Phillip Florian [Flo16]. In flowpipe-construction-based reachability analysis for linear hybrid systems, after having computed the first flowpipe segment, the next segments are obtained by repeated linear transformation (see Section 3.4) with the same matrix $\mathcal{A} \in \mathbb{R}^{d \times d}$. To reduce storage, we do not store $\mathcal{A}$ in each operation, but rather provide a container holding $\mathcal{A}$.

For each set, containment in the invariant, intersection with the set of bad states and enabledness of transitions have to be checked by applying an intersection operation with the respective state set. Provided these tests are performed directly, and intersection operations only added to the operation tree in case the intersection modifies the current state set and otherwise are left out, in many cases we can obtain sequences of linear transformation operations only as shown in Figure 6.23.

These sequences provide a convenient way to reduce the length of paths in the operation tree of a support function object. Sequences of n linear transformations

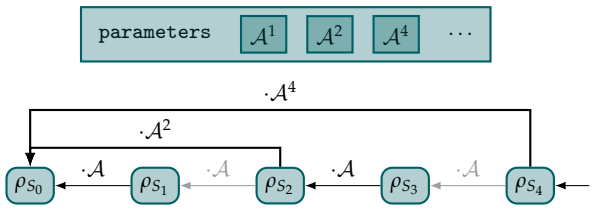


Figure 6.23: Reduction of the operation tree of a sequence of linear transformations of a support function using a base of 2. The actual parameters are stored in a shared parameter container to avoid duplicates.

with the same matrix $\mathcal{A}$ can be converted to one transformation using $\mathcal{A}^n$. While this reduces the path length effectively to one in any case, it involves an additional matrix multiplication for each new transformation $n+1$ to obtain $\mathcal{A}^{n+1}$, i.e., $\mathcal{A}^6 = ((((\mathcal{A}\cdot\mathcal{A})\cdot\mathcal{A})\cdot\mathcal{A})\cdot\mathcal{A})\cdot\mathcal{A}$. An extension of the parameter container to provide and compute powers of the initial matrix $\mathcal{A}$ effectively reduces the number of matrix multiplications, as powers of matrices can be computed effectively using fewer multiplications. When reusing parameters, the number of required multiplications to compute for instance $\mathcal{A}^6$ can be reduced to

$$\mathcal{A}^6 = \underbrace{(\underbrace{(\mathcal{A}\cdot\mathcal{A})}_{\mathcal{A}^2}\cdot\mathcal{A})}_{\mathcal{A}^3}\cdot((\mathcal{A}\cdot\mathcal{A})\cdot\mathcal{A}) = ((\mathcal{A}\cdot\mathcal{A})\cdot\mathcal{A})\cdot\mathcal{A}^3$$

which uses only three matrix multiplications as results from the left side ($\mathcal{A}\cdot\mathcal{A}$ and $((\mathcal{A}\cdot\mathcal{A})\cdot\mathcal{A})$) can be reused in comparison to five multiplications using the naive approach. We generalize the above approach by specifying a basis b of which powers will be stored and reused. The above example when using $b = 2$ collapses to

$$\mathcal{A}^6 = \underbrace{(\underbrace{(\mathcal{A}\cdot\mathcal{A})}_{\mathcal{A}^2}\cdot(\mathcal{A}^2))}_{\mathcal{A}^4}\cdot(\mathcal{A}^2).$$

In general, a chain of length n of consecutive linear transformations using the same matrix can be reduced to length $\log_b(n)$ where b denotes the base of the reduction. In the above case $b = 2$, i.e., the length of the chain is equal to the number of ones in the bit-representation of the exponent n, while we can also choose less-aggressive reductions by setting b to higher values.

Different chain lengths with relation to the basis are depicted in Figure 6.24. The plot relates the targeted exponent and the actual chain length for different choices of base b. In the plot, we can see that reduction allows us to keep the chain length in the search tree small even for high exponents.

The total number of multiplications m required including computing all intermediate matrices for basis b can be computed in the following way: Let $[n]_b$ be the representation of n using the basis b and let $[n]_b(i)$ be the digit of

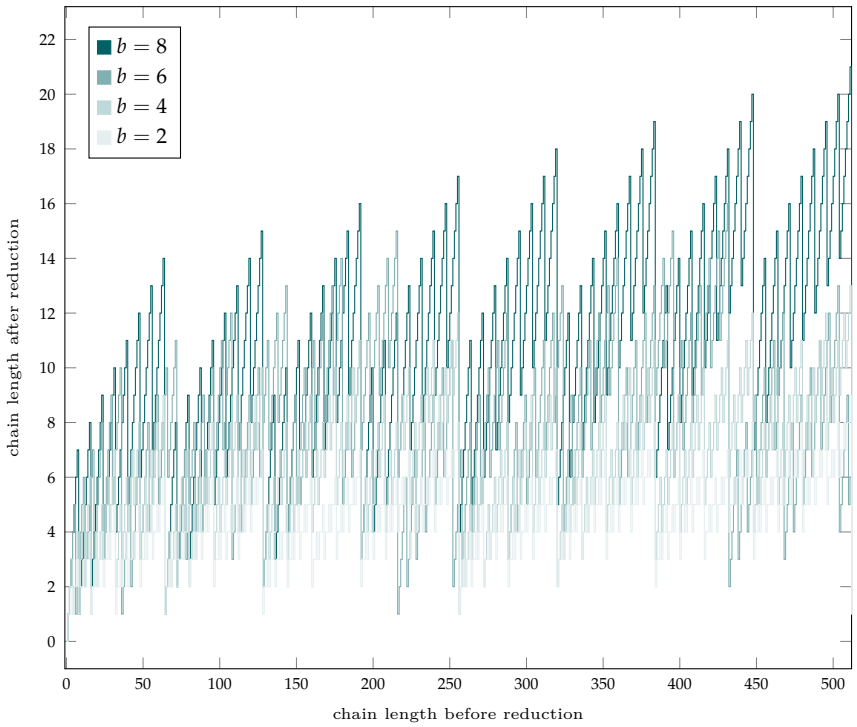


Figure 6.24: Transformation chain length depending on the reduction base. The plot shows the resulting chain length over the number of transformations up to 500 transformations.

$[n]_b$ at the i-th position (from highest to lowest) and $|[n]_b|$ refers to the total number of digits in $[n]_b$ with $[n]_b \neq 0$. Then it holds that

$$m = \underbrace{|[n]_b|}_{base} + \underbrace{[n]_b(0) - 1 + \sum_{i=1}^{|[n]_b|-1} [n]_b(i)}_{rest} \ .$$

The first part *base* refers to the number of multiplications required to compute the highest-valued digit in the representation of n in base-b, i.e., to compute all required intermediate matrices. The second part *rest* accounts for the number of multiplications using the previously computed matrices. Note that we subtract one as the computation of the highest-valued digit has been covered once in the first part already.

Example 6.4: Support Function Chain Reduction

If we want to compute $\mathcal{A}^{3142}$ using a basis $b = 2$, the total required number of multiplications can be computed as follows:

$$[3142]_2 = 110001000110 \ .$$

The highest non-zero exponent $[3142]_2(0)$ equals to 2048, i.e., 2^{11}, which requires 11 multiplications to compute

$$\mathcal{A}^{2048} = \underbrace{(\underbrace{(\mathcal{A}\cdot\mathcal{A})}_{\mathcal{A}^2}\cdot\mathcal{A}^2)}_{\mathcal{A}^4}\cdot\mathcal{A}^4\cdot\ldots\cdot\mathcal{A}^{1024}\ .$$

Once we have all intermediate matrices, we need to account for the non-zero digits which adds another four multiplications totaling in 15 required matrix-matrix multiplications to compute $\mathcal{A}^{3142}$ and results in a chain of length five.
If we choose $b = 4$, we obtain

$$[3142]_4 = 301012$$

which thus results in a chain of length seven but requires only six multiplications for the highest-valued digit.

Different bases provide an approach to reduce the required number of multiplications for the computation of the highest-valued digit at the cost of a longer chain. At first, this might result in fewer multiplications (see Example 6.4); however, in general, shorter chains are favorable. During flowpipe construction for a linear hybrid system, consecutive segments are related by a single linear transformation. Thus, a chain of transformations is built up during construction iteratively, which implies that the cost for the base case (*base*) is limited to at most one multiplication, in case a new matrix-power is required. Additionally, various samplings of a support function occur during computation, in which the length of a chain plays an essential role concerning the running times.

Note that this provides a method to reduce the operation tree depth on the fly during computation, as the links between the operation nodes can be updated upon request. A pseudo-code version of the reduction mechanism of the operation tree implemented in HyPro is shown in Algorithm 5.

Upon construction of a new node for a linear/affine transformation in the operation tree, chains of length 2^b where b is the chosen base are detected and iteratively collapsed. Each transformation node stores its exponent, which is also updated whenever a part of the chain is collapsed.

Tree Reset. Even though we provide methods to reduce the operation tree of a support function during computation as much as possible, e.g., collapsing linear transformations or removing intersection operations which do not affect the object, eventually the size of the operation tree will become too large. Large operation trees do not only require large amounts of storage but also have a strong influence on the running times when computing the support of a set S, as the whole operation tree has to be traversed.

To obtain an operation tree with a single node, we can compute an over-approximation of the current set S by using *template-based evaluation* i.e., we try

Algorithm 5: Chain reduction for linear transformations of support functions invoked whenever a new linear transformation is added to the operation tree.

Input: Operation tree root n, transformation matrix $\mathcal{A}$, base b

```
child := n                                              // set child node
exp := 1                                                // set current exponent
if n is a linear transformation operation then
    reduced := false
    repeat
        /* get number of consecutive linear transformations using exp.
           */
        tCount := getTrafosWithExp(exp,n)
        if tCount = 2^b then
            exp = exp·2^b                               // update exponent
            /* new child node: skip 2^b − 1 nodes in chain                */
            child = updateChild(2^b − 1) reduced = true
    until reduced=false
else
    createParContainer(A)                               // initialize param. container
```

to obtain a polyhedral approximation of S based on a template and use it as a fresh leaf node in the operation tree. A template T is a set of d-dimensional vectors $t \in T$. Using each vector t as a normal vector for a hyperplane, we can obtain a convex polytope S_H in $\mathcal{H}$-representation by computing the support of S for each $t \in T$:

$$Set(S_H) = \left\{ x \;\middle|\; \forall t \in T.\ t^T \cdot x \leq \rho_S(t) \right\} .$$

As we have to traverse the whole operation tree to obtain the offset for each supporting hyperplane, the number of vectors in T influences the computational cost of this reduction. On the other hand, having more vectors in T generally increases the precision of the approximation. The actual shape of S usually is not known, and the exact set S can be obtained only by sampling all vectors in $\mathbb{R}^d$. While this is in general not possible, it is common to use uniformly distributed vectors, e.g., an octagonal template over all pairs of dimensions $i, j \in \{0, \ldots, d-1\}, i \neq j$

$$T = \Big\{ t \in \mathbb{R}^d \;\Big|\; \exists 0 \leq i, j < d.\ i \neq j \wedge t_i \in \{-1, 1\} \wedge t_j \in \{-1, 0, 1\} \wedge (\forall 0 \leq k < d.\ (k \neq i \wedge k \neq j) \Rightarrow t_k = 0) \Big\} .$$

In general, other templates are possible as well, for instance using only two directions per dimension

$$T = \left\{ t \in \mathbb{R}^d \;\middle|\; t_k = 0, t_i \in \{-1, 1\}, i \neq k \right\},$$

i.e., a box-template allows to use a box as a leaf node, which provides more efficient methods to compute the support (see Section 6.2) at the cost of reduced

precision. In our implementation, we provide methods to syntactically detect whether a leaf node S of a support function object is box-shaped, i.e., whether the set of half-spaces describing S conform to a template with four directions as described above.

Experimental Results

In this section we will present selected experimental results from testing our implementation of support functions. All results were run on a machine with $4 \times 4\,\text{GHz}$ Intel Core i7 CPUs and a memory limit of $8\,\text{GiB}$. To benchmark operations on support functions, we consider two basic setups: one where just the operation is applied and a second one where a variety of samples is requested after the operation. The reason for this is that in general support functions only store an operation tree, i.e., only the sequence of applied operations. Thus, performing single operations is usually very fast, unless post-processing slows down the process. In contrast to this, requesting a sample for a given support function reveals the actual running time of the whole operation as the complete operation tree has to be traversed. To account for this, the second setup, which includes sampling the support function measures, not only the time to apply the operation is measured, but the total time to obtain the supporting hyperplane for the resulting support function as well.

Affine Transformation. Our implementation of affine and linear transformation features reduction of chains of consecutive applications of the same transformation. To benchmark applying an affine transformation to a support function, we use the following setup in which we consider chains of transformations of different lengths. Furthermore, to indicate the impact of our optimizations we consider different numbers of sampling-requests afterwards. The reason for this is that while in a very basic setup the application of a transformation itself (without sampling) is expected to be constant, i.e., only the cost for storing the operation parameters, our optimizations introduce costs for reduction and post-processing. However, we expect that these optimizations pay off in terms of running time with increased requests for sampling. We consider three different setups: A basic setup (`basic`) representing a straight-forward implementation in which each operation is stored along with its parameters in the operation tree; a setup (`redI`) in which a set of parameters is only stored once and shared each time it is used; a setup (`redII`) in which chains of consecutive transformations are reduced according to Section 6.4. While the setup `basic` does not add any additional post-processing cost in terms of running time it comes at an increased storage usage. Setup `redI` overcomes this by searching the already existing operation tree for the same set of parameters and uses them as a reference to reduce storage. Traversing the tree and comparison of parameters however adds additional running time. The third setup `redII` additionally tries to collect chains of consecutive transformations as presented in Section 6.4, which comes at an additional cost of reducing the operation tree. We expect that while setups `redI` and `redII` come at higher fixed costs when applying the operation, the cost of sampling can be reduced drastically using these approaches.

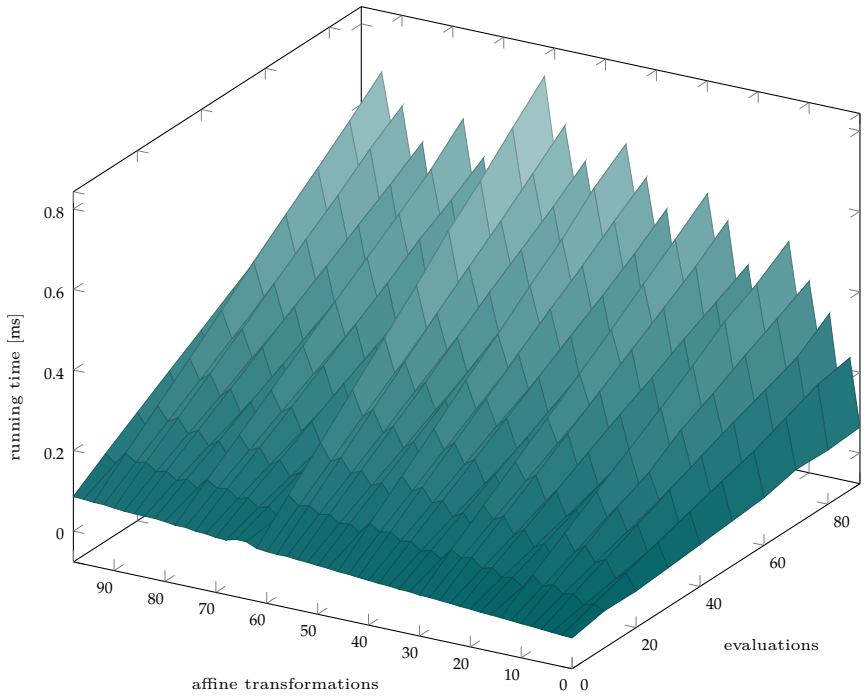


Figure 6.25: Running times obtained with different number of affine transformations and different number of samplings of a support function using chain reduction (`redII`) as presented in Section 6.4 with a base of two in a one-dimensional state space.

A plot relating the number of affine transformations in combination with different number of samplings using chain reduction with the running times can be seen in Figure 6.25. The results of Figure 6.25 were obtained for computations in a one-dimensional state space and the running times represent averages of 1000 repetitions of the respective setup. We chose a one-dimensional state space to be able to depict and analyze the effects of chain reduction only and minimize other influences such as caching as much as possible. Results from configurations with a higher-dimensional state space are shown later. Figure 6.26 shows the running times of the same setup but without chain reduction. From both plots, we can observe that the running times increase linearly with the increase of the number of samplings, which matches our expectations. Furthermore, a cross-section in Figure 6.25 for a fixed number of evaluations shows a similar structure as in Figure 6.24 and shows the strong relation between chain length and running time. The total speed-up by using chain reduction in comparison to the original method is depicted in Figure 6.27. The running times for this plot were obtained using a 100-dimensional state space. We can observe that for a small number of samplings the speed-up is below one, i.e., only for a reasonably large number of samplings the method pays off. Similar to the other plots, a cross-section in the speed-up vs. transformations plane resembles the structure as presented in Figure 6.24.

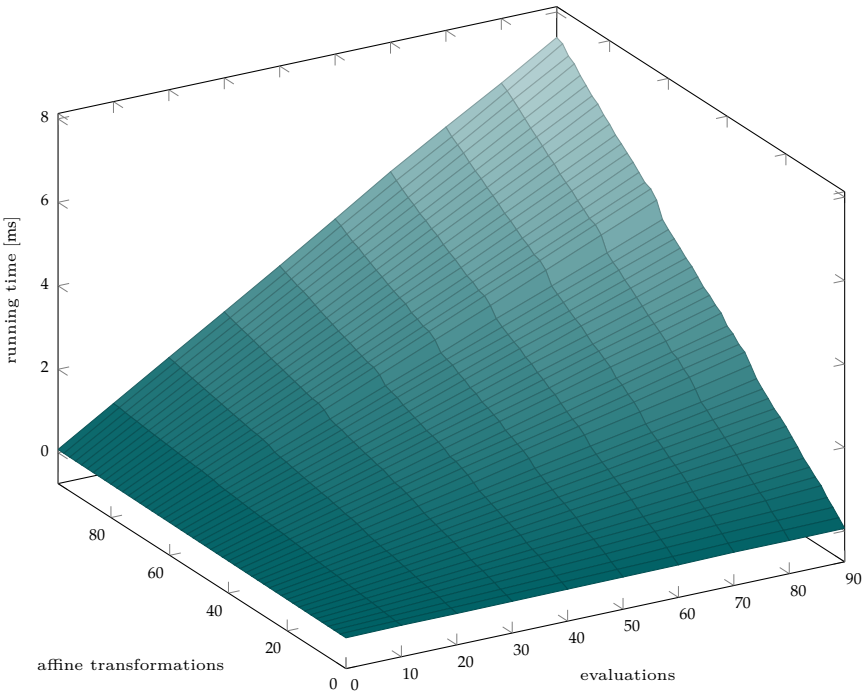


Figure 6.26: Running times obtained with different number of affine transformations and different number of samplings of a support function without chain reduction (`basic`). The underlying state space was one-dimensional.

Sampling. Sampling a support function itself highly depends on the operation tree but also on the set representation at the leaf-node of the operation tree. In this paragraph, we will discuss the sampling process of a leaf-node itself. Apart from representation types such as a ball (in some norm), we consider convex polytopes in $\mathcal{H}$-representation as a possible leaf-node. In general, the sampling of a polytope is equivalent to linear programming; in special cases we can speed up the process, e.g., when the polytope is a box (see Section 6.4).

Note that for all examples of hybrid systems we are aware of, the initial set is represented as a box, i.e., in any case improved sampling using boxes can be used on the first flowpipe. Furthermore, if operation tree reduction with a uniform template in four directions is used (see Section 6.4) the resulting polyhedral approximation is a box—using this approach we can significantly reduce running times. The effects on running times are similar to the effects observed for computing the support of a box (see Section 6.2), which is why we will not repeat the results here for single operations. Instead Table 6.2 shows the cumulative effects using this approach in a flowpipe-construction-based reachability analysis implementation on a selection of benchmarks from our collection. We consider three settings for support functions: (i) no detection of boxes (`plain`), (ii) detection of boxes (`detect`), and (iii) detection and reduction of and to boxes after discrete jumps (`reduce`). When using the setting `detect`, during the analysis sets which are box-shaped will be treated

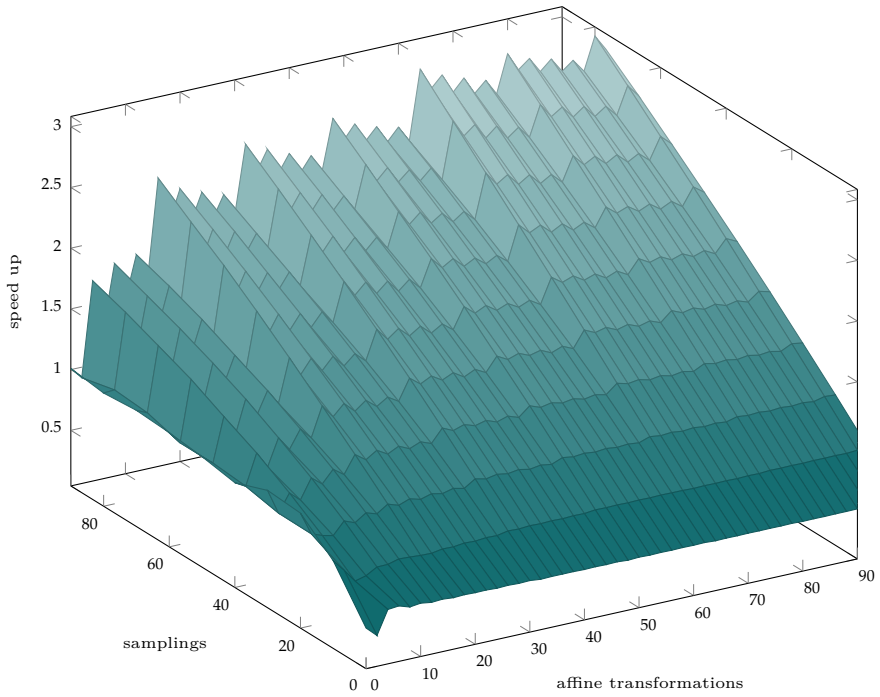


Figure 6.27: Speed-up comparing no chain reduction (`basic`) to running times obtained when using chain reduction techniques (`redII`) in a 100-dimensional state space.

Table 6.2: Running times in seconds for different support function sampling methods during flowpipe-construction-based reachability analysis. All benchmarks were run using a time step size $\delta = 0.001$ and full aggregation. We compare three settings: no optimization (`plain`), detection of box-shaped sets (`detect`), and reduction to boxes (`reduce`).

Model	`plain`	`detect`	`reduce`
`Ball`	0.49	0.46	**0.34**
`Sw5`	0.38	0.37	**0.13**
`2Tnk'`	**0.32**	0.59	0.85
`Pltn`	26.13	25.62	**1.12**
`Bld`	8.12	**2.21**	2.23

as boxes. In addition, when using the setting `reduce`, state sets will be reduced (template-based) to boxes after taking discrete jumps such that initial sets for new flowpipes are always box-shaped. The setting `plain` omits all of these optimizations and treats all sets which can be represented as a finite intersection of half-spaces as convex polytopes. All benchmarks were run using the same time step size $\delta = 0.001$ and full aggregation of sets satisfying a guard condition.

In general, detecting whether an initial set is box-shaped (`detect`) and in this case using a box representation already improves running times. As

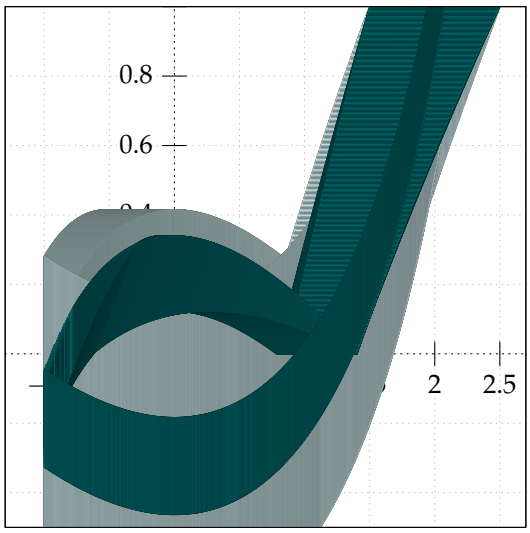


Figure 6.28: Comparison of precision using different reduction techniques for support functions. The plot shows both filling levels of a variant of the two tanks system computed using $\delta = 0.001$ and full aggregation. The approximation using the setting without box-reduction (`plain`) is depicted in petrol, the results when using box-reduction (`reduce`) are depicted in light petrol.

usually only the initial set of the first flowpipe is box-shaped, the box-detection approach only affects the computations of the first flowpipe-approximation. The building benchmark (`Bld`) which is a purely continuous system thus profits from this setting, as only one flowpipe is over-approximated. For all other problem instances the effect of a decreased running time is observable, however not significant, as further flowpipe computations cannot profit from this approach.

The running times improve considerably, if not only box-shaped initial sets are detected, but if also support function objects after a discrete jump are reduced (`reduce`) by using a box-shaped approximation instead of a polytopal approximation based on an octagonal template. Note that this setting may reduce the precision of the approximation, as the polytopal approximation using a box-template usually is less precise than using an octagonal template (see Figure 6.28). As a consequence of the loss of precision, it may happen that more flowpipe segments need to be computed, which results in longer running times, but also reflects in the plotted over-approximation of the set of reachable states, e.g., for a variant of the two tanks system (`2Tnk'`)[2].

Flowpipe approximations by support functions using reduction to boxes show improved running times, if the number of segments increases only by little, as the sampling of sets now is faster. Using detection of box-shaped initial sets in combination with reduction to boxes, we can observe improvements of up to 95 % (`Pltn`) in running times. The building system, which only requires approximation of one flowpipe consequently does not exhibit improved running times using this setting.

[2]This is the version without a PLC-controller attached.

6.5 Zonotopes

Zonotopes have been in focus of the research community for a long time and often are considered as a suitable state set representation, especially for purely continuous systems [Gir05; ASB10]. In this section, we briefly introduce zonotopes as a state set representation and highlight some of their features. The implementation of zonotopes in HyPro has been provided by Ibtissem Ben Makhlouf and is taken from her tool HyReach [BHK16]. Approaches implemented are mainly based on [ABC05; Gir05; GL08].

Definition 6.4: Zonotope

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

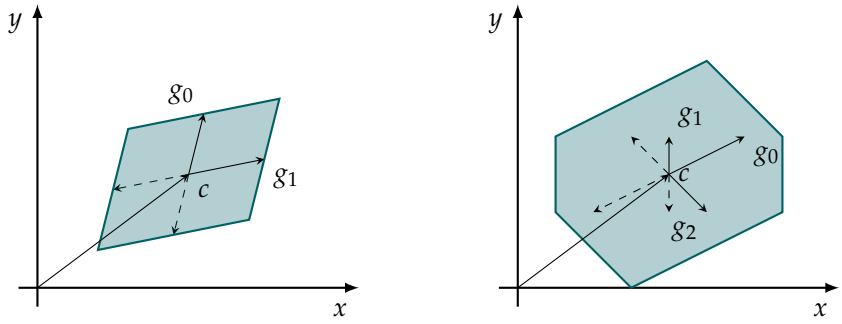
$$A_Z = \left\{ c + \sum_{i=0}^{n-1} \lambda_i \cdot g_i \;\middle|\; \lambda_i \in [-1,1] \subseteq \mathbb{R}^d \right\}$$

and we use $|A_Z|$ to denote the number n of generators used to represent A_Z .

We use the vector notation for a zonotope $Z = (c, g_0, \ldots, g_{n-1})$ where the first element always refers to the center and the remaining elements represent the generators. Intuitively, a zonotope is a convex polytope which is point-symmetric to the center c and it is spanned by the Minkowski sum of a finite set of line segments $l_i = \lambda_i \cdot g_i, \lambda_i \in [-1,1]$. From another point of view, a d-dimensional zonotope with $n \geq d$ generators can be seen as the projection of an n-dimensional box on a d-dimensional space (see for example Figure 6.29b). The more generators a certain zonotope has while the state space dimension remains fixed, the more complex the object gets and the less information per generator is added. Informally, the object becomes more and more "round" with an increased number of generators. The *zonotope order* $\frac{|Z|}{d}$ relates the dimension of the ambient space and the number of generators in a zonotope Z to each other and provides a convenient metric for the complexity of Z. Several reduction techniques have been developed to keep the order and thus the representation size within certain bounds during computation [Gir05; ASB10]. After a short overview of the implementation of operations on zonotopes, in Section 6.5 we will present two order reduction techniques implemented in HyPro; one of them which was provided by Ibtissem Ben Makhlouf while we added the other one.

Operations

In this section, we present the general idea and methods to perform the most commonly used operations required during flowpipe-construction-based reachability analysis when using zonotopes as a state set representation. In the following, assume two d-dimensional zonotope representations $A_Z = (c_a, a_0, \ldots, a_{n-1}), B_Z = (c_b, b_0, \ldots, b_{m-1})$.



a) A two-dimensional zonotope created from two generators.

b) A two-dimensional zonotope created from three generators.

Figure 6.29: Two-dimensional zonotopes spanned by different sets of generators.

Union

The convex hull of the union of A_Z and B_Z can be over-approximated by the zonotope

$$cl(A_Z \cup B_Z) = \left(\frac{c_a + c_b}{2}, \frac{a_0 + b_0}{2}, \ldots, \frac{a_{n-1} + b_{n-1}}{2}, \frac{c_b - c_a}{2}, \frac{a_0 - b_0}{2}, \ldots, \frac{a_{n-1} - b_{n-1}}{2} \right)$$

assuming $n = m$. If this is not the case, the additional generators are added to the result. This approach has been presented in [Gir05] and provides a rough, yet simple method to over-approximate $cl(A_Z \cap B_Z)$.

Intersection

Zonotopes are not closed under intersection which renders intersection operations (both, zonotope-zonotope and zonotope-half-space) a difficult task. The difficulty lies in constructing a small zonotope that contains the result. In HyPro, several methods [ABC05; Gir05] have been implemented for the zonotope-zonotope intersection as well as the zonotope-half-space intersection, which can be chosen by the user.

Minkowski Sum

Zonotopes specify a set of points by the Minkowski sum of a finite set of line segments (see Definition 6.4). Consequently, zonotopes are closed under the Minkowski sum. Thus, the Minkowski sum of two zonotopes representations A_Z and B_Z can be computed as

$$A \oplus B = (c_a + c_b, a_0, \ldots, a_{n-1}, b_0, \ldots, b_{m-1}) \ .$$

In essence, the Minkowski sum of two zonotopes can be computed by unifying the sets of generators and updating the center of the resulting zonotope.

Affine Transformation

Computing the result of applying a linear map $\mathcal{A} \in \mathbb{R}^{d \times d}$ on a d-dimensional zonotope representation $Z_Z = (c, z_0, \ldots, z_{n-1})$ is reflected in applying $\mathcal{A}$ on its generators and the center individually

$$\mathcal{A} \cdot Z_Z = (\mathcal{A} \cdot c, \mathcal{A} \cdot z_0, \ldots, \mathcal{A} \cdot z_{n-1}) \ .$$

The extension towards an affine transformation, i.e., adding a translation by a vector $b \in \mathbb{R}^d$ can simply be computed by adding the translation to the center point

$$A \cdot Z_Z + b = (\mathcal{A} \cdot c + b, \mathcal{A} \cdot z_0, \ldots, \mathcal{A} \cdot z_{n-1}) \ .$$

Computing the Support

The support for a zonotope $Z_Z = (c, z_0, \ldots, z_{n-1})$ for a direction $l \in \mathbb{R}^d$ can be computed component-wise by projecting l on each generator z_i and the center c to obtain

$$\rho_{Z_Z}(l) = |\langle l, c \rangle| + \sum_{i=0}^{n-1} |\langle l, z_i \rangle| \ .$$

Computing the Vertices

Especially for conversion methods, it is sometimes required to compute the vertices of a bounded set A whose convex hull defines A. For a zonotope representation A_Z of A where $Set(A_Z) \supseteq A$, we provide a recursive method in HyPro which is based on the iterative construction of a zonotope $A_Z = (c, a_0, \ldots, a_{n-1})$ with n generators as shown in Algorithm 6.

Order Reduction

In this section, we briefly present the ideas towards zonotope order reduction which have been implemented in HyPro. As a reminder, the order $p = \frac{|Z_Z|}{d}$ of a d-dimensional zonotope Z_Z reflects the relative complexity of the set represented. As operations such as the Minkowski sum of two zonotopes increase the order of the resulting polytope, reduction of the representation at the cost of over-approximation is performed in case the zonotope order exceeds a given bound.

The first method implemented in HyPro was first presented in [Gir05]. The general idea is to replace $2d$ generators of a d-dimensional zonotope Z_Z by d generators. To achieve this, the interval hull B of the $2d$ selected generators is computed by computing the sum of the absolute values of the generators component-wise. Afterwards those $2d$ generators can be replaced by d generators defining B.

A second approach is based on conversion methods for zonotopes in HyPro. The idea of this approach is to use a principal component analysis (PCA) of the vertices of a given zonotope Z to obtain an oriented rectangular hull Z' which contains Z. As rectangular hulls are a subclass of zonotopes, this approach provides a simple method to significantly reduce the number of generators in Z to d generators. We provide a detailed description of the approach in Section 6.8.

Algorithm 6: Recursive construction of $vertices(A_Z)$ of a zonotope A_Z.

Input: A zonotope representation $A_Z = (c, a_0, \ldots, a_{n-1})$
Output: The set of vertices V of A_Z

```
Function vertices(A_Z)
    cur = c
    if |A_Z| ≠ 0 then
        pos := cur + a_0                    // add positive generator
        neg := cur − a_0                    // add negative generator
        /* recursive calls with one generator less                  */
        if |A_Z| > 1 then
            vPos := vertices((pos, a_1, ..., a_{n−1}))
            vNeg := vertices((neg, a_1, ..., a_{n−1}))
            V := vPos ∪ vNeg
        else
            V := {pos, neg}
    else
        V := {cur}
    return V
```

6.6 Further State Set Representations

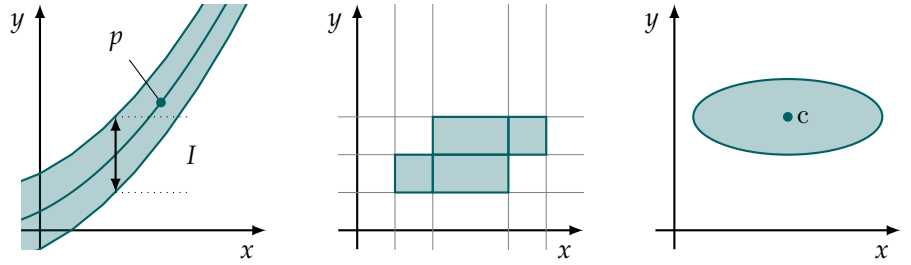
Here, we shortly present other state set representations which can also be used for flowpipe-construction-based reachability analysis but which are only partially implemented in HyPro.

Taylor Models

Taylor models as a symbolic representation for state sets have shown great potential for the analysis of non-linear hybrid systems [CÁS12; CÁS13], for an exemplary illustration see Figure 6.30a. A basic implementation of Taylor model arithmetic as implemented in Flow* [CÁS13] is shipped with HyPro. As reachability analysis for hybrid systems based on Taylor models has been successfully implemented in Flow*, the focus of our work on HyPro lies on state set representations for linear hybrid systems reachability analysis.

Orthogonal Polyhedra

HyPro offers a basic implementation of *orthogonal polyhedra*, which have been used in the early years of flowpipe-construction-based reachability analysis for hybrid systems [Dan00]. The advantage of orthogonal polyhedra lies in their capability of representing non-convex sets by a union of hyper-rectangles aligned on a non-uniform grid (see Figure 6.30b). Basic operations such as intersection or union on orthogonal polyhedra are implemented in HyPro, however, operations such as linear transformation and Minkowski sum are missing as the former requires to introduce heuristics to determine the resolution of the underlying



a) Taylor models approximate a set by a polynomial p of bounded degree and a remainder interval I building a tube around p.

b) Orthogonal polyhedra are non-convex sets as a union of hyper-rectangles on a non-uniform grid.

c) An Ellipsoid is an affinely transformed unit ball.

Figure 6.30: Illustration of other state set representations which are partially implemented in HyPro.

grid to represent non-axis-aligned constraints and for the latter non-convexity of the state sets renders an efficient implementation difficult. Consequently, the implementation of orthogonal polyhedra does not conform to the general interface, which renders them not usable in our generalized algorithm at the moment.

Ellipsoids

Ellipsoids have been also used as a state set representation for flowpipe-construction-based reachability analysis to represent state sets [KV00; KV07]. In recent tools, ellipsoids are rarely considered, as they are not closed under most operations such as intersection, union, or Minkowski sum.

The d-dimensional ellipsoid representation $A_E = (c, \mathcal{Q})$ represents the set

$$Set(A_E) = \left\{ x \in \mathbb{R}^d \;\middle|\; \forall l \in \mathbb{R}^d.\ \langle l, x \rangle \leq \langle l, c \rangle + \sqrt{\langle l, \mathcal{Q} \rangle} \right\}$$

where c denotes the center of the ellipsoid and $\mathcal{Q} \in \mathbb{R}^{d \times d}$ is a positive semi-definite matrix referred to as a *shape matrix*. Intuitively, an A_E is a ball centered at c, which is linear transformed using a matrix $\mathcal{Q}$ (see Figure 6.30c).

Ellipsoids are closed under linear and affine transformations. Furthermore, the Minkowski sum of two ellipsoids can be over-approximated efficiently and tightly for a given direction $l \in \mathbb{R}^d$ [KV07].

In HyPro, we provide an implementation of ellipsoids with an interface for linear and affine transformation as well as the Minkowski sum. In the context of the Master's thesis of Phillip Florian [Flo16], we have tested ellipsoids as a state set representation for computing the influence of external input during flowpipe-construction-based reachability analysis based on the method presented in [Le 09]. In this approach, the computation of external input for non-autonomous linear hybrid systems is decomposed into computing flowpipe

segments for the set of reachable states of the autonomous part (as presented in Section 3.4) and the computation of the influence of the external input. The later computation requires alternating sequences of Minkowski sum operations and linear transformations of state sets for which ellipsoids are well-suited.

6.7 General Optimizations

In this section, we will highlight general features implemented in HyPro which are relevant for implementing a flowpipe-construction-based reachability analysis method.

Linear Optimization and Numerical Precision

Several operations on different state set representations require linear programming (LP), e.g., computing the support of a convex polytope. In HyPro, we provide a generalized linear optimization fronted which allows integrating several different backend LP solvers. Currently supported solvers are GLPK, SOPLEX, SMT-RAT, and Z3. HyPro allows to switch between exact and inexact arithmetic through number representation, i.e., the user may choose to use native `C++ double` numbers or rational types, for instance, the GMP rational type `mpq_class`. Utilization of different number types is realized via templates, however, adjustments are needed for exact types, e.g., methods for the reduction of the number representation (see Section 6.3) or specialized methods for the comparison to `double` numbers need to be provided. Most linear optimization solvers require to specify the input using double numbers and correspondingly double precision is used during computation. However, solvers as GLPK or SOPLEX allow for both, floating-point arithmetic as well as problems specified by rational types. Furthermore, satisfiability modulo theories (SMT) solvers such as SMT-RAT or Z3, which also allow for optimization utilize rational arithmetic by default and provide precise results. In HyPro, the linear optimization frontend does not only provide wrappers for all mentioned solvers but furthermore allows to combine their usage (see Figure 6.31). The idea is similar to the one presented in [Mon09] in which a more precise SMT linear real arithmetic (LRA) solver working with rational arithmetic is initialized with a solution and the internal structure obtained by a less-precise solver using double arithmetic.

In our work we implement a similar approach for solving linear optimization problems $\max_{x \in X} c^T \cdot x$ with $c \in \mathbb{R}^d$ and the set $X \subseteq \mathbb{R}^d$ is given as an intersection of a finite set of half-spaces. We use a backend for linear optimization using floating-point arithmetic (here: GLPK) to obtain an initial solution $s \in X$. As floating-point computations are in general faster than rational arithmetic, this enables us to initialize a second solver quickly. We can use s to create a new constraint $c^T \cdot x \geq s$ and add it to X which enforces an improvement of s by adding a new half-space to the input problem. This approach may speed up the succeeding call to a linear optimization backend with higher precision (here either SMT-RAT, SOPLEX or Z3) which operates on the reduced search space using rational arithmetic to find a solution $s^* \geq s$. As a fallback, the

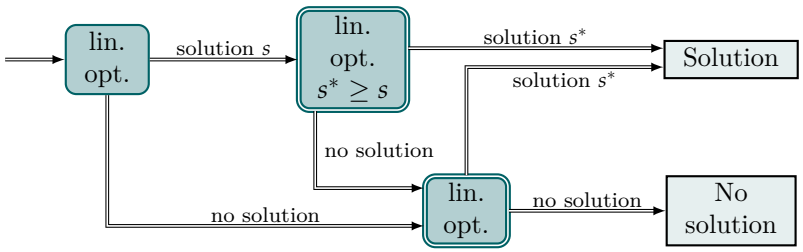


Figure 6.31: Portfolio approach for convex optimization using different backends. In a first attempt an optimization framework using double-precision is used (left). A second optimization framework (double lined) using rational arithmetic, which is potentially initialized with the first solution is used to either improve or validate the initial solution.

more precise backend is called on the initial problem if either the less precise backend is unable to find a solution or in case the attempt to improve s renders the problem infeasible. Both cases may occur in case the inexact optimization framework has returned a solution $s \notin X$, which is not contained in the search space of the original problem.

6.8 Conversion

One of the goals when providing a library of state set representations is to make different state set representations exchangeable on demand and even during computation with little effort. Using a common interface, which allows to use all provided state set representations in a similar fashion is one step towards this goal. To allow exchanging the used representation type even during reachability analysis computation, conversion methods are required which guarantee over-approximation. HyPro provides conversion methods for all implemented state set representations complying with the common interface. Once switching the state set representation during the reachability analysis is possible, we can implement more advanced approaches (see Chapters 7 and 8).

In this section, we are not going to present all conversion methods between all state set representations in detail, but focus on the more involved ones and will give an intuition on the remaining ones. For the interested reader, we refer to the Bachelor's thesis of Simon Froitzheim [Fro16], which contains detailed information about all presented approaches which are also implemented in HyPro.

Conversion to Polytopes

Convex polytopes as used in HyPro naturally require conversion methods between $\mathcal{H}$-representation and $\mathcal{V}$-representation, i.e., methods for *vertex enumeration* ($\mathcal{H}$-representation to $\mathcal{V}$-representation) or *facet enumeration* ($\mathcal{V}$-representation to $\mathcal{H}$-representation) to implement some of the required operations for reachability analysis (see Section 6.3). Furthermore, most other state set representations for a set S provide methods to compute either a set of vertices whose

convex hull contains S or a set of linear constraints whose conjunction contains S. For instance, as boxes are a subclass of convex polytopes in $\mathcal{H}$-representation the set of constraints defining a box B is given by the interval boundaries for each dimension. As another example, the vertices of a zonotope Z can be enumerated by a recursive algorithm (see Section 6.5) and used for conversion. Consequently, the conversion of state set representations to a convex polytope can be reduced to the construction of a polytope in either $\mathcal{V}$-representation or $\mathcal{H}$-representation.

For facet enumeration many algorithms were developed, but only a few extend beyond three dimensions. Two of the most well-known methods are *Graham's scan* [Gra72], and an optimized version of the *gift wrapping* algorithm, also known as *Jarvis-March* [Jar73]. These methods can be used for two-dimensional problem instances and compute a convex hull of n points in $\mathcal{O}(n \log n)$. Algorithms implementing general methods for arbitrary dimensional spaces can be split into two groups: constructive methods e.g., Quickhull [BDH96] and reversed search methods [AF92]. In HyPro, we provide several implementations: for plotting two-dimensional polytopes, an implementation of Graham's scan is included. For higher-dimensional polytopes, we use an implementation of the Quickhull algorithm to enumerate facets of a polytope in $\mathcal{V}$-representation. Currently, vertex enumeration is done by simple permutation and verification of the results.

To provide a more comprehensive collection of algorithms, an implementation of Avis and Fukuda's reversed search method [AF92] is planned. As this method can easily be adapted to solve the dual problem of vertex enumeration, we expect a significant improvement over the current implementation. Note that special care has to be taken when the state set in question does not span the whole state space, for instance, Quickhull requires a full-dimensional simplex as an initialization, i.e., the passed set of points needs to contain at least $d+1$ affinely independent points. We will present our approach to solving this problem in the next section.

Lesser-dimensional and Degenerated Polytopes. Special care has to be taken when working with lesser-dimensional polytopes, for instance, when trying to represent a line segment in a 2-dimensional space by a convex polytope in $\mathcal{H}$-representation (see Figure 6.32). In HyPro, we implement two approaches: an approach based on recursive projection and a fallback to a template-based approach (see Section 6.8). As the conversion via template-based objects is described below, we will provide an intuition on the method based on recursive projection here. The method takes a set of n points $P = \{p_0, \dots, p_{n-1}\}, p_i \in \mathbb{R}^d$ and the task is to compute the convex hull of P in $\mathcal{H}$-representation. The first step is to determine the dimension of the space the points in P span which is equivalent to computing the rank of the matrix $\mathcal{A} = (r_0, \dots, r_{n-2})^T$, where

$$r_i^T = (p_0 - p_{i+1})$$

are the row-vectors of $\mathcal{A}$. If $\mathsf{rank}(\mathcal{A}) = d$ holds, we can employ the previously mentioned convex hull algorithms, e.g., Quickhull. Otherwise, we know that the points in P all lie on a hyperplane $\bar{h} = \left\{ x \in \mathbb{R}^d \;\middle|\; n^T \cdot x = c \right\}$ of dimension

$\text{rank}(\mathcal{A})$ (see Figure 6.32a). Let $\text{rank}(\mathcal{A}) = k$. The idea is to recursively project j-dimensional vertices to a $(j-1)$ dimensional space until we achieve a k-dimensional vertex set. In our implementation, the choice of the dimension for projection is guided by the coordinate range for the respective dimension—iteratively we project out the dimension with the smallest coordinate range.

At this point, we can employ established convex hull algorithms to obtain a k-dimensional $\mathcal{H}$-representation of P (see Figure 6.32b). Recursively, each defining half-space is extended dimension-wise until a d-dimensional representation is fully constructed (see Figure 6.32c). A pseudo-code implementation of this algorithm can be found in Algorithm 7.

Algorithm 7: Polytope construction from a point set.

Input: Set of points P
Output: Convex polytope H containing P

```
Function constructPoly(P)
    A := constructMatrix(P)            // determine affine dimension
    if rank(A) = dim(P) then
        return facetEnumeration(P)     // classical facet enumeration
    else
        h̄ := constructCommonHyperplane(P)
        P' := P ↓_{dim(P)−1}
        H := constructPoly(P')
        addEmptyDimension(H)                    // undo projection
        return H ∩ h̄
```

Example 6.5: Recursive Projective Polytope Construction

Consider the set of points $P_3 = ((1,1,1)^T, (2,2,2)^T)$ describing a line segment in $\mathbb{R}^3$. Points in P_3 all lie on the hyperplane

$$\bar{h}_3 : x_0 - x_2 = 0.$$

In the first iteration, P is projected on the last two dimensions x_1, x_2 to obtain the point set $P_2 = ((1,1)^T, (2,2)^T)$ and the ambient space dimension is reduced to $\mathbb{R}^2$. One example for a hyperplane both points in P_2 lie on is

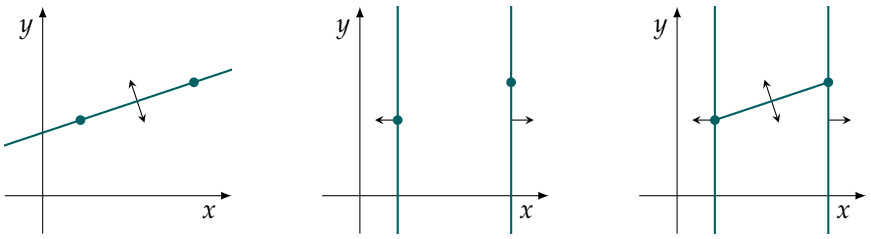
$$\bar{h}_2 : -x_1 + x_2 = 0.$$

Projecting on dimension x_2 results in $P_1 = ((1), (2))$ for which we can create the bounds

$$x_2 \geq 1 \wedge x_2 \leq 2.$$

Increasing the state space dimension by one accounting for x_1 and adding constraints for $\bar{h}_2$ we obtain

$$\begin{aligned} & x_2 \geq 1 \wedge x_2 \leq 2 \wedge -x_1 + x_2 \leq 0 \wedge x_1 - x_2 \leq 0 \\ \Leftrightarrow & x_2 \geq 1 \wedge x_2 \leq 2 \wedge x_1 = x_2. \end{aligned}$$



a) Determine shared hyperplane for all points in the current recursive call.

b) Obtain the constraints of the lesser-dimensional polytope after projection using facet-enumeration methods.

c) Lift polytope back to higher dimension and add constraints for the shared hyperplane.

Figure 6.32: Representation-conversion of a lesser-dimensional polytope: the set represented by two points in a two-dimensional state space is converted into $\mathcal{H}$-representation using a recursive conversion approach.

Again by increasing the state space dimension by one accounting for x_0 and adding constraints for $\bar{h}_3$ we obtain

$$\begin{aligned} & x_2 \geq 1 \wedge x_2 \leq 2 \wedge x_1 = x_2 \wedge -x_0 + x_2 \leq 0 \wedge x_0 - x_2 \leq 0 \\ \Leftrightarrow & x_2 \geq 1 \wedge x_2 \leq 2 \wedge x_1 = x_2 \wedge x_0 = x_2 \end{aligned}$$

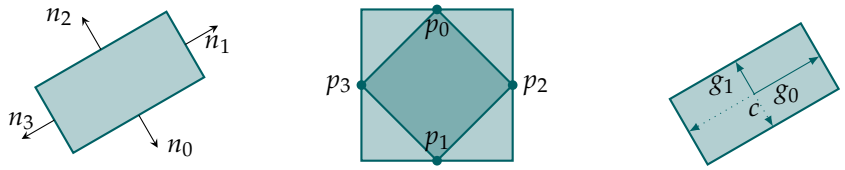
which describes the line segment between $(1,1,1)^T$ and $(2,2,2)^T$.

Conversion via Template-based Objects

We can use templates to provide conversion methods for representation types that provide the functionality to compute supporting hyperplanes for a given set of directions, e.g., for the conversion of support functions to any other representation. Template-based evaluation (see Section 6.4) for a support function S for instance allows to obtain a polyhedral over-approximation $S' \supseteq S$ in $\mathcal{H}$-representation with a fixed number of half-spaces defining S'. To achieve this, we use a *template* $T = (t_0, \ldots, t_{n-1})$ of vectors $t_i \in \mathbb{R}^d$ which are the normal vectors of the half-spaces bounding S'. The choice of the template influences the shape of the resulting polytope, and for set representation conversion allows to control the result. For instance, the conversion of a d-dimensional support function to a d-dimensional box can be achieved by choosing a template

$$\begin{aligned} T = \Big\{ t \in \{-1,0,1\}^d \;\Big|\; & \exists i \in \{0,\ldots,d-1\}.\ t_i \in \{-1,1\} \wedge \\ & \forall j \in \{0,\ldots,d-1\}.\ j \neq i \Rightarrow t_j = 0 \Big\} \end{aligned}$$

such that the resulting constraints are axis-aligned. Note that conversion via template-based objects can also be used to reduce the representational complexity of a set as this approach computes a polyhedral over-approximation of a state set with a potentially lower number of facets.



a) An oriented rectangular hull.

b) Ambiguity of the result of a PCA on $P = p_0, \ldots, p_3$.

c) The result of a PCA is also a zonotope.

Figure 6.33: Oriented rectangular hulls as the result of a PCA may be used to over-approximate any convex set represented by its vertices by a zonotope.

Conversion via Principal Component Analysis

Principal component analysis (PCA) [Pea01] as a method from statistics is originally used to make statements about the distribution of a finite set of data points $P = \{p_0, \ldots, p_{n-1}\}, p_i \subseteq \mathbb{R}^d$. The idea is to describe the shape of the cloud of data points by a linear combination of its main components. The result of a PCA of P is a set $C = \{c_0, \ldots, c_{n-1}\}, c_i \subseteq \mathbb{R}^d$ of d vectors, the *principal components* which are orthogonal to each other. The principal components play a role that is similar to the generators of a zonotope defining an oriented rectangular hull that contains all sample points.

Definition 6.5: Oriented rectangular hull

An oriented rectangular hull H in a d-dimensional Euclidean space can be described as a set $\{h_0, \ldots, h_{2d-1}\}$ of $2d$ half-spaces $h_i = (n_i, c_i)$ with the following properties:

- $\forall i \in \{0, \ldots, d-1\}.\ n_{i+d} = -n_i$—for each half-space there exists another half-space whose normal vector points into the opposite direction, which means that the bounding hyperplanes are parallel to each other.
- $\forall i, j \in \{0, \ldots, d-1\}.\ i \neq j \Rightarrow n_i \perp n_j$—all normal vectors that are not in the above relation are pairwise orthogonal to each other.

An example can be found in Figure 6.33a where the normal vectors n_0 and n_2 as well as n_1 and n_3 are pairwise inverse and orthogonal to the other normal vectors as described in Definition 6.5. Note that a box is a special case of an oriented rectangular hull in which all normal vectors n_i are axis-aligned. The result of a PCA in general is not unique, for instance the point set $P = p_0, \ldots, p_3$ as shown in Figure 6.33b may result in two different results. Intuitively, the more sphere-shaped the point cloud is, the more ambiguous the output result of the PCA may be.

The usage of a PCA to convert state sets provides a simple way to convert any convex set P represented by the vertices whose convex hull defines P (i.e., a polytope in $\mathcal{V}$-representation) into a zonotope in an over-approximating fashion [Fro16]. Each oriented rectangular hull naturally is a zonotope as

well (see Figure 6.33c), as the normal vectors returned by PCA can be used as generators and the center point is easily determined.

6.9 Utility

Organized around our collection of state set representations presented in this chapter, HyPro provides additional utility functions and data structures such as points, half-spaces, containers for linear constraints and many more (see also Figure 6.4).

Algorithms used throughout our implementation, for instance, to compute the convex hull of a set of points or Fourier-Motzkin variable elimination are modularized and are designed to be used elsewhere as well.

To test our library for performance, we have implemented a state of the art flowpipe-construction-based reachability analysis method similar to the one presented in [Le 09] for linear hybrid systems which is provided in HyPro and whose implementation we discuss below, as it is the basis for further modules that implement thesis contributions.

As a first step towards a tool prototype, HyPro also features a parser for Flow* model files and compositional interchange format (CIF) input format [BFH+14; Kie18]. To be able to visualize results, a simplified plotting class is provided, which creates gnuplot [WKm19] files for different output formats to allow to create pdf, png and LaTeX files containing plots of the over-approximation of the set of reachable states. To this end, the library of state set representations has been extended by utility and algorithms which allow users to set up flowpipe-construction-based reachability analysis methods on their own. The modular design allows us to replace or improve single parts easily without diverting the focus on other aspects during the development of new methods.

Concepts that will be introduced in the following chapters such as flowpipe-construction-based reachability analysis extended with a CEGAR-like refinement (see Chapter 7) or state space decomposition (see Chapter 8) along with the required data structures and algorithms have been integrated into HyPro as well.

Search Tree

General reachability analysis (see e.g., Algorithm 1) for hybrid automata analyzes all (possibly bounded) initial execution paths of a given hybrid automaton $\mathcal{H}$ (see Definition 3.3). As discrete jumps may be taken non-deterministically, reachability analysis induces a search tree where nodes represent the passage of time and the parent-child relation between nodes in the tree represent discrete jumps. The shape of the search tree depends on the utilized parameter configuration for the reachability analysis. Analysis parameters such as aggregation or the time step size may affect the branching structure (see Section 3.4) or render subtrees not reachable, depending on the introduced over-approximation errors. In the following, we give a formal definition of a search tree as we use in our implementation.

Definition 6.6: Search Tree

For a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ with dimension $d = |Var|$, a state set Σ, a time horizon $T \in \mathbb{R}_{\geq 0}$, and a parameter configuration Par, a *search tree* is a tuple

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

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 $State : Nodes \to (Loc \times 2^{\mathbb{R}^d})$ 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 n is *completed* if $Completed(n) = 1$;
- for each node $n \in Nodes$, either $Completed(n) = 0$ and n has no successors (i.e., $\forall (n', n'') \in Succ.\ n' \neq n$), or $Completed(n) = 1$ and for each $(n, n') \in Succ$ with $Trace((n, n')) = (I, e)$ we have that

$$FP(State(n), Par) = \left\{ State(n') \;\middle|\; (n, n') \in Succ \right\} .$$

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

We made the design decision to store only the initial sets $State(n)$ in the tree nodes for which flowpipes and jump successors are computed. This decision was made to reduce memory consumption during the analysis, however storing also the flowpipes might have some advantages. Though in our algorithms we never need the flowpipe again, we could recompute it in case it is required.

When we represent search trees graphically, we annotate the edges with the *Succ* information about the time interval and the jump taken to generate the given child node. When clear from the context, we omit the information about the jump and label with the timing information only.

During time successor computation starting from node $n \in Nodes$ we compute a sequence of flowpipe segments Ω_i for the given time horizon. For each edge $e \in Edge_{\mathcal{H}}$, we collect all segments, apply intersection with the guard, apply the reset, intersection with the target invariant and optionally aggregating or clustering during these computations, as described in Section 3.4. For each resulting symbolic state (ℓ, N) covering time successors from the interval I we

create a new node n' with $State(n) = (\ell, \mathrm{N})$ and $Trace((n, n')) = (I, e)$ in the search tree[3].

Note that this enables us to map each path

$$n_0 \overset{(I_0,e_0)}{\to} n_1 \overset{(I_1,e_1)}{\to} \cdots \overset{(I_{h-2},e_{h-2})}{\to} n_{h-1}$$

in the search tree with $State(n_i) = (\ell_i, \mathrm{N}_i)$ to a symbolic path

$$\Pi(n_{h-1}) = (\ell_0, \mathrm{N}_0) \overset{I_0}{\to} (\ell_0, \mathrm{N}_0') \overset{e_0}{\to} (\ell_1, \mathrm{N}_1) \cdots (\ell_{h-1}, \mathrm{N}_{h-1})$$

of $\mathcal{H}$ with some valuation sets N_i' that over-approximate the states reachable from N_i within time I_i. The valuation sets N_i' contain those flowpipe segments whose duration intersects with the time interval I_i and from which the given jump e_i was taken to obtain N_{i+1}. This information is not stored in the search tree but it can be reconstructed on demand (using the fixed parameter configuration for the analysis).

Though the definition allows different nodes with the same symbolic path, in our approach each node will have its unique symbolic path leading to it which we will call the *symbolic path of the corresponding search tree path*. We achieve this by including each jump successor of each flowpipe segment in precisely one child node.

Tasks and Workers

During the analysis, time and discrete transitions need to be explored alternatingly. For these computations, we introduce the concept of a task as a data type to store the information needed to compute one flowpipe and all of its potential jump successors.

Definition 6.7: Task

Assume a search tree generated by flowpipe-construction-based reachability analysis with a fixed parameter configuration *Par* for a given hybrid automaton. A task t

$$t = (n)$$

is specified by a node n in the search tree (see Definition 6.6).

A *worker* process is responsible for the execution of tasks. The processing of a task $t = (n)$ involves the computation of time successor states starting from the initial set $State(n)$ using configuration *Par*; as a consequence of enabled discrete transitions, new child nodes of n are added to the search tree, which require further processing after which $Completed(n) = 1$ is set.

Successors of discrete jumps, for which tree nodes have been created, can be stored for processing by creating new tasks. As several discrete transitions

[3]Instead of a single interval, it is straightforward to extend to sets of intervals. For the sake of simplicity we use only a single interval in this thesis.

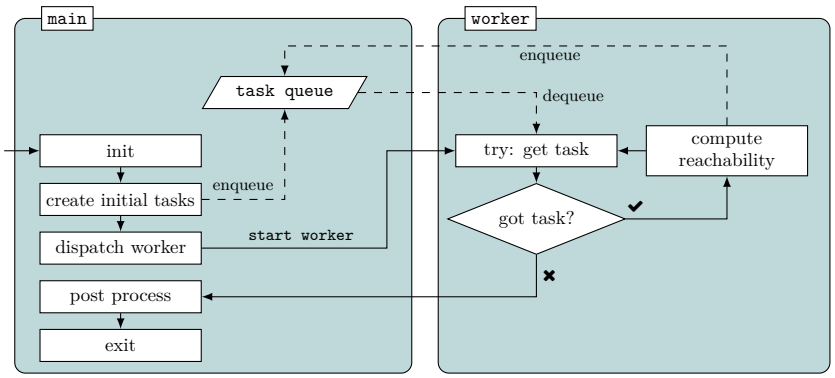


Figure 6.34: Modularized worker-based reachability analysis in HyPro. Dashed lines represent data flow.

may be enabled due to non-determinism in the model, we store new tasks in a queue equivalent to the set R_{new} in Algorithm 1.

As stated in [FR09], we could also consider specialized workers (e.g., applying different successor computation approaches dedicated to certain types of dynamics). In a later part of this work (see Chapter 8), we make use of specialized workers in the context of the decomposition of the state space as described in [SNÁ17].

Once a worker completes processing a task, it adds the resulting jump successor state sets as new nodes to the search tree and creates tasks for them to trigger their processing.

The whole process is illustrated in Figure 6.34. The main module (left) is responsible for pre- and post-processing which involves parsing input files, initializing data structures, and after the analysis creating plots. The actual flowpipe-construction-based reachability analysis is performed by the worker module (right) in which iteratively tasks from the task queue are executed, and resulting successor tasks are pushed to the task queue. Note that at this point, the worker is implemented as a module—we will extend this concept to worker-threads in Section 7.5.

In our implementation, the task queue is implemented as a first-in-first-out queue—consequently, the search tree is explored breadth-first. Note that changing the queue order to implement a first-in-last-out queue, i.e., a stack allows switching to depth-first exploration of the search tree. By heuristically adjusting the queue order, further mixed approaches may be implemented.

6.10 Experimental Evaluation

In this section, we present a small experimental evaluation of HyPro on a selection of benchmarks similar to the one presented in [SÁB+17]. We have chosen the same systems for safety verification and run our benchmarks on the same machine with $4 \times 4\,\text{GHz}$ Intel Core i7 CPUs and a memory limit of $8\,\text{GiB}$ and a timeout (to) of $15\,\text{min}$ but with the current version of HyPro which reflects the state presented in this work. The time horizons are the same, but

the jump depth for the rod reactor system has been increased from two to five jumps to have a consistent set of benchmarks throughout this work.

Benchmark Selection. Similar to our original evaluation[4] we have selected three benchmarks, the bouncing ball (`Ball`), the rod reactor (`Rods`), and the switching system (`Sw5`). For the bouncing ball as well as for the rod reactor system we use a time step size $\delta = 0.01$, while for the switching system a time step size $\delta = 0.001$ is required. The respective time horizons and maximal jump depths can be found in Section 4.1.

Comparison. In this evaluation we compare the performance of different state set representations implemented in HyPro, namely boxes (`box`), convex polytopes in $\mathcal{H}$-representation (`hpol`) and $\mathcal{V}$-representation (`vpol`), our wrapper to the PPL-library (`ppl`), support functions (`sf`), and zonotopes (`zono`). Note that we use boxes and support functions that implement all presented improvements. Furthermore, we compare the effects of different number representations on the running times; here, we compare a native `double` implementation to the widely used `mpq_class`-type provided by the GMP-library, which implements rational numbers. Additionally, we make use of different configurations for linear optimization in our evaluation: while GLPK is always used as a pre-solver in HyPro, we may optionally invoke further backends for linear optimization to increase the precision of the obtained results (see Section 6.7). Here, we compare an implementation that uses only GLPK with two further implementations that use SMT-RAT or Z3 as additional backends.

Furthermore, to put the obtained results in relation, we ran the tool SpaceEx on the same selection of benchmarks with similar parameters. For SpaceEx we vary two settings: the used algorithm (LGG [Le 09] and STC [FKL13]), and the number of template directions used for support functions during the computation. We use an octagonal template, which is also used in the support function implementation of HyPro and a box-shaped template—the results for box-shaped templates can be found in the box-rows although technically the underlying representation is not a box. Note that the underlying implementation of the LGG-algorithm is similar to the method implemented in HyPro which is why the results from HyPro using support functions with native `double` numbers and no secondary LP-solver backend are especially well-suited for comparison.

Results. The running times from our evaluation can be found in Table 6.3. In general, the effects of the choice of state set representation are reflected in the running times and behave as expected. While for instance boxes produce overall good results, with the chosen settings, it was not possible to prove safety for the five-dimensional switching system. In contrast to this, the running times for instance for convex polytopes are significantly longer, which is expected as operations on convex polytopes in general require more computational effort but produce more precise results. In comparison with the PPL implementation, our

[4] as in [SÁB+17]

implementation of convex polytopes shows room for improvements regarding running times and robustness; running times where the algorithm finished but produced an erroneous result are put in braces. Results obtained using support functions show running times that are comparable to the ones obtained with SpaceEx (LGG) when using native `double` numbers and no second optimization framework.

The effect of different number representations is observable among all representations, especially for those whose operations involve many arithmetic operations such as convex polytopes or zonotopes. The effect for boxes and support functions is smaller, as operations on boxes in general require less computational effort and for support functions sampling the support has the most significant effect on running times, which can be optimized in case the underlying set is box-shaped. Furthermore, we could observe that rounding errors play a significant role when using convex polytopes as a state set representation—both representations show increased erroneous behavior when using `double` numbers. In contrast to this, the polytopes implemented in the PPL internally use rational number representations and consequently do not exhibit these problems—all numbers need to be converted which adds slight overhead when using `double` numbers.

The effect of different additional solving backends can be observed for state set representations which strongly rely on linear optimization, e.g., support functions, where additional solving backends influence the running time. Note that this effect is diminished by using boxes as an underlying state set representation for support functions where most calls to linear optimization can be avoided. In contrast to this, zonotopes or boxes for instance do not rely on linear optimization and consequently are not affected by different linear optimization backends.

Table 6.3: Running times in seconds, time-out (to) was 15 min obtained with HyPro using different number type implementations and different linear optimization configurations. Results marked with "†" indicate that safety could not be proven with this configuration, entries in brackets reflect erroneous results.

		`mpq_class`			`double`			SpaceEx	
		GLPK	GLPK + SMT-RAT	GLPK + Z3	GLPK	GLPK + SMT-RAT	GLPK + Z3	LGG	STC
box	Ball	0.16	0.14	0.13	0.11	0.13	0.11	0.10	**0.07**
	Rods	2.01	1.78	1.16	**0.30**	0.40	0.42	2.57	0.32
	Sw5	†	†	†	†	†	†	†	**0.05**
hpol	Ball	**0.42**	(0.11)	10.62	to	(0.11)	(0.23)		
	Rods	**94.01**	to	692.81	(18.41)	(0.14)	to		
	Sw5	to	to	to	**15.13**	(0.25)	to		
vpol	Ball	0.22	0.32	0.30	**0.14**	0.16	0.30		
	Rods	**134.43**	680.45	149.38	to	(0.88)	to		
	Sw5	to	to	to	to	to	to		
ppl	Ball	0.35	0.25	0.23	**0.21**	0.24	**0.21**		
	Rods	9.28	10.56	9.18	6.44	7.19	**6.42**		
	Sw5	to	to	to	to	to	to		
sf	Ball	0.33	2.15	0.33	0.13	0.14	0.13	0.14	**0.08**
	Rods	690.78	to	674.60	4.30	4.63	5.97	4.40	**0.50**
	Sw5	33.19	34.71	33.07	**0.13**	0.15	**0.13**	0.36	0.32
zono	Ball	to	to	to	0.12	**0.11**	**0.11**		
	Rods	to	to	to	0.61	**0.33**	0.52		
	Sw5	137.92	138.31	150.61	0.14	0.14	**0.13**		

Part III

Improving Reachability Analysis

7

Counter Example Guided Abstraction Refinement in Hybrid Systems Reachability Analysis

Based on our implementation of HyPro as presented in the previous chapters, in this chapter, we consider an extension towards flowpipe-construction-based reachability analysis based on counterexample-guided abstraction refinement (CEGAR).

Hybrid systems reachability analysis based on flowpipe construction requires expert knowledge to be applied to real-world systems. The selection of suitable analysis parameter values, e.g., the time step size or the state set representation is crucial for the success of safety verification (see Section 3.4), but determining optimal parameter values is a difficult task: each implemented method comes with its own set of parameters, which all need to be chosen with respect to the system model. Parameter configurations, which result in a coarse approximation might not allow to verify the safety of a model, i.e., produce a *spurious counterexample*: we can observe a non-empty intersection of the over-approximation of the set of reachable states and the set of bad states, even though the actual system may be safe. On the other hand, parameter configurations resulting in small approximation errors might unnecessarily increase running times. Intuitively, an optimal parameter configuration thus is as coarse as possible and as precise as necessary.

In this chapter, we discuss a CEGAR-based symbolic path refinement approach, which aims at deriving the appropriate level of detail for every single node in a given search tree. While some symbolic paths in the search tree may represent paths that lead to potentially unsafe states, others can be declared safe. We propose an approach to restricting iterative refinement to the necessary case of potentially unsafe paths to refute spurious counterexamples. Refinement of a symbolic path can be achieved by recomputing the approximation of reachability following the same jumps within the same time intervals again but using a different parameter configuration, ideally chosen in such a way that the resulting over-approximation error is smaller than the previous one. To further reduce computational effort, we refine symbolic paths not individually but reuse computations for shared prefixes as much as possible. Furthermore, as we will show in Section 7.3, our approach is extensible by increased information

reuse between refinement levels, which can additionally reduce running times. Additionally, we show a natural way of parallelizing our approach in Section 7.5.

Related Work. Apart from other applications, CEGAR-approaches [CGJ+00] have been used in hybrid systems reachability analysis in the past in several ways. Different flavors of CEGAR have been used with varying refinement and abstraction methods, see for instance [ADI03; CFH+03; BDF+13; BDF+16; Nel16; BFG+17]. We will shortly review the basic ideas of available approaches before we introduce our method.

The authors in [ADI03] present a predicate-refinement method, in which predicates from a provided set as well as new predicates synthesized during the analysis are used to identify and rule out spurious counterexamples. The approach is based on refining an abstraction of the state space by adding predicates, which separate regions in the abstract state space. In a closely related approach [CFH+03], the authors refine their abstraction of a hybrid system iteratively based on counterexamples. Refinement is achieved by splitting symbolic states which are reachable in the abstraction based on their reachability using a tighter approximation. Similarly, enabled discrete transitions in the model abstraction can be refuted in case they are not enabled when using a more precise approximation.

In [BDF+13; BDF+16], the authors propose to use coarse approximations of the set of reachable states to guide the analysis towards potential counterexamples. In their work, an abstraction of the set of reachable states is obtained by specific analysis parameters, in this case the time step size and the template directions for the polyhedral approximation of the used support functions (see Section 6.4). The abstractions are stored in a pattern data base (PDB), which allows storing both the discrete and the continuous abstraction at the same time. The length of potential erroneous paths in the PDB is used as a cost function to guide the search using a more fine abstraction. Guidance in this context means that paths that may potentially lead to a bad state will be explored first during the analysis. A similar approach for guidance has been presented in [BFG+12], in which the Euclidean distance to the set of bad states without consideration of the discrete structure is used as a cost function.

In the Ph.D. thesis of Johanna Nellen [Nel16], she presents CEGAR-approaches for hybrid systems as an extension to the tool SpaceEx. In her approach, an abstract model of a given system which does not specify dynamics is refined iteratively by adding dynamics on the basis of observed potential counterexamples during the analysis.

In [BFG+17], the directions defining a template for template polyhedra are synthesized during analysis to iteratively rule out counterexamples by refining the state set representation used. To our knowledge, this is the first approach in which the state set representation is refined specifically, guided by a counterexample.

Compared to previous works, the presented method is more general and allows for an arbitrary number of refinements. As in the approach in [BDF+16], our method guides the search using partial information obtained during runtime by prioritizing path refinement over regular state space exploration.

In the following, we present our counterexample-guided approach towards the iterative refinement of the abstraction of the set of reachable states, which we refer to as *partial path refinement*. This chapter is based on our work [SÁ18a; SÁ18b] and contains excerpts of those works which are used in consent with the co-authors.

7.1 Partial Path Refinement

As presented in Sections 3.3 and 3.4, the result of a flowpipe-construction-based reachability analysis of a hybrid automaton $\mathcal{H}$ highly depends on the selected parameter configuration used to analyze $\mathcal{H}$. In case the analysis of $\mathcal{H}$ is successful using Par, the system can be declared safe, as the computed over-approximation $Reach'_{\mathcal{H}}$ of the set of reachable state $Reach_{\mathcal{H}}$ is not conflicting with the safety specification P_{bad}.

In case the analysis of $\mathcal{H}$ fails, i.e., a non-empty intersection between the over-approximation $Reach'_{\mathcal{H}}$ of the set of reachable states $Reach_{\mathcal{H}}$ and the set of bad states P_{bad} is detected in a node n^* of the search tree, the result is inconclusive. We refer to the symbolic path $\Pi(n^*)$, which may contain an execution of $\mathcal{H}$ conflicting with P_{bad} as a potential counterexample. In this case, either $\Pi(n^*)$ contains an unsafe path, or the computed counterexample is *spurious*, i.e., the computed over-approximation of the set of reachable states $Reach'_{\mathcal{H}}$ is too coarse, which causes the violation of the safety specification. While the first case, in general, cannot be proved unless under-approximative computations are used, a valid approach to overcome the second case is to use a different parameter configuration Par', which reduces the over-approximation error, usually at the cost of increased running time. In current approaches, this leaves the user to choose a suitable Par' manually and restart the whole analysis from scratch.

In this section, we will present our approach to overcome this problem. The general observation we can make is that up to the point in which an intersection with the set of bad states occurred, potentially much information has already been obtained. This information is discarded when restarting the analysis. Our approach aims at storing information during the analysis and uses it to improve consecutive analysis runs on the same system $\mathcal{H}$ using different parameter configurations Par_i. Similar to the approach presented in [BDF+16], from a first analysis that detected a potential counterexample, we already gain information on safe as well as potentially unsafe paths. When performing a fresh restart with another parameter configuration Par' in the hope of more precise results, typically this information is lost.

Our approach tries to avoid this information loss by building on previous results and *refining* only the approximation of $Reach_{\mathcal{H}}$ on potentially unsafe paths. With our approach of partial path refinement, we want to achieve that during the analysis we (i) only use cost-intensive parameter configurations where needed and (ii) we reuse information gained during previous analysis runs to speed up the search.

We want to make use of several analysis parameter configurations to detect and refute spurious counterexamples during analysis. In our setup, a collection of parameter configurations is provided a priori as a search strategy by the user.

Definition 7.1: Search Strategy

A *search strategy* is a non-empty, finite, ordered sequence of reachability analysis parameter configurations Par_i

$$(Par_0, Par_1, \ldots, Par_{m-1}) \ .$$

We use the term *refinement level* (starting with zero) to address the position of the parameter configuration in a given strategy.

In this section we consider the parameters state set representation rep, a time step size δ and aggregation settings agg. Thus, a parameter configuration in this section is a tuple $Par = (rep, \delta, agg)$. In general, depending on the underlying reachability analysis method, parameter configurations may contain other or further parameters.

In general, it is advisable to choose a search strategy, in which the parameter configurations at increasing levels result in an increase in precision of the approximation of the state set. This can be a difficult decision, as for some of the analysis parameters no total order exists with respect to induced precision. Note that, though advantageous, this property is not a necessary condition for the correctness of the method.

Consider a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ and a search strategy $(Par_0, \ldots, Par_{m-1})$ of m parameter configurations. Starting from a set of initial states[1] (ℓ_0, N_0), the search tree is created by a worker using flowpipe-construction-based reachability analysis with parameter configuration Par_0. If at a certain point the verification fails, for instance during the analysis in the current tree node n^* using configuration Par_i, a non-empty intersection between the over-approximation of the set of reachable states $Reach'_{\mathcal{H}}$ and the bad states P_{bad} is encountered, and a *partial path refinement* is triggered. As mentioned before, partial path refinement aims at improving the precision of the computed $Reach'_{\mathcal{H}}$ by using different parameter configurations collected in a search strategy.

To achieve this, the symbolic path

$$\Pi(n^*) = (\ell_0, \mathrm{N}_0) \xrightarrow{\tau_0} (\ell_0, \mathrm{N}'_0) \xrightarrow{e_0} (\ell_1, \mathrm{N}_1) \xrightarrow{\tau_1} \cdots \xrightarrow{\tau_{k-1}} (\ell_{k-1}, \mathrm{N}'_{k-1})$$

where $State(n^*) = (\ell_{k-1}, \mathrm{N}_{k-1})$ holds is refined using the next parameter configuration Par_{i+1} (if available). In case $i = m-1$ holds, i.e., there is no further parameter configuration in the current search strategy, the potential counterexample given by $\Pi(n^*)$ cannot be refuted and the algorithm terminates with an inconclusive result. Starting from the symbolic state (ℓ_0, N_0), we use flowpipe-construction-based reachability analysis with the next parameter configuration Par_{i+1} to obtain a new set of symbolic states for each element of $\Pi(n^*)$. To realize refinement of $\Pi(n^*)$, for the refinement of time transitions

[1] In general several initial state sets are possible, however for the presentation we consider single initial sets.

$\tau_r = [t_{l,r}, t_{u,r}]$ we use $t_{u,r}$ as a local time horizon. Accordingly, the guard of the discrete jump e_r is checked for enabledness within the flowpipe segments for the time interval τ_r only. The refinement on this path naturally ends, if a discrete transition e_r which was enabled on $\Pi(n^*)$ using Par_i is not enabled any more when using a different configuration Par_j, $j > i$.

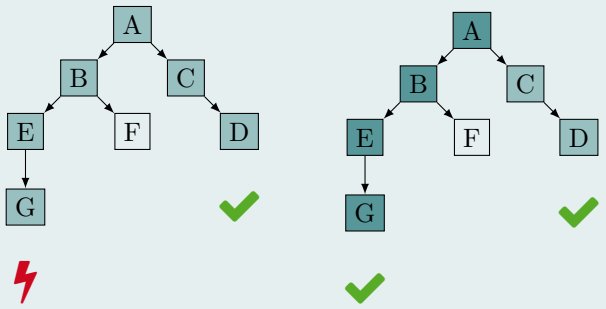
This way, solely the approximation of the set of reachable states on a selected path $\Pi(n^*)$ in the search tree is refined to be able to declare a potential counterexample spurious. The whole process is illustrated in Example 7.1.

Example 7.1: Partial Path Refinement

Consider the following exemplary search tree starting from the root node A. For the analysis a strategy

$$(Par_0, Par_1, Par_2)$$

is used. Starting with configuration Par_0 (very light petrol) the analysis is successful for the path from A to D but fails to successfully declare safety for the path from A to G, for which a refinement with configuration Par_1 is triggered (below left).



After successful validation (right) using a different parameter configuration Par_1 (light petrol), the analysis may continue.

The change between different parameter configurations $Par_i = (rep_i, \delta_i, agg_i)$ and $Par_j = (rep_j, \delta_j, agg_j)$ might require the conversion of the representation for the initial state set (see Section 6.8) in case $rep_i \neq rep_j$. Furthermore, when switching to a different time step size $\delta_j \neq \delta_i$ the matrices $\Phi_i = e^{\delta_i \mathcal{A}}$ need to be recomputed to account for δ_j.

As denoted before, the path $\Pi(n^*)$ is used to identify the path in the search tree, which is to be refined using the specified parameter configuration. In some cases, the refined computations are also represented by a single path in the search tree, as before. However, in some other cases, the refined computations might themselves be tree-shaped, for instance, if aggregation is turned off in the refinement, which requires more involved mechanisms to maintain the bookkeeping for all levels in the same search tree.

Assume a path $n_0, \ldots, n_{n-1}$ in the search tree from the root node n_0 to $n_{n-1} = n^*$ with $Trace(n_k, n_{k+1}) = (\tau_k, e_k)$, which is a counterexample path

$\Pi(n^*)$ to be refined with configuration Par_j. We start with the root node n_0, which is to be processed first at level j with the initial state (ℓ_0, N_0). We use the parameter configuration Par_j to compute time successors for the time interval τ_0 and take a discrete transition $e_0 = (\ell_0, g_0, r_0, \ell_1)$ during the time interval τ_0, if it is indeed enabled during that period from the refined flowpipe.

While the instructions for computing time successor states are easy to follow, we need to pay special attention to discrete jumps: depending on time step sizes and aggregation settings, this may result in several successor nodes in the search tree (see Figure 7.1) or in no successors at all, when e.g., new information has been gained during refinement.

That means we need to cope with a dynamic tree structure as a result of refinement. To provide an appropriate data structure, we have considered two options: (i) keep a separate search tree for each refinement level, or (ii) maintain a single tree with a dynamic structure. In this work, we follow the second option to ease the information exchange between different refinement levels. Consequently, search tree nodes become multi-dimensional, as information for each refinement level is stored in each node on the refinement path.

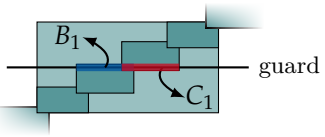
Definition 7.2: Refinement Search Tree

For a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ with dimension $d = |Var|$, state set Σ, a time horizon $T \in \mathbb{R}_{\geq 0}$, and a search strategy $(Par_0, \ldots, Par_{m-1})$, a *refinement search tree* is a tuple $(Nodes, Root, Succ, State, Trace, Completed)$ with the following components:

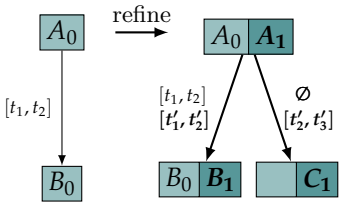
- a finite set $Nodes$ of *nodes* and a *root* node $Root \in Nodes$;
- a set $Succ \subseteq Nodes \times Nodes$ of *edges* such that $(Nodes, Root, Succ)$ is a tree;
- $State$ is a collection of m partial functions where for $i = 0, \ldots, m-1$ its i^{th} component $State_i : Nodes \to (Loc \times 2^{\mathbb{R}^d})$ assigns symbolic states of $\mathcal{H}$ as data to nodes;
- $Trace$ is a collection of m partial functions where for $i = 0, \ldots, m-1$ its i^{th} component $Trace_i : Succ \to (\mathbb{I} \times Edge)$ assigns an interval and a jump of $\mathcal{H}$ to edges of the search tree;
- $Completed$ is a collection of m partial functions where for $i = 0, \ldots, m-1$ its i^{th} component $Completed_i : Nodes \to \{0, 1\}$; we say that a node n is *completed at level* i if $Completed_i(n) = 1$;
- for each node $n \in Nodes$, either $Completed_i(n) = 0$ and $State_i$ is undefined on all children of n if any, or $Completed_i(n) = 1$ and for each $(n, n') \in Succ$ with $Trace_i((n, n')) = (I, e)$ we have that

$$FP(State_i(n), Par_i) = \left\{ State_i(n') \;\middle|\; (n, n') \in Succ \right\} .$$

A search tree is called *complete* for a jump depth $J \in \mathbb{N}_{\geq 0}$ if each node $n \in Nodes$ with depth less than J is completed on at least one level.



a) Coarse flowpipe segments (very light petrol) are refined (light petrol) such that two instead of one segment satisfy the guard predicate.



b) Refinement (light petrol) of a previous path (very light petrol) creates an additional node C_1 in the search tree.

Figure 7.1: Refinement with a smaller time step and no aggregation creates additional nodes in the search tree, stored time intervals assigned to the respective transition are updated.

In our algorithm, when a new node is created, all partial functions are undefined for this node to avoid meaningless assignments. Note that we do not remove nodes from the search tree during the analysis, even if they are declared unreachable.

Similarly to the approach without refinement we can define for each path

$$n_0 \overset{(I_0,e_0)}{\to} n_1 \overset{(I_1,e_1)}{\to} \cdots \overset{(I_{h-2},e_{h-2})}{\to} n_{h-1}$$

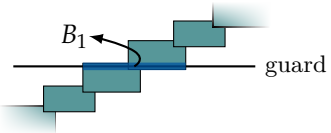
in the search tree and each refinement level j where $State_j(n_i) = (\ell_i, \mathrm{N}_i)$ a symbolic path

$$\Pi_j(n_{h-1}) = (\ell_0, \mathrm{N}_0) \overset{I_0}{\to} (\ell_0, \mathrm{N}'_0) \overset{e_0}{\to} (\ell_1, \mathrm{N}_1) \cdots (\ell_{h-1}, \mathrm{N}_{h-1})$$

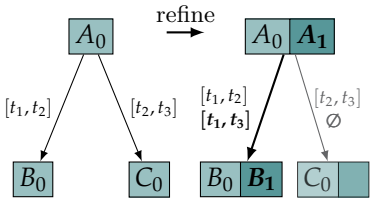
of $\mathcal{H}$. Our algorithm will ensure, that if $State_j(n_{h-1})$ is defined, then it is defined for all n_i with $i < h - 1$.

Technically we could use m-dimensional arrays to uniquely identify data for the different levels in the search tree. In our implementation, we use vectors and push a dedicated *dummy refinement* to indicate that the data is undefined on the given level. In the illustrations (see e.g., Figures 7.1b and 7.2b), the tree nodes are represented by sequences of boxes, similar to an array: the number of boxes is the number of applied refinements for this node, and undefined data at level i is indicated by the i-th box being empty. In contrast to this, in the illustrations we use uncolored node parts to indicate, that the node has not been processed on the respective refinement level.

As each refinement run starts at the root node, two different refinements for the paths to nodes n_m and n_n with configuration Par_k may share a common prefix, for which the computations only need to be done once. We illustrate this with the following example.



a) When switching aggregation on by a refinement, two sets satisfying the guard predicate are unified (aggregated).

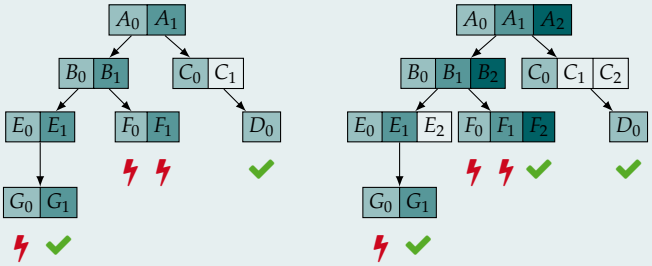


b) One instead of two jump successors in the search tree as a result of refinement.

Figure 7.2: The sets satisfying the guard predicate are unified (aggregated) during refinement resulting in only one instead of two discrete jumps successor nodes in the search tree.

Example 7.2: Refinement reuse

Consider the search tree from Example 7.1 along with the same refinement strategy as before. The analysis of node F fails using the first parameter configuration Par_0. The scheduled refinement for nodes A and its follow-up refinement for node B with parameter configuration Par_1 can be skipped as the information is already available.

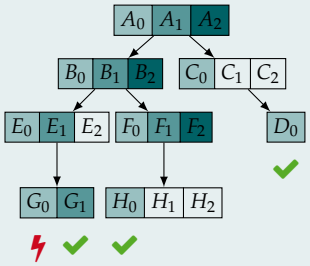


Though it would be possible to continue with the same refinement level after successful falsification of a potential counterexample, in our implementation the analysis continues with a lower refinement level (see node H in Example 7.2) to reduce the effort for future computations. That means if a counterexample can be refuted at refinement level i then the i-th state set is used to generate initial states for all children for all refinement levels $0 \leq k \leq i$. Note that these are generated, but only processing of level zero is scheduled. The other initial sets are only needed in case a further counterexample candidate is detected in the subtree below. The reason why we add them is to avoid the recomputation of the flowpipe.

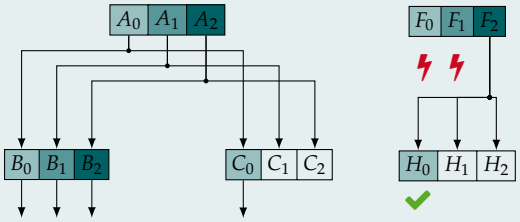
Example 7.3: Refuted counterexample

Consider the search tree from Example 7.2. After successful refutation of the potential counterexample in node F, a successor node H is added as a child node with all initial states up to the maximal level required to

refute the counterexample in F (see below). Only the analysis for level zero is scheduled (and in this case successfully applied).



In the previous examples we have used a simplified representation of the edges of the search tree for illustrative purposes. In the following, we show two excerpts of the above search tree with arrows indicating the actual relation between the different refinement levels inside the tree nodes.



The first excerpt (left) shows how the different refinement levels are related in the root node. After the successful refutation of a counterexample, the refinement level is reduced. All intermediate initial sets are created from the smallest refinement level which was able to refute the counterexample (right).

7.2 Data Structures and General Concepts

Next, we present how we adapt the HyPro analysis module as described in Section 6.9 to fit the need of the partial path refinement approach.

We extend the definition of tasks (see Definition 6.7) by two fields to obtain *refinement tasks*. For the sake of simplicity, in the following, we use the notion of a task synonymous to refinement task.

Definition 7.3: Refinement Task

Assume a search tree generated by flowpipe-construction-based reachability analysis with partial path refinement for a given hybrid automaton using a search strategy $(Par_0, \ldots, Par_{m-1})$. A *refinement task* t is a tuple

$$t = (n_i, Par_j, n^*),$$

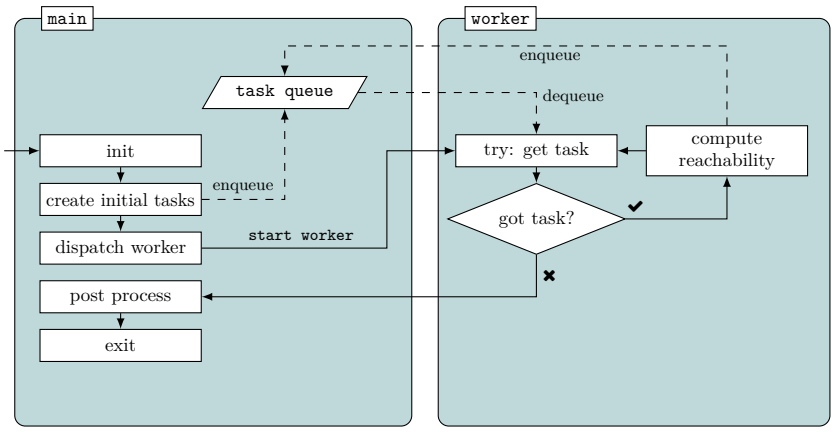


Figure 7.3: Modularized worker-based reachability analysis in HyPro. Dashed lines represent data flow.

which contains two nodes n_i and n^* of the search tree and an analysis parameter configuration Par_j with $j \in 0, \ldots, m-1$.

As before, tasks are held in a (possibly ordered) globally accessible task queue. A task $t = (n_i, Par_j, n^*)$ contains all information required to compute time successor states starting from $State_j(n_i)$ using the parameter configuration Par_j. Each task (n_i, Par_j, n^*) that is to be processed specify nodes such that the flowpipe $FP(State(n^*), Par_{j-1})$ of n^* has a non-empty intersection with P_{bad} and the depth of n^* is at least the depth of n_i. However, as explained before, due to different branching structures at different refinement levels, the path in the search tree from the root node to n_i is not necessarily a prefix of the path from the root to n^*. Nevertheless, we can definitely state that the sequence of jumps attached to the path to n_i is a prefix of the sequence of jumps on the path to n^*. An analogous rule cannot be stated for the sequence of timing information.

To avoid different representations of tasks at refinement level zero and higher, that means initial reachability computations and refining reachability computations, we represent all tasks as refinement tasks (n_i, Par_j, n^*) by setting $n^* = \bot$ for $j = 0$.

Refinement Worker. A refinement worker provides functionality to process a (refinement) task $t = (n_i, Par_j, n^*)$, i.e., to over-approximate the reachable time successor states using analysis parameters from the configuration Par_j. The symbolic state $State_j(n_i)$ is used as the set of initial states for the time successor computation. Discrete jump successors for each segment satisfying a guard condition are computed afterward, which results in the update respectively creation of child nodes in the search tree and the potential creation of tasks for their processing. It is essential to mention that similar to the original case without refinement, when computing successors for a node at a positive refinement level, we compute successors for *all* outgoing jumps, instead of

computing it only for the counterexample path. The reason for this is to avoid multiple computations of the same flowpipe, in case several counterexamples are detected at the same level with a common prefix. While this can be expected to save computational effort, this also brings some complications as explained in Example 7.4.

A pseudo-code version of the partial path refinement method, which is based on the flowpipe-construction-based reachability analysis method presented in Algorithm 2 can be found in Algorithm 8.

Algorithm 8: Partial path refinement algorithm inside a worker.

```
Data: Task list T
Output: Answer to R ∩ P_bad = ∅
while true do
    if T is empty then
        return safe
    take an element (n_i, Par_j, n*) from T
    R := computeFlowpipe(State_j(n_i), Par_j)
    if R contains unsafe states then
        if (j = m − 1) then
            return unknown
        addToTaskList(n_0, Par_{j+1}, n_i);
    else
        if jump depth not yet reached then
            computeJumpSuccessorsAndUpdateTaskList(n_i, Par_j, n*, R)
```

Refinement Task Priority. The fact that a potential counterexample cannot be refuted should be detected as early as possible to be able to stop further computation. Therefore, during the analysis, we want to prioritize refinement tasks with respect to basic reachability analysis tasks. This can either be achieved by increasing the task priority, i.e., its position in a task queue, or by introducing a second queue for priority tasks, which is accessed prior to the basic task queue. In the following, we assume a single queue.

Remember that in the previous non-refining setting, the task queue was implemented as a first-in-first-out queue, which corresponds to a breadth-first search. In the refinement setting, prioritizing refinement tasks is not sufficient to ensure breadth-first search at all refinement levels, but we need to add as second priority the depths of the node which needs to be refined. A third priority criterion first-in-first-out defines the order of tasks of the same type and equal depth. This priority definition has a similar effect as described in the approach described in [BDF+16] where the authors use a fast, box-based detection of possible counterexample traces to compute a guiding metric to decide which branch to process first.

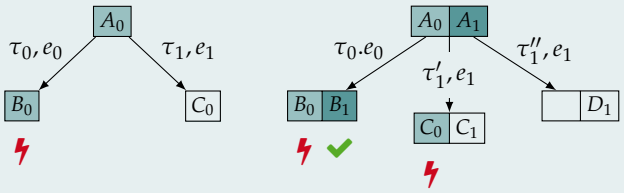
Alternative Entry Points for Refinement. In the presented approach refinement of a node n^* starts at the root node and the set of reachable states on the path in the search tree to n^* is refined according to the next parameter configuration in the defined strategy. At the first refinement, the root node has not been yet processed at the desired level. However, for later refinements, sharing a prefix with an already refined path in the search tree, the targeted node in the refinement task might already be processed at the desired refinement level. Such nodes, which are already on the desired refinement level could theoretically be skipped for refinement.

Therefore, it is tempting to assume that the first node on the path from n^* to the root node, which has the desired refinement level can be used as an entry point for refinement to avoid starting refinement from the root node. However, this is rarely possible due to the changing search tree structure, as we will illustrate in the following.

Without aggregation, refinement may change the branching structure of the search tree for the new refinement level, because the number of successor nodes in the search tree depends on e.g., the time step size used. As a consequence, following a single refinement path as described before is not sufficient. We illustrate this in Example 7.4.

Example 7.4: Refinement Level Relation

Consider the excerpt of a search tree as depicted below (left). Starting with the initial state set A_0 at refinement level zero, we assume two successors B_0, C_0 are added as children in the search tree resulting from two different discrete jumps e_0, e_1. Assume that processing B_0 detects a potential counterexample. During refinement of A_0, for the discrete jump e_1 two successor nodes C_1, D_1 are added (right). After successful refutation of the first counterexample, during the processing of C_0, another potential counterexample is detected.



For the refinement of C_0 it is not sufficient to use only the already initialized C_1 but also D_1 has to be considered, as both, C_1 and D_1 result from a refinement of A_0.

There is another case we have to keep in mind that leads to additional branching in the search tree structure in the presence of refinements, even if the refinement does not change the aggregation setting. Assume that clustering is switched on, collecting successors into at most $n > 1$ clusters (into n clusters if there are at least n successors, otherwise each successor is added separately). If at level i there are less than n successors but due to e.g., smaller time step

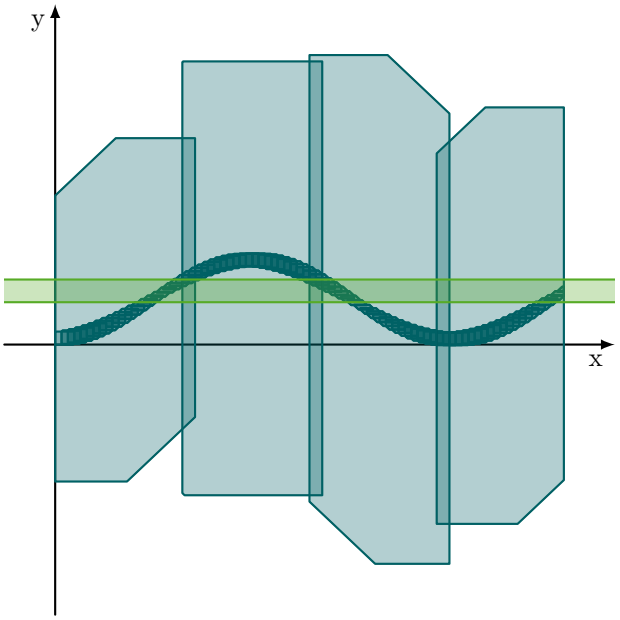


Figure 7.4: Depending on the dynamics, the search tree branching structure may change when using different time step sizes and aggregating all segments (petrol) which consecutively enable a transition (guard: green). We use light petrol to depict flowpipe segments in a coarse approximation and dark petrol for a more precise approximation.

size at the next level there are more than n successors computed, the number of children changes, thus the tree structure is modified.

A similar effect appears, if we consider another case of uniting successors, in which time consecutive successors are aggregated, such that the number of successors is determined by the number of changes in the enabledness of a discrete jump. Coarser approximations might not recognize all changes in the enabledness, for example, if the time step size is longer than the duration of continuous disabledness. A refinement step, for example, reducing the time step size below that duration, might increase the number of successors as shown in Figure 7.4.

To avoid tedious checks to detect such cases we chose the safe way always to start refinement from the root node and when already refined nodes are encountered, collect all relevant children as specified in the following definition.

Definition 7.4: Relevant Children

Assume a hybrid automaton, a search strategy $(Par_0, \ldots, Par_{m-1})$, and a refinement search tree $(Nodes, Root, Succ, State, Trace, Completed)$. Assume furthermore a refinement task $t = (n, Par_j, n^*)$ where $j > 0$, and the depths of n respectively n^* in the search tree are i resp. h and where

$$\Pi_{j-1}(n^*) = (\ell_0, \mathrm{N}_0) \xrightarrow{\tau_0} (\ell_0, \mathrm{N}'_0) \xrightarrow{e_0} (\ell_1, \mathrm{N}_1) \cdots (\ell_{h-1}, \mathrm{N}_{h-1})$$

such that $State_j(n)$ is defined and $Completed_j(n) = 1$. We define the set of relevant child nodes for t in the search tree as follows:

$$rel(t) = \{n' \in Nodes \mid (n, n') \in Succ \wedge Trace_{j-1}(n, n') = (\tau, e) \wedge \tau_i \cap \tau \neq \emptyset \wedge e_i = e\}$$

If the conditions of Definition 7.4 hold, meaning that the node of a task t is already completed at the requested level, then $rel(t)$ is the set of nodes for which refinement tasks are added to the queue in Algorithm 8 in the function `computeJumpSuccessorsAndUpdateTaskList`.

7.3 Information Reuse

During the reachability analysis of a given hybrid system $\mathcal{H}$, we store certain information such as time intervals of jumps in the search tree. Apart from plotting, computed flowpipe segments are only used temporarily to validate predicates such as guards, invariant conditions, or bad states. In the previous section, we have presented an idea on how we can use this information to identify which computations are needed to refine a counterexample, without recomputing the whole reachability.

In general, timing information from events in which predicates such as guards, invariant conditions, or bad states were satisfied can be further exploited during refinement, beyond what was explained in the previous section. The main idea is based on the fact that the timing information stored in the search tree is also *over-approximating*. For example, if a guard is enabled for a part of the time duration of a flowpipe segment, but not during the whole time interval, the successors are still labeled with the whole time interval. Thus, when we change the time step size, we get different timing information with different precision. We can exploit this fact in the refinement by taking the strongest statement we can derive from the information collected at the previous refinement levels.

To illustrate this, we are going to use the intersection with guard predicates as a running example. Consider a path $n_0, \ldots, n_{h-1} = n^*$ in the search tree such that

$$\Pi_j(n^*) = \cdots (\ell_i, \mathrm{N}_i) \xrightarrow{\tau_i} (\ell_i, \mathrm{N}_i') \xrightarrow{e_i} \cdots$$

is a counterexample with discrete jumps $e_i = (\ell_i, g_i, r_i, \ell_{i+1})$ in which e_i was taken during the time interval $\tau_i = [t_{l,i}, t_{u,i}]$. Upon refinement of n_i, we can use the information in the above symbolic path not only to identify the set of discrete successor states which match the path segment accounting for e_i but also to speed up the computation. We enter location ℓ at local time point 0 and we are interested in a refined computation for the successor via e_i with guard g_i only for the time interval $[t_{l,i}, t_{u,i}]$. Consequently, we can skip checking whether g_i is satisfied by the current flowpipe segment for time points $0 \leq t' < t_{l,i}$ and also for time points after $t_{u,i}$, which may significantly reduce the computational effort for refinement, depending on the computational effort required to verify a predicate (intersection and test for emptiness). A similar approach can be used for verifying the intersection with bad states or for finding an upper bound on

the number of time successors based on the time interval the current invariant was satisfied.

Incremental Refinement. From the previous results we can derive that we only need to validate predicates C during refinement, if the current time interval has a non-empty intersection with the stored time interval I which over-approximates the time interval I' in which C is informative for the analysis. Note that the relevance of a predicate C during analysis depends on the type of predicate we are considering (guards, invariants or bad states) as well as on the type of intersection result (full containment, no full containment but non-empty intersection, or empty intersection). Our implemented algorithms are capable of detecting the following types of intersection results for a flowpipe segment Ω and a predicate C:

- Full containment F: $Sat(\Omega \cap C) = \Omega$.
- Partial containment P: $Sat(\Omega \cap C) = \Omega' \neq \emptyset$, $\Omega' \subset \Omega$.
- No containment N: $Sat(\Omega \cap C) = \emptyset$.

As mentioned above we consider predicates C defining either guards, invariants or bad states. A predicate C is *informative* for a flowpipe segment Ω over-approximating a time interval I if the intersection type for the time interval I denotes *full* or *no* containment. If we know from previous computations that a condition is either fully satisfied or not satisfied at all for a given time interval we expect not to get new information when revalidating this information at a higher refinement level. In contrast to this, the information of partial containment can be refined—as the time intervals associated with each flowpipe segment represent an over-approximation of the actual time interval covered, we can refine stored information (see Figure 7.5). We illustrate this with an example:

Example 7.5: Event Timing Refinement

Assume a parameter configuration Par_0 for which a counterexample has been detected. The flowpipe segments which over-approximate the flow for the time interval $[0, T]$ in location ℓ were computed using Par_0. The flowpipe segments with corresponding time intervals $[t_0, t_1], [t_1, t_2], [t_2, t_3], [t_3, t_4]$ (see Figure 7.5) contain states which satisfy a predicate C. In a refinement, a different configuration Par_1 with reduced time step size is applied. For the refined flowpipe, only those segments whose time interval intersects the time intervals $[t_0, t_2]$ and $[t_3, t_4]$ need to be checked against C. The segments before t_0 and after t_4 might satisfy C but then only due to over-approximation. For the time interval $[t_2, t_3]$ we know that all reachable states in the exact segment satisfy C; though it can happen that the refinement detects partial unsatisfaction due to over-approximation, for a refinement typically leading to more precise results we expect this not to happen frequently and save the effort for these checks.

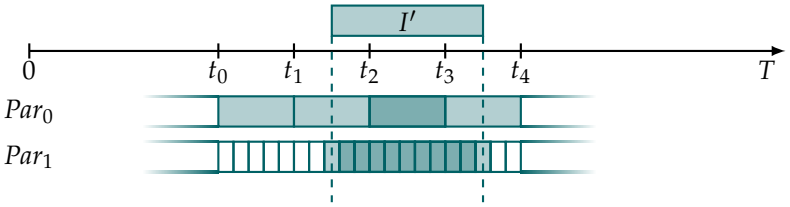


Figure 7.5: Incremental refinement of the interval I' in which some predicate is satisfied. The intervals where some partial containment is observed (light petrol) and the ones where full containment is observed (petrol) are color coded. White intervals correspond to flowpipe segments which do not contain states satisfying the predicate.

To fully exploit the observed time intervals in which a predicate is informative for flowpipe segments at different refinement levels, we merge information from several refinement levels for each predicate of an arbitrary type as follows.

In the following explanation, we use a data structure called *timing tree* which is similar to a (non-refinement) search tree but the data stored by the function *State* stores a set of real-valued intervals instead of symbolic states.

Initially, before the analysis starts, for each predicate in the model and for both of the cases F, N, we create a separate timing tree with a single (root) node containing the empty set of intervals. For example, in the F-tree for an invariant, we store information on paths and time intervals for which we surely know that the invariant is satisfied. In the following we use as a running example the F-tree of an invariant; the other cases are analogous.

Assume that the analysis starts for a given initial state set and we compute the first flowpipe. At the basic (non-refinement) level, we compute for each flowpipe segment its intersection type with each relevant predicate (invariant, guards, bad states). We store this information in the respective timing trees by adding the time intervals to the root node set.

For example, if the invariant fully contains a flowpipe segment for the time interval τ (which we call the τ-flowpipe segment at level zero), then we add τ to the interval set of the F-tree of the invariant. When a counterexample is detected, and the root node of the search tree gets refined, then we can reuse this stored information because we know that all states reachable within the interval τ satisfy the invariant; therefore we refrain from computing the intersection of the τ-flowpipe segment at level one with the invariant. In general, for a τ-flowpipe segment the invariant satisfaction can be derived if the union of the intervals in the root of the F-tree of the invariant contains τ.

The presented approach requires more effort, when the computations happen after a sequence of discrete jumps. Assume that we compute the flowpipe for a node n of the search tree with symbolic path $\Pi(n)$. In that case, the knowledge that the invariant fully contains a flowpipe segment can be reused only for another node n' if the symbolic path of n' is fully contained in $\Pi(n)$. We can assure this, by storing not only the time intervals for which the invariant is satisfied but also a sequence containing the durations of the flowpipe segments

from which jumps were taken and the identities of the jumps themselves. This is stored in the timing tree introduced above.

We formalize the algorithm to add new information to these trees in Algorithm 9. Informally, for a search tree node for which we compute the flowpipe and detect containment in the invariant for a time interval τ, we extract the timing and jump information $\tau_0, e_0, \ldots, \tau_i, e_i$ from the symbolic path of the node. Starting from the root node of the F-timing tree for the invariant, we recursively check whether an outgoing edge of the root node is labeled with τ_0', e_0 such that $\tau_0' = \tau_0$. If yes we recursively descend to add the information in that subtree. To avoid redundant information, if τ_0 and τ_0' are not identical but have a non-empty intersection, we split cases to their intersection and the remaining parts. Otherwise, if no intersection of τ_0 and τ_0' is detected, we create a new child of the root node and label the edge leading to it with τ_0, e_0, and descend to its subtree to add the information. When we reach a leaf, we add the interval τ to its interval set.

We can now use these timing trees to extract timing information for a predicate, after a symbolic path described by $\tau_0, e_0, \ldots, \tau_i, e_i$. To do so, we can use Algorithm 10, which works as follows. We try to find a set of child nodes of the root node which together cover the whole time interval τ_0 for the discrete jump e_0, that means each of them assures information for a specific part of τ_0. Now we recursively collect information in those subtrees, but since we want definitive results for the whole τ_0, we need to extract information common to all of them. Concretely this means we need to intersect their individually assured time durations.

The algorithm to extract information from the timing tree can be executed concurrently by several threads, as long as no thread modifies the timing tree. The modification, however, must be mutually exclusive.

Correctness. Assume we compute the flowpipe in a node n of the search tree with symbolic path $\Pi(n) = (\ell_0, \mathrm{N}_0) \xrightarrow{\tau_0} (\ell_0, \mathrm{N}_0') \xrightarrow{e_0} (\ell_1, \mathrm{N}_1) \cdots (\ell_i, \mathrm{N}_i)$. We call `getIntervals(`$B, \tau_0, e_0, \ldots, \tau_i, e_i$`)` to obtain a set of intervals, where B is the F-tree of the respective invariant. Correctness means that for all concrete paths in the symbolic path $\Pi(n)$ and all time points t in any of those intervals, if we extend the path with the elapse of t time units, then the invariant will be satisfied. We do not show correctness formally. The proof idea is based on the fact, that we only add intervals to the timing tree, for which we know invariance and all other changes to the tree only split symbolic paths into their parts.

Further Remarks. We mention that there are some exceptional cases, for example, when the invariant is violated, such that not all flowpipe segments need to be computed at a given level. To remember that no jumps can be taken from this time point on, we need to add the given time interval to the no intersection class (N) for all predicates of all types even though the intersection was not explicitly checked.

If during refinement for a certain time interval we know that no jump enabledness needs to be checked, theoretically we could also avoid computation of the flowpipe segments. Instead, we could use a dedicated linear transformation

Algorithm 9: Adding of informative intervals.

Input: Timing tree B for a certain predicate, sequence $\tau_0, e_0, \ldots, \tau_i, e_i$
Output: Set of intervals.

Function `addIntervals`(B,$\tau_0, e_0, \ldots, \tau_i, e_i$, τ)

- $n :=$ `root`(B)
- **if** `isLeaf`(n) **then**
  - $State(n) := State(n) \cup \{\tau\}$
- **else if** *exists* $n' \in Nodes$ *with* $(n, n') \in Succ$ *and* $Trace(n, n') = (\tau_0, e_0)$ **then**
  - $B' :=$ `subtree`(n')
  - `addIntervals`(B', $\tau_1, e_1, \ldots, \tau_i, e_i$, τ)
- **else if** *exists* $n' \in Nodes$ *with* $(n, n') \in Succ$ *and* $Trace(n, n') = (\tau_0', e_0)$ *and* $\tau_0 \cap \tau_0' \neq \emptyset$ **then**
  - $l_1 := \min(\tau_{0,l}, \tau'_{0,l})$
  - $l_2 := \max(\tau_{0,l}, \tau'_{0,l})$
  - $u_1 := \min(\tau_{0,u}, \tau'_{0,u})$
  - $u_2 := \max(\tau_{0,u}, \tau'_{0,u})$
  - replace the single child n' of n by three copies n_A, n_B, n_C (including the subtree below), and label the edges $Trace(n, n_A) = ([l_1, l_2], e_0)$, $Trace(n, n_B) = ([l_2, u_1], e_0)$, and $Trace(n, n_C) = ([u_1, u_2], e_0)$.
  - **if** $[l_1, l_2] \subseteq \tau_0$ **then**
    - `addIntervals`(`subtree`(n_A), $\tau_1, e_1, \ldots, \tau_i, e_i$, τ)
  - **if** $[l_2, u_1] \subseteq \tau_0$ **then**
    - `addIntervals`(`subtree`(n_B), $\tau_1, e_1, \ldots, \tau_i, e_i$, τ)
  - **if** $[u_1, u_2] \subseteq \tau_0$ **then**
    - `addIntervals`(`subtree`(n_C), $\tau_1, e_1, \ldots, \tau_i, e_i$, τ)
- **else**
  - create new child n' of n with $Trace(n, n') = (\tau_0, e_0)$.
  - `addIntervals`(`subtree`(n'), $\tau_1, e_1, \ldots, \tau_i, e_i$, τ)

Algorithm 10: Extracting informative intervals.

Input: Timing tree B, sequence $\tau_0, e_0, \ldots, \tau_i, e_i$
Output: Set of intervals.

Function `getIntervals`(B,$\tau_0, e_0, \ldots, \tau_i, e_i$)

- $n :=$ `root`(B)
- **if** `isLeaf`(n) **then**
  - **return** $State(n)$
- **if** *exists* $S = \{n_0, \ldots, n_{k-1}\} \subseteq \{n' \in Nodes \mid (n, n') \in Succ\}$ *such that* $\tau_0 \subseteq \bigcup_{n' \in S, Trace(n,n') = (\tau, e)} \tau$ **then**
  - **for** $i = 0, \ldots, k-1$ **do**
    - $B_i :=$ `subtree`(n_i)
    - $I_i :=$`getIntervals`(B_i, $\tau_1, e_1, \ldots, \tau_i, e_i$)
  - **return** $\left\{\bigcap_{i=0,\ldots,k-1} \tau_i' \;\middle|\; \forall i = 0, \ldots, k-1.\ \tau_i' \in I_i\right\}$

to compute reachability directly for the start of the interval for the next time point of possible enabledness. However, this requires the computation of dedicated matrix exponentials, depending on the bridged time duration, which may not necessarily pay off in terms of running time. We did not implement this option in HyPro.

As a last remark regarding the implementation, we developed a dedicated data structure for the efficient insertion and lookup of knowledge regarding time-dependent predicate satisfaction, without explicitly building the union respectively intersection of intervals but maintaining information symbolically by lists of annotated intervals.

7.4 Further Ideas

This section presents further ideas for future work to improve the presented approach towards partial path refinement and its extensions.

Dynamic Strategies. In the current setup, the strategy with its available parameter configurations is fixed—not only the order of configurations but the configurations themselves have to be provided a priori. A small part of the motivation behind the development of this approach was to move closer towards a push-button approach for reachability analysis for hybrid systems. Being able to synthesize a strategy during computation automatically would help to pursue this goal. As we have several parameters we can tune, automatically creating a strategy by creating combinations of a set of fixed parameter values for each parameter can reduce the complexity of strategy creation but does not necessarily improve the approach. As we cannot give a total order on the parameters, we can still provide patterns for strategies, for instance, after having used boxes as a state set representation, using support functions has been proven to be successful in the past. Additionally, we can refute specific configurations, for instance, if the state space dimension is large, we can refute using convex polytopes as a representation since most operations on those do not scale very well for large state space dimension. On the other hand, we can also promote configurations; for instance, boxes have been useful for many existing systems to obtain a first idea of the system's behavior. Another example of model-driven strategy-creation are zonotopes as they show excellent performance for purely continuous systems, as the weakness of zonotopes in reachability analysis lies in computing intersections with polyhedral guards which does not occur in purely continuous systems.

Similarly, we can use experience for other parameters as well: a rule of thumb for choosing a suitable time step size δ suggests to use the reciprocal of the largest eigenvalue of the matrix describing the system's dynamics.

Based on the state set representation, we can also make statements about sequences of configurations. For instance, using boxes after having used support functions will most likely not increase the precision while using support functions after boxes might.

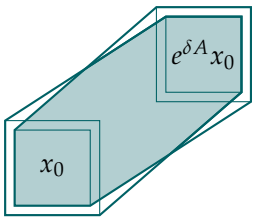


Figure 7.6: An adaption of first segment computation to include the results from backward analysis for the autonomous dynamics of a linear hybrid system.

Other works involving aggregation and de-aggregation techniques have been successfully applied [BD17b] and focus on dynamic de-aggregation as a refinement based on simulation traces of the system.

In-flow Refinement. The refinement method as presented is based on CEGAR-approaches, i.e., we require a counterexample to be able to refine the abstraction. Based on the distance of a flowpipe segment Ω to a polyhedral set C specifying a predicate, for instance a guard, another improvement could adapt the current analysis settings. This requires an efficient approximation of the Hausdorff-distance between C and Ω. As mentioned in the introduction to this chapter, an approach where the authors use box-approximations and evaluate the distance between the center of the box-shaped flowpipe segment and the predicate has been presented before [BFG+12].

Adaption of analysis parameters based on the distance of a flowpipe segment to a particular predicate allows to adjust the current parameter configurations while computing time successor states, i.e., refinement implemented like this will result in the partial refinement of time transitions opposed to the method presented before, which fully refines time transitions.

Spatial Refinement. The results of a flowpipe-construction-based reachability analysis method do not depend on the utilized parameter configurations only, but also on the analyzed system. As mentioned in the introduction, approaches were developed, which starting from the purely discrete components of the system refine its abstraction by gradually adding dynamics [Nel16]. Instead of refining the dynamics of a system, we can consider the initial variable valuation for refinement. A subdivision of the set of initial states might lead to more precise analysis results. Furthermore, in case a system still cannot be declared safe, it could be made safe if it is possible to exclude initial regions with inconclusive verification results.

Backward Analysis. A step away from the refinement approach presented here, which was based on modifying the parameter configurations for the analysis would be to add backward-analysis as a refinement step. Since both, forward-analysis, as well as backward-analysis over-approximate the set of reachable states, intersecting the results obtained by both methods may lead to more precise results than obtained by using each method individually. Note that in

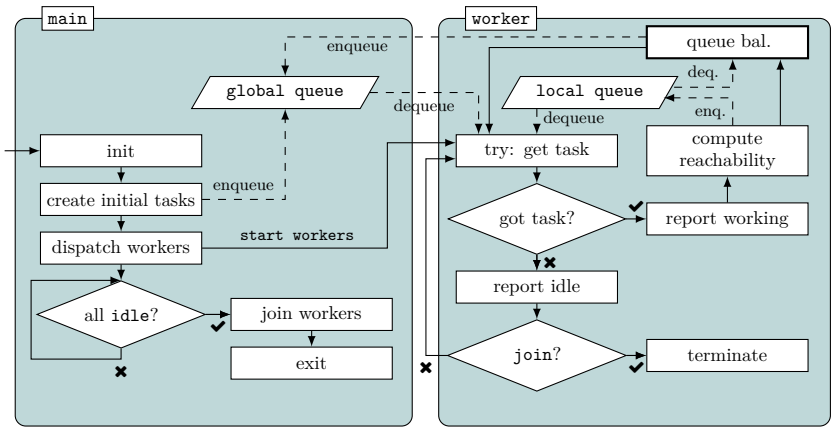


Figure 7.7: Parallelized flowpipe-construction-based reachability analysis in HyPro. Workers and threads exist in a one-to-one correspondence. Dashed lines represent data flow. The communication to the global queue happens in a synchronized way while the thread-local queues can be accessed without locking.

case the reset function of a discrete jump is not invertible, backward-analysis cannot give precise results on the discrete predecessor state of that respective jump. In this case, the whole state space (intersected with the location's invariant and the guard) is considered as a predecessor state which most likely will not aid in reducing over-approximation errors. Note that for autonomous linear hybrid systems, i.e., systems in which the flow is given as $\dot{x} = Ax$ it is possible to adapt the method for computing the first flowpipe segment to directly include results for backward analysis (see Figure 7.6).

7.5 Parallelization

This section is based on our work [SÁ18b] and may contain excerpts that are not explicitly marked as citations in the text.

The concept of partial path refinement integrates well into classical flowpipe-construction-based reachability analysis as presented before. During the analysis tasks are created which hold information about symbolic states for which successors need to be computed. Branching in the search tree which is caused by modes having several outgoing transitions or the usage of clustering or no aggregation when processing discrete jumps leads to the creation of several nodes in the search tree and consequently several tasks as the result of processing one task. The order in which those tasks are processed is usually not of importance; it only affects the order in which the search tree is explored (breadth-first vs. depth-first). As seen before, it may help to prioritize refinement to avoid spurious counterexamples and to guide the search towards potentially safety-critical paths. Nonetheless, to verify safety up to a certain bound, all

paths have to be analyzed regardless of the order[2]. In our framework, tasks are stored in a priority queue which implicitly results in a breadth-first exploration of the search tree.

In this section, we present our approach towards parallelization based on *multi-threading*. The tasks are natural units for parallel processing: multiple threads can implement workers (in a one-to-one correspondence between threads and workers) processing different tasks in parallel. In the following, we will illustrate the parallelization of our partial path refinement method (see also Figure 7.7) and provide details about certain aspects of the implementation afterward.

Local and Global Queues. As in the sequential case, each worker has a *local task queue* to implement refinement task prioritization. Access to these local queue is restricted to the owning worker, therefore it does not require any synchronization and is thus expected to be fast. During the analysis, each worker only communicates with its local queue when scheduling new tasks.

Additionally, for work balancing between the workers, we need a mechanism to distribute tasks between threads. For this purpose, we use a *global task queue*, which can be accessed by all workers in a synchronized fashion. Initially, the main thread schedules a task for the initial state set (or all initial state sets, if there are several) of the system and adds it to the global queue. The thread-local queues are empty.

After the creation of the workers, they compete for the initial task as follows. When idle, each worker tries first to obtain a task to process from its local task queue to keep the synchronization overhead as small as possible. Only if its local queue is empty, the worker tries to obtain a task from the global task queue, using synchronized access. If the global queue is also empty, the worker rechecks the global queue regularly, until it is filled or until also all other local queues are empty, which leads to a synchronized termination of the algorithm.

If a worker processes a task, new tasks will be added to the worker's local queue according to enabled discrete jumps. Consequently, without further task sharing, the subtree under the currently processed node in the search tree will be analyzed by this worker only.

To allow for work-balancing, workers can *move tasks* from their local queue to the global queue. We consider three heuristics for this balancing step, which are used after each completion of a task: (i) the worker pushes all but one tasks from its local queue to the global queue; (ii) only when the local queue size is larger than a certain threshold, tasks exceeding that threshold are moved from the local to the global queue; (iii) push a certain ratio of tasks from the local queue to the global queue. We expect that approaches (i) and (iii) will result in balanced work distribution at higher synchronization costs while approach (ii) should be better suited to limit these costs but lead to a less balanced work share. Note that queue balancing happens after the completion of each

[2]There are some exceptional cases where a scheduled task gets superfluous without being executed, for instance when switching on aggregation, but this is atypical and we do not detect these cases.

task by a worker, i.e., when potential successor tasks have been added to the thread-local queue.

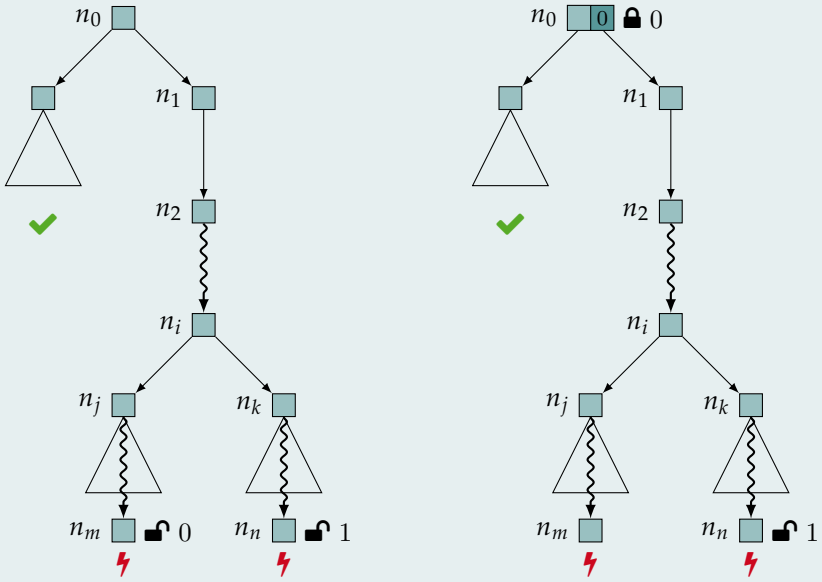
To compare our implementation, we also consider a different setting, where *only a global queue* is present. In this setting, work is inherently well-balanced but getting new tasks from the queue and adding new tasks to it requires synchronization for each action. While this setting might inflict more synchronization effort, it is also the case that accessing only the global queue allows for a more direct work-sharing even during the analysis—in all other configurations tasks are distributed to the global queue only after the analysis.

Node Synchronization. All workers w_i operate on a single shared search tree. Without path refinement, the workers need to synchronize on the access to the global queue, but not on search tree nodes: each search tree node n will be referred to by precisely one task t, which will be processed by exactly one worker w_i. New nodes n'_j will be added to the search tree as a result of the analysis of w_i and those new nodes will be added as child nodes to the current node n referenced by t. However, this is not the case for path refinement, as counterexample paths might share a prefix and workers may compete for access to a specific tree node n during a simultaneous refinement of the shared prefix. To ensure thread-safety during path refinement, each worker first attempts to get a lock on the tree node n referenced in the refinement task, and if successful processes the task referencing n. The lock is released free before starting to process any other node to avoid deadlocks. The *hold-and-wait* property, which requires to keep a lock for a shared ressouce while acquiring a secondary lock for a further shared ressource, is broken, which is sufficient to ensure deadlock-freedom.

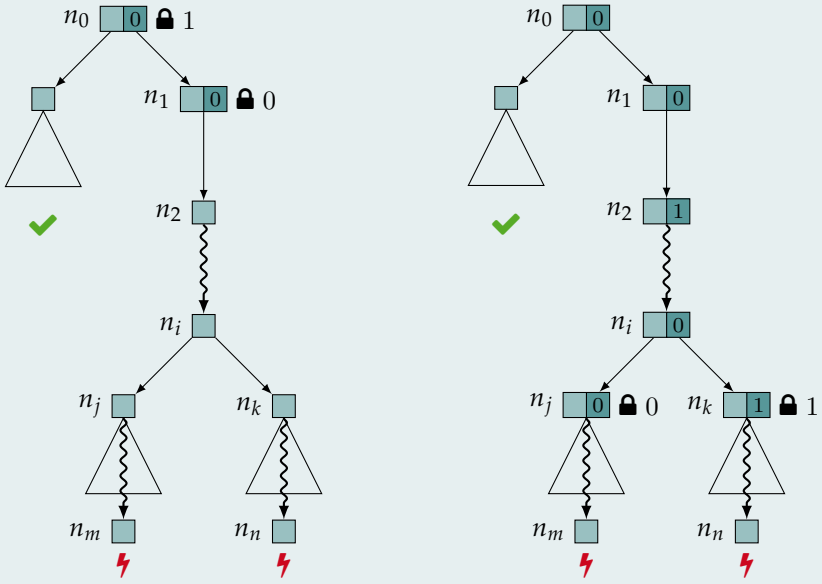
As a consequence, during refinement in a multi-threaded environment two workers which refine a path with a shared prefix may take turns in refining said path, depending on who acquires the lock for a node on the prefix first (see Example 7.6). Naturally, once a lock is acquired, the respective worker has to check again whether the current node's refinement level is below the targeted level or the node's level has been increased while waiting for the lock.

Example 7.6: Multi-threaded Refinement

Consider a setup in which two workers w_0, w_1 perform flowpipe-construction-based reachability analysis extended by partial path refinement in a parallelized environment. Assume the following excerpt of the search tree that represents the current state of the analysis (left). In these trees, we also want to show which worker processes which node; to avoid too much information, instead of the state sets, we indicate the worker in the nodes.



The two workers w_0, w_1 both found a potential counterexample in the subtree below n_j respectively n_k. The refinement paths share a common prefix which refers to the node sequence $n_0, n_1, n_2, \ldots, n_i$ in the search tree. To start refinement, both workers release the locks on the currently held nodes if they are refining[a]. The workers compete for the lock of the root node n_0, which is taken by w_0 in this example (right).



After releasing the lock on n_0, w_1 may access the node, but can skip refinement there as the node is already on the desired refinement level

(left). In the meanwhile w_0 has acquired the lock for a subsequent node n_1 on its refinement path.
Note that depending on the work-sharing mechanisms it either will happen that the first worker to obtain a lock on the shared prefix will fully refine it locally or both may take turns as in this example (right, node n_2 has been refined by worker w_1).

[a] Otherwise they work non-synchronized at level zero and hold no locks.

7.6 Experimental Results

We have implemented the presented approaches of *partial path refinement* together with an extension for the *parallelization* thereof in our tool prototype HyDRA [SÁ18c]. In this section, we present experimental results using HyDRA on a selected set of benchmarks. The results presented in this section were originally published in our works on partial path refinement [SÁ18a] and on the parallelization [SÁ18b].

Table 7.1: Parameter settings: Refinement strategies are specified by triplets containing (1) state set representation (`box`, support functions (`sf`), convex polytope in $\mathcal{H}$-representation (`hpol`)), (2) time step size, (3) aggregation (`agg`)/clustering in k clusters (`cl`.k).

	refinement strategy		
S_0:(box, 10^{-2}, agg)	(sf, 10^{-3}, agg)	(sf, 10^{-4}, agg)	
S_1:(box, 10^{-2}, agg)	(box, 10^{-3}, agg)	(sf, 10^{-2}, agg)	(sf, 10^{-3}, agg)
S_2:(box, 10^{-2}, agg)	(box, 10^{-3}, agg)	(sf, 10^{-2}, agg)	(sf, 10^{-4}, agg)
S_3:(box, 10^{-1}, agg)	(sf, 10^{-3}, agg)		
S_4:(box, 10^{-1}, agg)	(hpol, 10^{-3}, agg)		
S_5:(box, 10^{-1}, noAgg)	(box, 10^{-2}, cl.3)	(sf, 10^{-2}, cl.3)	

To show the general applicability of our approach, we have conducted several experiments on an implementation of the method presented in Section 7.1. We have used our implementation to verify the safety of several well-known benchmarks using different strategies (see Table 7.1). As all benchmarks are described in detail in Section 4.2, we will only provide a short reminder of the model and its properties. All experiments were carried on a machine with $4 \times 4\,\text{GHz}$ Intel Core i7 CPUs and a memory limit of 8 GiB. Results for the used strategies can be found in Table 7.2.

Benchmark Selection. The following benchmarks from the area of hybrid systems verification are selected.

The well-known bouncing ball benchmark (`Ball`) models the height and velocity of a falling ball bouncing off the ground. The added set of bad states constrains the height of the ball after the first bounce.

The 5-D switching system (`Sw5`) is an artificially created model with five locations and five variables that shows more complex dynamics and is well-suited to show the differences in resulting over-approximation errors between different state set representations. We added a set of bad states in the last location where the system's trajectories converge to a certain point.

The navigation benchmark [FI04] models the velocity and position of a point mass moving through cells on a two-dimensional plane (we used variations of instances `Nav09` and `Nav11`). Each cell[3] exhibits different dynamics that influence the acceleration of the mass. The goal is to show that a set of good states can potentially be reached while a set of bad states will always be avoided (see Figure 7.9). The initial position of the mass is chosen from a set large enough, such that this benchmark demonstrates non-determinism for the discrete transitions which results in a more complex search tree.

The platoon benchmark [BKG+09] models a vehicle platoon of three cars where two controlled cars follow the first one while keeping the distance e_i between each other within a certain threshold (label: `Pltn`). The vehicles communicate via radio which breaks down in specific patterns, depending on the instance of the benchmark. For the evaluation, we chose an instance as also used in [ABC+17; ABC+18] in which communication between the cars breaks down and comes up deterministically every five units of time. This benchmark was chosen, as it unifies a higher dimensional state space with more complex dynamics.

Strategies. During the development of our approach we tested several strategies with varying parameters (a) the state set representation, (b) the time step size, and (c) aggregation settings. In general, other parameters (e.g., initial set splitting) could also be considered, but our prototype currently does not yet support these. For this evaluation we selected six strategies $S_0, \ldots, S_5$ which mostly vary (a) and (b) (see Table 7.1). Changing aggregation settings has shown to be challenging for the tree update mechanism, but the exponential blow-up of the number of tree nodes did not render this method useful in practice. Furthermore, when aggregation is turned off in certain settings, the most significant precision gain can be observed for boxes while for all other tested state set representations the effect can be neglected. Note that our prototype implements the general approach as presented before, where partial path refinement starts from the root node.

Comparison. We compare our refinement algorithm (1) with a classical approach where no refinement is performed. To achieve this, we specify only a single strategy element for our algorithm. We give results for (2) the fasted successful setting (of the respective strategy), an experienced user would choose, and for (3) the setting with the highest precision level, a conservative user would select. The three entries per cell in Table 7.2a show the running times for our dynamical approach (light petrol), the fastest successful setting and the conservative approach. Additionally, we have analyzed the resulting search tree

[3] Cells and locations are in a one-to-one correspondence.

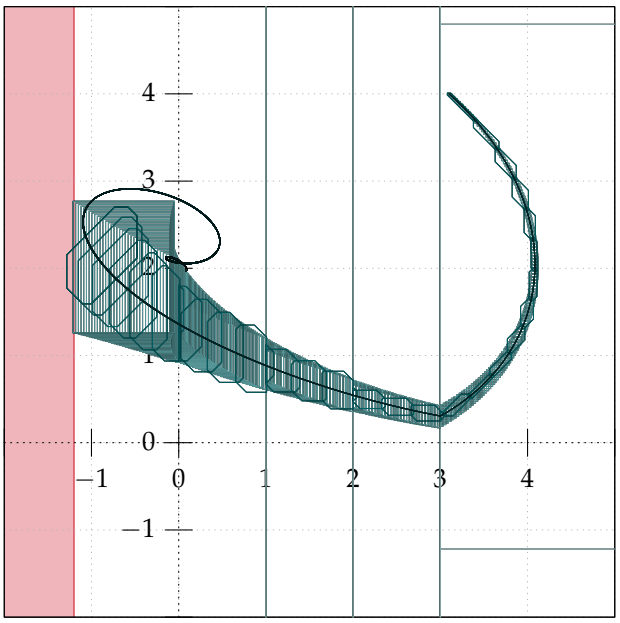


Figure 7.8: Approximation of the set of reachable states (strategy S_2, first two variables) for the linear switching system (`Sw5`). The first three configurations in S_2 result in reachable set over-approximations which intersect the set of bad states (red) in the last location. The finest configuration (dark petrol) allows to refute the counterexample and declare the system safe.

which is explored during the analysis. The results of this analysis are depicted in Table 7.2b, where we give the number of nodes in the search tree. For refinement, we provide the number of nodes for each refinement level separately.

Results. The results obtained during our experimental evaluation let us make several observations when considering the running times as well as the structure of the explored search tree.

The results in Table 7.2a were obtained using an implementation where timing information for discrete jumps was collected and reused during refinement. This feature allows us to partly compensate for the overhead, which is introduced as a result of maintaining the search tree. We expect that for benchmarks with a search tree with only one path, i.e., the models `Ball`, `Sw5`, and `Pltn`, the optimal setting outperforms our refinement approach. However, we can observe that the running times can compete with the running times of an optimal setting which is attributed to the timing information reuse.

When considering the search tree structure for the analyzed models, we observe that benchmarks which induce a high branching in the search tree, i.e., models which have a high level of non-determinism as the models of the navigation benchmark (`Nav`) profit from using partial path refinement. For instance, for `Nav09` using the search strategy S_1, we observe that large parts of the search tree already can be verified using the first analysis parameter configuration in S_1. Consequently, those safe nodes do not need to be considered during

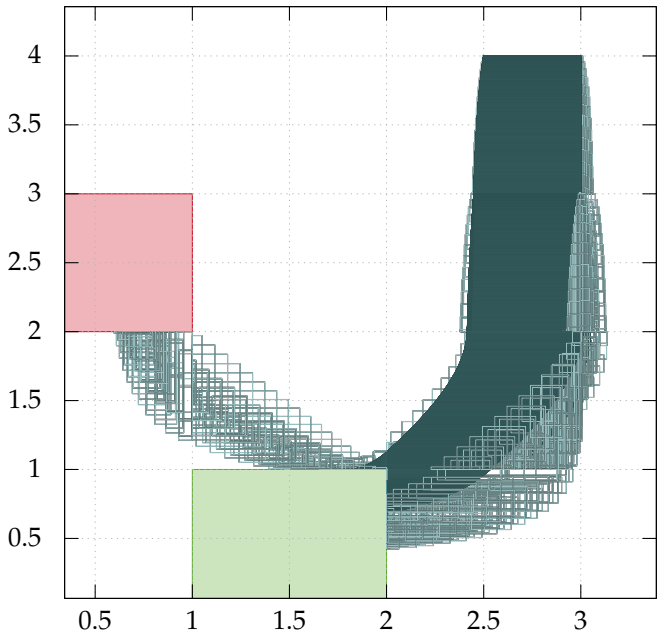


Figure 7.9: Plot of the approximation of the set of reachable states of the navigation benchmark (`Nav09`) using partial path refinement (strategy S_3, x, y position of the point mass). The first computations (very light petrol) intersect the set of bad states (red) while refinement (light petrol) constitutes that all system trajectories end in the safe cell (green).

refinement. Only six nodes require refinement using the second configuration of which again only four need a refinement using the third configuration. As the third configuration is the fastest one which allows verifying `Nav09`, using an approach without partial path refinement requires to explore the whole search tree using this analysis parameter configuration which takes considerably longer (5.76 s vs. 118 s).

As a side-effect of coarser approximation, we can observe that the search tree using refinement usually has more nodes than the ones from successful validation without partial path refinement using a classical approach (`Nav09`, S_1: 279 vs. 244 nodes). Together with the running times, this confirms our assumption that putting effort in selective, partial refinement of single branches pays off in terms of computational effort.

Additionally, the length of the counterexample significantly influences the outcome of the partial path refinement approach—in the bouncing ball benchmark, the set of bad states is reachable after one discrete transition and from then on never again, i.e., after refining the early spurious counterexample analysis can be proceeded using a coarse approximation. In contrast to that in the 5-D switching system (`Sw5`) the set of bad states is reachable only in the last location in a linear setup (see Figure 4.5b) which causes a refinement of the whole explored search tree and a recovery to a lower refinement level afterward is not possible (see Figure 7.8).

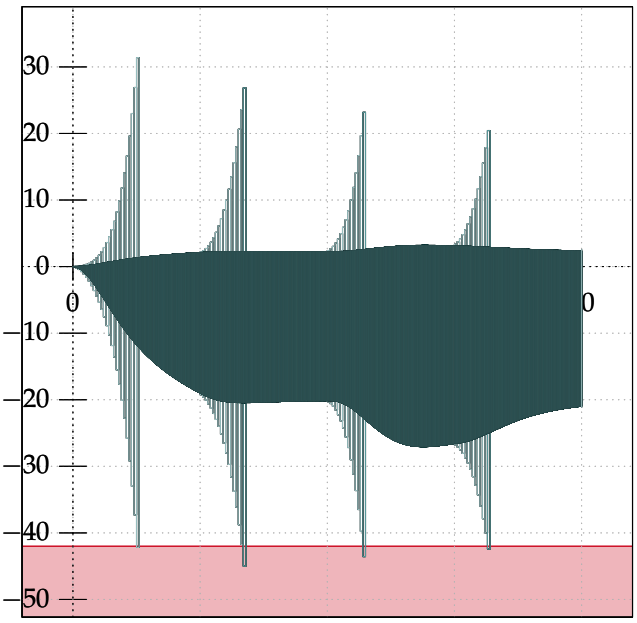


Figure 7.10: Plot of the approximation of the set of reachable states of the platoon benchmark using partial path refinement (S_3, distance first to second vehicle over time). The first refinement level (very light petrol) intersects the set of bad states (red). After refinement (light petrol) the counterexample is refuted and the refinement level is reset. Reducing the refinement level after successful invalidation of a counterexample causes repeated refinement, as the bad states are intersected repeatedly.

Stepping back to lower refinement levels can also be counter-productive in some cases. In the platoon benchmark, stepping back to a lower refinement level does not provide any advantages, as an intersection with the set of bad states occurs repeatedly (see Figure 7.10).

Parallelization

In this section, we present experimental results specific to our approach towards the parallelization of partial path refinement in flowpipe-construction-based reachability analysis. All experiments were run on a machine with $48 \times 2.1\,\text{GHz}$ AMD Opteron CPUs and a memory limit of 8 GiB; the single cores each are slower than the cores of the machine we used before, but the goal here was to investigate the effects of parallelization.

Benchmark Selection. For our experiments on parallelization, we have selected benchmarks with a potentially large search tree to be able to quantify the effects of different work-sharing strategies. The models of the navigation benchmark (`Nav09` and `Nav11`) are suitable, as the chosen initial set is large enough to enable multiple discrete transitions in the initial cell. Additionally, we select the model in which several processes compete for mutually exclusive

access to a shared resource (`Fish`). Similar to the other selected benchmarks, this model naturally exhibits non-determinism and thus induces branching in the search tree during analysis. In our evaluation, we did not include benchmarks with little non-deterministic choices. Additionally, our results show that the overhead caused by synchronization is small (see below) so we can expect little influence on running times for benchmarks with little branching.

Settings. For our experiments we consider eight different settings (see Table 7.3). Even though path refinement is not the main focus of our presented approach, all eight settings support path refinement as this requires node synchronization (see Section 7.5).

Each setting specifies a refinement strategy and a work queue balancing heuristics as before, i.e., as a sequence of triplets, each triplet specifying (1) the state set representation used, (2) the time step size for flowpipe construction and (3) settings for aggregation/clustering. The settings are chosen to result in large branching in the search tree: settings S_2, S_4–S_7 use clustering instead of full aggregation. Setting S_0, S_1, and S_3 are added for comparison.

To test different queue balancing methods, we did experiments with pushing all tasks above a fixed threshold from the local queue to the global queue, but this was far less stable in efficiency than pushing a certain percentage of the local queue contents, therefore here we include only experiments with the latter. In Table 7.3, the work queue balancing heuristics specifies which portion of the local queue is moved to the global queue after the completion of each task, i.e., 100 % means all but one. At least one task is always left in the local queue which can directly be processed by the respective worker and thus reduces synchronization effort for corner-cases, for instance, if only one discrete successor has been discovered.

Settings S_0–S_4 differ in their refinement heuristics, but they are all eager in pushing all but one task from the local to the global queue after the completion of each task. Contrary, settings S_4–S_7 share the same refinement heuristics but they differ in their work balancing method. Notably, setting S_7 completely avoids thread-local queues: every worker operates on the global queue directly. The difference is that while in all other settings the work distribution takes place at the end of the flowpipe computation in a batch, S_7 pushes single successor tasks to the global queue during its computations such that idle workers potentially could start computation earlier. As the experimental results show, this works surprisingly good, even though the increased synchronization effort is recognizable.

Results. The running times of our implementation using different settings and different numbers of threads for parallelization are given in Table 7.4. In general, we can observe a speed-up when increasing the number of worker threads—we could achieve a speedup of up to factor 33 (`Nav09`) which in this case results in $\sim 70.1\,\%$ efficiency ($\mathit{efficiency} = \frac{\mathit{speedup}}{\#\mathit{threads}}$) of the parallelization (`Nav11`: maximal factor 30, `Fish`: maximal factor 25). Furthermore, we notice that the running times of some instances (e.g., `Nav09`, S_0) stabilize at some point. This behavior can be caused by several issues: either the work distribution is not well-balanced,

or the synchronization overhead is too large in comparison to the time it takes to process a single task. Note that an unbalanced work distribution does not necessarily result from poor task distribution strategies but may also be caused by single tasks requiring significantly more running time to be processed in comparison to the majority of tasks in this setup.

For interpreting the results, it is important to mention that processing every single task is in general computationally expensive: the time required to compute a flowpipe is usually long in comparison to the time it takes to acquire a lock for synchronization and move tasks to the global queue. Consequently, the running times using one thread in our implementation resemble the running times of a purely sequential approach. Furthermore, with aggregation/clustering, the number of generated new tasks is often relatively small as both settings effectively limit the branching of the induced search tree. For example, for a deterministic system, a task might generate just a single successor task, in which case no work balancing would take place at all. This might lead to insufficient work balancing and explain why for some benchmarks and some settings involving more workers does not lead to any additional speedup.

To further investigate this, we ran the benchmarks with up to 48 threads on a machine with $48 \times 2.1\,\mathrm{GHz}$ AMD Opteron CPUs and a memory limit of $8\,\mathrm{GiB}$. For benchmark instances such as the navigation benchmark in combination with settings where aggregation was used (S_0, S_1, S_3) we can observe that the running times already converge for a low number of threads as there are not enough tasks created during analysis such that most threads idle. The running times for these settings do not significantly increase when using more threads which indicates that our implementation successfully minimizes the synchronization effort required. An exception is setting S_7 on benchmark `Nav09`, where the running times increase when using more than eight threads; as this setting only uses a global queue, the increased need for synchronization is reflected in the running times.

To investigate the actual work distribution, we collected the number of tasks processed by each worker thread. Table 7.5a shows the coefficient of variation (CoV) of these results to allow for statements about variance in the work distribution. The CoV as a relative measure for variance gives the influence of the variance of data on the mean in percent. Lower percentages hereby indicate a lower variance in data—in our case a better work distribution.

We can observe the influence of different queue balancing methods for benchmarks with settings that produce a lot of tasks (S_4–S_7) reflected in the CoV. With increasing number of threads the average number of processed tasks per worker decreases. When using settings that produce too few tasks, many worker threads idle, thus increasing the variance of processed tasks per worker (see e.g., `Nav09`, S_0). As expected, the setting using only a global queue shows the lowest CoV throughout the experiments as all available tasks are *immediately* shared.

Settings with local queues where 100 % of the created tasks are shared are expected to exhibit a similar CoV as when using a global queue only, however there are only two differences: firstly, when using a global queue only, tasks are shared immediately after their creation, whereas in the presence of local

queues sharing happens after task completion; secondly, 100 % sharing with local queues does not yield exactly 100 % as one single task is kept for further processing in a local queue to reduce synchronization. Strategies where a worker only shares part of its created tasks (S_5, S_6) show a larger variance, i.e., work is less equally distributed. With regard to the observed running times, we can deduce that sharing work comes at a price—even though setting S_7 has the lowest variance, the running times in comparison to settings S_4–S_6, which share the same analysis parameters are longer.

Note that a low CoV can also be achieved when many threads are taking turns in processing a small number of such tasks. Therefore, we also analyzed the average share of idle time for all threads (see Table 7.5b). We can conclude that the increased running time for setting S_7 indeed can be amounted to synchronization, as the idle time for the workers is among the lowest ones.

Observed Side-effects. During our experiments, we could observe the search tree size may vary, depending on the number of used workers and the benchmark instance. While the search tree size is deterministic in a single-threaded environment, as the execution order of tasks is deterministic, this does not hold for a multi-threaded approach. As described in Algorithm 8 we prioritize partial path refinement to refute a spurious counterexample. Hence, in a single-threaded environment, path refinement is completed before any other task is processed, which does not necessarily hold in a multi-threaded environment.

Table 7.2: Experimental results in s for different strategies. Timeout (to) was set to 10 min, memout (mo) to 4 GiB, "err" marks numerical errors. Three results per cell: (1) dynamic refinement (light petrol), (2) fastest successful setting only, (3) most precise setting.

a) Running times for different strategies in seconds.

Model	Strategy					
	S_0	S_1	S_2	S_3	S_4	S_5
`Ball`	**0.15**	**0.15**	**0.15**	0.46	1.58	0.21
	0.22	0.18	0.18	0.97	3.45	1.71
	11.93	0.97	9.90	0.97	3.45	9.47
`Nav09`	to	5.76	**5.09**	549.00	err	to
	to	118.00	118.00	to	to	mo
	to	to	to	to	to	to
`Nav11`	to	7.15	7.61	63.40	err	120.00
	to	**6.40**	**6.40**	395.00	to	130.00
	to	395.00	to	395.00	to	to
`Sw5`	2.27	0.49	2.30	0.39	15.31	0.45
	2.35	0.38	2.36	0.38	to	**0.37**
	2.35	0.38	2.36	0.38	to	**0.37**
`Pltn`	173.00	3.67	3.60	18.70	to	19.16
	to	**3.48**	**3.48**	18.90	to	18.80
	to	18.90	to	18.90	to	18.80

b) Number of nodes in the search tree for different strategies, refinement runs give the number of nodes on each level.

Model	Strategy					
	S_0	S_1	S_2	S_3	S_4	S_5
`Ball`	5|2|0	5|2|0|0	5|2|0|0	5|2	5|2	29|4|0
	5	5	5	5	5	121
	5	5	5	5	5	63
`Nav09`	to	279|6|4|0	317|17|6|0	549	err	to
	to	244	244	to	to	mo
	to	to	to	to	to	to
`Nav11`	to	45|8|7|0	75|16|7|0	73|11	err	75|4168|0
	to	24	24	24	to	4170
	to	24	to	24	to	to
`Sw5`	5|5|5	5|5|5|5	5|5|5|5	5|5	5|5	5|64|5
	5	5	5	5	to	5
	5	5	5	5	to	5
`Pltn`	5|4|4	5|4|4|0	5|4|4|5	5|4	to	5|4|4
	to	5	5	5	to	5
	to	5	to	5	to	5

Table 7.3: Parameter settings: Refinement strategies are lists of configurations, each configuration specified by a triplet (1) state set representation (box, support functions (sf)), (2) time step size, (3) aggregation (`agg`)/clustering in k clusters (`cl`.k). Additionally, the last column specifies the queue balancing rate.

	refinement strategy			work balancing
S_0:	$(\texttt{box}, 10^{-1}, \texttt{agg})$	$(\texttt{sf}, 10^{-2}, \texttt{agg})$	$(\texttt{sf}, 10^{-3}, \texttt{agg})$	100 %
S_1:	$(\texttt{sf}, 10^{-1}, \texttt{agg})$	$(\texttt{sf}, 10^{-2}, \texttt{agg})$		100 %
S_2:	$(\texttt{sf}, 10^{-1}, \texttt{agg})$	$(\texttt{sf}, 10^{-2}, \texttt{cl}.5)$		100 %
S_3:	$(\texttt{box}, 10^{-1}, \texttt{agg})$	$(\texttt{box}, 10^{-2}, \texttt{agg})$		100 %
S_4:	$(\texttt{box}, 10^{-1}, \texttt{agg})$	$(\texttt{box}, 10^{-2}, \texttt{cl}.3)$		100 %
S_5:	$(\texttt{box}, 10^{-1}, \texttt{agg})$	$(\texttt{box}, 10^{-2}, \texttt{cl}.3)$		10 %
S_6:	$(\texttt{box}, 10^{-1}, \texttt{agg})$	$(\texttt{box}, 10^{-2}, \texttt{cl}.3)$		50 %
S_7:	$(\texttt{box}, 10^{-1}, \texttt{agg})$	$(\texttt{box}, 10^{-2}, \texttt{cl}.3)$		global queue

Table 7.4: Running times in seconds for settings S_0–S_7, timeout (to) = 10 min, memout (mo) = 8 GiB, † = safety cannot be shown. Running times averaged over 10 runs. This table was originally published in [SÁ18b].

model	setting	#threads 1	2	4	8	16	32	48
Nav09	S_0	**21.99**	20.32	20.32	20.40	20.34	20.29	20.35
	S_1	24.87	**15.72**	**11.87**	**11.70**	**11.68**	11.70	11.72
	S_2	to	to	to	to	mo	mo	mo
	S_3	†	†	†	†	†	†	†
	S_4	263.80	134.90	69.34	36.87	21.68	16.70	15.63
	S_5	252.80	127.90	64.79	32.85	17.00	**10.41**	**7.51**
	S_6	263.50	132.80	68.70	36.20	20.90	15.53	13.95
	S_7	78.52	46.60	32.01	29.52	34.21	42.23	45.03
Nav11	S_0	70.49	45.72	45.39	45.42	45.41	45.47	45.44
	S_1	**18.47**	**9.81**	**6.15**	**5.03**	**4.68**	4.49	4.50
	S_2	to	290.70	146.40	75.53	39.92	22.45	16.50
	S_3	†	†	†	†	†	†	†
	S_4	95.73	47.05	24.04	12.21	6.42	**3.60**	3.13
	S_5	93.68	45.85	23.28	12.03	6.57	4.02	3.54
	S_6	92.11	47.16	24.02	12.20	6.62	3.74	**3.02**
	S_7	95.92	49.12	25.12	13.03	8.02	6.16	6.49
Fish	S_0	40.66	20.46	**10.43**	**5.49**	**2.96**	1.84	**1.61**
	S_1	to	to	to	393.90	201.50	107.20	79.02
	S_2	to	to	to	394.30	201.40	107.40	79.07
	S_3	40.57	20.44	10.47	5.54	2.97	**1.82**	1.78
	S_4	**40.56**	20.45	10.49	5.55	2.97	1.79	1.83
	S_5	40.63	20.47	10.87	6.76	4.56	3.92	3.96
	S_6	40.67	**20.42**	10.47	5.53	**2.96**	1.84	1.70
	S_7	42.73	21.79	11.26	6.06	3.68	3.45	3.93

Table 7.5: Coefficient of variation and idle time in percent for settings S_0-S_7, failures are marked (timeout=to, memout=mo). Unsuccessful settings are left out.

a) Coefficient of variation in percent for the work distribution among worker threads using settings S_0–S_7.

bench-mark	setting	#threads 2	4	8	16	32	48
Nav09	S_0	87.5	85.4	102.2	133.5	197.2	220.6
	S_1	32.7	43.6	39.0	44.8	92.5	118.4
	S_4	**0.1**	0.7	1.0	**1.3**	**1.7**	**2.2**
	S_5	0.4	1.1	1.8	2.7	4.0	4.9
	S_6	0.2	0.4	1.0	1.4	1.8	**2.2**
	S_7	0.4	**0.6**	**0.9**	**1.3**	2.2	2.7
Nav11	S_0	45.3	44.3	70.4	130.8	175.1	215.8
	S_1	24.0	15.3	29.2	45.1	90.5	121.0
	S_2	**0.4**	**0.9**	**1.9**	3.6	6.0	7.6
	S_4	0.9	1.7	10.3	11.0	15.7	13.5
	S_5	2.2	3.2	5.3	8.8	13.6	16.2
	S_6	1.4	2.1	2.6	17.3	11.9	12.6
	S_7	2.5	3.0	3.6	**3.3**	**3.9**	**5.5**
Fish	S_0	0.6	3.6	7.6	8.8	11.6	14.3
	S_1	to	to	6.8	8.0	10.0	13.2
	S_2	to	to	7.6	8.5	10.0	12.7
	S_3	0.8	3.3	7.4	9.3	11.8	13.9
	S_4	0.8	3.4	7.1	8.0	11.3	13.9
	S_5	**0.3**	2.6	14.9	24.8	67.6	99.8
	S_6	0.9	3.4	8.1	8.4	11.6	14.0
	S_7	0.5	**1.2**	**2.6**	**2.9**	**4.1**	**4.9**

b) Idle time in percent for settings S_0–S_7.

bench-mark	setting	#threads 2	4	8	16	32	48
Nav09	S_0	18.72	34.96	36.52	33.50	15.28	12.20
	S_1	10.63	28.78	36.52	28.29	12.76	10.21
	S_4	**0.04**	**0.18**	0.44	0.85	1.09	1.22
	S_5	0.16	0.46	1.07	2.30	4.33	6.16
	S_6	0.05	**0.18**	0.45	0.86	1.33	1.69
	S_7	0.11	0.23	**0.30**	**0.41**	**0.38**	**0.41**
Nav11	S_0	7.52	6.05	3.54	2.44	1.36	0.74
	S_1	4.13	20.95	40.91	47.22	33.84	25.93
	S_2	0.11	0.51	2.33	5.76	12.10	17.20
	S_4	0.11	0.44	1.21	3.45	5.79	6.17
	S_5	0.17	0.64	1.43	3.82	6.62	7.75
	S_6	0.07	0.46	0.81	3.35	6.35	7.74
	S_7	**0.01**	**0.30**	**0.72**	**1.63**	**2.67**	**2.78**
Fish	S_0	0.32	1.22	3.32	5.94	10.21	12.33
	S_1	to	to	**0.44**	**1.09**	**2.44**	3.70
	S_2	to	to	**0.44**	**1.06**	**2.65**	3.80
	S_3	0.29	1.43	3.84	5.88	10.45	11.37
	S_4	0.29	1.19	4.39	6.41	9.90	11.70
	S_5	0.24	2.00	1.96	7.73	15.30	14.81
	S_6	0.32	1.29	3.98	6.01	10.42	11.85
	S_7	**0.23**	**0.67**	1.44	2.56	2.83	**2.50**

8

Subspace Decomposition

Among other use cases, hybrid systems might contain a digital controller interacting with a continuous environment. Formal methods to analyze the discrete controller, i.e., the underlying state transition system induced by the control program, have been developed for a long time and resulted in many successful approaches in the program verification community. The analysis of the continuous counterpart itself, as an independent system, has been worked on in several research communities, for instance in control engineering where dynamical systems and their stability are under interest, or in biology, where the development of a population of specimen can be described as a dynamic system. Further relevant fields for the analysis of dynamical systems include physics, chemistry, or electrical engineering.

A hybrid system allows combining both worlds and models for hybrid systems, thus unifying both behaviors allow putting the controller in the loop with the plant in a single model. As a consequence, both the model of the plant and the model of the controller can be verified as a composed system, in which the controller and the plant interact instead of being analyzed individually.

This chapter is based on our previous work on state space decomposition [SNÁ17], which was done together with my colleague Johanna Nellen, and the extensions described in [SWÁ18; Win18; SÁ19]. All excerpts of the previously mentioned works are used in consent with the co-authors. As a motivating example, we consider the verification of a PLC-controlled hybrid plant.

Programmable logic controllers (PLCs) are digital controllers, which are widely used in industrial applications, for instance in production chains. A PLC has input and output pins that are connected with the sensors and the actuators of a plant. Control programs running on a PLC specify the output of the PLC in dependence of its input readings. These control programs are executed in a cyclic manner with a fixed cycle time, which is guaranteed by design. In the first step, the PLC *reads* the current state of the sensors and the actuators of the plant and stores this information in input registers. Next, all programs on the PLC *execute* in parallel to compute the next output values based on the last input, and store the results in the corresponding output registers. These computations might use local variables, stored in local registers. In the last step

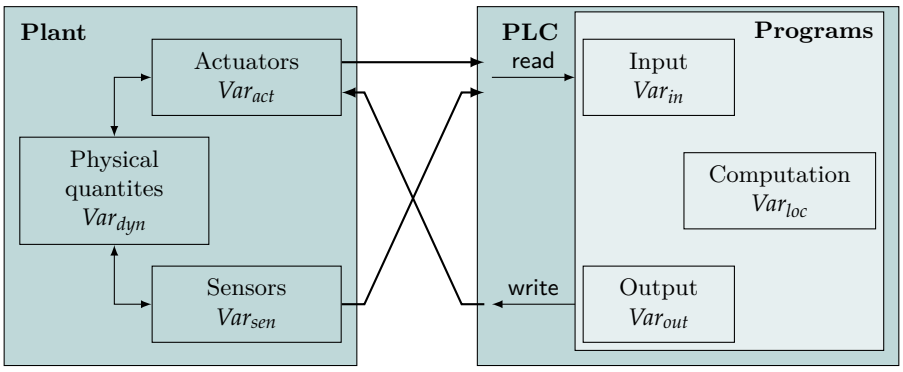


Figure 8.1: Visualization of the plant-PLC interface and interaction between controller and plant (Image credit: Johanna Nellen).

of the cycle, the PLC *writes* the computed output values to the output pins that are connected to the actuators of the plant. Note that industrial PLCs guarantee a constant cycle time, i.e., the time it takes to execute one cycle is known a priori and fixed. General controllers which may not guarantee accurate cycle times still may be modeled in this way by exchanging the fixed cycle time by some time interval.

To model a PLC-controlled plant, we introduce variable sets Var_{dyn}, Var_{act}, and Var_{sen} to represent the state of physical quantities, the actuators, respectively the sensors (see Figure 8.1) of the plant. For the modeling of a controller, we consider variable sets Var_{in}, Var_{out}, and Var_{loc} to represent the PLC registers for input, output respectively local variables. Additionally, we introduce one clock variable for each PLC to account for and ensure the PLC cycle time.

For modeling PLC-controlled plants, we make use of the fact that the PLC execution between reading the input and writing the output has no influence on the plant's state: we model the plant evolution and the concurrent cyclic PLC execution by toggling between a controller model and a plant model, assuming that all controller actions are executed instantaneously after the input is read, the plant evolves for the duration of the PLC cycle, and the output is written at the end of the cycle. We refer to [Nel16] for more information on the modeling of PLC-controlled plants.

Related Work. In the past, some approaches for the state space decomposition for hybrid systems reachability analysis have been developed. In [CS16], the authors present their method for decomposed reachability analysis for non-linear systems using Taylor-model flowpipe construction.

A similar approach for linear hybrid systems, which uses fixed decompositions to two-dimensional subspaces has been presented in [BFF+18]. The authors exploit efficient algorithms for two-dimensional polyhedra to speed up the computation in high-dimensional state spaces.

Both works [BFF+18; CS16] use static decompositions, which do not reflect the structure of the model but which also do not depend on syntactical relations between variables. On the one hand, this allows a more flexible decomposition;

on the other hand, the methods make some assumptions about the dynamical developments in the individual subspaces in order to be able to compute locally.

In the following, we present an approach for the verification of such PLC-controlled plants, which allows making use of domain-specific knowledge to speed up the analysis. Section 8.1 presents preliminary knowledge about different classes of hybrid automata required to understand our approach. The basic idea is presented in Section 8.2. Section 8.3 shows implications on the design and implementation of a flowpipe-construction-based reachability analysis method which implements the presented approach and indicates how to exploit domain-specific knowledge. We present experimental results in Section 8.4 and a prospect towards future development and possible improvements in Section 8.5.

8.1 Subclasses of Hybrid Automata

Hybrid automata can be classified into several subclasses, depending on the dynamics and the type of conditions used (see Table 3.1). In this section, we will shortly reconsider relevant subclasses that are less expressive than linear hybrid automaton II (LHA II) and provide further detail on specialized reachability analysis methods for those subclasses.

Purely Discrete Automata

Technically not hybrid, we still do consider automata where variables are not subject to any flow, i.e., $\dot{x} = 0$ in our analysis, motivated by systems in which digital controllers along with their control program and program variables are part of the model. Discrete variables may only be updated when the control takes a discrete jump, which renders this type of automaton similar to a labeled state transition system (LSTS). Digital controllers can be modeled as discrete automata. Analogous to a LSTS in program analysis, the program variables and thus the state of the controller, respectively the control program only change discretely, i.e., when taking a discrete transition. We refer to variables subject to this behavior as *discrete variables*.

Reachability Analysis. As variable valuations may only be modified when taking a discrete jump, the reachability analysis of purely discrete automata does not require flowpipe computation.

Timed Automata

A *timed automaton (TA)* $\mathcal{T}$ allows to model the most simplistic subclass of hybrid systems, in which all variables $c \in C$ act as *clocks*, i.e., the slope for all variables is fixed to one. We refer to C as the set of clocks. All conditions, e.g., guard conditions or invariant conditions compare single clocks $c_i \in C$ to constants, i.e., are of the shape $c_i \sim k \in \mathbb{Q}$, $\sim \in \{\leq, <, =, >, \geq\}$. The reset function for discrete jumps allows clocks to be reset to zero only. An example of a timed automaton is shown in Figure 8.2.

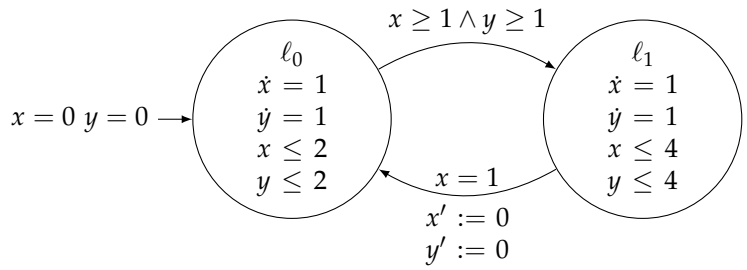


Figure 8.2: Example of a timed automaton with two clocks x, y. Note that clocks are per definition *syntactically independent*.

Timed automata have been studied for a long time, and efficient approaches for computing the set of reachable states for timed automata have been developed, for example, by computing their region transition system or by using zone-based abstraction techniques.

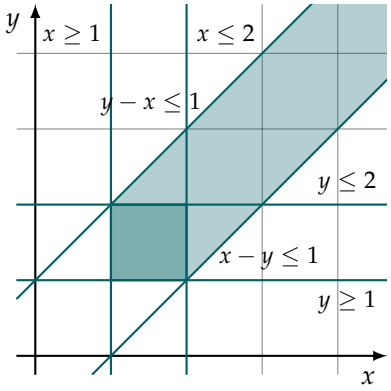
Reachability Analysis. The reachable states of a timed automaton can be computed as a finite union of *zones* storing sets of states. Zones are represented by symbolic state sets whose predicates are conjunctions of constraints of the form $x_i \sim c$ or $x_i - x_j \sim c$ with $\sim \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{Q}$. As the slope for each clock is fixed, zones can be defined by particular types of convex polytopes in $\mathbb{R}^d$ (see Figure 8.3).

Based on the restricted form of the defining constraints, a difference bound matrix (DBM) [Dil90; BY04] offers an efficient representation for a single zone. For instance, the zone in Figure 8.3b can be represented by the DBM

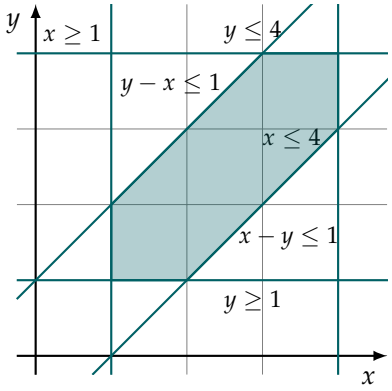
$$D = \begin{array}{c} \\ \mathbf{0} \\ x \\ y \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{0} & x & y \end{array} \\ \begin{pmatrix} (0,\leq) & (-1,\leq) & (-1,\leq) \\ (4,\leq) & (0,\leq) & (1,\leq) \\ (4,\leq) & (1,\leq) & (0,\leq) \end{pmatrix} \end{array}. \tag{8.1}$$

Each constraint $x_i - x_j \sim c$ is represented by an entry $D_{i,j} = (c, \sim)$ in the DBM where an auxiliary dimension $\mathbf{0}$ with constant zero value has been introduced to allow a normalized representation $x_i - \mathbf{0} \sim c$ of constraints $x_i \sim c$. Thus, for a set of n clocks a DBM of size $(n+1) \times (n+1)$ is required to represent a zone.

To compute the set of reachable states of a timed automaton, flow and jump successors of the initial state set represented by a DBM can be computed in an alternating fashion. To compute flow successors of a given zone in a given location of a timed automaton, we increase all upper bounds in the entries $D_{i,0}$ for each clock x_i to the largest value still allowed by the invariant (which might be $+\infty$). For instance, the zone describing the set of reachable states of location ℓ_1 of the automaton in Figure 8.2 can be described by the zone depicted in Figure 8.3b.



a) Initial set for a location in a timed automaton together with flow cone (light shaded).



b) Time successor states with respect to the location's invariant.

Figure 8.3: Graphical representation of the time successor computation in location ℓ_1 of the timed automaton in Figure 8.2 together with the constraints defining the according DBMs.

Example 8.1: Timed Automata Time Successor Computation

The set of states after taking the first discrete jump $\ell_0 \to \ell_1$ (see Figure 8.3a) can be described by the DBM

$$D = \begin{array}{c} \\ \mathbf{0} \\ x \\ y \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{0} & x & y \end{array} \\ \begin{pmatrix} (0,\leq) & (-1,\leq) & (-1,\leq) \\ (2,\leq) & (0,\leq) & (1,\leq) \\ (2,\leq) & (1,\leq) & (0,\leq) \end{pmatrix} \end{array}$$

To represent all time successors, the entries $(x, \mathbf{0})$ and $(y, \mathbf{0})$ representing the constraints $x - \mathbf{0} \leq 2$ and $y - \mathbf{0} \leq 2$ are updated to the largest value still satisfying the invariant condition for each clock respectively, i.e.,

$$\begin{aligned} (x, \mathbf{0}) &= \min(4, \infty) \\ (y, \mathbf{0}) &= \min(4, \infty)\ . \end{aligned}$$

This results in the DBM in Equation (8.1), which represents all time successor states in location ℓ_1 after the first discrete jump (see Figure 8.3b).

Similarly, for discrete jumps intersections with guards as well as clock resets can be represented by adjusting the DBM entries. For further details about timed automata model checking, we refer to [BK08].

Constant Derivatives

Hybrid automata with constant derivatives (rectangular automata and LHA I), i.e., $\dot{x} = c$ represent a super-class to timed automata and can be seen as timed automata with skewed clocks. In this work, we refer to this type as *linear hybrid automata I (LHA I)*. Similar to timed automata, all conditions are represented as linear real arithmetic constraints. In contrast to timed automata, where variables only may be reset to zero, variables can be set to arbitrary constants upon taking a discrete jump. As shown in [HKP+98] unbounded reachability for this class is already undecidable in case the LHA I is not initialized (see Definition 3.5); otherwise, a reduction to an equivalent timed automaton is possible.

Reachability Analysis. Consider a LHA I $\mathcal{H}$ and an initial state represented symbolically by a location $\ell \in Loc(\mathcal{H})$ and a predicate φ being a conjunction of linear real-arithmetic constraints over the variables of the automaton. From the given initial set specified by (ℓ, φ) the set of reachable states can be described by a (possibly infinite) union of such symbolic states. Flow successors of states represented symbolically by (ℓ, φ) can be computed by transforming (ℓ, φ) to another symbolic state $T^+((\ell, \varphi))$. Quantified variables x^{pre} are used in the transformed formula to represent time predecessor states:

$$\exists t.\ \exists x^{pre}.\ t \geq 0 \wedge \varphi\,[x/x^{pre}] \wedge Flow(\ell)\,[x,x'/x^{pre},x] \wedge Inv(\ell)\ .$$

Using variable elimination techniques such as Gaussian elimination and Fourier-Motzkin variable elimination (see Section 2.2), the quantified variables can be eliminated to compute a description of the time successor states. Discrete jump successor states are computed in a similar way by quantifying variables representing the state before the jump.

Example 8.2: Constant Derivatives Flow

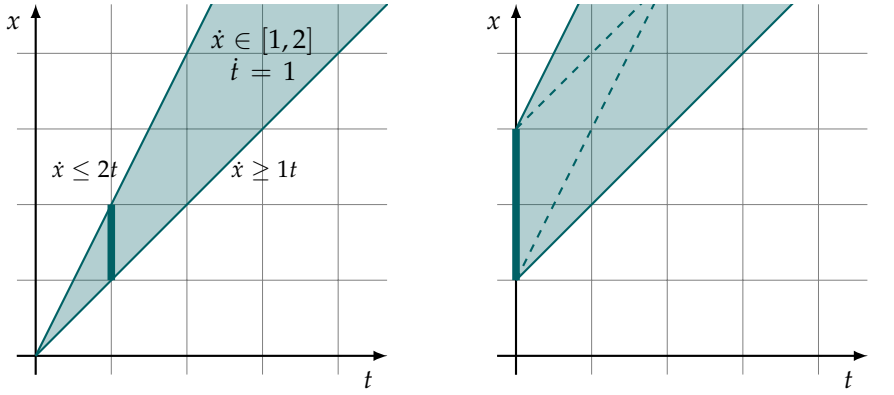
Consider a symbolic state $\sigma = (\ell, \varphi)$ of a given hybrid automaton over variables $Var = \{x\}$ with constant flow $\dot{x} = 4$ in location ℓ. Starting from $\varphi = x \in [2,3]$ we can describe the time successors of σ as

$$\exists t.\exists x^{pre}.\ t \geq 0 \wedge 2 \leq x^{pre} \wedge x^{pre} \leq 3 \wedge x = x^{pre} + 4t\ .$$

We can use Gaussian elimination to eliminate x^{pre} and use Fourier-Motzkin variable elimination to eliminate t:

$$\begin{aligned} &\exists t.\exists x^{pre}.\ t \geq 0 \wedge 2 \leq x^{pre} \wedge x^{pre} \leq 3 \wedge x = x^{pre} + 4t \\ \Leftrightarrow &\exists t.\ t \geq 0 \wedge 2 \leq x - 4t \wedge x - 4t \leq 3 \\ \Leftrightarrow &2 \leq x\ . \end{aligned}$$

Alternatively to the above approach, we can also apply flowpipe construction as presented before, however, there is no need for a segmentation: segmentation was introduced to reduce the over-approximation error due to non-linear



a) Construction of the flow cone for the dynamics $\dot{x} \in [1,2], \dot{t} = 1$ for unbounded t.

b) Adding (Minkowski sum) the cone for the flow of x and t to the initial set $x \in [1,3] \wedge t = 0$ yields the set of time successors.

Figure 8.4: Example for the construction of time successor state sets for rectangular hybrid automata.

evolution but linear over-approximation. For constant derivatives the evolution is linear, thus the whole flowpipe is a linear set. The previous flowpipe construction is replaced by a more simple approach, starting from the initial set, computing a cone for the dynamics, computing the Minkowski sum and building an intersection with the invariant (see Figure 8.4 for an example).

8.2 Syntactic Decomposition

In this section, we show how to make use of the previous observations about different subclasses of hybrid automata in general reachability analysis for hybrid automata. Using the previously introduced PLC-controlled plant as a motivation, we will develop an extension to flowpipe-construction-based reachability analysis as presented in Section 3.4, which exploits the presence of different types of dynamics in one model.

For practically relevant applications, the current modeling approach, where LHA II are used to model for instance PLC-controlled plants by hybrid automata, usually leads to huge models as locations corresponding to steps in the control program need to be modeled in the hybrid automaton as well. Apart from potentially many locations in the corresponding hybrid automaton $\mathcal{H}$, the increased state space dimension poses a serious problem for the verification of such automata. In our example of a PLC-controlled plant, the variable set contains variables modeling the plant dynamics Var_{dyn}, the states of sensors and actuators (Var_{sen}, Var_{act}), the input and output values of the PLC (Var_{in}, Var_{out}), the local variables used in program executions (Var_{loc}), and clocks for PLC cycle synchronization. As most state set representations are sensitive to the state space dimension in terms of computational effort required to perform certain operations, the resulting high dimensional state space leads to computationally

expensive operations and heavy memory consumption during reachability analysis. With the help of our representative example of a PLC-controlled plant, we identify several system properties that can be exploited, leading to a more scalable approach for its verification.

Firstly, as the PLC along with its control program is part of the model, the program variables of the PLC behave in a *discrete* way (see Section 8.1). The values for discrete variables do not change dynamically over time but only when taking discrete jumps from one location in the controller to another controller location in the composed hybrid automaton. For all discrete variables d_i the flow in every location is zero, i.e., $\dot{d}_i = 0$ for all $\ell \in Loc$. Additionally, the variables representing the state of actuators and sensors can be modeled by discrete variables too, as actuator states change discretely (when writing the output) and the sensor values are relevant only at the beginning of each cycle (when reading the plant state). Thus, only the physical quantities modeled in the plant and the cycle clocks evolve continuously, which coincides with the natural perception of the modeled system. Finally, computing flowpipes for clocks and other variables with constant derivatives can usually be done using specialized approaches instead of using approaches for dynamics specified by linear ordinary differential equations (ODEs) (see Section 8.1).

In the presented approach, we divide the variable set $Var_{\mathcal{H}}$ of the hybrid automaton $\mathcal{H}$ into disjoint subsets $Var_{\mathcal{H}} = V_0, \ldots, V_{n-1}$. The decomposition of $Var_{\mathcal{H}}$ is based on properties shared between the variables in V_i, which are relevant for the reachability analysis of $\mathcal{H}$. Furthermore, we may analyze the subspaces $S_0, \ldots, S_n$ induced by the variable subsets $V_0, \ldots, V_{n-1}$ independently. As we will show later, the decomposition is based on the properties relevant for classifying a particular subclass of hybrid automata (see Section 8.3). In the following, we will first describe the general idea of subspace decomposition informally, then provide a formal description, and finally show the relation to the subclasses of hybrid automata.

To be able to compute the set of reachable states in one of the subspaces S_i, its variables $v \in V_i$ need to be independent from all other variables in the sense that their continuous as well as discrete evolution is not influenced by other variables $w \notin V_i$ "directly" but only "indirectly" through time as formalized below. Furthermore, we ensure that the variables $v \in V_i$ do not influence other variables not being part of V_i. Formally, all predicates $\varphi \in Pred_{Var_{\mathcal{H}}}$ present in the hybrid automaton $\mathcal{H}$ must be decomposable to a conjunction $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_{n-1}$ of predicates $\varphi_i \in Pred_{V_i}$ over the respective variable subsets V_i. Similarly, the same property for jump resets specified by $Pred_{X \cup X'}$ and flows given as $Pred_{X \cup \dot{X}}$ needs to hold for the same subspace decomposition. We refer to this property as *syntactic independence*.

Definition 8.1: Syntactic Independence

Assume a hybrid automaton $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$. A decomposition $Var_{\mathcal{H}} = V_0 \cup \ldots \cup V_{n-1}$ of the variable set of $\mathcal{H}$ into disjoint subsets is called *syntactically independent*, if the following holds:

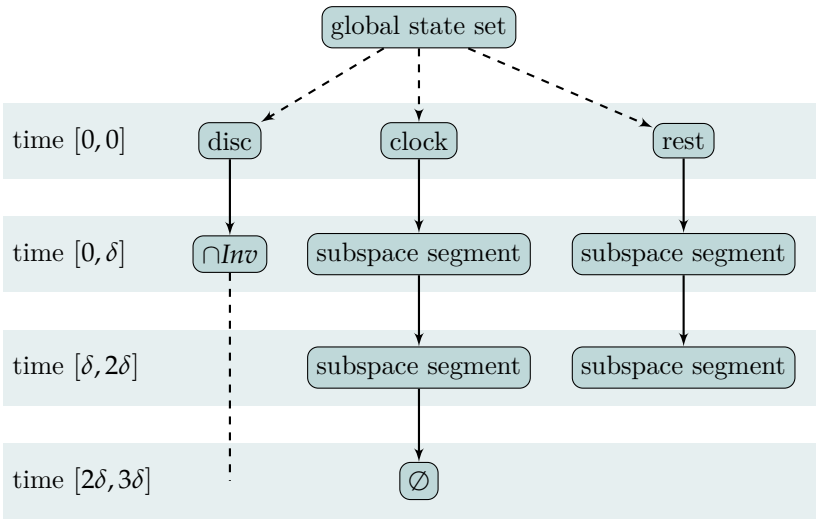


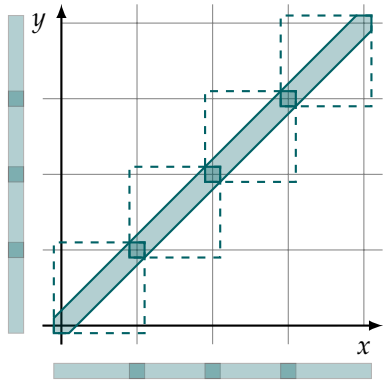
Figure 8.5: Example of a decomposition into a discrete subspace (disc), a timed subspace (clock) and a linear subspace (rest). Implicit time synchronization allows to stop computation as soon as the invariant is violated in one subspace (timed subspace).

- each invariant and each guard in $\mathcal{H}$ can be written as a conjunction $\varphi_0 \wedge \ldots \wedge \varphi_{n-1}$ with $\varphi_i \in Pred_{V_i}$ for each $i = 0, \ldots, n-1$;
- each flow in $\mathcal{H}$ can be written as a conjunction $\varphi_0 \wedge \ldots \wedge \varphi_{n-1}$ with $\varphi_i \in Pred_{V_i \cup \dot{V}_i}$ for each $i = 0, \ldots, n-1$;
- each reset in $\mathcal{H}$ can be written as a conjunction $\varphi_0 \wedge \ldots \wedge \varphi_{n-1}$ with $\varphi_i \in Pred_{V_i \cup V_i'}$ for each $i = 0, \ldots, n-1$.

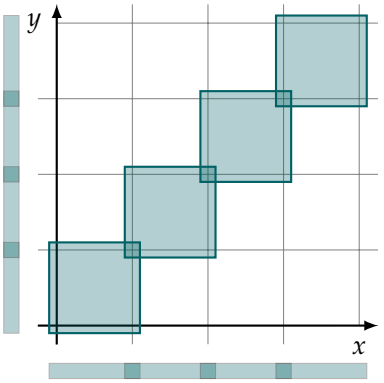
We also name the variable subsets themselves *syntactically independent*.

The decomposition of predicates such as invariant or guard conditions implies temporal synchronization between the different subspaces in the sense that as soon as one invariant condition is violated in one subspace, the control is forced to leave the respective decomposed location in all corresponding subspaces. Correspondingly, a guard is only enabled if it is enabled in all subspaces at the same time. This is what we described above as "indirectly influenced". In the following, we assume that the time step size δ used for the continuous time successor computation is the same in all decomposed subspaces. Each computed flowpipe segment over-approximates a certain time interval (see Section 3.4). We can use the associated time intervals to realize an over-approximative synchronization between the subspaces as depicted in Figure 8.5.

A decomposition of the variable set $Var_{\mathcal{H}}$ into syntactically independent subsets $V_0, \ldots, V_{n-1}$ allows us to represent (global) state sets $(\ell, \mathrm{N}) \subseteq Loc \times \mathbb{R}^d$ by their projections $\mathrm{N}_i = \mathrm{N}{\downarrow_{V_i}} \subseteq \mathbb{R}^{|V_i|}$ to the subspaces S_i; we call $(\ell, \mathrm{N}_0, \ldots, \mathrm{N}_{n-1})$ the *projective representation* of (ℓ, N) with respect to the



a) Projective representation of two variables which are connected results in an over-approximation in the global state space (dashed).



b) Projective representation of two variables which are not connected. The projective representation is exact as the variables are not connected in the global state space.

Figure 8.6: Projective representation of a two-dimensional space onto two one-dimensional spaces.

variable decomposition $V_0, \ldots, V_{n-1}$. Note that the projective representation drops the connection between the subspaces and is therefore *over-approximative*, i.e., $\mathrm{N} \subseteq \mathrm{N}_0 \times \cdots \times \mathrm{N}_{n-1}$ but in general $\mathrm{N} \neq \mathrm{N}_0 \times \cdots \times \mathrm{N}_{n-1}$ (see Figure 8.6a). Equivalence can only be achieved if state sets are represented by boxes, as boxes naturally cannot represent dependence between variables. The Cartesian product of the projections of sets of interval-valued variables, i.e., a box is the box itself. Therefore the projective representation of boxes is exact (see Figure 8.6b). We will come back to this property when describing the technical realization of our approach in Section 8.3.

Once the variables are separated as presented, we modularize the reachability analysis computation by analyzing the subspaces $S_0, \ldots, S_{n-1}$ induced by the variable subsets $V_0, \ldots, V_{n-1}$ independently. Intuitively, our goal is to replace computations in the global, high-dimensional state space by multiple independent computations in lesser-dimensional subspaces.

Graph-based Decomposition

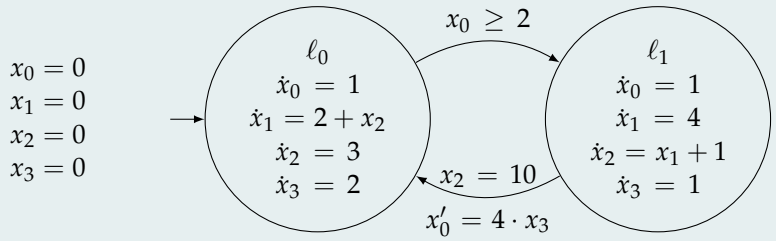
To compute a syntactically independent decomposition, we use a graph-based approach in which subspaces are discovered in a bottom-up fashion by iteratively adding edges representing syntactic dependency. Initially, we assume that all variables $Var_{\mathcal{H}} = \{x_0, \ldots, x_{d-1}\}$ of a given hybrid automaton $\mathcal{H}$ are syntactically independent, i.e., we start with a decomposition $Var_{\mathcal{H}} = V_0 \cup \ldots \cup V_{d-1}$ where $V_i = \{x_i\}$ for all subsets V_i. Traversing the automaton, we can iteratively unite sets of variables V_i, V_j whenever one of the conditions for syntactic independence (see Definition 8.1) between V_i and V_j is violated. During the process, we create an undirected graph $G = (L, E)$ with a set of nodes L and a set of edges E. Each location $v_i \in L$ represents a variable x_i of $\mathcal{H}$ while an edge

between two locations v_i, v_j indicates syntactic dependence of the corresponding variables $x_i, x_j \in Var_{\mathcal{H}}$. For each location ℓ in $\mathcal{H}$, we add edges connecting v_i and v_j in G whenever x_i depends on x_j in the flow specification in ℓ, i.e., the entry $\mathcal{A}_{ij}$ in the flow matrix $\mathcal{A}$ of ℓ is non-zero. Similarly, we add edges (v_i, v_j) connecting v_i and v_j in case in the invariant condition $Inv(\ell) = \bigcap h_k$ one of the h_k relates x_i and x_j. A similar approach can be used for the discrete jumps in $\mathcal{H}$. For each jump, an edge connecting v_i and v_j is added whenever x_i and x_j are dependent in the guard condition or in case the reset function creates a dependency for both variables. This is the case when either one of the variables x_i, x_j is on the left-hand-side and the other one on the right-hand-side of the reset assignment, for instance $x_i' = x_j$ or when both occur on the right-hand-side for the assignment of an additional (also dependent) variable x_k, for example $x_k' = x_i + x_j$ (transitivity).

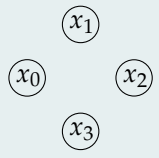
After having traversed the whole automaton, the set of locations of each connected component of the graph G represents the set of variables inducing a common subspace in $\mathcal{H}$. Note that this approach uses an undirected graph that is created based on the whole automaton. Consequently, the subspace decomposition obtained by this procedure thus holds for the whole automaton. Later on, we present ideas on how to strengthen this decomposition by using local decompositions instead (see Section 8.5).

Example 8.3: Decomposition

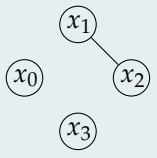
Consider the hybrid automaton $\mathcal{H}$ as given below:



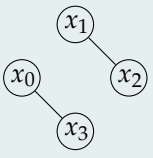
We create the dependency graph $G = (L, E)$ for the variables $x_0, x_1, x_2, x_3 \in Var_{\mathcal{H}}$:



Initial graph.



Adding edges for flow in ℓ_0 and ℓ_1.



Adding edges for discrete jumps.

From the connected components of G we can derive two syntactically independent subspaces S_0, S_1 induced by the variable sets $Var_0 = \{x_0, x_3\}$ and $Var_1 = \{x_1, x_2\}$.

8.3 Modular Reachability Analysis

The approach of syntactic decomposition of a hybrid automaton $\mathcal{H}$ with variables $Var_{\mathcal{H}}$ into the subspaces $S_0, \ldots, S_{n-1}$ effectively decreases the dimension of the underlying state space for flowpipe-construction-based reachability analysis as each subspace is analyzed individually. While more flowpipes need to be computed, i.e., a separate analysis is performed on each subspace S_i, the reduced dimension of the single subspaces has a positive effect on the running time as the influence of the state space dimension on the state set operations often is non-linear. We can observe that each subspace itself can be classified into one of the hybrid automata subclasses (see Section 8.1). As each subspace is analyzed individually, this observation enables us to use dedicated analysis methods depending on the classification of the respective subspace. Note that some of the subclasses of hybrid automata even naturally induce their own subspace. For instance, for TA, all clocks naturally are syntactically independent, as their dynamics is always one and clocks are reset only to zero. In any condition, i.e., guards or invariants, clocks are only compared to constants. Consequently, all variables in $\mathcal{H}$ that behave like clocks of a timed automaton induce their separate subspace when $\mathcal{H}$ is decomposed.

Not only timed automata but also other subclasses of hybrid automata can be analyzed with specialized methods, which are usually more efficient than the flowpipe-construction-based reachability analysis for linear hybrid automata as described in Section 8.1. Subspace decomposition opens an opportunity to create a flowpipe-construction-based reachability analysis framework, which allows increasing the efficiency of the analysis by using the most appropriate approach available depending on the dynamics of the subspace which is to be analyzed. In the following, we will use the terms *discrete subspace*, *timed subspace*, *LHA I subspace*, and *LHA II subspace* to refer to the type of dynamics used in a subspace.

To realize this behavior in a full reachability analysis framework, we introduce the concept of *context dependent workers*. The general idea of a worker is identical to the one described in Section 7.2, which means that a worker processes tasks t_i, i.e., computes a single flowpipe based on the information contained in t_i. In contrast to the initial definition of a worker, we extend this concept by putting the worker into the respective context of the current subspace, which allows using specialized workers, depending on the subspace and its dynamics. Following this idea, we can create workers for all relevant dynamics, for instance one worker type which allows to analyze timed subspaces, or another worker which implements a method for LHA II as presented before. In the following, we give technical details on how we realize context-dependent workers in a modularized way in our tool prototype HyDRA.

Handler-based Workers. Computing a single flowpipe in a specific location of a hybrid automaton starting from a predefined initial set requires several steps. We can sub-divide the whole process of computing a flowpipe into several steps which are almost independent of the actual dynamics or approach which is used to compute the flowpipe. In classical flowpipe construction for

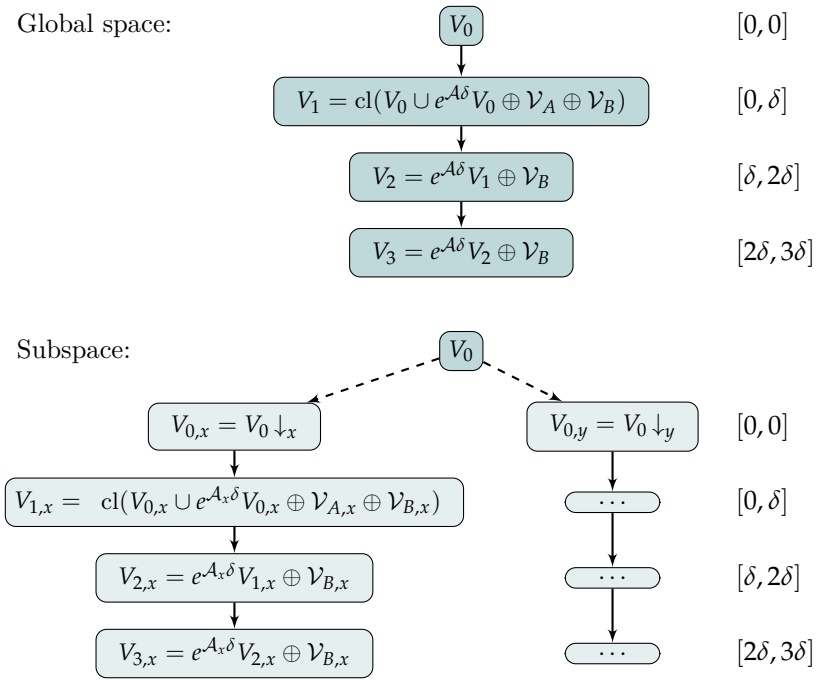


Figure 8.7: General flowpipe-construction-based reachability analysis for hybrid systems (top) in comparison to the analysis of a decomposed system (bottom) using projective representations of state sets in the subspaces x, y resulting from decomposition.

linear hybrid systems (LHA II), a first flowpipe segment needs to be computed. Afterward, the following segments are obtained by a recurrence relation (see Section 3.4). Intersections with guards, invariant conditions and bad states have to be computed during the analysis for each computed flowpipe segment and potential discrete jump successor sets have to be computed.

In our implementation, we exploit the modular nature of this approach towards instantiating modularized workers, which are composed of a collection of handlers dealing with the respective subtask. The general structure of a modular worker is given in pseudo-code in Algorithm 11. Note that each function call in Algorithm 11 delegates the respective task to the handlers stored in the context. Based on the subspace decomposition for the current system, each function call is forwarded to a suitable handler for each subspace.

Designing workers in such a modular way allows to easily exchange handlers dedicated to a specific task in case a new method has been implemented. Furthermore, this allows us to reuse handlers; for instance, guard intersection handling for timed, constant, and rectangular contexts is similar and thus can be done by the same handler.

As handlers maintain an internal state, this enables handlers for different subspaces to co-exist in the same worker. For instance, a handler implementing LHA II time successor computation based on time discretization will compute one flowpipe segment at a time, each over-approximating a time interval $[i\delta, (i +$

Algorithm 11: Handler-based worker structure.

Input: Task $t = (n_i, Par_j, n^*)$
Output: over-approximation of reachable states R

```
firstSegmentHandler()
invariantHandler()
badStateHandler()
jumpHandler()
while !isFinished() do
    timeHandler()
    invariantHandler()
    badStateHandler()
    jumpHandler()
```

$1)\delta]$ of size δ. On the other hand, a time successor handler for a timed subspace only requires one step to compute all time successor states. By maintaining an internal state, both handlers may co-exist in the same reachability worker, the time successor handler for the timed subspace will only compute a set of reachable states at its first call and otherwise will idle while the worker for the LHA II flowpipe construction will continue computing its segments. Another example is depicted in Figure 8.7, where for a discrete subspace the invariant is tested only once as there is no flow in this subspace while in other subspaces flowpipe segments are computed. In a handler-based worker, the handler responsible for validating the invariant in a discrete subspace thus will skip the computation using its internal state which stores the result obtained on the first call.

8.4 Examples and Experimental Results

In this section, we will present experimental results obtained using our tool-prototype HyDRA equipped with state set decomposition methods. The results on PLC-controlled plants were obtained together with Johanna Nellen who provided the models and the static decomposition. The original results and benchmark descriptions were published together in [SNÁ17] and are presented here in consent with the co-authors. For further details on the decomposition we refer to [Nel16]. The results on the automatic decomposition using specialized analysis approaches presented afterward were originally published in [SWÁ18; SÁ19], where we evaluated our implementation of said approaches in HyDRA. To focus on the effects of the decomposition, all results were obtained using the presented decomposition but without partial path refinement or parallelization from Section 7.5.

PLC-controlled Plants

In this section, we present our first experimental results on syntactic decomposition as published in [SNÁ17] together with Johanna Nellen. The approach presented in this work is based on a user-defined static decomposition which

Table 8.1: Model sizes of the benchmarks before and after decomposition.

Model	Type	#variables			#modes		#jumps
		disc.	clocks	rest	contr.	plant	
`LTnk`	original	0	0	12	8	3	34
	timed	0	2	10	8	3	34
	discrete	9	0	3	8	3	34
	timed & discrete	9	2	1	8	3	34
`2Tnk`	original	0	0	22	20	14	296
	timed	0	3	19	20	14	296
	discrete	17	0	5	20	14	296
	timed & discrete	17	3	2	20	14	296
`Thmo`	original	0	0	8	6	2	18
	timed	0	2	6	6	2	18
	discrete	5	0	3	6	2	18
	timed & discrete	5	2	1	6	2	18

is provided a priory. The motivation originates from the verification of PLC-controlled plants where the model contains both the controller and the plant. Consequently, the state space contains variables of the controller as well as plant variables where the controller variables do not exhibit dynamic behavior and thus unnecessarily increase the state space dimension.

Benchmark Selection. To analyze the capabilities of the decomposition approach, we consider three well-known entry-level benchmarks: the leaking tank benchmark (`LTnk`); the two tanks benchmark (`2Tnk`); and the thermostat benchmark (`Thmo`). Each of the benchmarks is extended by a controller behaving like a PLC-controller, i.e., in each cycle the controller reads the plant state, executes a small control-program (here: opening/closing valves or switching the mode of the thermostat), and writes output which influences the dynamic behavior of the plant accordingly. Aside from adding modes and transitions for the controller, we also introduce variables accounting for the program variables in the controller as well as actuator and sensor states red by the controller and one clock for cycle synchronization for each introduced PLC-controller to model cycle time. A schematic overview of the interaction between a general plant and its controller is depicted in Figure 8.1.

We use a static decomposition where we fix the subspace types for each variable a priory, i.e., controller variables behave like discrete variables (zero derivatives), a clock is added to ensure cycle timings and the plant variables behave as in the original versions of the benchmarks. In our experiments, we compare the analysis of the benchmarks without variable separation ("original") with variable-set-separation-based analysis separating only clocks ("timed"), only discrete variables ("discrete"), and both ("timed & discrete"). The sizes in terms of locations, variables, and transitions of the resulting benchmarks are shown in Table 8.1, where we also provide the dimension of the resulting subspaces. In the following, we shortly describe the extensions to the benchmarks behavior caused by the added PLC-controller.

As in the original version the PLC-equipped version of the *leaking tank* benchmark models a leaking water tank, i.e., it has a constant outflow. The tank can be refilled from an unlimited external resource with a constant inflow that is larger than the outflow. The added PLC-controller triggers refilling (by switching a pump on) if a sensor indicates a low water level ($h \leq 6$). If the water level is high ($h \geq 12$) the controller stops refilling (switches the pump off). Adding the controller to the model introduces two variables representing controller input for low and high water levels, variables for the actuator (pump) state in the plant and the controller, and a variable to store the controller mode. Furthermore, a new clock is added to model the PLC cycle time. Besides the controller, we also model a user that can manually switch the pump on and off as far as the water level allows it. In our implementation, the user constantly toggles between the pump states on and off. We analyze the system behavior over a global time horizon of 40 s using a PLC cycle time of 2 s.

The PLC-equipped version of the *two tanks* benchmark models the water levels of two water tanks in a closed system. Each tank has a constant inflow and a constant outflow. The tanks are connected via pipes, such that the amount of water outflow of the first tank is equal to the inflow of the second tank and vice versa. One pump per pipe allows to enable/disable the water flow. We add a PLC-controller to the two tank system that controls the pumps. A pump is switched off if the water level of the source tank is low ($h \leq 8$) or if the water level of the target tank is high ($h \geq 32$). Each time a pump is switched off by the controller, the other pump is switched on to balance the water levels in the tanks. The introduction of the controller adds variables to model sensing low and high water levels of both tanks and variables to model the actuator (pump) states in the plant and the controller. Moreover, we add a variable to store the controller mode and a new clock to model the PLC cycle time. Again, we model a user who switches the pumps manually on or off as far as the water levels allow it. We implemented a user that toggles the state of each pump in each PLC cycle. The global time horizon and a PLC cycle time were set to 20 s respectively 1 s.

The third extended benchmark is a variant of the thermostat benchmark, where a room heater with a thermostat controller is modeled. Initially, the temperature is $t = 20\,°\mathrm{C}$ and the heater is on. The controller keeps the temperature t between 16 °C and 24 °C. The heater is switched off if the temperature rises above 23 °C and it is switched on at a temperature below 18 °C. Adding a controller to the model introduces new variables for the low and high temperature sensors in the controller, a variable for the actuator (heater) state in the plant and the controller, and a variable to store the controller mode. Additionally, we introduce a new clock for the cycle time of the PLC. The global time horizon is 10 s and the PLC cycle time is 0.5 s.

Results. All computations were carried out on a machine with $4 \times 4\,\mathrm{GHz}$ Intel Core i7 CPUs and a memory limit of 8 GiB. We used the time step size $\delta = 0.01$ and an unlimited jump depth. To be able to express a global time horizon, each model is equipped with a global clock and according invariant constraints. The running times can be found in Table 8.2a, the number of

Table 8.2: Benchmark results for different separation set-ups. Running times are in seconds, time-out (to) was 20 min, the second table lists the number of flowpipes computed.

a) Running times in seconds.

				HyPro			SpaceEx
Model	Rep.	Agg	original	timed	disc.	timed & disc.	original
LTnk	box	agg	2.70	2.08	1.06	1.13	3.67
	box	none	2.62	2.09	1.06	1.13	3.82
	sf	agg	to	to	161.12	37.03	448.30
	sf	none	to	1044.97	19.49	5.84	444.82
2Tnk	box	agg	4.39	2.60	0.97	1.15	5.49
	box	none	4.46	2.68	1.02	1.16	5.53
	sf	agg	to	to	900.11	329.80	to
	sf	none	to	to	35.04	14.64	to
Thmo	box	agg	0.07	0.09	0.06	0.06	0.57
	box	none	0.11	0.09	0.06	0.06	0.57
	sf	agg	35.87	22.69	1.17	0.29	9.89
	sf	none	30.41	20.19	1.18	0.30	9.91

b) Number of computed flowpipes.

				HyPro			SpaceEx
Model	Rep.	Agg	original	timed	disc.	timed & disc.	original
LTnk	box	agg	662	662	662	662	200
	box	none	662	662	662	662	200
	sf	agg	to	to	662	662	425
	sf	none	to	662	662	662	425
2Tnk	box	agg	470	470	470	470	195
	box	none	470	470	470	470	195
	sf	agg	to	to	470	470	to
	sf	none	to	to	470	470	to
Thmo	box	agg	95	95	95	95	95
	box	none	95	95	95	95	95
	sf	agg	95	95	95	95	84
	sf	none	95	95	95	95	84

computed flowpipes is shown in Table 8.2b. We have compared our approach to the tool SpaceEx (version 0.9.8f) using a similar configuration for the analysis. In our tool, we used boxes and support functions (with an octagonal template) to represent state sets, whereas in SpaceEx we used support functions with four and eight directions, as SpaceEx does not support explicit box representations. Furthermore, SpaceEx maintains a fixed-point detection method that our implementation does not feature. For the leaking tank and the two tanks benchmarks this affects the results as both benchmarks exhibit branching of paths during the execution where the branches can be merged later during the analysis, i.e., it suffices to analyze one of the branches after said merging point. Methods implementing fixed-point detection can recognize this, while otherwise

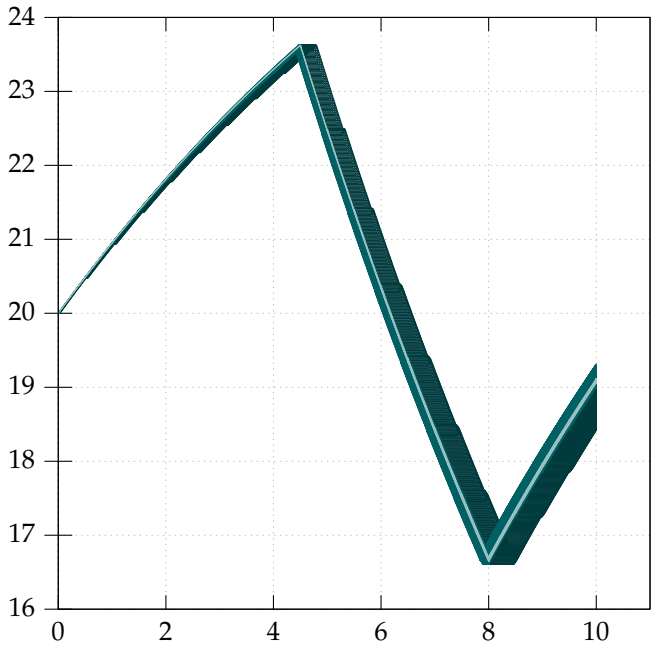


Figure 8.8: Results on the thermostat benchmark using a support function representation (octagonal template). The plot shows the temperature over time for SpaceEx (very light petrol), HyPro using either no or only discrete variable separation (which have the same precision and are both colored in light petrol), and HyPro with discrete variable- and clock-separation (dark petrol).

both branches which are identical after the merging point will be fully analyzed. This is reflected in the number of computed flowpipes (see Table 8.2b).

A further difference is that we may use different state set representations for the separate subspaces in HyPro: in our case, we use boxes (`box`) for clocks and discrete variables and only vary the representation for the plant variables between boxes (`box`) and support functions (`sf`).

From the results in Table 8.2a, we can see that subspace decomposition has a significant effect on the running times as a result of the lower dimension in the subspaces. On the other hand, as mentioned before subspace decomposition may introduce additional over-approximation errors as segments of different subspaces are only implicitly connected via the time interval they over-approximate. We can observe this behavior in Figures 8.8 and 8.9, where we show plots for leaking tank and thermostat.

The observed speed-up due to the discrete variable separation is in general larger than the influence of a clock separation, as in our benchmarks the discrete variables outnumber the clocks. Furthermore, this implementation already skips repeated tests for guards and invariant conditions on discrete variables as they do not change over time. Nonetheless, a separation of clocks already shows a speed-up of about 30 %. As mentioned before, we used boxes as a state set representation for the set of discrete variables, which does not introduce any further over-approximation error, as the discrete variables themselves are all

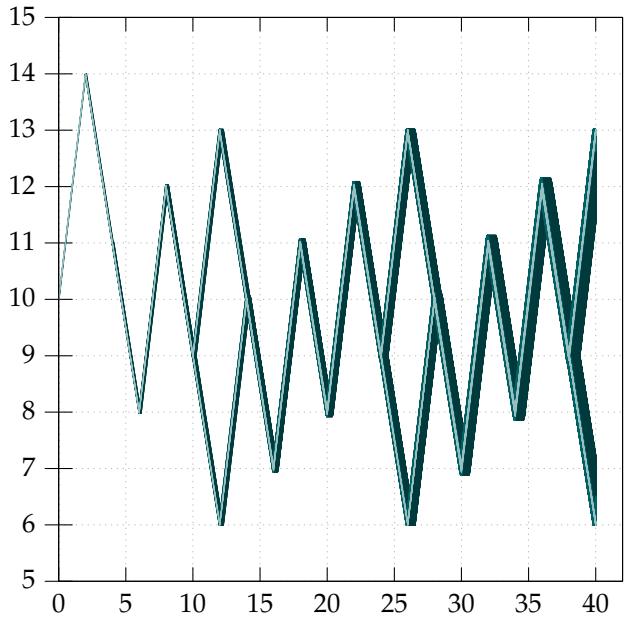


Figure 8.9: Plotted approximation of the set of reachable states (water-level over time) of the PLC-variant of the leaking tank system (`LTnk`) using state space decomposition on clocks and discrete variables (dark petrol), no decomposition or decomposition on discrete variables only (petrol) and results obtained using SpaceEx (very light petrol). All results were obtained using $\delta = 0.01$, full aggregation and support functions with an octagonal template.

syntactically independent. We can observe that using boxes as a state set representation, our implementation outperforms SpaceEx (even when a lot more flowpipes are computed) in terms of running time, which is expected, as boxes in general require less computational effort than support functions (evaluated in four directions) in reachability analysis.

In HyPro, aggregation causes longer running times because in the current implementation aggregation is realized by a conversion of the single sets (which are to be aggregated) to polytopes, which is computationally expensive, especially in higher dimensions.

Automated Subspace Decomposition

In this section, we present experimental results obtained by an extension of the previous method by automated detection of subspaces as presented in Section 8.2. Furthermore, we evaluate the usage of specialized approaches towards the reachability analysis of specific subclasses of hybrid automata such as *timed-* and *rectangular automata*. The following results were originally published in [SWÁ18; SÁ19] and are presented here with the consent of the co-authors.

Table 8.3: Running times in seconds for automated decomposition, time out (to) was set to 20 min, models which could not be verified are marked with "†".

a) Running times for benchmark instances with non-rectangular subspaces. The upper part shows results for benchmarks which are not decomposable (`Ball`, `Sw5`) and the lower part shows running times for decomposable models. This table was originally published in [SWÁ18].

Model	`box` $\delta = .01$		`box` $\delta = .001$		`sf` $\delta = .01$		`sf` $\delta = .001$	
	no dec.	dec.	no dec.	dec.	no dec.	dec.	no dec.	dec.
`Ball`	†	†	**0.21**	**0.21**	**0.17**	**0.17**	0.85	**0.81**
`Sw5`	†	†	†	†	†	†	**0.32**	**0.32**
`Fish`	**5.21**	9.68	**59.20**	110.00	to	**27.30**	to	to
`Pltn`	†	†	†	†	3.21	**1.78**	**19.80**	to
`Rods`	**0.82**	0.94	**7.78**	9.11	38.40	**7.69**	to	to
`2Tnk`	**0.98**	1.37	**5.84**	11.50	to	**1.22**	to	**8.78**

b) Running times for benchmark instances with rectangular subspaces.

Model	`rectangular`
`Fish R`	20
`5var_system`	19.40

Benchmark Selection. To test the influence of automated decomposition on further examples, we have selected a set of commonly known benchmarks for evaluation. Among them the bouncing ball (`Ball`) , an instance of Fisher's mutual exclusion protocol (`Fish`), the model of a vehicle platoon (`Pltn`), the simplified model of a temperature control of a reactor (`Rods`), an artificial 5D linear switching system (`Sw5`), and a model of two leaking tanks with a controlled inflow (`2Tnk`). As before, all experiments were carried on a machine with 4×4 GHz Intel Core i7 CPUs and a memory limit of 8 GiB and a timeout (to) of 10 min. The resulting running times for our experiments can be found in Table 8.3a.

Due to the lack of published benchmarks for rectangular automata, we used an artificial model with five variables (`5var_system`) taken from [CÁF11]. Furthermore, we created an equivalent instance of `Fish` using a rectangular automaton model to test our approach (`Fish R`), as the original dynamics are constant and thus may be expressed by point-intervals. The running times for those experiments can be found in Table 8.3b.

Settings. The chosen analysis parameter configurations vary the state set representation between boxes (`box`) and support functions (`sf`) and the time step size between $\delta = 0.01$ and $\delta = 0.001$. All settings used aggregation (`agg`) for the discrete jump successor computation. As before, these settings are used for the analysis of the LHA II automata and their decomposition. Note that we use dedicated methods for the analysis of timed subspaces as presented in Section 8.1, which do not comply with these settings as they do not use time

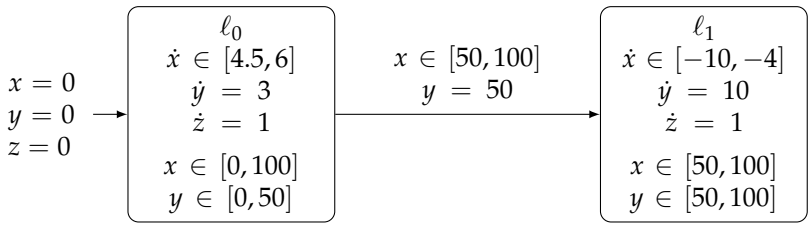
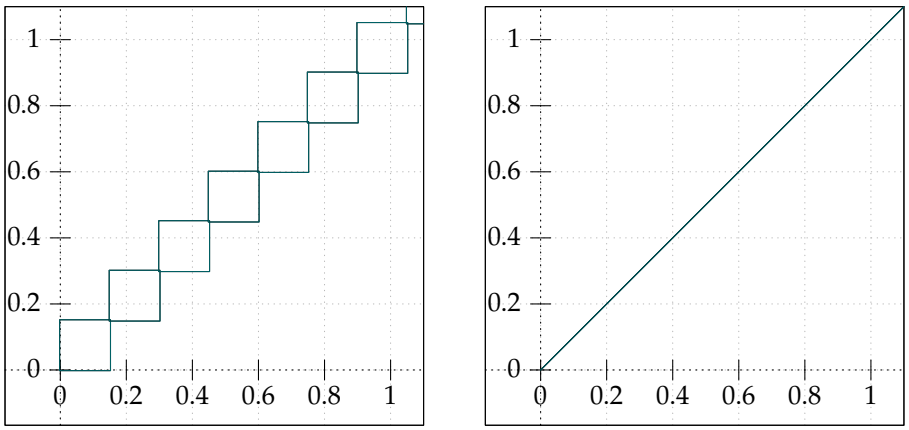


Figure 8.10: Small toy example (`toy`) to test the decomposition of mixed-dynamics hybrid automata.

discretization and in our case operate on DBMs as a state set representation. Configurations denoted by "dec." use the presented approach of graph-based subspace decomposition while configurations marked with "no dec." denote the classical approach without decomposition. As before we mark configurations that could not prove safety with "†" and configurations where the analysis exceeded the time limit by "to".

Results. The benchmark instances `Fish` (4×1), `Pltn` (2×1, 1×10), `Rods` (3×1) and `2Tnk` (10×1, 6×2) can be decomposed into subspaces as indicated in the brackets (number of sets $V_i \times |V_i|$) while `Ball` and `Sw5` are not decomposable. The rectangular version `Fish R` is decomposable into subspaces of dimension 1×3 and 1×1 where the first subspace is rectangular. The state set representation for the analysis of rectangular subspaces is fixed (see Section 8.1) to polytopes in $\mathcal{H}$-representation, i.e., state sets are described as a conjunction of linear inequalities. As shown before reachability analysis methods for rectangular automata do not use time discretization as the set of reachable states can be computed in one step per location using logical formulas as state set representations and Fourier-Motzkin variable elimination techniques. Consequently, we do consider neither time step size nor state set representation as analysis parameters for this approach.

We can observe that the effect on running times is the largest when using support functions. The reason for this lies in the complexity of operations on support functions in comparison to the overhead introduced when computing a subspace decomposition. The most crucial operation, sampling the support, highly depends on the dimension of the state space, such that the speed-up resulting from multiple, but lower-dimensional state spaces compensates for the decomposition overhead. Even small reduction, as for instance in the platoon benchmark (two dimensions) is noticeable, the decomposition of `Fish` into one-dimensional subspaces allows to obtain results while the analysis using the original four-dimensional state space exceeds the time limit. Note that in the non-rectangular Fisher-model, the decomposed subspaces all require the usage of the linear context (LHA II) while the separated one-dimensional subspaces in the platoon-model can both be computed using a timed context. The subspaces in the two tanks benchmark (`2Tnk`) are mostly of discrete nature, which explains the huge speed-up when using support functions. Note that the results presented in Table 8.3a were obtained using a more evolved version of HyPro to represent



a) First segments for `Fish` using LHA II reachability analysis methods ($\delta = 0.1$).

b) Excerpt from reachable set for the rectangular variant of `Fish`.

Figure 8.11: Plots of the computed set of reachable states for the first location in `Fish` using different analysis methods.

support functions (see Section 6.4) which had a considerable impact on the running times on common instances (`2Tnk`) as well.

The observed behavior when using boxes together with subspace decomposition differs significantly with respect to running times in comparison to the case where support functions were used. In general, boxes are among the fastest state set representations available such that the overhead introduced by decomposition as well as the overhead caused by instantiating multiple handlers computing flowpipes instead of one single handler is noticeable and explains the increased running times.

The analysis results obtained for the rectangular version of `Fish` give longer running times for the analysis than when using boxes as a state set representation for the non-rectangular version. However, the state sets can be computed precisely when using the analysis method dedicated to rectangular automata, which is not possible when using LHA II analysis methods in the non-rectangular version (see Figure 8.11).

The model of `5var_system` could be verified for one jump only; when increasing the jump limit, the memory limit was exceeded. This can be explained by the repeated application of Fourier-Motzkin variable elimination (see Section 2.2), which in the worst case introduces quadratically many new constraints when eliminating a variable. We expect that adding heuristics to reduce the number of redundant constraints during the elimination along with redundancy removal after the variable elimination may improve running times and memory consumption.

8.5 Future Work

In this section, we will present some ideas on future work and potential extensions of the method presented.

Strong Decomposition. In its current state, a subspace decomposition $\mathcal{D}$ using syntactic independence is computed based on the whole model of a hybrid system. Consequently, if two variables v_i and v_j are syntactically dependent in one location $\ell \in \mathit{Loc}$, they will be assigned to the same subspace in $\mathcal{D}$.

Using a local decomposition on a per-location basis which considers the local flow, as well as invariant conditions and guards on outgoing jumps, could result in a more versatile decomposition using stronger, local criteria for variable independence. Consequently, if the decomposition $\mathcal{D} = V_0, \ldots, V_{n-1}$ in some location ℓ is different from the computed decomposition $\mathcal{D}' = V'_0, \ldots, V_{m-1}$ for a different location ℓ', additional effort is required when taking a discrete jump $e \in \mathit{Edge}$ from ℓ to ℓ'. The projective representation for each subspace S'_i induced by the variable sets V'_i in decomposition $\mathcal{D}'$ needs to be recomputed from the discrete successor states according to the current decomposition $\mathcal{D}$. When the decomposition changes, we can build the Cartesian product of the subspace representations and project this set from the global state space to the new subspaces in the next decomposition.

Decomposed Workers. In the current setup, each context maintains a collection of workers that process a task, i.e., compute time- and discrete successor states simultaneously in all induced subspaces. Following the relation between the variable set decomposition and the parallel composition of hybrid automata, we can also analyze different subspaces independently from each other (similar to ideas presented in Section 5.8) in different tasks. As a result, workers may be decomposed and even operate in parallel as presented in Section 7.5. This requires a more involved task handling framework. Tasks may now depend on each other, for instance if the guard condition for some discrete transition $e = (\ell, g, r, \ell')$ contains constraints g which need to be handled in more than one subspace, this dependency between the subspaces needs to be taken to the task-level. Consequently, potential successor tasks resulting from taking e can only be created if all subspaces which are relevant for validating g and applying r agree on taking e during the same period of (execution) time (see Figure 8.7). Similarly, tasks need to synchronize on invariants—to avoid too much overhead it might be advisable to compute as many time successor states as possible per subspace individually in one location and afterward synchronize on the duration the individual invariant constraints were satisfied in the subspaces. This is analogous to the method presented, only the analysis of different subspaces can be done in several worker-threads in parallel.

Decomposition Criteria. In this work, we focus on variable set decomposition based on syntactic independence. Recently other approaches with other decomposition criteria have been presented [BFF+18]. In their work, the authors ignore the syntactic dependence of variables and compute a strict

2D-decomposition of a given system using suitable over-approximations of state sets for dependent variables in different subspaces.

A different way of decomposing a hybrid automaton which is not based on syntactic independence (see Definition 8.1) is by using an eigenvalue decomposition of the matrix $\mathcal{A}$ defining the flow of a location ℓ. For details on eigenvalue decompositions, we point the interested reader to [HS74] which provides a very intuitive introduction and will now limit ourselves to sketch the idea of this approach.

For a given flow matrix $\mathcal{A} \in \mathbb{R}^{n \times n}$ describing the dynamics $\dot{x} = \mathcal{A}x$, the eigenvalue decomposition of $\mathcal{A}$ into three matrices $\mathcal{A} = \mathcal{Q}\Lambda\mathcal{Q}^{-1}$ allows to decompose the system of n linear ODEs into n independent linear differential equations, i.e., the resulting system is *decoupled* after decomposition. This has the advantage that the matrix Λ, which is used to describe the flow has a diagonal form. The solution for each of the linear differential equations is a univariate exponential function for which the approximation error by a convex set can be determined more easily. Our preliminary work [Haf18] indicates the general applicability but also shows that there are open problems that need to be addressed before we can develop an automated method.

To be able to use eigenvalue decomposition, the matrix $\mathcal{A}$ describing the dynamics needs to be *diagonalizable*. If this is not the case, small perturbations applied to $\mathcal{A}$ could be used to obtain a diagonalizable matrix. However, a robustness analysis is required afterward to be able to make statements about the original dynamics. Furthermore, since the decomposition depends on $\mathcal{A}$, similar to the proposal of a *strong decomposition* (see above), a decomposition is only local for each location and needs to be computed for each location.

Decomposition in Refinement Strategies. Currently, the implementation provides decomposition separated from the other contributions. A promising idea is to embed this decomposition approach into the partial path refinement method as a special analysis method in parameter configurations. This way, the speed-up could be well exploited and in case the over-approximation that it introduces hinders verification, the next refinement level could switch off decomposition. It would even be possible a partial path refinement method inside subspaces. Additionally, parallelization could also be used for the refinement part, additionally to parallel computations in the subspaces.

9

Conclusion

This work presents results on hybrid systems safety verification using flowpipe-construction-based reachability analysis obtained during the past years. Starting from fundamental concepts such as how to represent sets up to higher-level extensions of state of the art reachability analysis methods we have provided several contributions to the research community.

Various state set representations have been considered in flowpipe-construction-based reachability analysis over the past years ranging from boxes over convex polytopes to support functions. Our contribution, the C++-library HyPro collects many of those commonly used representations under a unified interface and simplifies the usage in flowpipe-construction-based reachability analysis for tool developers. While our collection is not complete, experimental results have indicated that HyPro already provides a basis for answering one of our initial questions: "How can we efficiently represent state sets?".

Following this development, we were able to successfully exploit the diversity of HyPro by presenting a generalized CEGAR-based approach for partial path refinement during the analysis and a parallelization thereof. The obtained contributions may provide one answer to the questions on how to incorporate different state set representations in a scalable flowpipe-construction-based reachability analysis method. Not only does this approach increase the effectiveness of reachability analysis approaches, but it also paves the way towards more user-friendly tools that do not require expert knowledge for their application.

A different path of development starting from HyPro was implemented by our approach towards state space decomposition based on syntactic independence and the automation thereof. Motivated by industrial applications, we have shown that the problem of hybrid systems safety verification in real-world application has many layers to consider, ranging from efficient analysis methods towards the effective usage of domain-specific knowledge.

We are aware that the journey is not over yet—in this work we have presented our ideas and approaches which are all publicly available in HyPro and hope that this work may serve as a stepping stone for other researchers and promising future development. We have already given detailed information on future ideas to improve certain aspects of our methods in the respective chapters and will confine ourselves to provide a more general prospect here.

Apart from safety, also criteria such as robustness and stability of a system are of interest for developers in industry and academia. Methods from control engineering in which purely continuous (dynamic) systems are analyzed have been used for many years but less prominently used for hybrid systems as well. In our option both communities can profit from a vivid exchange of approaches which is why this can be a promising direction for future development.

Another interesting direction aims at improving usability and the general applicability of the developed methods to make research contributions available for industry as well. Work in this direction may improve rapid prototyping techniques by supporting further programming languages, or to make approaches available to the broad public by moving closer towards push-button approaches.

Recent advances in the analysis and verification of probabilistic systems have put this area of research more into the spotlight. Extensions towards probabilistic hybrid systems and the analysis thereof are currently under investigation and development in this area may bridge the gap between the two areas leading to further interesting development.

Bibliography

[ABC05] Teodoro Alamo, José M Bravo, and Eduardo F. Camacho. "Guaranteed State Estimation by Zonotopes". In: *Automatica* 41.6 (2005), pp. 1035–1043. DOI: 10.1016/j.automatica.2004.12.008.

[AD98] Hassane Alla and René David. "Continuous and Hybrid Petri Nets". In: *Journal of Circuits, Systems, and Computers* 8.01 (1998), pp. 159–188. DOI: 10.1142/S0218126698000079.

[Alt15] Matthias Althoff. "An Introduction to Cora 2015". In: *Proc. of ARCH'14-15*. Vol. 34. EPiC Series in Computing. EasyChair, 2015, pp. 120–151. DOI: 10.29007/zbkv.

[ABC+17] Matthias Althoff, Stanley Bak, Dario Cattaruzza, Xin Chen, Goran Frehse, Rajarshi Ray, and Stefan Schupp. "ARCH-COMP17 Category Report: Continuous and Hybrid Systems with Linear Continuous Dynamics". In: *Proc. of ARCH'17*. Vol. 48. EPiC Series in Computing. EasyChair, 2017, pp. 143–159. DOI: 10.29007/4dcn.

[ABC+18] Matthias Althoff, Stanley Bak, Xin Chen, Chuchu Fan, Marcelo Forets, Goran Frehse, Niklas Kochdumper, Yangge Li, Sayan Mitra, Rajarshi Ray, Christian Schilling, and Stefan Schupp. "ARCH-COMP18 Category Report: Continuous and Hybrid Systems with Linear Continuous Dynamics". In: *Proc. of ARCH'18*. Vol. 54. EPiC Series in Computing. EasyChair, 2018, pp. 23–52. DOI: 10.29007/73mb.

[ABF+19] Matthias Althoff, Stanley Bak, Marcelo Forets, Goran Frehse, Niklas Kochdumper, Rajarshi Ray, Christian Schilling, and Stefan Schupp. "ARCH-COMP19 Category Report: Continuous and Hybrid Systems with Linear Continuous Dynamics". In: *Proc. of ARCH'19*. Vol. 61. EPiC Series in Computing. EasyChair, 2019, pp. 14–40. DOI: 10.29007/bj1w.

[AD14] Matthias Althoff and John M. Dolan. "Online Verification of Automated Road Vehicles Using Reachability Analysis". In: *IEEE Transactions on Robotics* 30.4 (2014), pp. 903–918. DOI: 10.1109/TRO.2014.2312453.

[AF14] Matthias Althoff and Goran Frehse. *Benchmarks of the Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH)*. 2014.

[ASB10] Matthias Althoff, Olaf Stursberg, and Martin Buss. "Computing Reachable Sets of Hybrid Systems Using a Combination of Zonotopes and Polytopes". In: *Nonlinear Analysis: Hybrid Systems* 4.2 (2010), pp. 233–249. DOI: 10.1016/j.nahs.2009.03.009.

[ADI03] Rajeev Alur, Thao Dang, and Franjo Ivani. "Counter-example Guided Predicate Abstraction of Hybrid Systems". In: *Proc. of TACAS'03*. Vol. 2619. LNCS. Springer, 2003, pp. 208–223. DOI: 10.1007/3-540-36577-X_15.

[AD94] Rajeev Alur and David L. Dill. "A Theory of Timed Automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183–235. DOI: 10.1016/0304-3975(94)90010-8.

[ACD90] Rajev Alur, Costas Courcoubetis, and David L. Dill. "Model-checking for Real-time Systems". In: *Proc. of LICS'90*. 1990, pp. 414–425. DOI: 10.1109/LICS.1990.113766.

[ACH+95] Rajev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Peihsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. "The Algorithmic Analysis of Hybrid Systems". In: *Theoretical Computer Science* 138.1 (1995), pp. 3–34. DOI: 10.1016/0304-3975(94)00202-t.

[AS05] Aaron D. Ames and Shankar Sastry. "Characterization of Zeno Behavior in Hybrid Systems Using Homological Methods". In: *Proc. of ACC'05*. IEEE Computer Society Press, 2005, pp. 1160–1165. DOI: 10.1109/acc.2005.1470118.

[Art91] Michael Artin. *Algebra*. Prentice Hall, 1991.

[AF92] David Avis and Komei Fukuda. "A Pivoting Algorithm for Convex Hulls and Vertex Enumeration of Arrangements and Polyhedra". In: *Discrete & Computational Geometry* 8.3 (1992), pp. 295–313. DOI: 10.1007/bf02293050.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21. DOI: 10.1016/j.scico.2007.08.001.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[BD17a] Stanley Bak and Parasara Sridhar Duggirala. "Direct Verification of Linear Systems with Over 10000 Dimensions". In: *Proc. of ARCH'17*. Vol. 48. EPiC Series in Computing. EasyChair, 2017, pp. 114–123. DOI: 10.29007/dwj1.

[BD17b] Stanley Bak and Parasara Sridhar Duggirala. "Hylaa: A Tool for Computing Simulation-equivalent Reachability for Linear Systems". In: *Proc. of HSCC'17*. ACM Press. 2017. DOI: 10.1145/3049797.3049808.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. "The Quickhull Algorithm for Convex Hulls". In: *ACM Transactions on Mathematical Software* 22.4 (1996), pp. 469–483. DOI: 10.1145/235815.235821.

[BFH+14] Dirk A. van Beek, W.J. Fokkink, Dennis Hendriks, A. Hofkamp, Jasen Markovski, Jasen van de Mortel-Fronczak, and Michel A. Reniers. "CIF 3: Model-based Engineering of Supervisory Controllers". In: *Proc. of TACAS'14*. Vol. 8413. LNCS. Springer, 2014, pp. 575–580. DOI: 10.1007/978-3-642-54862-8_48.

[BDK12] Ibtissem Ben Makhlouf, Hilal Diab, and Stefan Kowalewski. "Safety Verification of a Controlled Cooperative Platoon under Loss of Communication Using Zonotopes". In: *Proc. of ADHS'12*. IFAC-PapersOnLine, 2012, pp. 333–338. DOI: 10.3182/20120606-3-nl-3011.00057.

[BHK16] Ibtissem Ben Makhlouf, Norman Hansen, and Stefan Kowalewski. "Hyreach: A Reachability Tool for Linear Hybrid Systems Based on Support Functions". In: *Proc. of ARCH'16*. Vol. 43. EPiC Series in Computing. EasyChair, 2016, pp. 68–79. DOI: 10.29007/7ncn.

[BKG+09] Ibtissem Ben Makhlouf, Stefan Kowalewski, Martin Guillermo, Chávez Grunewald, and Dirk Abel. "Safety Assessment of Networked Vehicle Platoon Controllers– Practical Experiences with Available Tools". In: *Proc. of ADHS'09*. Elsevier, 2009, pp. 292–297. DOI: 10.3182/20090916-3-es-3003.00051.

[BY04] Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". In: *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*. Springer, 2004, pp. 87–124. DOI: 10.1007/978-3-540-27755-2_3.

[BHM+09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bir27] George David Birkhoff. *Dynamical Systems*. Vol. 9. American Mathematical Society, 1927.

[BDF+13] Sergiy Bogomolov, Alexandre Donzé, Goran Frehse, Radu Grosu, Taylor T. Johnson, Hamed Ladan, Andreas Podelski, and Martin Wehrle. "Abstraction-based Guided Search for Hybrid Systems". In: *Proc. of SPIN'13*. Vol. 7976. LNCS. Springer, 2013, pp. 117–134. DOI: 10.1007/978-3-642-39176-7_8.

[BDF+16] Sergiy Bogomolov, Alexandre Donzé, Goran Frehse, Radu Grosu, Taylor T. Johnson, Hamed Ladan, Andreas Podelski, and Martin Wehrle. "Guided Search for Hybrid Systems Based on Coarse-grained Space Abstractions". In: *International Journal on Software Tools for Technology Transfer* 18.4 (2016), pp. 449–467. DOI: 10.1007/s10009-015-0393-y.

[BFF+18] Sergiy Bogomolov, 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: *Proc. of HSCC'18*. ACM Press. 2018, pp. 41–50. DOI: 10.1145/3178126.3178128.

[BFG+17] Sergiy Bogomolov, Goran Frehse, Mirco Giacobbe, and Thomas A. Henzinger. "Counterexample-guided Refinement of Template Polyhedra". In: *Proc. of TACAS'17*. Vol. 10205. LNCS. Springer. 2017, pp. 589–606. DOI: 10.1007/978-3-662-54577-5_34.

[BFG+12] Sergiy Bogomolov, Goran Frehse, Radu Grosu, Hamed Ladan, Andreas Podelski, and Martin Wehrle. "A Box-based Distance between Regions for Guiding the Reachability Analysis of SpaceEx". In: *Proc. of CAV'12*. Vol. 7358. LNCS. Springer, 2012, pp. 479–494. DOI: 10.1007/978-3-642-31424-7_35.

[Bon16] Igor Bongartz. "Over-approximative Reduction of Polytopes in the Context of Hybrid Systems Reachability Analysis". Bachelor's thesis. RWTH Aachen University, 2016.

[BCD13] Olivier Bouissou, Alexandre Chapoutot, and Adel Djoudi. "Enclosing Temporal Evolution of Dynamical Systems Using Numerical Methods". In: *Proc. of NFM'13*. Vol. 7871. LNCS. Springer, 2013, pp. 108–123. DOI: 10.1007/978-3-642-38088-4_8.

[BMP99] Olivier Bournez, Oded Maler, and Amir Pnueli. "Orthogonal Polyhedra: Representation and Computation". In: *Proc. of HSCC'99*. Vol. 1569. LNCS. Springer. 1999, pp. 46–60. DOI: 10.1007/3-540-48983-5_8.

[BRS17] Lei Bu, Rajarshi Ray, and Stefan Schupp. "ARCH-COMP17 Category Report: Bounded Model Checking of Hybrid Systems with Piecewise Constant Dynamics." In: *Proc. of ARCH'17*. Vol. 48. EPiC Series in Computing. EasyChair, 2017, pp. 134–142.

[BRS18] Lei Bu, Rajarshi Ray, and Stefan Schupp. "ARCH-COMP18 Category Report: Bounded Model Checking of Hybrid Systems with Piecewise Constant Dynamics". In: *Proc. of ARCH'18*. Vol. 54. EPiC Series in Computing. EasyChair, 2018, pp. 14–22. DOI: 10.29007/q5tq.

[BL06] Manuela Bujorianu and John Lygeros. "Toward a General Theory of Stochastic Hybrid Systems". In: *Stochastic Hybrid Systems*. Vol. 337. LNCS. Springer, 2006, pp. 3–30. DOI: 10.1007/11587392_1.

[Che15] Xin Chen. "Reachability Analysis of Non-linear Hybrid Systems Using Taylor Models". PhD thesis. RWTH Aachen University, Germany, 2015.

[CÁF11] Xin Chen, Erika Ábrahám, and Goran Frehse. "Efficient Bounded Reachability Computation for Rectangular Automata". In: *Proc. of RP'11*. Vol. 6945. LNCS. Springer, 2011, pp. 139–152. DOI: 10.1007/978-3-642-24288-5_13.

[CÁS12] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. "Taylor Model Flowpipe Construction for Non-linear Hybrid Systems". In: *Proc. of RTSS'12*. IEEE Computer Society Press, 2012, pp. 183–192. DOI: 10.1109/rtss.2012.70.

[CÁS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. "Flow*: An Analyzer for Non-linear Hybrid Systems". In: *Proc. of CAV'13*. Vol. 8044. LNCS. Springer, 2013, pp. 258–263. DOI: 10.1007/978-3-642-39799-8_18.

[CS16] Xin Chen and Sriram Sankaranarayanan. "Decomposed Reachability Analysis for Nonlinear Systems". In: *Proc. of RTSS'16*. IEEE Computer Society Press, 2016, pp. 13–24. DOI: 10.1109/RTSS.2016.011.

[CSB+15] Xin Chen, Stefan Schupp, Ibtissem Ben Makhlouf, Erika Ábrahám, Goran Frehse, and Stefan Kowalewski. "A Benchmark Suite for Hybrid Systems Reachability Analysis". In: *Proc. of NFM'15*. Vol. 9058. LNCS. Springer, 2015, pp. 408–414. DOI: 10.1007/978-3-319-17524-9_29.

[CFH+03] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. "Abstraction and Counterexample-guided Refinement in Model Checking of Hybrid Systems". In: *International Journal of Foundations of Computer Science* 14.04 (2003), pp. 583–604. DOI: 10.1142/S012905410300190X.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided Abstraction Refinement". In: *Proc. of CAV'00*. Vol. 1855. LNCS. Springer, 2000, pp. 154–169. DOI: 10.1007/10722167_15.

[CBG+12] Pieter Collins, Davide Bresolin, Luca Geretti, and Tiziano Villa. "Computing the Evolution of Hybrid Systems Using Rigorous Function Calculus". In: *Proc. of ADHS'12*. IFAC-PapersOnLine, 2012, pp. 284–290. DOI: 10.3182/20120606-3-nl-3011.00063.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. "SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving". In: *Proc. of SAT'15*. Vol. 9340. LNCS. Springer, 2015, pp. 360–368. DOI: 10.1007/978-3-319-24318-4_26.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints among Variables of a Program". In: *Proc. of SIGACT-SIGPLAN*. ACM Press, 1978, pp. 84–96. DOI: 10.1145/512760.512770.

[Dan00] Thi Xuan Thao Dang. "Vérification Et Synthese Des Systemes Hybrides". PhD thesis. institut National Polytechnique de Grenoble, France, 2000.

[Dan63] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

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

[Dil90] David L. Dill. "Timing Assumptions and Verification of Finite-state Concurrent Systems". In: *Proc. of CAV'89*. Vol. 407. LNCS. Springer, 1990, pp. 197–212. DOI: 10.1007/3-540-52148-8_17.

[DF13] Alexandre Donzé and Goran Frehse. "Modular, Hierarchical Models of Control Systems in SpaceEx". In: *Proc. of ECC'13*. IEEE Computer Society Press. 2013, pp. 4244–4251. DOI: 10.23919/ECC.2013.6669815.

[Egg14] Andreas Eggers. "Direct Handling of Ordinary Differential Equations in Constraint-solving-based Analysis of Hybrid Systems". PhD thesis. Universität Oldenburg, Germany, 2014.

[EFH08] Andreas Eggers, Martin Fränzle, and Christian Herde. "SAT Modulo ODE: A Direct SAT Approach to Hybrid Systems". In: *Proc. of ATVA'08*. Vol. 5311. LNCS. Springer. 2008, pp. 171–185. DOI: 10.1007/978-3-540-88387-6_14.

[FI04] Ansgar Fehnker and Franjo Ivancic. "Benchmarks for Hybrid Systems Verification". In: *Proc. of HSCC'04*. Vol. 2993. LNCS. Springer, 2004, pp. 326–341. DOI: 10.1007/978-3-540-24743-2_22.

[Fis91] Michael E. Fisher. "A Semiclosed-loop Algorithm for the Control of Blood Glucose Levels in Diabetics". In: *IEEE Transactions on Biomedical Engineering* 38.1 (1991), pp. 57–61. DOI: 10.1109/10.68209.

[Flo16] Phillip Florian. "Optimizing Reachability Analysis for Non-autonomous Hybrid Systems Using Ellipsoids". MA thesis. RWTH Aachen University, 2016.

[Fou27] Jean Babtiste Joseph Fourier. "Analyse Des Travaux De Lacadémie Royale Des Sciences Pendant Lannée 1824". In: *Partie Mathématique* (1827).

[Fre05] Goran Frehse. "PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech". In: *Proc. of HSCC'05*. Vol. 3414. LNCS. Springer, 2005, pp. 258–273. DOI: 10.1007/978-3-540-31954-2_17.

[FKL13] Goran Frehse, Rajat Kateja, and Colas Le Guernic. "Flowpipe Approximation and Clustering in Space-time". In: *Proc. of HSCC'13*. ACM Press, 2013, pp. 203–212. DOI: 10.1145/2461328.2461361.

[FLD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. "SpaceEx: Scalable Verification of Hybrid Systems". In: *Proc. of CAV'11*. Vol. 6806. LNCS. Springer, 2011, pp. 379–395. DOI: 10.1007/978-3-642-22110-1_30.

[FR09] Goran Frehse and Rajarshi Ray. "Design Principles for an Extendable Verification Tool for Hybrid Systems". In: *Proc. of ADHS'09*. IFAC-PapersOnLine, 2009, pp. 244–249. DOI: 10.3182/20090916-3-es-3003.00043.

[FR12] Goran Frehse and Rajarshi Ray. "Flowpipe-guard Intersection for Reachability Computations with Support Functions". In: *Proc. of ADHS'12*. Vol. 45. 9. 2012, pp. 94–101. DOI: https://doi.org/10.3182/20120606-3-NL-3011.00053.

[Fro16] Simon Froitzheim. "Efficient Conversion of Geometric State Set Representations for Hybrid Systems". Bachelor's thesis. RWTH Aachen University, 2016.

[Fuk04] Komei Fukuda. "From the Zonotope Construction to the Minkowski Addition of Convex Polytopes". In: *Journal of Symbolic Computation* 38.4 (2004), pp. 1261–1272. DOI: 10.1016/j.jsc.2003.08.007.

[Gao12] Sicun Gao. "Computable Analysis, Decision Procedures, and Hybrid Automata: A New Framework for the Formal Verification of Cyber-physical Systems". PhD thesis. Carnegie Mellon University, 2012.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M Clarke. "dReal: An SMT Solver for Nonlinear Theories Over the Reals". In: *Proc. of CADE-24*. Vol. 7898. LNCS. Springer, 2013, pp. 208–214. DOI: 10.1007/978-3-642-38574-2_14.

[GJ00] Ewgenij Gawrilow and Michael Joswig. "Polymake: A Framework for Analyzing Convex Polytopes". In: *Polytopes – combinatorics and computation*. Vol. 29. DMV Sem. Birkhäuser, Basel, 2000, pp. 43–73. DOI: 10.1007/978-3-0348-8438-9_2.

[Gir04] Antoine Girard. "Algorithmic Analysis of Hybrid Systems". PhD thesis. Institut National Polytechnique de Grenoble - INPG, France, 2004.

[Gir05] Antoine Girard. "Reachability of Uncertain Linear Systems Using Zonotopes". In: *Proc. of HSCC'05*. Vol. 3414. LNCS. Springer, 2005, pp. 291–305. DOI: 10.1007/978-3-540-31954-2_19.

[GL08] Antoine Girard and Colas Le Guernic. "Zonotope/hyperplane Intersection for Hybrid Systems Reachability Analysis". In: *Proc. of HSCC'08*. Vol. 4981. LNCS. Springer, 2008, pp. 215–228. DOI: 10.1007/978-3-540-78929-1_16.

[Gra72] Ronald L. Graham. "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set". In: *Information Processing Letters* 1.4 (1972), pp. 132–133. DOI: 10.1016/0020-0190(72)90045-2.

[GRB+18] Amit Gurung, Rajarshi Ray, Ezio Bartocci, Sergiy Bogomolov, and Radu Grosu. "Parallel Reachability Analysis of Hybrid Systems in XSpeed". In: *International Journal on Software Tools for Technology Transfer* (2018). DOI: 10.1007/s10009-018-0485-6.

[Haf18] Jan Philipp Hafer. "Using Eigenvalue Decomposition in Hybrid Systems Reachability Analysis". Bachelor's thesis. RWTH Aachen University, 2018.

[Hen96] Thomas A. Henzinger. "The Theory of Hybrid Automata". In: *Proc. of LICS'96*. IEEE Computer Society Press, 1996, pp. 278–292. DOI: 10.1007/978-3-642-59615-5_13.

[HKP+98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. "What's Decidable about Hybrid Automata?" In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 94–124. DOI: 10.1006/jcss.1998.1581.

[HM99] João P. Hespanha and A. Stephen Morse. "Stabilization of Nonholonomic Integrators Via Logic-based Switching". In: *Automatica* 35.3 (1999), pp. 385–393. DOI: 10.1016/s0005-1098(98)00166-6.

[HS74] Morris W Hirsch and Stephen Smale. *Differential Equations, Dynamical Systems, and Linear Algebra*. Academic, 1974.

[HSR+17] Jannik Hüls, Stefan Schupp, Anne Remke, and Erika Ábrahám. "Analyzing Hybrid Petri Nets with Multiple Stochastic Firings Using Hypro". In: *Proc. of VALUETOOLS'17*. ACM Press, 2017, pp. 178–185. DOI: 10.1145/3150928.3150938.

[Hüt16] Dustin Hütter. "Adaptive Dynamic Reachability Analysis for Linear Hybrid Automata". MA thesis. RWTH Aachen University, 2016.

[Imb90] Jean-Louis Imbert. "About Redundant Inequalities Generated by Fourier's Algorithm". In: *Artificial Intelligence IV*. Elsevier, 1990, pp. 117–127. DOI: 10.1016/B978-0-444-88771-9.50019-2.

[IAC+18] Fabian Immler, Matthias Althoff, Xin Chen, Chuchu Fan, Goran Frehse, Niklas Kochdumper, Yangge Li, Sayan Mitra, Mahendra Singh Tomar, and Majid Zamani. "ARCH-COMP18 Category Report: Continuous and Hybrid Systems with Nonlinear Dynamics". In: *Proc. of ARCH'18*. Vol. 54. EPiC Series in Computing. EasyChair, 2018, pp. 53–70. DOI: 10.29007/mskf.

[IUH11] Daisuke Ishii, Kazunori Ueda, and Hiroshi Hosobe. "An Interval-based SAT Modulo ODE Solver for Model Checking Nonlinear Hybrid Systems". In: *International Journal on Software Tools for Technology Transfer* 13.5 (2011), pp. 449–461. DOI: 0.1007/s10009-011-0193-y.

[Izh07] Eugene M. Izhikevich. *Dynamical Systems in Neuroscience*. MIT Press, 2007. DOI: 10.7551/mitpress/2526.001.0001.

[Jar73] Ray A. Jarvis. "On the Identification of the Convex Hull of a Finite Set of Points in the Plane". In: *Information Processing Letters* 2.1 (1973), pp. 18–21. DOI: 10.1016/0020-0190(73)90020-3.

[Kie18] Sabrina Kielmann. "Comparing the Expressivity and Usabiliy of Hybrid Systems' Modeling Languages". MA thesis. RWTH Aachen University, 2018.

[Kug14] Christopher Kugler. "A Polytope Library for the Reachability Analysis of Hybrid Systems". MA thesis. RWTH Aachen University, 2014.

[KV00] Alexander B. Kurzhanski and Pravin Varaiya. "Ellipsoidal Techniques for Reachability Analysis". In: *Proc. of HSCC'00*. Vol. 1790. LNCS. Springer, 2000, pp. 202–214. DOI: 10.1007/3-540-46430-1_19.

[KV07] Alexander B. Kurzhanski and Pravin Varaiya. "Ellipsoidal Techniques for Reachability Analysis of Discrete-time Linear Systems". In: *IEEE Transactions on Automatic Control* 52.1 (2007), pp. 26–38. DOI: 10.1109/TAC.2006.887900.

[Lam87] Leslie Lamport. "A Fast Mutual Exclusion Algorithm". In: *ACM Transactions on Computer Systems* 5.1 (1987), pp. 1–11.

[Le 09] Colas Le Guernic. "Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics". PhD thesis. Université Joseph-Fourier-Grenoble I, France, 2009.

[LG10] Colas Le Guernic and Antoine Girard. "Reachability Analysis of Linear Systems Using Support Functions". In: *Nonlinear Analysis: Hybrid Systems* 4.2 (2010), pp. 250–262. DOI: 10.1016/j.nahs.2009.03.002.

[LSA+19] Francesco Leofante, Stefan Schupp, Erika Abraham, and Armando Tacchella. "Engineeering Controllers for Swarm Robotics Via Reachability Analysis in Hybrid Systems". In: *Proc. of ECMS'19*. to appear. 2019.

[LWS+11] Jó Ágila Bitsch Link, Christoph Wollgarten, Stefan Schupp, and Klaus Wehrle. "Perfect Difference Sets for Neighbor Discovery: Energy Efficient and Fair". In: *Proc. of ExtremeCom'11*. ACM Press, 2011, 5:1–5:6. DOI: 10.1145/2414393.2414398.

[MFK09] Hitashyam Maka, Goran Frehse, and Bruce H. Krogh. "Polyhedral Domains and Widening for Verification of Numerical Programs". In: *Proc. of NSV-II*. 2009.

[Mak18] Andrew Makhorin. *GNU Linear Programming Kit Home Page*. 2018.

[MB09] Kyoko Makino and Martin Berz. "Rigorous Integration of Flows and ODEs Using Taylor Models". In: *Proc. of SNC'09*. ACM Press, 2009, pp. 79–84. DOI: 10.1145/1577190.1577206.

[MF14] Stefano Minopoli and Goran Frehse. "Non-convex Invariants and Urgency Conditions on Linear Hybrid Automata". In: *Proc. of FORMATS'14*. Vol. 8711. LNCS. Springer, 2014, pp. 176–190. DOI: 10.1007/978-3-319-10512-3_13.

[Mon09] David Monniaux. "On Using Floating-point Computations to Help an Exact Linear Arithmetic Decision Procedure". In: *Proc. of CAV'09*. Vol. 5643. LNCS. Springer, 2009, pp. 570–583. DOI: 10.1007/978-3-642-02658-4_42.

[MKC09] Ramon E. Moore, Ralph Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009. DOI: 10.1137/1.9780898717716.

[Mot36] Theodore Samuel Motzkin. *Beitrage Zur Theorie Der Linearen Ungleichungen*. Azriel, 1936.

[MB08] Leonardo M. de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. of TACAS'08*. Vol. 4963. LNCS. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[Nel16] Johanna Nellen. "Analysis and Synthesis of Hybrid Systems in Engineering Applications". PhD thesis. RWTH Aachen University, 2016, pp. 1–168.

[NÁC+13] Johanna Nellen, Erika Ábrahám, Xin Chen, and Pieter Collins. "Counterexample Generation for Hybrid Automata". In: *Proc. of FTSCS'13*. Vol. 419. Communications in Computer and Information Science. Springer, 2013, pp. 88–106. DOI: 10.1007/978-3-319-05416-2_7.

[Neu16] Johannes Neuhaus. "Development of a Modular Approach for Hybrid Systems Reachability Analysis". MA thesis. RWTH Aachen University, 2016.

[NOS+92] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. "An Approach to the Description and Analysis of Hybrid Systems". In: *Proc. of HS'93*. Vol. 736. LNCS. Springer, 1992, pp. 149–178. DOI: 10.1007/3-540-57318-6_28.

[Pea01] Karl Pearson. "LIII. on Lines and Planes of Closest Fit to Systems of Points in Space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572.

[Pla08] André Platzer. "Differential Dynamic Logic for Hybrid Systems". In: *Journal of Automated Reasoning* 41.2 (2008), pp. 143–189. DOI: 10.1007/s10817-008-9103-8.

[PQ08] André Platzer and Jan-David Quesel. "A Hybrid Theorem Prover for Hybrid Systems (system Description)". In: *Proc. of IJCAR'08*. Vol. 5195. LNCS. Springer, 2008, pp. 171–178. DOI: 10.1007/978-3-540-71070-7_15.

[Poi92] Henry Poincaré. *Les Méthodes Nouvelles De La Méchanique Céleste*. 3 vols. Gauthier-Villars, 1892.

[RMC09] Nacim Ramdani, Nacim Meslem, and Yves Candau. "A Hybrid Bounding Method for Computing an Over-approximation for the Reachable Set of Uncertain Nonlinear Systems". In: *IEEE Transactions on Automated Control* 54.10 (2009), pp. 2352–2364. DOI: 10.1109/TAC.2009.2028974.

[RS05] Stefan Ratschan and Zhikun She. "Safety Verification of Hybrid Systems by Constraint Propagation-based Abstraction Refinement". In: *Proc. of HSCC'05*. Vol. 3414. LNCS. Springer, 2005, pp. 573–589. DOI: 10.1007/978-3-540-31954-2_37.

[Rat96] Dietmar Ratz. "On Extended Interval Arithmetic and Inclusion Isotonicity". In: *Submitted for Publication in Siam Journal on Numerical Analysis* (1996).

[RW03] Michal Rewienski and Jacob White. "A Trajectory Piecewise-linear Approach to Model Order Reduction and Fast Simulation of Nonlinear Circuits and Micromachined Devices". In: *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 22.2 (2003), pp. 155–170. DOI: 10.1109/tcad.2002.806601.

[SDI08] Sriram Sankaranarayanan, Thao Dang, and Franjo Ivani. "Symbolic Model Checking of Hybrid Systems Using Template Polyhedra". In: *Proc. of TACAS'08*. Vol. 4963. LNCS. Springer, 2008, pp. 188–202. DOI: 10.1007/978-3-540-78800-3_14.

[Sch13] Stefan Schupp. "Interval Constraint Propagation in SMT-compliant Decision Procedures". MA thesis. RWTH Aachen University, 2013.

[SÁ18a] Stefan Schupp and Erika Ábrahám. "Efficient Dynamic Error Reduction for Hybrid Systems Reachability Analysis". In: *Proc. of TACAS'18*. Vol. 10806. LNCS. Springer, 2018, pp. 287–302. DOI: 10.1007/978-3-319-89963-3_17.

[SÁ18b] Stefan Schupp and Erika Ábrahám. "Spread the Work: Multithreaded Safety Analysis for Hybrid Systems". In: *Proc. of SEFM'18*. Vol. 10886. LNCS. Springer, 2018, pp. 89–104. DOI: 10.1007/978-3-319-92970-5_6.

[SÁ18c] Stefan Schupp and Erika Ábrahám. "The HyDRA Tool – a Playground for the Development of Hybrid Systems Reachability Analysis Methods". In: *Proc. of the PhD Symposium at iFM18 (PhD-iFM18)*. University of Oslo, 2018, pp. 22–23.

[SÁ19] Stefan Schupp and Erika Ábrahám. "Context-dependent Reachability Analysis for Hybrid Systems". In: *Reuse in Intelligent Systems* (2019). to appear.

[SÁB+17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. "HyPro: A C++ Library for State Set Representations for Hybrid Systems Reachability Analysis". In: *Proc. of NFM'17*. Vol. 10227. LNCS. Springer, 2017, pp. 288–294. DOI: 10.1007/978-3-319-57288-8_20.

[SÁC+15] Stefan Schupp, Erika Ábrahám, Xin Chen, Ibtissem Ben Makhlouf, Goran Frehse, Sriram Sankaranarayanan, and Stefan Kowalewski. "Current Challenges in the Verification of Hybrid Systems". In: *Proc. of CyPhy'15*. Vol. 9361. Information Systems and Applications, incl. Internet/Web, and HCI. Springer, 2015, pp. 8–24. DOI: 10.1007/978-3-319-25141-7_2.

[SLÁ+18] Stefan Schupp, Francesco Leofante, Erika Ábrahám, and Armando Tacchella. "Robot Swarms as Hybrid Systems". In: *Proc. of SNR'18*. to appear. EPTCS, 2018.

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

[SWÁ18] Stefan Schupp, Justin Winkens, and Erika Ábrahám. "Context-dependent Reachability Analysis for Hybrid Systems". In: *Proc. of FMI'18*. IEEE Computer Society Press, 2018, pp. 518–525. DOI: 10.1109/IRI.2018.00082.

[SL13] Zhucheng Shao and Jing Liu. "Spatio-temporal Hybrid Automata for Cyber-physical Systems". In: *Proc. of ICTAC'13*. Vol. 8049. LNCS. Springer, 2013, pp. 337–354. DOI: 10.1007/978-3-642-39718-9_20.

[Spr00] Jeremy Sproston. "Decidable Model Checking of Probabilistic Hybrid Automata". In: *Proc. of FTRTFT'00*. Vol. 1926. LNCS. Springer, 2000, pp. 31–45. DOI: 10.1007/3-540-45352-0_5.

[SK03] Olaf Stursberg and Bruce H. Krogh. "Efficient Representation and Computation of Reachable Sets for Hybrid Systems". In: *Proc. of HSCC'03*. Springer. 2003, pp. 482–497. DOI: 10.1007/3-540-36580-x_35.

[TNJ16] Hoang-Dung Tran, Luan Viet Nguyen, and Taylor T Johnson. "Large-scale Linear Systems from Order-reduction (benchmark Proposal)". In: *Proc. of ARCH'16*. Vol. 43. EPiC Series in Computing. EasyChair, 2016.

[VMK97] Pascal Van Hentenryck, David McAllester, and Deepak Kapur. "Solving Polynomial Systems Using a Branch and Prune Approach". In: *Siam Journal on Numerical Analysis* 34.2 (1997), pp. 797–827. DOI: 10.1137/s0036142995281504.

[WF05] Christophe Weibel and Komei Fukuda. "Computing Faces up to K Dimensions of a Minkowski Sum of Polytopes". In: *Proc. of CCCG'05*. 2005, pp. 256–259.

[WKm19] Thomas Williams, Colin Kelley, and many others. *Gnuplot 5.2: An Interactive Plotting Program*. http://gnuplot.sourceforge.net/. 2019.

[Win18] Justin Winkens. "Context-dependent Reachability Analysis for Hybrid Automata". MA thesis. RWTH Aachen University, 2018.

[Wun96] Roland Wunderling. “Paralleler Und Objektorientierter Simplex-algorithmus”. PhD thesis. Technische Universität Berlin, 1996.

[ZSR+10] Lijun Zhang, Zhikun She, Stefan Ratschan, Holger Hermanns, and Ernst Moritz Hahn. “Safety Verification for Probabilistic Hybrid Systems”. In: *Proc. of CAV'10*. Vol. 6174. LNCS. Springer, 2010, pp. 196–211. DOI: 10.1007/978-3-642-14295-6_21.

[Zie95] Günter M. Ziegler. *Lectures on Polytopes*. Vol. 152. Graduate Texts in Mathematics. Springer, 1995. DOI: 10.1007/978-1-4613-8431-1.

Index