Sergiy Bogomolov

Abstraction-based Analysis of Hybrid Automata

Dissertation

zur Erlangung des Doktorgrades
der Technischen Fakultät
der Albert-Ludwigs-Universität Freiburg

Tag der Disputation:
2. April 2015

Dekan:
Prof. Dr. Georg Lausen, *Albert-Ludwigs-Universität Freiburg*

Referenten:
Prof. Dr. Andreas Podelski, *Albert-Ludwigs-Universität Freiburg*
Prof. Dr. Radu Grosu, *Technische Universität Wien*
Prof. Dr. Christoph Scholl, *Albert-Ludwigs-Universität Freiburg*
Prof. Dr. Bernhard Nebel, *Albert-Ludwigs-Universität Freiburg*

Abstract

Hybrid automata are used to model systems with both discrete and continuous dynamics. A prototypical example is a thermostat which switches between the modes "heating" and "cooling". We present new methods to construct and utilize abstractions for the analysis of hybrid automata. For guiding the exploration of the region space of a hybrid automaton, we propose distance functions which are based on abstractions of the original hybrid automaton and can be computed automatically. To analyze networks of hybrid automata, we lift the notion of assume-guarantee abstraction refinement to the class of hybrid automata. For this purpose, we introduce an abstraction of a hybrid automaton which can efficiently shrink the discrete structure. In order to apply symbolic reachability analysis to hybrid planning based on the planning language PDDL+ as studied in Artificial Intelligence, we present a semantics preserving translation of PDDL+ into a hybrid automaton. We have implemented our methods as extensions to the symbolic hybrid model checker SpaceEx. As the experiments show, our methods make it possible to analyze hybrid automata which were out of scope of existing methods.

Zusammenfassung

Hybrid-Automaten werden dazu verwendet, Systeme zu modellieren, die sowohl diskretes als auch kontinuerliches Verhalten aufweisen. Ein typisches Beispiel für ein solches System ist ein Thermostat, der zwischen den Modi “Heizen” and “Kühlen” wechselt. In dieser Arbeit stellen wir abstraktionsbasierte Verfahren zur symbolischen Analyse von Hybrid-Automaten vor. Um das Explorieren des Zustandsraumes eines Hybrid-Automaten zu steuern, schlagen wir Abstandsfunktionen vor, die auf Abstraktionen des ursprünglichen Hybrid-Automaten beruhen und automatisch berechnet werden können. Um Netzwerke von Hybrid-Automaten zu analysieren, erweitern wir das Konzept von “Assume-Guarantee Abstraction Refinement” auf die Klasse der Hybrid-Automaten. Zu diesem Zweck führen wir eine Abstraktion ein, die die diskrete Struktur des Automaten effizient verkleinern kann. Weiterhin präsentieren wir eine semantik-erhaltende Übersetzung der Planungsspache PDDL+, die im Bereich der künstlichen Intelligenz untersucht wird, in einen Hybrid-Automaten, um symbolische Erreichbarkeitsanalyse für Handlungsplanung in hybriden Domänen anwenden zu können. Wir haben unsere Verfahren als Erweiterungen des symbolischen hybriden Modellprüfers SpaceEx implementiert. Unsere Experimente zeigen, dass unsere Verfahren es ermöglichen, Hybrid-Automaten zu analysieren, die mit existierenden Verfahren bisher nicht analysiert werden konnten.

Acknowledgments

First and foremost, I would like to thank my supervisor Andreas Podelski for guiding my research during all the years of my PhD studies. This thesis would not have been possible without his support and encouragement. Andreas taught me how to combine bold creativity and constructive criticism in order to pose and tackle exciting scientific problems. His immense influence on my style of research and generally on my attitude to life will last for many years to come.

I thank Radu Grosu for a very productive collaboration over many years. Radu's vision of the area and its future developments has shaped my research interests a lot and led to many fascinating projects.

I am grateful to Goran Frehse for introducing me to the world of the SpaceEx model checker, insightful discussions and answering my numerous questions. I owe a large part of my technical knowledge in the area of hybrid automata to Goran.

I am thankful to Martin Wehrle for a fruitful collaboration in the area of heuristics and planning which already resulted in many papers and hopefully will lead to even more papers in the future. Thanks to Martin I learnt (or at least hope so) how to write briefly and succinctly. His friendly and professional advice was invaluable at many different stages of my research.

I wish to thank Marius Greitschus for working together on reachability analysis of hybrid automata. It was not easy to meet conference deadlines sometimes; however, it is very nice to remember our joint pizzas and hamburgers in the days before deadlines. Marius's deep knowledge of software development was always instrumental in our projects.

I also wish to express my gratitude to Christian Schilling for our regular discussions on hybrid automata which were always helpful and enjoyable.

I would like to thank my office colleague Corina Mitrohin for supporting my first steps in the research of hybrid automata. I really enjoyed our dis-

cussions on all possible topics and can only wish everybody to have such an office colleague.

Also, my thanks go to all the members of the Chair of Software Engineering at the University of Freiburg for a friendly and creative work environment.

Finally, I want to thank my family for their support. In particular, I am grateful to my wife Eugenia and daughter Sophia for their patience and love. I thank my parents Olga and Nikolai for all their help, belief in me and for always being there whenever I needed them. I dedicate this thesis to my family.

Contents

1

Introduction

1.1 Abstractions for Hybrid Automata

Nowadays, our life cannot be imagined without computers. In many areas, e.g., in the automobile industry, incorrect behavior of a computer-assisted system can cost human lives. Currently, engineers mostly rely on *testing* paradigms in order to ensure that the system fulfills its intended purpose. Unfortunately, the fact that no bugs have been found in a testing phase does not guarantee their absence as the testing results rely only on a *finite* number of tests. This situation is unsatisfactory and motivated the development of a new research discipline called *"Model Checking"* [26]. The aim is to provide *automatic* techniques for checking system correctness: Given a model $\mathcal{M}$ and a property φ, a model checking algorithm checks whether the relation $\mathcal{M} \models \varphi$ holds, i.e., the model $\mathcal{M}$ *satisfies* the property φ. The distinguishing feature of model checking is the fact that it can provide a mathematical *proof* of the property satisfaction. However, it is usually necessary to *exhaustively* explore the system state space for this purpose. The state space grows exponentially in the size of the system which can make model checking of large systems computationally intractable. Therefore, big efforts have been devoted [25, 12, 39] to ensuring the *scalability* of model checking approaches. In this thesis, we investigate model checking algorithms for models $\mathcal{M}$ which belong to the class of hybrid automata. A hybrid automaton [1] is a mathematical model which unifies the notions of a non-deterministic finite automaton (NFA) [44] and continuous dynamical system [65]. More formally, every location of an NFA is augmented with a system of differential equations which governs the state evolution of the hybrid automaton in this particular location. On the one hand, this leads to a large expressive power and thus makes hybrid automata an appropriate modelling framework for a large range of application domains such as embedded systems [6] and

systems biology [13]. On the other hand, the resulting complex system behavior requires careful handling to ensure that the analysis algorithms are still computationally amenable.

Symbolic model checking techniques enable the analysis of infinite state space systems by utilizing finite representations of sets of states in form of constraints, binary decision diagrams, etc. We investigate symbolic model checking techniques for hybrid automata. For this class of systems, sets of states are represented by *regions*. In our research, we focus on methods for *symbolic reachability analysis*, a subclass of symbolic model checking techniques. In this setting, reachability of some given *bad* states is equivalent to system malfunction.

We scale the reachability analysis by making a number of technical contributions. They have in common a core ingredient: a model *abstraction*. In other words, we consider a *simplified* version of the original system. Here, the main idea is to *automatically* compute such an abstraction which provides enough information to reason about the given property φ, yet can be analyzed in an efficient way. Therefore, an important research question is to find a *trade-off* between the abstraction precision and the required analysis time.

In the following, we give a short overview of the thesis contributions:

1. **Guided Search for Hybrid Automata.** A hybrid model checker like SpaceEx [39] is typically optimized towards proving the absence of errors. In some settings, e.g., when the verification tool is employed in a feedback-directed design cycle, it is desirable to have an option to call a version that is optimized towards finding an error trajectory in the region space. A possible way to reach this goal is to employ *guided search* [34]. Guided search relies on a cost function that indicates which states are promising to be explored, and in the first instance explores more promising states first.

   We propose two abstraction-based cost functions [20, 18, 17]. The first one works by approximating symbolic regions and measuring their distance to the bad states. Here, we mainly take the continuous part of the state into account. The second one is based on *coarse-grained space abstractions* for guiding the reachability analysis. For this purpose, a suitable abstraction technique that exploits the flexible granularity of modern reachability analysis algorithms is introduced. The new cost function is an effective extension of the pattern database approach [28] which was originally developed in the scope of Artificial Intelligence.
2. **Assume Guarantee Abstraction Refinement for Hybrid Automata.** Compositional verification techniques in the assume-guarantee

style have been successfully applied to transition systems [57] to efficiently reduce the search space by leveraging the compositional nature of the systems under consideration. We adapt these techniques to the domain of hybrid automata with affine dynamics [19]. To build assumptions we introduce an *abstraction based on location merging*. We integrate the assume-guarantee style analysis with automatic abstraction refinement.

3. **Hybrid Planning.** Planning is an area of artificial intelligence which studies the problem of finding a sequence of actions leading to a goal state. Hybrid planning considers planning problems with actions which exhibit continuous dynamics. Planning in hybrid domains is an important and challenging task, and various planning algorithms [27, 66, 31] have been proposed in the last years. From an abstract point of view, hybrid planning domains are based on hybrid automata. However, despite the quest for more scalable planning approaches, model checking algorithms have not been applied to planning in hybrid domains so far. We note that planning represents an instantiation of a *falsification* problem in a specific problem domain. In particular, in the planning setting, we are interested in reaching a *goal* state, whereas in the falsification setting we are interested in detecting the paths towards a *bad* state. Still, the task of exploring a system state space in an efficient manner towards a given set of states stays the same.
   We make a first step in bridging the gap between these two worlds. We provide a formal translation scheme from PDDL+, a formalism to describe planning domains, to the standard formalism of hybrid automata [21], as a solid basis for using hybrid automata model-checking tools for dealing with hybrid planning domains. As a case study, we use the SpaceEx model checker, showing how we can address PDDL+ domains that are out of the scope of state-of-the-art planners.

1.2 Proof of Concept

All the techniques presented in this thesis have been implemented as extensions of the hybrid model checker SpaceEx. The standard version of SpaceEx can verify the safety of a given hybrid automaton. The analysis is based on the symbolical representation of reachable regions. In order to evaluate our approaches, we have considered a number of challenging hybrid automata benchmark models. The tools and the benchmarks can be found at `http://swt.informatik.uni-freiburg.de/tool/spaceex`.

1.3 Outline

Most of the contributions of the thesis have already been published. The relevant papers are listed below:

[BDF+] 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. Submitted to *International Journal on Software Tools for Technology Transfer (STTT)*.

[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 *Model Checking Software (SPIN 2013)*, volume 7976 of *LNCS*, pages 117–134. Springer, 2013.

[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 *Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*, pages 479–494. Springer, 2012.

[BFG+14] Sergiy Bogomolov, Goran Frehse, Marius Greitschus, Radu Grosu, Corina S. Pasareanu, Andreas Podelski, and Thomas Strump. Assume-guarantee abstraction refinement meets hybrid systems. In *Haifa Verification Conference (HVC 2014)*, volume 8855 of *LNCS*, pages 116–131. Springer, 2014.

[BMPW14] Sergiy Bogomolov, Daniele Magazzeni, Andreas Podelski, and Martin Wehrle. Planning as model checking in hybrid domains. In *AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 2228–2234. AAAI Press, 2014.

The thesis is organized as follows. In Chapter 2, we introduce the necessary notions that we use throughout the thesis. Chapter 3 describes our approaches to guide the search in the state space of a hybrid automaton. This material has been presented at CAV'12 [BFG+12], SPIN'13 [BDF+13] and STTT [BDF+]. Then, we present our assume guarantee framework in Chapter 4 which is also published at HVC'14 [BFG+14]. Chapter 5 presents our translation from PDDL+ to hybrid automata. This chapter is based on the work presented at AAAI'14 [BMPW14]. Finally, in Chapter 6, we conclude the thesis and discuss some possible lines of research for the future.

2

Preliminaries

In this chapter, we introduce the preliminaries that are needed for this thesis. In Section 2.1, we introduce hybrid automata and their semantics. In Section 2.2, we discuss the symbolic reachability algorithms we use in our work.

2.1 Hybrid Automaton

We consider models that can be represented by hybrid automata. A hybrid automaton is formally defined as follows.

Definition 2.1 (Hybrid Automaton) *A* hybrid automaton *is a tuple* $\mathcal{H} = (Loc, Var, Init, Flow, Trans, Inv)$ *defining*

- *the finite set of locations* Loc*,*
- *the set of continuous variables* $Var = \{x_1, \ldots, x_n\}$ *from* $\mathbb{R}^n$*,*
- *the initial condition, given by the constraint* $Init(\ell) \subset \mathbb{R}^n$ *for each location* ℓ*,*
- *for each location* ℓ*, a relation called* $Flow(\ell)$ *over the variables and their derivative,*
- *the discrete transition relation, given by a set* $Trans$ *of discrete transitions; a discrete transition is formally defined as a tuple* (ℓ, g, ξ, ℓ') *defining*
  - *the source location* ℓ *and the target location* ℓ'*,*
  - *the guard, given by a linear constraint* g*,*
  - *the update, given by an affine mapping* ξ*, and*
- *the invariant* $Inv(\ell) \subset \mathbb{R}^n$ *for each location* ℓ*.*

In this thesis, we consider $Flow(l)$ to be continuous dynamics of the following two forms:

1. If $\dot{x}(t) \in P$ where P is a polytope, then the HA is called a linear hybrid automaton (LHA).
2. If $\dot{x}(t) = Ax(t) + u(t), u(t) \in \mathcal{U}$, where $x(t) \in \mathbb{R}^n$, A is a real-valued $n \times n$ matrix and $\mathcal{U} \subseteq \mathbb{R}^n$ is a closed and bounded convex set, then the HA is called an affine hybrid automaton (AHA).

The semantics of a hybrid automaton $\mathcal{H}$ is defined as follows. A *state* of $\mathcal{H}$ is a tuple $(\ell, \mathbf{x})$, which consists of a location $\ell \in \mathit{Loc}$ and a point $\mathbf{x} \in \mathbb{R}^n$. More formally, $\mathbf{x}$ is a valuation of the continuous variables in Var. For the following definitions, let $\mathcal{T} = [0, \Delta]$ be an interval for some $\Delta \geq 0$. A *trajectory* of $\mathcal{H}$ from state $s = (\ell, \mathbf{x})$ to state $s' = (\ell', \mathbf{x}')$ is defined by a tuple $\rho = (L, \mathbf{X})$, where $L : \mathcal{T} \to \mathit{Loc}$ and $\mathbf{X} : \mathcal{T} \to \mathbb{R}^n$ are functions that define for each time point in $\mathcal{T}$ the location and values of the continuous variables, respectively. Furthermore, we will use the following terminology for a given trajectory ρ. A sequence of time points where location switches happen in ρ is denoted by $(\tau_i)_{i=0\ldots k} \in \mathcal{T}^{k+1}$. In this case, we define the *length* of ρ as $|\tau| = k$. Trajectories $\rho = (L, \mathbf{X})$ (and the corresponding sequence $(\tau_i)_{i=0\ldots k}$) have to satisfy the following conditions:

- $\tau_0 = 0$, $\tau_i < \tau_{i+1}$, and $\tau_k = \Delta$ – the sequence of switching points increases, starts with 0 and ends with Δ
- $L(0) = \ell$, $\mathbf{X}(0) = \mathbf{x}$, $L(\Delta) = \ell'$, $\mathbf{X}(\Delta) = \mathbf{x}'$ – the trajectory starts in $s = (\ell, \mathbf{x})$ and ends in $s' = (\ell', \mathbf{x}')$
- $\forall i\ \forall t \in [\tau_i, \tau_{i+1}) : L(t) = L(\tau_i)$ – the location is not changed during the continuous evolution
- $\forall i\ \forall t \in [\tau_i, \tau_{i+1}) : \quad (\mathbf{X}(t), \dot{\mathbf{X}}(t)) \in \mathit{Flow}(L(\tau_i))$, i.e. $\dot{\mathbf{X}}(t) = A\mathbf{X}(t) + u(t)$ holds and thus the continuous evolution is consistent with the differential equations of the corresponding location
- $\forall i\ \forall t \in [\tau_i, \tau_{i+1}) : \mathbf{X}(t) \in \mathit{Inv}(L(\tau_i))$ – the continuous evolution is consistent with the corresponding invariants
- $\forall i \quad \exists (L(\tau_i), g, \xi, L(\tau_{i+1})) \in \mathit{Trans} : \mathbf{X}_{end}(i) = \lim_{\tau \to \tau_{i+1}^-} \mathbf{X}(\tau) \wedge \mathbf{X}_{end}(i) \in g \wedge \mathbf{X}(\tau_{i+1}) = \xi(\mathbf{X}_{end}(i))$ – every continuous transition is followed by a discrete one, $\mathbf{X}_{end}(i)$ defines the values of continuous variables right before the discrete transition at the time moment τ_{i+1} whereas $\mathbf{X}_{start}(i) = \mathbf{X}(\tau_i)$ denotes the values of continuous variables right after the switch at the time moment τ_i.

A state s' is *reachable* from state s if there exists a trajectory from s to s'.

In the following, we mostly refer to *symbolic states*. A symbolic state $s = (\ell, \mathcal{R})$ is defined as a tuple, where $\ell \in \mathit{Loc}$, and $\mathcal{R}$ is a convex and bounded set consisting of points $\mathbf{x} \in \mathbb{R}^n$. The continuous part $\mathcal{R}$ of a symbolic state is

also called *region*. The symbolic state space of $\mathcal{H}$ is called the *region space*. The initial set of states $\mathcal{S}_{init}$ of $\mathcal{H}$ is defined as $\bigcup_\ell(\ell, Init(\ell))$. The reachable state space Reach($\mathcal{H}$) of $\mathcal{H}$ is defined as the set of symbolic states that are reachable from an initial state in $\mathcal{S}_{init}$, where the definition of reachability is extended accordingly for symbolic states.

A *network* $\mathcal{N} = \{\mathcal{H}_1, \ldots, \mathcal{H}_m\}$ of hybrid automata is a set of hybrid automata. The semantics of $\mathcal{N}$ is defined based on the semantics of single hybrid automata, with the following extensions. Every automaton in $\mathcal{N}$ is associated with a finite set of *synchronization labels*, including a special label τ in all label sets. The discrete component of a *state* s of $\mathcal{N}$ is defined as a *vector* of locations that denotes the current locations of every component in $\mathcal{N}$. Similarly, in addition to single automata, a *trajectory* of $\mathcal{N}$ maps time points to vectors of locations of each automaton. For a time point t, changes in the location vectors in a trajectory can either be caused by a single transition labelled with τ of one automaton in $\mathcal{N}$ ("interleaving transition"), or there there are several automata in $\mathcal{N}$ that simultaneously fire transitions with equal synchronization label $\neq \tau$ ("synchronized transition"). We refer to the work by Donzé et al. [32]

In this thesis, we assume there is a given set of symbolic bad states $\mathcal{S}_{bad}$ that violate a given property. Our goal is to find a sequence of symbolic states which contains a trajectory from $\mathcal{S}_{init}$ to a symbolic *error state*, where a symbolic error state s_e has the property that there is a symbolic bad state in $\mathcal{S}_{bad}$ that agrees with s_e on the discrete part, and that has a nonempty intersection with s_e on the continuous part. A trajectory that starts in a symbolic state s and leads to a symbolic error state is called an *error trajectory* $\rho_e(s)$.

2.2 Symbolic States Representation

The representation of symbolic states plays a crucial role for the reachability analysis of hybrid automata. As outlined in the previous section, a symbolic state consists of a discrete location and a continuous region. The handling of continuous regions within the reachability analysis poses a special challenge as a number of operations on polyhedra (such as linear maps, Minkowski sum, and convex hull computation) need to be performed efficiently in practice. The LGG scenario [54] which is implemented in SpaceEx [39] relies on two main ingredients for this purpose: support functions [14] and template polyhedra. In the following, we will describe them in more detail. Note that the support function representation is currently applicable only to affine hybrid automata. However, we can still apply the LGG scenario to multiaffine hybrid automata by approximating their dynamics (see Chapter 3).

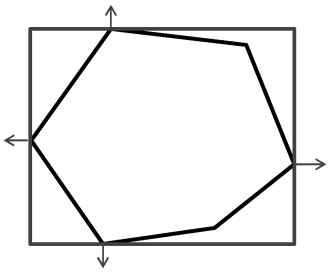


Fig. 2.1: Region representation using box directions

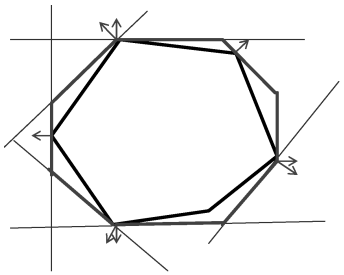


Fig. 2.2: Region representation using octagonal directions

The support function $\rho_{\mathcal{R}}(\ell)$ of a region $\mathcal{R}$ with respect to the direction $\ell \in \mathbb{R}^n$ is defined as follows:

$$\rho_{\mathcal{R}}(\ell) = \max_{x \in \mathcal{R}} \ell \cdot x$$

We can represent an arbitrary convex closed set $\mathcal{R}$ by using support functions in the following way:

$$\mathcal{R} = \bigcap_{\ell \in \mathbb{R}^n} \{x \mid \ell \cdot x \leq \rho_{\mathcal{R}}(\ell)\}$$

The representation based on support functions allows for efficiently computing all the above mentioned polyhedra operations, hence reachability algorithms in turn benefit from this representation.

As the consideration of an infinite number of directions is clearly infeasible from the computational point of view, SpaceEx also makes use of a continuous set representation derived from the support functions: *template* polyhedra. In this setting, we *predefine* from the very beginning a set of directions taken into account in course of the reachability analysis. In other words, a user provides a set of directions $D = \{\ell_1, \ldots, \ell_m\}$ used for the reachability analysis. Based on D, the region $\mathcal{R}$ can be over-approximated by the following polyhedron:

$$\mathcal{R}_D = \{x \in \mathbb{R}^n \mid \bigwedge_{\ell_i \in D} \ell_i \cdot x \leq \rho_{\mathcal{R}}(\ell_i)\}$$

SpaceEx supports a number of predefined direction sets such as, e.g., box directions (directions parallel to axes; see Figure 2.1) and octagonal directions (the union of directions parallel to axes and diagonal ones; see

Figure 2.2). Obviously, by increasing the number of considered directions, we can improve the approximation precision.

In the rest of this section, we briefly recapitulate the computation of continuous successors for a given symbolic state, i.e. the states which are reachable according to the continuous dynamics. As the continuous post operator does not change the discrete part of a symbolic state, we consider only the continuous region of a symbolic state.

The LGG scenario computes the continuous successors only for a finite time horizon. Therefore, we use a *time bounded version* of the reachable region $\text{Reach}_{t_1,t_2}(\mathcal{R})$ for a given starting region $\mathcal{R} \subseteq \mathbb{R}^n$, dynamics $\dot{x}(t) = Ax(t) + u(t), u(t) \in \mathcal{U}$ (*) and a time interval $[t_1, t_2] \subseteq \mathbb{R}^{\geq 0}$:

$$\text{Reach}_{t_1,t_2}(\mathcal{R}) = \{x(\tau) \mid t_1 \leq \tau \leq t_2, x(0) \in \mathcal{R}, \\ x(\tau) \text{ is the solution of (*)}\}$$

SpaceEx performs an over-approximating time-bounded reachability analysis of $\text{Reach}_{0,T}(\mathcal{R})$, where $T \in \mathbb{R}^{\geq 0}$ is a user-provided *time horizon*. In more detail, as the reachability analysis of hybrid automata is generally undecidable, SpaceEx over-approximates the successor regions by iteratively computing over-approximations based on *discretizing* the time up to the time horizon: First, the time interval $[0, T]$ is partitioned in a number of small time intervals $[\delta_i, \delta_{i+1}]$, where $\delta_i = i \cdot T_\delta$ $(i = 0, \ldots, N-1)$ and $T_\delta = T/N$ $(N \in \mathbb{N})$ is a user provided *sampling time*. Second, given this partitioning, SpaceEx covers the exact reachability set with the sequence $\Omega_i \subseteq \mathbb{R}^n, i = 0, \ldots, N-1$ where Ω_i defines the over-approximation of the states reachable within the time interval $[\delta_i, \delta_{i+1}]$. In other words, the following inclusion holds:

$$\text{Reach}_{0,T}(Init) \subseteq \bigcup_{i=0}^{N-1} \Omega_i$$

The set Ω_{i+1} can be expressed in terms of the "predecessor set" Ω_i by using a linear map and Minkowski sum. Therefore, we only need to provide a routine to compute Ω_0 which in turn can be done in two steps. First, we compute the convex hull of the union of the region $\mathcal{R}$ and its image at the moment T_δ. Second, we observe that the continuous dynamics non-linearities can lead to some reachable states being outside of the computed convex hull. In order to account for this phenomena, we bloat the resulting convex hull to ensure the over-approximation. Clearly, a larger sampling time T_δ makes a possibly larger bloating necessary, which worsens the approximation precision (see Figure 2.3 and Figure 2.4 for a comparison).

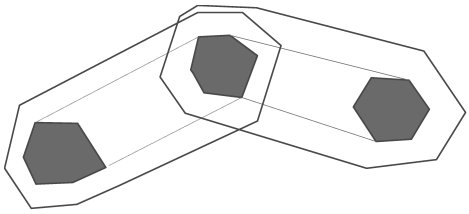


Fig. 2.3: Region representation using a large sampling time

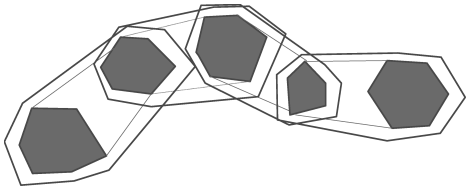


Fig. 2.4: Region representation using a small sampling time

To summarize, we observe that the adjustment of the template directions used in the support function representation and the sampling time in the continuous post operator crucially impacts the precision, i.e. the *abstraction level*, of the symbolic state representation. Clearly, an improved precision leads to an increased analysis time on the downside. Based on this representation, we will present an algorithm which leverages different abstraction levels to efficiently explore the region space.

3

Guided Search for Hybrid Automata

Guided search is an approach that has recently attracted much attention for finding errors in large systems [51]. As suggested by the name, guided search performs a search in the state space of a given system. In contrast to standard search methods like breadth-first or depth-first search, the search is guided by a cost function that estimates the search effort to reach an error state from the current state. This information is exploited by preferably exploring states with lower estimated costs. If accurate cost functions are applied, the search effort can significantly be reduced compared to uninformed search. Obviously, the cost function therefore plays a key role within the setting of guided search, as it should be as accurate as possible on the one hand, and as cheap to compute as possible on the other. Cost functions that have been proposed in the literature are mostly based on *abstractions* of the original system. An important class of abstraction-based cost functions is based on *pattern databases (PDBs)*. PDBs have originally been proposed in the area of Artificial Intelligence [28] and also have successfully been applied to model checking discrete and timed systems [63, 50, 51, 69]. Roughly speaking, a PDB is a data structure that contains abstract states together with abstract cost values based on an abstraction of the original system. During the concrete search, concrete states s are mapped to corresponding abstract states in the PDB, and the corresponding abstract cost values are used to estimate the costs of s. Overall, PDBs have demonstrated to be powerful for finding errors in different formalisms. The open question is if guided search can be applied equally successfully to finding errors in hybrid automata.

In this chapter, we first describe the box-based distance measure [20] to estimate the cost of a symbolic state based on the Euclidean distance from its continuous part to a given set of error states. This approach appears to be best suited for systems whose behavior is strongly influenced by the (continuous) differential equations. However, it suffers from the fact that dis-

crete information like mode switches is completely ignored, which can lead to arbitrary degeneration of the search. To see this, consider the example presented in Figure 3.1. It shows a simple hybrid automaton with one continuous variable which obeys the differential equation $\dot{x} = 1$ in every location (differential equations are omitted in the figure). The error states are given by the locations $l_{e1}, \ldots, l_{en}$ and invariants $0 \leq x \leq 8$. In this example, the box-based distance heuristic wrongly explores the whole lower branch first (where no error state is reachable) because it only relies on the continuous information given by the invariants. More precisely, for the box-based distance heuristic, the invariants suggest that the costs of the "lower" states are equal to 0, whereas the costs of the "upper" states are estimated to be equal to 4 (i. e., equal to the distance of the centers of the bounding boxes of the invariants).

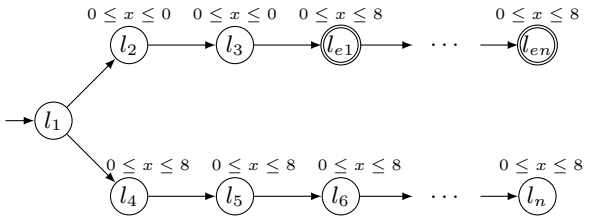


Fig. 3.1: A motivating example

To overcome these limitations, we furthermore introduce an abstraction-based cost function for hybrid systems [18, 17] which is motivated by PDBs. In contrast to the box-based approach based on Euclidean distances, this cost function is able to properly reflect the discrete part of the system. Compared to the "classical" discrete setting, the investigation of PDBs for hybrid automata becomes more difficult for several reasons. First, hybrid automata typically feature both discrete and continuous variables with complex dependencies and interactions. Therefore, the question arises how to compute a suitable (accurate) abstraction of the original system. Second, computations for symbolic successors and inclusion checks become more expensive than for discrete or timed systems – can these computations be performed or approximated efficiently to get an overall efficient PDB approach as well? In this chapter, we provide answers to these questions, leading to an efficient guided search approach for hybrid systems. In particular, we introduce an abstraction technique leveraging properties of the set representations used in modern reachability algorithms. By simply using coarser parameters for the explicit representation, we obtain suitable and cheap *coarse-grained space*

abstractions for the behaviors of a given hybrid automaton. Furthermore, we adapt the idea of *partial* PDBs, which has been originally proposed for solving discrete search problems [8], to the setting of hybrid automata in order to reduce the size and computation time of "classical" PDBs. Our implementation in the SpaceEx tool [39] shows the practical potential.

The remainder of the chapter is organized as follows. After introducing the necessary background for this work in Section 3.1, we present our box-based distance measure in Section 3.2 and PDB approach for hybrid systems in Section 3.3. This is followed by a discussion about related work in Section 3.4. Afterwards, we present our experimental evaluation in Section 3.5. Finally, we conclude the chapter in Section 3.6.

3.1 Preliminaries

We discuss the basic framework of the guided search in Section 3.1.1. Furthermore, we introduce the ideas behind the pattern databases in Section 3.1.2.

3.1.1 Guided Search

In this section, we introduce a guided search algorithm (Algorithm 1) along the lines of the reachability algorithm used by the current version of SpaceEx [39]. It works on the region space of a given hybrid automaton. The algorithm checks if a symbolic error state is reachable from a given set of initial symbolic states $\mathcal{S}_{init}$. As outlined above, we define a symbolic state s_e in the region space of $\mathcal{H}$ to be a symbolic error state if there is a symbolic state $s \in \mathcal{S}_{bad}$ such that s and s_e agree on their discrete part, and the intersection of the regions of s and s_e is not empty (in other words, the error states are defined with respect to the given set of bad states). Starting with the set of initial symbolic states from $\mathcal{S}_{init}$, the algorithm explores the region space of a given hybrid automaton by iteratively computing symbolic successor states until an error state is found, no more states remain to be considered, or a (given) maximum number of iterations i_{max} is reached. The exploration of the region space is guided by the *cost* function such that symbolic states with lower cost values are considered first.

In the following, we provide a conceptual description of the algorithm using the following terminology. A symbolic state s' is called a symbolic *successor state* of a symbolic state s if s' is obtained from s by first computing the continuous successor of s (according to iteratively over-approximating the successor regions of s with sets Ω_i as described in the previous section), and then by computing a discrete successor state of the resulting (intermediate)

Algorithm 1 A guided symbolic reachability algorithm

Input: Set of initial symbolic states $\mathcal{S}_{init}$, set of symbolic bad states $\mathcal{S}_{bad}$, cost function *cost*

Output: Can a symbolic error state be reached from a symbolic state in $\mathcal{S}_{init}$?

```
 1: compute cost(s) for all s ∈ S_init
 2: PUSH (L_waiting, {(s, cost(s)) | s ∈ S_init})
 3: i := 0
 4: while (L_waiting ≠ ∅ ∧ i < i_max) do
 5:    s_curr := GETNEXT (L_waiting)
 6:    i := i + 1
 7:    s'_curr := CONTINUOUSSUCCESSOR(s_curr)
 8:    if s'_curr is a symbolic error state then
 9:       return "Error state reached"
10:    end if
11:    PUSH (L_passed, s'_curr)
12:    S' := DISCRETESUCCESSORS(s'_curr)
13:    for all s' ∈ S' do
14:       if s' ∉ L_passed then
15:          compute cost(s')
16:          PUSH (L_waiting, (s', cost(s')))
17:       end if
18:    end for
19: end while
20: if i = i_max then
21:    return "Maximal number of iterations reached"
22: else
23:    return "Error state not reachable"
24: end if
```

state. Therefore, for a given symbolic state s_{curr}, the function CONTINUOUSSUCCESSOR (line 7) returns a symbolic state which is an over-approximation of the symbolic state reachable from s_{curr} within the given time horizon according to the continuous evolution. Accordingly, the function DISCRETESUCCESSORS (line 12) returns the symbolic states that are reachable due to the outgoing discrete transitions.

A symbolic state s is called *explored* if its symbolic successor states have been computed. A symbolic state s is called *visited* if s has been computed but not yet necessarily explored. To handle encountered states, the algorithm maintains the data structures $\mathcal{L}_{passed}$ and $\mathcal{L}_{waiting}$. $\mathcal{L}_{passed}$ is a list containing symbolic states that are already explored; this list is used to avoid exploring cycles in the region space. $\mathcal{L}_{waiting}$ is a priority queue that contains visited symbolic states together with their cost values that are candidates to be explored next. The algorithm is initialized by computing the cost values for the initial symbolic states and pushing them accordingly into $\mathcal{L}_{waiting}$ (lines 1 – 2). The main loop iteratively considers a best symbolic state s_{curr}

from $\mathcal{L}_{waiting}$ according to the cost function (line 5), computes its symbolic continuous successor state s'_{curr} (line 7), and checks if s'_{curr} is a symbolic error state (lines 8 – 10). (Recall that s'_{curr} is defined as a symbolic error state if there is a symbolic bad state $s \in \mathcal{S}_{bad}$ such that s and s'_{curr} agree on their discrete part, and the intersection of the regions of s and s'_{curr} is not empty.) If this is the case, the algorithm terminates. If this is not the case, then s'_{curr} is pushed into $\mathcal{L}_{passed}$ (line 11). Finally, for the resulting symbolic state s'_{curr}, the symbolic discrete successor states are computed, prioritized and pushed into $\mathcal{L}_{waiting}$ if they have not been considered before (lines 12 – 18). As a side remark, if a successor state $s' = \langle l, \mathcal{R} \rangle$ is not contained in $\mathcal{L}_{passed}$ (line 14), but instead there is a symbolic state $s'' = \langle l, \mathcal{R}' \rangle \in \mathcal{L}_{passed}$ with $\mathcal{R} \subset \mathcal{R}'$, then s' is discarded as well because all transitions enabled in s' have already been enabled in s'' which is already explored. Finally, the check if the given maximal number of iterations has been reached (line 4 and line 20) ensures termination, which would not be generally guaranteed otherwise (e. g., because of Zeno behavior).

Obviously, the search behavior of Algorithm 1 is crucially determined by the cost function that is applied. In the next section, we give a generic description of *pattern database* cost functions.

3.1.2 General Framework of Pattern Databases

For a given system $\mathcal{S}$, a pattern database (PDB) in the classical sense (i. e., in the sense PDBs have been considered for discrete and timed systems) is represented as a table-like data structure that contains abstract states together with abstract cost values. The PDB is used as a cost estimation function by mapping concrete states s to corresponding abstract states $s^{\#}$ in the PDB, and using the abstract cost value of $s^{\#}$ as an estimation of the cost value of s. The computation of a classical PDB is performed in three steps. First, a subset $\mathcal{P}$ of variables and automata of the original system $\mathcal{S}$ is selected. Such subsets $\mathcal{P}$ are called *pattern*. Second, based on $\mathcal{P}$, an abstraction $\mathcal{S}^{\#}$ is computed that only keeps the variables occurring in $\mathcal{P}$. Third, the entire state space of $\mathcal{S}^{\#}$ is computed and stored in the PDB. More precisely, all reachable abstract states together with their abstract cost values are enumerated and stored. The abstract cost value for an abstract state is defined as the shortest length of a trajectory from that state to an abstract error state. The resulting PDB of these three steps is used as the *cost* function during the execution of Algorithm 1; in other words, the PDB is computed *prior* to the actual model checking process, where the resulting PDB is used as an input for Algorithm 1.

A straight-forward adaptation of such classical PDBs to the area of hybrid automata is the following. For a given hybrid automaton $\mathcal{H}$, compute an abstract system $\mathcal{H}^\#$ as the basis for the PDB, where $\mathcal{H}^\#$ is obtained from $\mathcal{H}$ by removing some of the variables in $\mathcal{H}$ (the pattern corresponds to the remaining variables in $\mathcal{H}^\#$). Based on $\mathcal{H}^\#$, the PDB is represented by a data structure that contains abstract states together with corresponding cost values. The abstract states and cost values are obtained by a region space exploration of $\mathcal{H}^\#$. The abstract cost value of an abstract state $s^\#$ is defined as the length of a shortest found trajectory in $\mathcal{H}^\#$ from $s^\#$ to an abstract error state. The PDB computes the cost function

$$cost^P(s) := cost^\#(s^\#),$$

where s is a symbolic state, $s^\#$ is a corresponding abstract state to s in the PDB, and $cost^\#$ is the length of the corresponding trajectory from $s^\#$ to an abstract error state as defined above.

3.2 The Box-Based Distance Measure

In this section, we present the box-based heuristic. Section 3.2.1 discusses the idea of order-preserving measures. In Section 3.2.2, we provide a conceptual description of an idealized distance measure based on the length of trajectories. This idealized distance measure is used as the basis for our box-based distance measure which is presented in Section 3.2.3.

3.2.1 Order-Preserving Measures

In the following, we discuss a desirable property of cost measures in the context of guided search. As already outlined, we intend to design a cost measure that guides the search well in the region space. To achieve good guidance, the relative error of a cost measure h to the *cost* function as defined in the previous section is not necessarily correlated to the accuracy of h. In other words, h may accurately guide the search although the relative error of h's cost estimations is high. This is because it suffices for h to always select the "right" state to be explored next.[1] Based on this observation, we give the definition of *order-preserving*.

[1] As a simple example, consider two states s and s' with real costs 100 and 200, respectively. Furthermore, consider a cost measure that estimates the costs of these states as 1 and 2, respectively. We observe that the relative error is high, but the better state is determined nevertheless.

Definition 3.1 (Order-Preserving). *Let $\mathcal{H}$ be a hybrid automaton. A cost measure h is* order-preserving *if for all states s and s' with $cost(s) < cost(s')$, then also $h(s) < h(s')$.*

Cost measures that are order-preserving lead to perfect search behavior with respect to the *cost* function. Therefore, it is desirable to have cost measures that satisfy this property. We will come back to this point in the next section.

3.2.2 A Trajectory-Based Distance Measure

In this section, we formulate a distance measure *dist* that can be expressed in terms of the length of trajectories (see below for a justification of the name). For states s and s', the distance measure $dist(s, s')$ is defined as the minimal length of a trajectory ρ that is obtained from the continuous flow and discrete switches of trajectories that lead from s to s'. To define this more formally, we denote the set of trajectories that lead from s to s' with $\mathcal{T}(s, s')$. Moreover, $dist_{eq}(\mathbf{x}, \mathbf{x}')$ denotes the Euclidean distance between points $\mathbf{x}$ and $\mathbf{x}'$. Using this notation, we give the definition of our trajectory-based distance measure.

Definition 3.2 (Trajectory-Based Distance Measure). *Let $\mathcal{H}$ be a hybrid automaton, let s and s' be states of $\mathcal{H}$. We define the distance measure*

$$dist(s, s') := \min_{\rho \in \mathcal{T}(s,s')} \sum_{i=0}^{|\tau|-1} \left(\int_{\tau_i}^{\tau_{i+1}} \sqrt{\dot{x}_1^2(t) + \cdots + \dot{x}_n^2(t)}\ dt + dist_{eq}(i, i+1) \right),$$

where $\rho = (L, \mathbf{X})$, $\mathbf{X}(t) = (x_1(t), \ldots, x_n(t))$, and $dist_{eq}(i, i+1)$ is a shorthand for $dist_{eq}(\mathbf{X}_{end}(i), \mathbf{X}_{start}(i+1))$.

Informally speaking, the distance between states s and s' is defined as the length of a shortest trajectory ρ from s to s' induced by the differential equations and discrete updates of the visited locations $L(\tau_i)$ in ρ. Obviously, the trajectory-based distance measure can be applied to error states in a straight-forward way by setting s' to an error state. We call the trajectory-based error distance measure $dist_E(s) := \min_{s_e} dist(s, s_e)$, where s_e ranges over the set of given error states of $\mathcal{H}$.

In the following, we show that for a certain class of hybrid automata $\mathcal{H}$, $dist(s)$ is indeed correlated to the costs of s for all states s of $\mathcal{H}$. In fact, this correlation can be established for hybrid automata such that

1. all differential equations in $\mathcal{H}$ are of the form $\dot{x}_i(t) = \pm c_i$ for every continuous variable $x_i \in Var$ and a constant $c_i \in \mathbb{N}$, and

2. all guards in $\mathcal{H}$ do not contain discrete updates.

We call hybrid automata that satisfy the above requirements *restricted automata*. Specifically, we observe that a necessary condition for hybrid automaton $\mathcal{H}$ to be a restricted automaton is that for every continuous variable x_i in $\mathcal{H}$, there is a global constant $c_i \in \mathbb{N}$ such that *all* differential equations in $\mathcal{H}$ that talk about x_i only differ in the sign. It is not difficult to see that for the class of restricted automata, the length of the obtained flow is linearly correlated with the time. Therefore, the error distance measure $dist_E$ is order-preserving.

Proposition 3.3. *For restricted automata $\mathcal{H}$, $dist_E$ is order-preserving.*

Proof. We show that from $cost(s) < cost(s')$, it follows that $dist_E(s) < dist_E(s')$. As $\mathcal{H}$ is a restricted automaton, the square root of $\dot{x}_1^2(t) + \cdots + \dot{x}_n^2(t)$ is constant and $dist_{eq}(i, i+1)$ is equal to zero. Thus, $dist_E(s) = \min_{s_e} \min_{\rho \in \mathcal{T}(s,s_e)} c \sum \delta_i$, which is equal to $c \cdot cost(s)$. This proves the claim.

Proposition 3.3 leads to an interesting and important observation. Roughly speaking, we have reduced the problem of computing (dwell time) costs in the state space to the problem of computing "shortest" flows between regions. Therefore, Proposition 3.3 shows that under certain circumstances, we can choose between *cost* and *dist* without loosing precision. However, although still hard to compute, the representation of *dist* based on lengths of flows lends itself to an approximation based on *estimated* flow lengths. This approximation is presented in the next section.

3.2.3 The Box-Based Approximation

In the following, we propose an effective approximation of the *dist* function that we have derived in the last section. While the *dist* measure has been defined for concrete states, our box-based approximation is defined for symbolic states. The approximation is based on the following two ingredients.

1. Instead of computing the exact length of trajectories between two points $\mathbf{x}$ and $\mathbf{x}'$ (as required in Definition 3.2), we use the Euclidean distance between $\mathbf{x}$ and $\mathbf{x}'$.
2. As we are working in the region space, we approximate a given region R with the smallest box $\mathcal{B}$ such that R is contained in $\mathcal{B}$. This corresponds to the well-known principle of Cartesian abstraction.

In the following, we will discuss these ideas and make them precise. As stated, we define the estimated distance between points $\mathbf{x}$ and $\mathbf{x}'$ as the

Euclidean distance between $\mathbf{x}$ and $\mathbf{x}'$. Unfortunately, the Euclidean distance is not order-preserving for restricted automata, but only for even more restricted automata that allow even less behavior. This is formalized in the following proposition. For a state $s = (\ell, \mathbf{x})$, we define $dist_E^{eq}(s) := \min_{s_e} dist_{eq}(\mathbf{x}, \mathbf{x}_e)$, where $s_e = (\ell_e, \mathbf{x}_e)$ ranges over the error states, and $dist_{eq}$ is the Euclidean distance function as introduced earlier.

Proposition 3.4. *For restricted automata $\mathcal{H}$ with $\dot{x}_i(t) = c_i$, i. e., for restricted automata where all locations have the same continuous behavior, $dist_E^{eq}$ is order-preserving.*

Proof. We show that from $cost(s) < cost(s')$, it follows that $dist_E^{eq}(s) < dist_E^{eq}(s')$. By assumption, $\mathcal{H}$ is a restricted automaton where every location has the same continuous dynamics. Therefore, the Euclidean distance $dist_{eq}(s, s_e)$ is equal to $\int_0^{\tau_k} \sqrt{\dot{x}_1^2(t) + \cdots + \dot{x}_n^2(t)}\, dt$, where τ_k is equal to the accumulated dwell time of the trajectory from s to s_e. Furthermore, the square root of $\dot{x}_1^2(t) + \cdots + \dot{x}_n^2(t)$ is some constant c. Thus $dist_E^{eq}(s) = \min_{s_e} dist_{eq}(s, s_e) = \min_{s_e} c \cdot \tau_k = c \cdot \min_{s_e} \tau_k$ which in turn is equal to $c \cdot cost(s)$.

The above proposition reflects that the Euclidean distance is a coarse approximation of the trajectory-based distance measure because it is effectively only order-preserving for automata that allow behavior that corresponds to automata with only one location. Indeed, it is the coarsest approximation one can think of on the one hand. However, on the other hand, we have shown that there *exist* automata for which it *is* order-preserving, which suggests (together with Proposition 3.3) that the Euclidean distance could be a good heuristic to estimate distances also for richer classes of hybrid automata. Moreover, it is efficiently computable which is particularly important for distance heuristics that are computed on-the-fly during the state space exploration. Obviously, one can think of arbitrary more precise approximations based on piecewise linear functions. However, such approximations also become more expensive to compute.

For our distance heuristic, we approximate a given symbolic state $s = (\ell, R)$ with the smallest box $\mathcal{B}(s)$ that contains R. Formally, this corresponds to the requirement

$$R \subseteq \mathcal{B}(s) = [x_1, x_1'] \times \ldots \times [x_n, x_n'] \subseteq \mathbb{R}^n \wedge \forall \mathcal{B}' \neq \mathcal{B}(s) : R \subseteq \mathcal{B}' \Rightarrow \mathcal{B}(s) \subseteq \mathcal{B}'.$$

To be efficiently computable, it is essential that tight over-approximating boxes can be computed efficiently. This can be achieved using linear programming techniques. Our distance heuristic h^{eq} is defined as the Euclidean

distance between the center of two boxes. Formally, for a symbolic state $s = (\ell, R)$, we define

$$h^{eq}(s) := \min_{s_e} dist_{eq}(\textsc{Center}(\mathcal{B}(R)), \textsc{Center}(\mathcal{B}(R_e))),$$

where $s_e = (\ell_e, R_e)$ ranges over the set of error states of $\mathcal{H}$, $dist_{eq}$ is the Euclidean distance metric, and $\textsc{Center}(B)$ denotes the central point of box B. Obviously, central points of boxes can be computed efficiently as the arithmetic average of its lower and upper bounds for every dimension.

Overall, our distance heuristic h^{eq} determines distance estimations for symbolic states $s = (\ell, R)$ by first over-approximating R with the smallest box $\mathcal{B}$ that contains R, and then computing the minimal Euclidean distance between $\mathcal{B}$'s center and the center of an error state. This procedure is summed up by Alg. 2.

Algorithm 2 Compute Distance Heuristic h^{eq}

Input: State $s = (\ell, R)$
Output: Estimated distance to a closest error state in S_{error}

```
1: d_min ← ∞
2: B ← B(R)
3: for all s' = (ℓ', R') ∈ S_error do
4:     B' ← B(R')
5:     d_curr ← dist_eq(Center(B), Center(B'))
6:     if d_curr < d_min then
7:         d_min ← d_curr
8:     end if
9: end for
10: return d_min
```

3.3 Pattern Databases for Hybrid Automata

In Section 3.1.2, we have described the general approach for computing and using a PDB for guiding the search. However, for hybrid automata, there are several challenges using the classical PDB approach. First, it is not clear how to effectively design and compute suitable abstractions $\mathcal{H}^{\#}$ for hybrid automata $\mathcal{H}$ with complex variable dependencies. Second, in Section 3.3.2, we address the general problem that the precomputation of a PDB is often quite expensive, where in many cases, only a small fraction of the PDB is actually needed for the search [43]. This is undesirable in general, and specifically becomes problematic in the context of hybrid automata because the reachability analysis in hybrid automata is typically much more expensive

than, e. g., for discrete systems. In Section 3.3.2, we introduce a variant of *partial* PDBs for hybrid automata to address these problems.

3.3.1 Coarse-Grained Space Abstractions

A general question in the context of PDBs is how to compute suitable abstractions of a given system. In particular, for hybrid automata where variables often have rather complex dependencies, projection abstractions based on removing variables (as done for classical PDBs) can be too coarse to achieve accurate heuristics. In this chapter, we propose a simple, yet elegant alternative to the classical PDB approach to obtain a coarse grained and fast analysis: As described in Section 2.2, the LeGuernic-Girard (LGG) algorithm implemented in SpaceEx [39] uses support function representation (based on the chosen set of template directions) to compute and store over-approximations of the reachable states. Therefore, a reduced number of *template directions* and an increased *sampling time* results in an abstraction of the original region space in the sense that the dependency graph of the reachable abstract symbolic states is a discrete abstraction of the automaton. The granularity of the resulting abstraction is directly correlated with the parameter selection: Choosing coarser parameters (fewer template directions, larger sampling time) in the reachability algorithm makes this abstraction coarser, whereas finer parameters lead to finer abstractions as well. In more detail, for a given set of template directions D and sampling time N, a subset $D' \subset D$ and a larger sampling time $N' > N$ induce *coarse-grained space abstractions* with respect to the abstractions obtained by D and N: the over-approximation of regions based on D' and N' are coarser than for D and N. As an example for template directions, consider again Figure 2.1 and Figure 2.2: the set of box directions in Figure 2.1 is a coarse-grained space abstraction of the set of octagonal directions in Figure 2.2. Similarly, as an example for the sampling time, consider again Figure 2.3 and Figure 2.4, where Figure 2.3 shows a coarse-grained space abstraction based on increased sampling time of the regions in Figure 2.4.

In the following, we apply coarse-grained space abstractions to obtain abstractions as the basis for pattern databases. This is a significant difference compared to classical PDB approaches (see Section 3.1.2): Instead of computing an explicit (projection) abstraction $\mathcal{H}^{\#}$ based on a *subset* of all variables, we *keep* all variables (and hence, the original automaton $\mathcal{H}$), and instead choose a coarser exploration of the abstract region space of $\mathcal{H}$ to obtain the abstraction used for the PDB. (In practice, we apply unguided search provided by SpaceEx to compute this coarser abstraction.) As an ad-

ditional difference to classical PDBs, we will apply a variant of *partial* PDBs, which are introduced in the next section.

3.3.2 Partial Pattern Databases

As already outlined, a general drawback of classical PDBs is the fact that their precomputation might become quite expensive. Even worse, in many cases, most of this precomputation time is often unnecessary because only a small fraction of the PDB is actually needed during the symbolic search in the region space [43]. One way that has been proposed in the literature to overcome this problem is to compute the PDB on demand: So-called *switch-back search* maintains a family of abstractions with increasing granularity; these abstractions are used to compute the PDB to guide the search in the next-finer level [53].

In the following, we apply a variant of *partial* PDBs [8] based on coarse-grained space abstractions to address this problem: Instead of computing the whole abstract region space for a given abstraction, we restrict the abstract search to explore only a fraction of the abstract region space while focusing on those abstract states that are likely to be sufficient for the concrete search. In the following definition, we call an abstract state $s^\#$ *corresponding* to state s if s and $s^\#$ agree on their discrete part, the region of s is included in region of $s^\#$, and $s^\#$ is an abstract state with minimal abstract costs that satisfies these requirements.

Definition 3.5 (Partial Pattern Database) *Let $\mathcal{H}$ be a hybrid automaton. A* partial pattern database *for $\mathcal{H}$ is a pattern database for $\mathcal{H}$ that contains only abstract state/cost value pairs for abstract states that are part of some trajectory of shortest length (in terms of number of location switches) from an initial state to some abstract error state. The partial pattern database computes the function*

$$cost^{PP}(s) := \begin{cases} cost^\#(s^\#) & \text{if ex. corresponding } s^\# \text{ to } s \\ +\infty & \text{otherwise} \end{cases}$$

where s, $s^\#$, and $cost^\#$ are defined as above, and $+\infty$ is a default value indicating that no corresponding abstract state to s exists.

Informally, a partial PDB for a hybrid automaton $\mathcal{H}$ exactly contains those abstract states that are explored on some *shortest* trajectory (instead of containing *all* abstract states of a complete abstract region space exploration to *all* abstract error states as it would be the case for a classical PDB). In other words, partial PDBs are incomplete in the sense that there might

exist concrete states, but the corresponding abstract states are not contained the PDB. In such cases, the default value $+\infty$ is returned with the intention that corresponding concrete states are only explored if no other states are available. Obviously, this might worsen the overall search guidance compared to the fully computed PDB. However, in special cases, a partial PDB is already sufficient to obtain the same cost function as obtained with the original PDB or even obtained with a perfect cost function (that allows for exploring the region space without backtracking to find an error state). For example, this is the case when only abstract states are excluded from which no abstract error state is reachable anyway. More generally, under the idealized assumption that the abstraction is fine enough such that no spurious behavior occurs on shortest possible error trajectories, the partial PDB already delivers the same search behavior as a perfect search algorithm that finds an error trajectory without backtracking.

Proposition 3.6. *Let $\mathcal{H}$ be a hybrid automaton. Let $n \in \mathbb{N}_0$ be the length of a shortest concrete error trajectory. If all shortest abstract error trajectories in $\mathcal{H}$ (obtained by a coarse-grained space abstraction to build a pattern database) correspond to concrete error trajectories of the same length, then guided search with Algorithm 1 and $cost^{PP}$ finds an error trajectory after n steps.*

Proof. By construction, the partial PDB contains exactly those symbolic abstract states that are part of shortest possible error trajectories. By assumption, these abstract states correspond to concrete states on concrete error trajectories of the same length. Hence, for every concrete state s on a shortest error trajectory, there is a corresponding entry in the partial PDB for all concrete successor states s' of s that are part of a shortest concrete error trajectory, and $cost^{PP}(s') = cost^{PP}(s) - 1$. In addition, for all concrete successor states s'' that are not part of a shortest concrete error trajectory, $cost^{PP}(s'') = \infty$. Overall, the claim follows by an inductive argument: Let s_0 be an initial state such that $cost^{PP}(s_0) = n$ is minimal among the costs of all initial states, i. e., n is the length of a shortest concrete error trajectory. Furthermore, all concrete states on a shortest concrete error trajectory have a concrete successor state with a cost value decreased by one, whereas all other successor states have a cost value of infinity. Hence, Algorithm 1 with the $cost^{PP}$ function finds a concrete error trajectory within n steps.

Under the idealized assumptions of Proposition 3.6, it follows immediately that guided search applying the full PDB cannot improve over the partial PDB.

Corollary 3.7. *Under the assumptions of Proposition 3.6, guided search with Algorithm 1 and* $cost^P$ *explores at least as many states as with* $cost^{PP}$*.*

Proposition 3.6 and Corollary 3.7 show that partial PDBs can provide effective search guidance in an idealized setting where the applied abstraction only introduces spurious behavior on non-relevant parts of the region space. Clearly, in practice, these assumptions will mostly not be satisfied for abstractions that are efficiently computable. However, we rather consider Proposition 3.6 as a proof of concept showing that the basic concept of partial PDBs is meaningful in our setting. (In our experimental analysis, we will show that partial PDBs yield an effective and efficient approach for a number of practical and challenging problems as well – we will come back to this point in Section 3.5.) Overall, we will see that although in case the requirements of Proposition 3.6 are not fulfilled, partial PDBs can still be a good heuristic choice that lead to cost functions that are efficiently computable and accurately guide the concrete search.

3.3.3 Discussion

Our pattern database approach for finding error states exploits abstractions in a different way than in common approaches for verification (see Section 3.4 for a discussion on related work). Most notably, the main focus of our abstraction is to provide the basis for the cost function to guide the search, rather than to prove correctness (although, under certain circumstances, it can be efficiently used for verification as well – we will come back to this point in the experiments section). As a short summary of the overall approach, we first compute a symbolic abstract region space (as described in Section 3.3.1), where the encountered symbolic abstract states $s^\#$ are stored in a table together with the corresponding abstract cost values of $s^\#$. To avoid the (possibly costly) computation of an *entire* PDB, we only compute the PDB partially (as described in Section 3.3.2). This partial PDB is then used as the cost function of our guided reachability algorithm. As in many other approaches that apply abstraction techniques to reason about hybrid automata, the abstraction that is used for the PDB is supposed to accurately reflect the "important" behavior of the automaton, which results in accurate search guidance of the resulting cost function and hence, of our guided reachability algorithm.

An essential feature of the PDB-based cost function is the ability to reflect the continuous *and* the discrete part of the automaton. To make this more clear, consider again the motivating example from the introduction (Figure 3.1). As we have discussed already, the box-based distance function

first wrongly explores the whole lower branch of this automaton because no discrete information is used to guide the search. In contrast, a partial PDB is also able to reflect the discrete behavior of the automaton. In this example, the partial PDB consists of an abstract trajectory to the first reachable error state, which is already sufficient to guide the (concrete) region space exploration towards to first reachable error state as well. In particular, this example shows the advantage of partial PDBs compared to fully computed PDBs (recall that fully computed PDBs would include *all* error states, whereas the partial PDB only contains the trajectory to a shortest one). In general, our PDB-approach is particularly well suited for hybrid automata with a non-trivial amount of discrete behavior. However, the continuous behavior is still considered according to our abstraction technique as introduced in Section 3.3.1. Overall, partial PDBs appear to be an accurate approach for guided search because they accurately balance the computation time for the cost function on the one hand, and lead to efficient and still accurately informed cost functions on the other hand.

3.4 Related Work

Abstraction techniques for hybrid automata have been mostly considered in a verification setting, i. e., in a setting where the focus is on proving that a given set of bad states cannot be reached. For this purpose, abstractions have been applied in different ways. On the one hand, a number of approaches to abstract the *regions* of symbolic states within the reachability analysis have been suggested, including constraint polyhedra [38], ellipsoids [52] and orthogonal polyhedra [22]. In this chapter, we use the support function representation [54]. These approaches have in common that the structure of the considered hybrid automaton is left intact. On the other hand, it also possible to abstract a hybrid automata structure. Alur et al. [2] suggest to use predicate abstraction for the hybrid automata analysis. In addition, Tiwari et al. [68] introduce a method based on the quantifier elimination decision procedure for real closed fields. Furthermore, Tiwari [67] investigates Lie derivatives and their application to the abstraction generation. Jha et al. [46] computes abstractions by removing some of the continuous variables. Finally, Bogomolov et al. [19] abstract hybrid automata by merging locations. The abstract dynamics is computed by eliminating the state variables and computing a convex hull. Our pattern database approach belongs to the first group outlined above as we exploit the parametrization of the symbolic region representation.

A prominent model checking approach for hybrid automata is based on counterexample-guided abstraction refinement (CEGAR) [4, 3]. In a nutshell,

CEGAR iteratively refines the considered abstraction until the abstraction is fine enough to prove or refute the property. Our PDB approach shares with CEGAR the general idea of using an abstraction to analyze a concrete automaton. However, in contrast to CEGAR, where abstract counterexamples have to be validated and possibly used in further abstraction refinement, abstractions for PDBs are never refined and only used as a heuristic to *guide* the search within the concrete automaton. In other words, in contrast to CEGAR, the accuracy of the abstraction influences the *order* in which concrete states are explored, and hence, the accuracy in turn influences the *performance* of the resulting model checking algorithm. Therefore, a crucial difference lies in the fact that CEGAR does the search in the abstract space, replays the counterexample in the concrete space, and stops if the error trajectory cannot be followed. In contrast, our approach does the search in the concrete space and uses the PDBs for guidance, only. If an abstract trajectory cannot be followed, the search does not stop, but tries other branches until either a counterexample is found, or all trajectories have been exhausted.

Considering more specialized techniques to find error states in faulty hybrid automata, Bhatia and Frazzoli [15] propose using rapidly exploring random trees (RRTs). In the context of hybrid automata, the objective of a basic RRTs approach is to efficiently cover the region space in an "equidistant" way in order to avoid getting stuck in some part of the region space. Recently, RRTs were extended by adding guidance of the input stimulus generation [29]. However, in contrast to our approach, RRTs approaches are based on numeric simulations, rather than symbolic executions. Applying PDBs to RRTs would be an interesting direction for future work. In a further approach, Plaku, Kavraki and Vardi [59] propose to combine motion planning with discrete search for falsification of hybrid automata. The discrete search and continuous search components are intertwined in such a way that the discrete search extracts a high-level plan that is then used to guide the motion planning component. In a slightly different setting, Ratschan and Smaus [64] apply search to finding error states in hybrid automata that are deterministic. Hence, the search reduces to the problem of finding an accurate initial state.

SpaceEx [39] is a recently developed, yet already prominent model checking tool for hybrid automata. As suggested by the name, it explores the region space by applying (symbolic) search. The most related approach to this thesis has recently been presented by Bogomolov et al. [20], who propose a cost function based on Euclidean distances of the regions of the current state and error states. The resulting guided search algorithm is implemented in SpaceEx and has demonstrated to achieve significant guidance and performance improvements compared to the uninformed search of SpaceEx. In

contrast to the presented PDB approach in this thesis, the Euclidean distances are solely based on the continuous part of the automaton, whereas PDBs are able to reflect both discrete and continuous parts.

Moreover, guided search has been applied to finding error states in a subclass of hybrid automata, namely to *timed* systems. In particular, PDBs have been investigated in this context [50, 51, 69]. In contrast to this chapter, the PDB approaches for timed systems are "classical" PDB approaches, i. e., a subset of the available automata and variables are selected to compute a projection abstraction. To select this subset, Kupferschmid et al. [50] compute an abstract error trace and select the automata and variables that occur in transitions in this abstract trace. In contrast, Kupferschmid and Wehrle [51, 69] start with the set of all automata and variables (i. e., with the complete system), and iteratively remove variables as long as the resulting projection abstraction is "precise enough" according to a certain quality measure. In both approaches, the entire PDB is computed, which is more expensive than the partial PDB approach proposed in this chapter.

3.5 Evaluation

We have implemented $cost^{PP}$ in the SpaceEx tool [39] and evaluated it on a number of challenging benchmarks. The experiments have been performed on a machine running with AMD Opteron 6174 processors. We set a time limit of 30 minutes per run. In the following, we report results for our PDB implementation of $cost^{PP}$ in SpaceEx. We compared $cost^{PP}$ with uninformed depth-first search as implemented in SpaceEx, and with the recently proposed box-based distance function [20] on several challenging benchmark problems. We compare the number of iterations of SpaceEx, the length of the error trajectory found as well as the overall search time (including the computation of the PDB for $cost^{PP}$) in seconds. In the following, we will shortly denote partial PDBs with PDBs.

3.5.1 Results for Navigation Benchmarks

As a first set of benchmarks, we consider a variant of the well-known navigation benchmark [35]. This benchmark models an object moving on the plane which is divided into a grid of cells. The dynamics of the object's planar position in each cell is governed by the differential equations $\dot{x} = v$, $\dot{v} = A(v - v_d)$ where v_d stands for the targeted velocity in this location. Compared to the originally proposed navigation benchmark problem, we address a slightly more complex version with the following additional constraints.

Table 3.1: Results for the navigation benchmarks. Abbreviations: Uninformed DFS: Uninformed depth-first search, Box-heuristic: box-based distance heuristic, PDB: our PDB cost function $cost^{PP}$, #loc: number of locations, #it: number of iterations, length: length of the found error trajectory, time: total time in seconds including any preprocessing. For our PDB approach, the fraction of the total time that is needed for the PDB computation is additionally reported in parenthesis.

Inst.	#loc	Uninformed DFS			Box-heuristic			PDB		
		#it	length	time	#it	length	time	#it	length	time (time abs.)
1	400	122	15	206.1	62	15	99.883	16	15	**28.325** (2.714)
2	400	183	33	262.565	86	33	168.815	34	33	**75.626** (10.153)
3	625	75	33	99.758	34	33	**52.222**	34	33	62.283 (10.234)
4	625	268	158	368.545	231	158	296.89	159	158	**178.705** (13.992)
5	625	85	79	167.502	26	25	**53.164**	26	25	58.417 (5.002)
6	625	96	53	155.458	101	53	148.448	54	53	**106.283** (13.267)
7	625	227	34	280.406	105	34	137.363	35	34	**66.315** (12.682)
8	625	178	25	371.8	86	25	192.71	26	25	**60.639** (9.609)
9	625	297	17	502.049	102	17	187.003	18	17	**42.785** (10.232)
10	625	440	30	753.488	136	30	282.914	31	30	**84.031** (18.114)
11	900	234	72	378.906	129	21	208.789	22	21	**45.085** (10.973)
12	900	317	43	473.785	174	61	277.467	44	43	**86.936** (21.097)
13	900	367	37	596.671	148	37	266.718	38	37	**97.456** (26.926)
14	900	411	32	608.962	278	32	419.827	33	32	**79.987** (14.934)
15	900	379	44	625.685	107	44	194.535	45	44	**97.138** (12.302)

First, we add inputs allowing perturbation of object coordinates, i. e., the system of differential equations is extended to: $\dot{x} = v + u$, $\dot{v} = A(v - v_d)$, $u_{min} \leq u \leq u_{max}$. Second, to make the search task even harder, the benchmark problems also feature obstacles between certain grid elements. This is particularly challenging because, in contrast to the original benchmark system, one can get stuck in a cell where no further transitions can be taken, and consequently, backtracking might become necessary. The size of the problem instances varies from 400 to 900 locations, and all instances feature 4 variables.

The results for the navigation benchmark problem class are provided in Table 3.1, where the best results are given in bold fonts with respect to the total runtime. The fraction of the total time to compute the PDB is given in parenthesis. As a general picture, they show that the precomputation time for the PDB mostly pays off in terms of guidance accuracy and overall runtime. Specifically, the overall runtime could (sometimes significantly) be reduced compared to uninformed search and also compared to the box-based heuristic. For example, in navigation instance 1, the precomputation for the

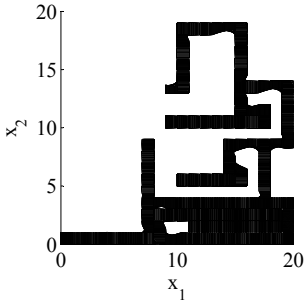


Fig. 3.2: Navigation benchmark: uninformed search error trajectory for instance 1.

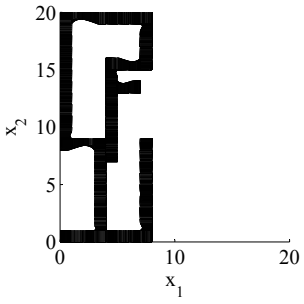


Fig. 3.3: Navigation benchmark: box-based heuristic search error trajectory for instance 1.

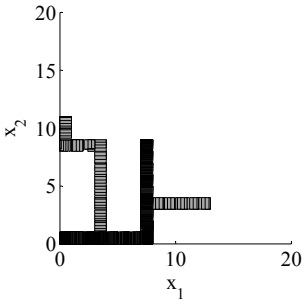


Fig. 3.4: Navigation benchmark: PDB search error trajectories for instance 1 (abstract: light gray, concrete: dark gray).

PDB only needs around 3 seconds, leading to an overall runtime of around 28 seconds, compared to around 99 seconds with the box-based heuristic and about 206 seconds with uninformed search. This search behavior for instance 1 is also visualized in Figure 3.2, Figure 3.3, and Figure 3.4, showing the trajectories (i. e., the parts of the covered region space) with the different search approaches. We observe the following: While uninformed depth-first search explores quite a large number of unnecessary trajectories, the box-based heuristic already guides the search more accurately and finds an error state with much fewer detours. Considering the PDB approach, we observe that PDBs can guide the search even more accurately in the sense that no detours are explored at all, and hence, no backtracking is needed either. Furthermore, the covered parts of the region space is again much lower than both with uninformed search and the box-based heuristic. In addition, we observe that even the abstract run (shown in light gray) is already rather accurate, covering only little more of the region space than the concrete run. Overall, the PDB approach finds an accurate balance between the computation time and the accuracy of the resulting cost function.

3.5.2 Results for Navigation Benchmarks with Additive Vanishing Perturbation

We consider another variant of the navigation benchmark with an additive vanishing perturbation (see, e.g., [49]). We use this variation for evaluating

the scalability of our approach with respect to increased continuous complexity given a constant discrete complexity. For this, the benchmark is modified to model a vanishing perturbation $w \in \mathbb{R}^p$ with increasing model order (i. e., $p = 1$, $p = 2$, etc.). In more detail, we extend the system of differential equations to $\dot{x} = v+u$, $\dot{v} = A(v-v_d)+\sum_{i=1}^{p} w_i$, $\dot{w} = A_w w$, where $u_{min} \leq u \leq u_{max}$ as before and $A_w \in \mathbb{R}^{p \times p}$ is Hurwitz to ensure it is a vanishing perturbation.

Table 3.2 presents results of the same scenarios evaluated in the earlier navigation benchmark for $p = 2$ additional vanishing perturbation variables (i. e., 2 additional state variables compared to the earlier navigation benchmark, yielding $n = 6$ continuous variables overall). We observe a similar picture as for the previous results: the PDB approach outperforms uninformed DFS and also the box-based heuristic in the majority of the problems. This is also reflected in Figs. 3.5, 3.6, and 3.7, which exemplary show the corresponding reachable states in the second instance for the three approaches, respectively.

In addition, Figure 3.8 presents the navigation benchmark instance 1 scaling the number of additional state variables from $p = 1$ through $p = 8$ (for a total of $n = 5$ through $n = 12$ continuous variables), while keeping all else constant, using a timeout of 30 minutes. We observe that also with increasing number of additional continuous variables, the runtime scalability of our PDB approach is considerably better compared to the box-based heuristic and uninformed DFS—even for $p = 8$ additional variables, PDBs is able to find an error state in less than 30 minutes. In contrast, both the uninformed DFS and box-based heuristic methods cannot find the error states in less than 30 minutes beyond $p = 3$ and $p = 5$ additional variables, respectively.

3.5.3 Results for Satellite Benchmarks

In this section, we consider benchmarks that result from *hybridization*. For a hybrid automaton $\mathcal{H}$ with nonlinear continuous dynamics, hybridization is a technique for generating a hybridized hybrid automaton from $\mathcal{H}$. The hybridized automaton has simpler continuous dynamics (usually affine or rectangular) that over-approximate the behavior of $\mathcal{H}$ [10], and can be analyzed by SpaceEx. For our evaluation, we consider benchmarks from this hybridization technique applied to nonlinear *satellite orbital dynamics* [48], where two satellites orbit the earth with nonlinear dynamics described by Kepler's laws. The orbits in three-dimensional space lie in a two-dimensional plane and may in general be any conic section, but we assume the orbits are periodic, and hence circular or elliptical. Fixing some orbital parameters (e.g., the orientations of the orbits in three-space), the states of the satellites in three-dimensional space $x_1, x_2 \in \mathbb{R}^3$ can be completely described in

Table 3.2: Results for the navigation benchmarks with two additional continuous variables modeling an additive vanishing perturbation. Abbreviations: OOT: out of time (max 30 minutes). Other abbreviations as in Table 3.1.

Inst.	#loc	Uninformed DFS			Box-heuristic			PDB		
		#it	length	time	#it	length	time	#it	length	time (time abs.)
1	400	122	15	709.416	62	15	368.986	16	15	**116.804** (4.448)
2	400	183	33	868.497	86	33	655.368	34	33	**274.823** (16.898)
3	625	75	33	373.06	34	33	**210.928**	34	33	228.217 (16.978)
4	625	268	158	1175.14	231	158	**960.585**	268	158	1187.27 (23.566)
5	625	85	79	563.22	26	25	**219.284**	26	25	227.956 (8.31)
6	625	96	53	536.192	101	53	535.117	54	53	**357.119** (22.137)
7	625	227	34	1061.49	105	34	549.549	35	34	**247.273** (22.147)
8	625	201	25	1496.8	89	25	818.356	26	25	**222.398** (15.639)
9	625	298	17	1734.63	102	17	708.366	18	17	**147.447** (16.875)
10	625	n/a	n/a	OOT	151	30	1223.23	31	30	**287.34** (29.607)
11	900	234	72	1288.58	129	21	787.449	22	21	**153.204** (18.309)
12	900	317	43	1624.18	174	61	995.995	161	43	**778.93** (34.855)
13	900	n/a	n/a	OOT	148	37	897.707	38	37	**311.623** (44.663)
14	900	n/a	n/a	OOT	279	32	1488.05	33	32	**280.072** (24.497)
15	900	n/a	n/a	OOT	107	44	687.813	45	44	**327.326** (20.426)

terms of their true anomalies (angular positions). Likewise, one can transform between the three-dimensional state description and the angular position state description. The nonlinear dynamics for the angular position are $\dot{\nu}_i = \sqrt{\mu/p_i^3}(1 + e_i \cos \nu_i)^2$ for each satellite $i \in \{1, 2\}$, where μ is a gravitational parameter, $p_i = a_i(1 - e_i^2)$ is the semi-latus rectum of the ellipse, a_i is the length of the semi-major axis of the ellipse, and $0 \leq e_i < 1$ is the eccentricity of the ellipse (if $e_i = 0$, then the orbit is circular and p_i simplifies to the radius of the circle). These dynamics are periodic with a period of 2π, so we consider the bounded subset $[0, 2\pi]^2$ of the state-space $\mathbb{R}^2$, and add invariants and transitions to create a hybrid automaton ensuring $\nu_i \in [0, 2\pi]$. For the benchmark cases evaluated, we fixed $\mu = 1$ and varied p_i and e_i for several scenarios. For more details, we refer to the work of Johnson et al. [48]. The size of the problem instances varies from 36 to 1296 locations, and all instances feature 4 variables.

The verification problem is *conjunction avoidance*, i. e., to determine whether there exists a trajectory where the satellites come too close to one another and may collide. Some of the benchmark instances considered are particularly challenging because they feature several sources of non-determinism, including several initial states and several bad states. As an additional source of non-determinism, some benchmarks model thrusting. A change in a satellite's orbit is usually accomplished by firing thrusters. This is usually modeled

Fig. 3.5: Navigation benchmark with additive vanishing perturbation for $p = 2$: uninformed search error trajectory for instance 2.

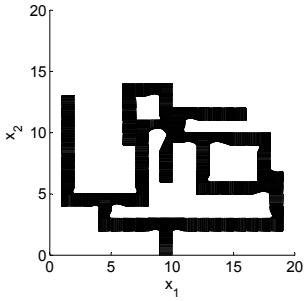


Fig. 3.6: Navigation benchmark with additive vanishing perturbation for $p = 2$: box-based heuristic search error trajectory for instance 2.

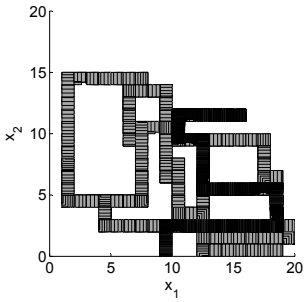


Fig. 3.7: Navigation benchmark with additive vanishing perturbation for $p = 2$: PDB search error trajectories for instance 2 (abstract: light gray, concrete: dark gray).

as an instantaneous change in the orbital parameters e_i and a_i. However, the angular position ν_i in this new orbit does not, in general, equal the angular position in the original orbit, and a change of variables is necessary, which can be modeled by a reset of the ν_i values when the thrusters are fired. The transitions introduced for thrusting add additional discrete non-determinism to the system.

The results for the satellite benchmark class are provided in Table 3.3. In general, we observe a similar search behavior to what we have observed in the navigation problems: The precomputation of the PDB pays off in the sense that much better search behavior can be achieved, leading to a fewer number of iterations and a lower overall runtime. For example, in instance 5, the precomputation time for the PDB amounts to roughly 5 seconds, leading to an overall time of around 92 seconds for the concrete run. In contrast, uninformed search and the box-based heuristic need around 426 and 272 seconds, respectively. The search behavior of the concrete and abstract run in instance 5 is also visualized in Figure 3.9, Figure 3.10, and Figure 3.11. We observe that the part of the covered search space with our PDB approach is again lower compared to the box-based heuristic and uninformed search. Figure 3.11 again particularly shows the part of the search space that is covered by the abstract run (which can be performed efficiently due to our abstraction described in Section 3.3.1), showing that our PDB approach finds

an accurate balance between the computation time and the accuracy of the resulting cost function.

Furthermore, we have also been able to effectively and efficiently prove the absence of errors in the instances 6 and 14, where the abstract run already revealed that no concrete error trajectory exists. As our abstraction is an over-approximation, we can safely conclude that no reachable error state in the concrete system exists either, and do not need to start the concrete search at all. Being able to efficiently verify hybrid automata with PDBs is a significant advantage compared to the box-based heuristic.

3.5.4 Results for Water-Tank Benchmarks

This benchmark consists of variants of the tank benchmark [5, 47]. The tank benchmark (see Figure 3.12) consists of some $N \in \mathbb{N}$ tanks, where each tank

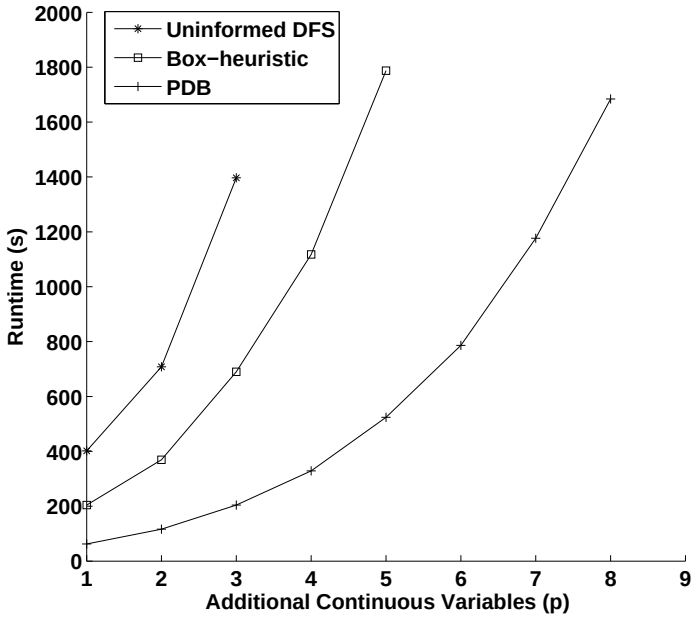


Fig. 3.8: Navigation benchmark with additive vanishing perturbation for $1 \leq p \leq 9$ with the same discrete structure as instance 1 (i. e., all else constant except the number of additive perturbation terms). Total number of continuous variables is $n = 4 + p$. Runs that exceeded the 30 minutes timeout are not plotted.

Table 3.3: Results for the satellite benchmarks. Abbreviations: OOT: out of time (max 30 minutes). Other abbreviations as in Table 3.1.

Inst.	#loc	Uninformed DFS			Box-heuristic			PDB		
		#it	length	time	#it	length	time	#it	length	time (time abs.)
1	36	116	32	37.501	75	10	18.393	16	10	**14.03** (10.05)
2	36	464	49	138.149	473	19	162.666	30	13	**21.882** (16.427)
3	64	719	87	42.198	281	91	**14.897**	264	121	27.067 (12.591)
4	100	111	106	51.705	45	15	30.602	23	14	**20.461** (8.106)
5	100	109	104	426.37	45	15	272.133	23	14	**92.393** (8.082)
6	159	2170	∞	107.68	1354	∞	68.107	0	∞	**21.051** (21.051)
7	324	580	135	251.016	1289	144	649.345	25	24	**42.316** (11.921)
8	557	1637	42	61.754	936	42	**35.523**	156	42	60.027 (54.115)
9	574	7113	41	298.601	561	10	23.977	14	10	**8.811** (8.309)
10	575	9092	4	376.935	387	5	16.467	15	4	**3.182** (2.651)
11	576	1485	775	273.265	253	13	50.172	15	13	**13.385** (7.899)
12	576	1005	775	172.192	796	13	160.342	15	13	**13.324** (7.841)
13	576	1317	1147	1410.17	484	54	775.325	52	51	**217.534** (104.304)
14	1293	13691	∞	526.483	7790	∞	312.288	0	∞	**170.428** (170.428)
15	1296	n/a	n/a	OOT	n/a	n/a	OOT	206	139	**784.986** (526.336)

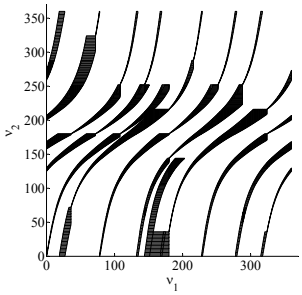


Fig. 3.9: Satellite benchmark: uninformed search error trajectory for instance 5.

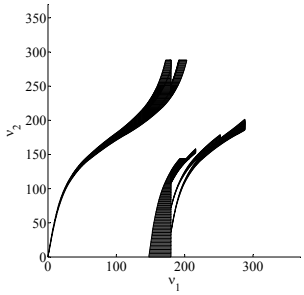


Fig. 3.10: Satellite benchmark: box-based heuristic search error trajectory for instance 5.

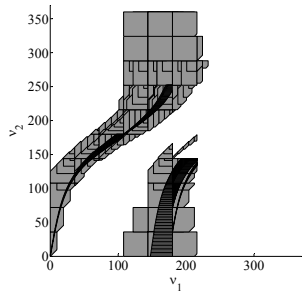


Fig. 3.11: Satellite benchmark: PDB search error trajectories for instance 5 (abstract: light gray, concrete: dark gray).

$i \in \{1, \ldots, N\}$ loses volume x_i at some constant flow rate v_i, so tank i has dynamics $\dot{x}_i = -v_i$, for a real constant $v_i \geq 0$. One of the tanks is filled from an external inlet at some constant flow rate w, so it has dynamics $\dot{x}_i = w - v_i$, for a real constant $w \geq 0$. In our variant, the volume lost by each tank simply vanishes and does not move from one tank to another. This benchmark class is qualitatively different than either the navigation or satellite benchmarks, as the discrete state space may be small.

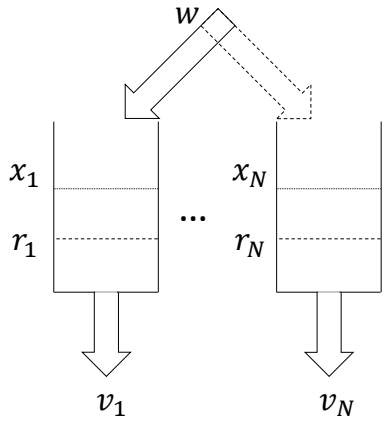


Fig. 3.12: Water tank benchmark with N tanks

The two variations we consider are complete and linear topologies with regard to the inlet tank choice. The inlet pipe w *may* be moved to some tank j with volume $x_i \leq r_i$ from some tank i, where: (a) $j \neq i$ is any other tank for the complete topology, or (b) $j \in \{i+1, i-1\}$ is an adjacent tank for the linear topology. The invariants in our variant of the benchmark are that the volumes of all tanks are non-negative: $\forall i \in \{1, \ldots, N\} : x_i \geq 0$. We consider variants where the aggregate out flow rate equals the in flow rate, so the sum of the flow rates out of all tanks equals the inlet flow rate: $w = \sum_{i=1}^{N} v_i$. Hence, the total volume is constant, so for all $t \geq 0$:

$$\sum_{i=1}^{N} x_i(t) = \sum_{i=1}^{N} x_i(0).$$

In these instances, the purpose of the inlet is to effectively move net volume between tanks, and the search problem is to find an appropriate order of such moves to reach a specific volume level in all of the N tanks.

The results for the water tank problem class are provided in Table 3.4. Again, the results are similar to the results in the navigation and the satellite benchmark classes: We observe that PDBs can help significantly in guiding the search towards error states. By comparing, e.g., Figure 3.13, Figure 3.14, and Figure 3.15, which show an execution of uninformed search, the box-based heuristic, and PDBs, respectively, we observe that our PDB-based approach is able to exploit the abstract run to more quickly find the correct sequence of tanks to fill to reach a certain region of the state-space (again, the light gray regions are covered in the abstract run only and can be computed efficiently). Generally, PDBs can particularly help for the water-tank

Table 3.4: Results for the tank benchmarks. Abbreviations: N: number of tanks (numbers of locations #loc and continuous variables), Top: topology (C=complete, L=linear), OOT: out of time (max 30 minutes). Other abbreviations as in Table 3.1.

Inst.	N	Top.	Uninformed DFS			Box-heuristic			PDB		
			#it	length	time	#it	length	time	#it	length	time (time abs.)
1	3	C	4	3	254.032	4	3	305.309	4	3	**166.516** (6.285)
2	3	C	4	3	237.33	4	3	238.311	4	3	**124.195** (4.71)
3	3	C	4	3	483.083	4	3	**479.847**	4	3	524.132 (41.855)
4	3	C	n/a	n/a	OOT	5	2	828.846	2	1	**190.139** (7.856)
5	3	C	6	5	508.693	8	2	76.68	3	2	**32.141** (11.351)
6	3	C	n/a	n/a	OOT	5	2	312.54	3	2	**220.108** (9.272)
7	3	L	6	5	506.076	5	2	281.351	3	2	**218.484** (7.791)
8	3	L	3	2	280.473	3	2	**276.498**	3	2	289.545 (6.156)
9	4	L	n/a	n/a	OOT	5	4	**6.171**	13	5	24.972 (8.688)
10	3	L	6	5	270.648	5	2	144.673	2	1	**41.345** (0.84)
11	4	L	18	6	30.95	11	6	23.81	4	3	**8.696** (3.199)
12	5	L	10	7	23.949	14	7	63.518	4	3	**18.138** (9.289)
13	6	L	39	27	64.091	28	21	51.595	4	2	**21.208** (6.283)
14	7	L	53	34	130.636	44	22	**117.732**	9	5	117.763 (9.536)
15	8	L	37	29	**108.7**	46	21	164.349	33	29	140.968 (30.271)

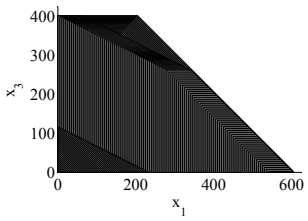


Fig. 3.13: Water tank benchmark: Uninformed search error trajectory for instance 10.

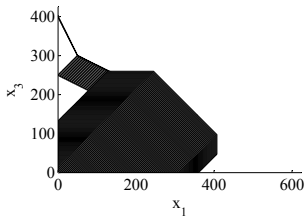


Fig. 3.14: Water tank benchmark: box-based heuristic search error trajectory instance 10.

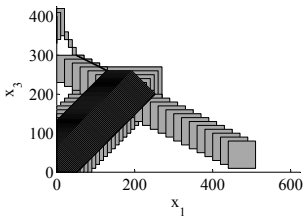


Fig. 3.15: Water tank benchmark: PDB search error trajectories instance 10 (abstract: light gray, concrete: dark gray).

problems because of the non-determinism that occurs in this problem class (which is important to be resolved accurately, corresponding to the choice of which tanks to fill in which order). However, we also observe that in 4 cases, the overall runtime is higher than the runtime with the box-based heuristic. In these cases, the precomputation of the PDB does not pay off – we will discuss such cases in more detail below.

Table 3.5: Results for the heater benchmarks. Abbreviations: OOT: out of time (max 30 minutes). Other abbreviations as in Table 3.1.

Inst.	#loc	Uninformed DFS			Box-heuristic			PDB		
		#it	length	time	#it	length	time	#it	length	time (time abs.)
1	4	7	6	**148.4**	12	6	212.651	7	6	149.117 (0.625)
2	4	9	8	305.21	5	0	146.311	1	0	**10.506** (7.982)
3	4	4	∞	27.476	4	∞	27.579	4	∞	**27.467** (0.042)
4	4	7	∞	178.781	7	∞	177.748	7	∞	**176.566** (1.1)
5	4	21	20	**84.779**	n/a	n/a	OOT	21	20	98.173 (12.303)
6	4	4	1	**1.284**	n/a	n/a	OOT	4	1	1.705 (0.4)
7	4	4	2	34.493	4	2	34.525	3	1	**32.07** (3.148)
8	4	7	6	**89.724**	49	48	907.52	7	6	90.778 (0.475)
9	4	4	2	8.772	3	2	**8.183**	3	1	8.28 (4.687)
10	4	5	4	**27.164**	15	8	65.249	5	4	27.851 (0.635)
11	4	13	8	25.771	25	14	48.844	12	8	**23.708** (0.435)
12	4	3	0	10.603	3	0	10.601	2	0	**8.212** (0.544)
13	4	n/a	n/a	OOT	n/a	n/a	OOT	10	6	**640.441** (240.583)
14	4	7	6	**58.533**	36	22	284.592	7	6	59.157 (0.55)
15	4	9	8	**38.06**	42	24	150.263	9	8	41.948 (3.752)

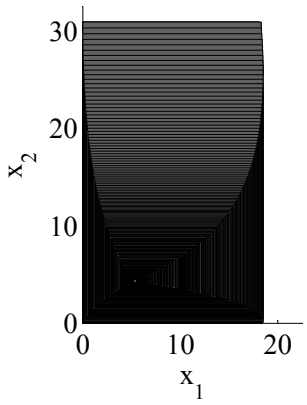


Fig. 3.16: Heater benchmark: Uninformed search error trajectory for for instance 2.

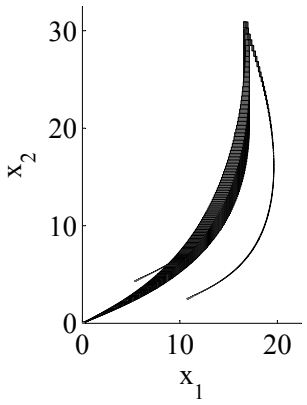


Fig. 3.17: Heater benchmark: box-based heuristic search error trajectory for instance 2.

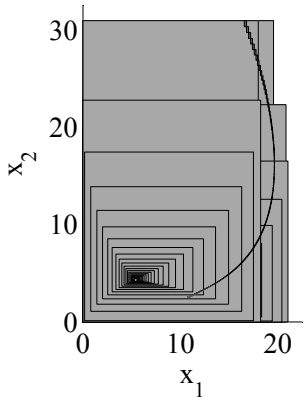


Fig. 3.18: Heater benchmark: PDB search error trajectories for instance 2 (abstract: light gray, concrete: dark gray).

3.5.5 Results for Heater Benchmarks

This benchmark consists of variants of the heater benchmark [35]. In our variation, we consider three rooms with one heater. The automaton is

modeled with four locations, consisting of no heaters on in any room, or the heater is on in one of the three rooms. The size of the problem instances have 4 locations, and all instances feature 3 temperature variables, 1 time variable, and 16 real constants. The temperature dynamics are linear and there is coupling between temperatures in different rooms. If the heater is on in a room, its temperature rate of change has a positive additive term c_i, but otherwise does not, so the temperature may decrease (subject to the temperatures in different rooms). Specifically, for room 1 (and symmetrically rooms 2 and 3), if the heater is on, the dynamics are: $\dot{x}_1 = b_1(u - x_1) + a_{1,2}(x_2 - x_1) + a_{1,3}(x_3 - x_1) + c_1$, but if the heater is off, the dynamics are the same except without the c_1 term. In our variant, the invariants specify only that the temperatures are all non-negative and bounded. The heater may be turned on in room $i \in \{1, 2, 3\}$ if $x_i \leq T_{on}$ for some real threshold T_{on}, and turned off if $x_i \geq T_{off}$ for some real threshold T_{off}. There is non-determinism in choosing to turn off or on the heater once the threshold condition is met, and there is a potential delay in changing the state of the heater from off to on and vice-versa.

The results for the heater benchmark are provided in Table 3.5. We observe that, unlike the results for the other benchmarks, the results for the heater are more diverse. While the PDB approach overall performs best in 7 out of 15 problem instances, it is somewhat slower than uninformed depth-first search (DFS) in other 7 instances. Having a closer look, we observe that the error trajectories with DFS are found with equally many iterations by SpaceEx, and additionally, their length is the same compared to those found with the PDB approach. In such cases where the PDB cannot improve over the search behavior of DFS, DFS is naturally more efficient because of the PDB's computational overhead (in fact, the difference in search time is almost exactly due to this overhead). However, obtaining such an informed search behavior with DFS is rather based on having good luck, whereas PDBs provide a more principled approach to achieve this. Furthermore, despite the sometimes higher runtimes in this benchmark class, we observe that our PDB approach is able to solve one more problem than DFS, and three more problems than the box-based heuristic within our time limit of 30 minutes. In addition, similar to the satellite benchmarks, we have been able to effectively prove the absence of errors in two cases (heater instance 3 and instance 4).

Finally, for the last time, let us have a look at the covered region space by DFS, by the box-based heuristic and by PDBs in Figure 3.16, Figure 3.17, and Figure 3.18, respectively. We observe that the concrete run with PDBs (indicated in dark grey) boils down to a small curve in this instance, whereas the other approaches cover a (much) larger fraction.

3.5.6 Runtimes of Partial PDBs vs. Full PDBs

Considering the runtime to build the partial PDBs compared to computing full PDBs, we observed that strong reductions of several orders of magnitude can indeed be obtained. In particular, the computation of the full abstract state space sometimes exceeds our time bound of 30 minutes, whereas the partial PDBs can still be computed efficiently. This happens in the water tank problem class, where full PDBs could not be computed within 30 minutes in any instance, whereas partial PDBs could be computed within less than a minute in all of the 15 instances (less than 10 seconds in 9 of these instances). In this respect, we conclude that the notion of partial PDBs particularly makes the overall approach tractable on a larger class of problems. In cases where full PDBs can be computed within 30 minutes, the runtime can be significantly higher than with partial PDBs: For example, in the satellite domain instance 10, computing the full PDB needs around 175 seconds, compared to roughly 3 seconds for computing the partial PDB.

3.5.7 Discussion

We have observed that PDBs can provide more informed search behavior than uninformed search or than the box-based heuristic. A potential problem is the computational overhead due to its precomputation time. We will discuss advantages and drawbacks of our PDB approach in this section.

As a general picture, we first observe that the number of iterations of SpaceEx and also the length of the found error trajectories are mostly at most as high with PDBs as with uninformed search and the box-based heuristic. In particular, our PDB approach could solve several problem instances where uninformed search and the box-based heuristic ran out of time. In some cases, the precomputation of the PDB does not pay off compared to DFS and the box-based heuristic – however, in such cases, the pure concrete search time with PDBs is still mostly similar to the pure search time of DFS and the box-based approach.

We further observe that the length of the trajectories found by the box-based heuristic and the PDB heuristic is often similar or equal, while the number of iterations is mostly decreased. This again shows that the search with the PDB approach is more focused than with the box-based heuristic in such cases, and less backtracking is needed. In particular, the box-based heuristic always tries to find a “direct” trajectory to an error state, while ignoring possible obstacles. Therefore, the search can get stuck in a dead-end state if there is an obstacle, and as a consequence, backtracking becomes necessary. Furthermore, the box-based heuristic can perform worse than the

PDB if several bad states are present. In such cases, the box-based heuristic might "switch" between several bad states, whereas the better accuracy of the PDB heuristic better focuses the search towards one particular bad state. In problems that are structured more easily (e. g., where no "obstacles" exist and error states are reachable "straight ahead"), the box-based heuristic might yield better performance because the precomputation of the PDB does not pay off.

Finally, a general advantage of PDBs compared to the box-based heuristic which we did not discuss in detail so far is the broader applicability of PDBs. By definition, the box-based heuristic estimates distances by computing Euclidean distances between the region of the current and the error state. However, in problems where error states are defined solely by (discrete) locations, there is no such error region, and the box-based distance heuristic is not effectively applicable. In contrast, PDBs are more general, and applicable for all kinds of error states.

3.6 Conclusion

In this section, we have introduced two heuristics to guide the search in the state space of hybrid automata. The first one, the box-based heuristic, works by estimating the distance between the center of the enclosing box of the considered region and the center of the box enclosing the bad region. We have shown that for a particular class of hybrid systems, this metric is an appropriate approximation of an idealized trajectory metric. Our experimental evaluation additionally shows that this metric can serve as an informed cost heuristic even for richer classes of hybrid systems. In particular, it has shown good results on systems with mainly continuous behavior. We end up with such systems, e.g., as a result of hybridization process, i.e., when some complex continuous dynamic is approximated by simpler one by state space partitioning.

The second heuristic applies coarse-grained space abstractions to compute pattern databases (PDBs) for hybrid systems. For a given safety property and hybrid system with linear dynamics in each location, we compute an abstraction by coarsening the over-approximation SpaceEx computes in its reachability analysis. The abstraction is used to construct a PDB, which contains abstract symbolic states together with their abstract error distances. These distances are used in guiding SpaceEx in the concrete search. Given a concrete symbolic state, the guiding heuristics returns the smallest distance to the error state of an enclosing abstract symbolic state. This distance is used to choose the most promising concrete symbolic successor. In

our implementation, we have taken advantage of the SpaceEx parametrization support, and were able to report a significant speedup in counterexample detection and even for verification. Our new PDB support for SpaceEx can be seen as a nontrivial extension of our previous work on guided reachability analysis for hybrid systems where the discrete system structure was ignored completely [20]. For the future, it will be interesting to further refine and extend our approach by, e. g., considering even more fine grained abstraction techniques, or by combinations of *several* abstraction techniques and therefore, by combining several PDBs. We expect that this will lead to even more accurate cost functions and better model checking performance.

4

Assume Guarantee Abstraction Refinement for Hybrid Automata

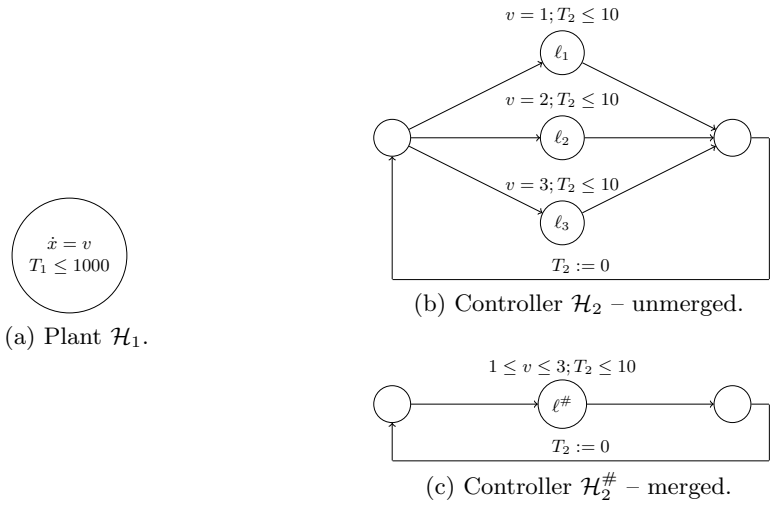
Assume-guarantee (AG) reasoning [61] is a well-known methodology for the verification of large systems. The idea behind is to decompose the verification of a system into the verification of its components, which are smaller and therefore easier to verify. A typical example of such systems would be a system comprised of a controller and a plant. In this work, we mainly concentrate on hybrid automata [1] with *stratified* controllers, i.e., controllers consisting of multiple strata (layers), where each of them is responsible for some particular plant parameter. Assume-guarantee reasoning can be performed using the following rule, ASym, where P is a safety property and $\mathcal{H}_1 \parallel \mathcal{H}_2$ denotes the parallel composition of components $\mathcal{H}_1$ and $\mathcal{H}_2$, where $\mathcal{H}_1$ is a plant and $\mathcal{H}_2$ is a controller.

$$\frac{\begin{array}{l}1 : \mathcal{H}_1 \parallel A \models P \\ 2 : \mathcal{H}_2 \models A\end{array}}{\mathcal{H}_1 \parallel \mathcal{H}_2 \models P}$$

Rule ASym

In this rule, A denotes an *assumption* about the controller of $\mathcal{H}_1$. Premise 1 ensures that when $\mathcal{H}_1$ is a part of a system that satisfies A, the system also guarantees P. Premise 2 ensures that any system that contains $\mathcal{H}_2$ satisfies A. Together the two premises imply the conclusion of the rule. The rule ASym is applicable if the assumption A is more abstract than $\mathcal{H}_2$, but still reflects $\mathcal{H}_2$'s behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for $\mathcal{H}_1$ to satisfy P in premise 1.

The most challenging part of applying assume-guarantee reasoning is to come up with appropriate assumptions to use in the application of the assume-guarantee rules. Several learning and abstraction-refinement tech-



(a) Plant $\mathcal{H}_1$.

(b) Controller $\mathcal{H}_2$ – unmerged.

(c) Controller $\mathcal{H}_2^{\#}$ – merged.

Fig. 4.1: A motivating example

niques [16, 57] have been proposed for automating the generation of assumptions for the verification of transition systems.

In this chapter, we focus on the automated generation of assumptions in the context of hybrid automata. Similar to the work by Bobaru et al. [16] we use abstraction-refinement techniques to iteratively build the assumptions for the rule ASym. In our case, $\mathcal{H}_2$, i.e., the controller of $\mathcal{H}_1$, is abstracted. The use of over-approximations guarantees that the assumption describes the component correctly and hence premise 2 holds by construction. However, it is possible that premise 1 does not hold, in which case a counterexample is provided. The counterexample is analyzed to see if it is spurious, in which case the abstraction of $\mathcal{H}_2$ is refined to eliminate it. If the counterexample is real, then $\mathcal{H}_1 \parallel \mathcal{H}_2$ violates P.

We present a framework which can efficiently handle the class of affine hybrid automata [19]. Due to the mixed discrete-continuous nature of hybrid automata, we need to pay special attention on the abstraction of continuous dynamics. We illustrate the idea of our compositional analysis on a toy example. Figure 4.1 shows a simple hybrid automaton consisting of the plant $\mathcal{H}_1$ in Figure 4.1a and controller $\mathcal{H}_2$ in Figure 4.1b. We observe that the derivative of variable x in plant $\mathcal{H}_1$ depends on the value of v governed by the controller $\mathcal{H}_2$. Furthermore, we see that the controller operates in iterations of length 10. The possible controller options are grouped in a stratum. While analyzing this system, a hybrid model checker will consider all the three options on every controller iteration which results in 3^n branches for n iterations. By noting that for some properties only the minimal and max-

imal values of v are of relevance, we come up with an abstracted version of the automaton $\mathcal{H}_2$ in Figure 4.1c. We replace the three alternative options by only one *coarser* option. To ensure that the resulting automaton is indeed an over-approximation of the original system, we use $1 \leq v \leq 3$ as an invariant of the merged location $\ell^{\#}$, i.e., we replace the exact values of v with its bounds. This abstraction will be especially useful to prove, e.g., that within the first 1000 seconds of system operation the state $x = 4000$ will still not be reached. In the abstraction we will reduce an exponential number of branchings to a linear one. Note that this kind of location-merging abstractions is especially useful for the class of stratified controllers. The reason is that the controller structure can be exploited to efficiently generate an initial abstraction by merging locations belonging to the same stratum. Intuitively, this step allows us to adjust the precision level at which the system parameters are taken into account. If the resulting abstraction is too coarse, a finer-grained abstraction is generated in the refinement step.

The lesson we learn from this example is that merging of locations is a promising approach to generate abstractions in scope of the assume-guarantee reasoning paradigm. To ensure the conservativeness of the resulting abstraction, we compute the invariants as a convex hull of the original locations. Note that the computation of minimal and maximal values of v shown above represents a simple case of a general convex hull computation. Given the continuous, affine dynamics of the form $\dot{x}(t) = Ax(t) + u(t)$, the merged locations are computed by first eliminating the (unprimed) state variables and consequently computing the convex hull of the resulting polytopes over the derivatives. As outlined above, sometimes we might end up with *spurious* counterexamples. To overcome this issue we proceed to the phase of spuriousness checking. If the found path is indeed spurious, we refine the system by splitting one or multiple locations and continue with the analysis of this new system. Note that the assume-guarantee reasoning methodology is a variant of the CEGAR approach [25]. The essential difference of AGAR compared to CEGAR is the compositional handling of the system. We develop our approach along these lines by ensuring that the proposed algorithms work in the compositional fashion, e.g., we only abstract a part of the system and the refinement algorithm considers a projection of the found counterexample on the abstracted component. Our implementation in SpaceEx [39] shows the practical potential.

The remainder of the chapter is organized as follows. In Section 4.1, we introduce our compositional framework. This is followed by a discussion about related work in Section 4.2. Afterwards, we present our experimental evaluation in Section 4.3. Finally, we conclude the chapter in Section 4.4.

4.1 Compositional Framework for Hybrid Automata

In this section, we introduce the main ingredients of our compositional framework: the abstraction of a hybrid automaton, an algorithm for spuriousness check, and a refinement algorithm.

4.1.1 Abstraction Algorithm

We construct our abstraction by partially merging system locations. To formally define the abstraction, we introduce a location abstraction function α and a location concretization function α^{-1} as follows.

Definition 4.1 (Location abstraction function). *Location abstraction function $\alpha : Loc \to Loc^{\#}$ provides a mapping from every concrete location in Loc to its abstract counterpart. Furthermore, we require $|Loc^{\#}| \leq |Loc|$, i.e., the abstract system should have at most the same number of locations as the original one.*

Definition 4.2 (Location concretization function). *Location concretization function $\alpha^{-1} : Loc^{\#} \to 2^{Loc}$ provides a mapping from every abstract location in $Loc^{\#}$ to the set of concrete locations which were merged into it.*

If $\ell \in \alpha^{-1}(\ell^{\#})$, then ℓ is a *corresponding* location to the abstract location $\ell^{\#}$. Furthermore, we abuse the notation and apply a concretization function not only to abstract locations, but also to abstract symbolic states and abstract symbolic paths. We define an abstract hybrid automaton $\mathcal{H}^{\#}$ induced by the location abstraction function α and concrete hybrid automaton $\mathcal{H}$ as follows:

Definition 4.3 (Location-merging abstraction).
Let $\mathcal{H} = (Loc, Var, Init, Flow, Trans, I)$ be a hybrid automaton and $\alpha : Loc \to Loc'$ be a location abstraction function. The abstract automaton $\mathcal{H}^{\#} = (Loc^{\#}, Var^{\#}, Init^{\#}, Flow^{\#}, Trans^{\#}, I^{\#})$ induced by the location-merging abstraction with respect to the location function α is defined as follows:

- *$Loc^{\#} = Loc'$, i.e., the location abstraction function provides which locations of $\mathcal{H}$ are to be merged. We assume that α keeps the bad location ℓ_{bad} as a singleton.*
- *$Var^{\#} = Var$, i.e., the abstraction preserves the continuous variables of the original system.*
- *$\forall \ell^{\#} \in Loc^{\#} : Init^{\#}(\ell^{\#}) = \mathcal{CH}(\bigcup_{\ell \in \alpha^{-1}(\ell^{\#})} Init(\ell))$, i.e., the regions describing the initial values in concrete locations are first merged into one*

(possibly non-convex) set and afterwards are over-approximated by a convex hull.
Note that if an abstract location is a singleton and $Init(\ell)$ is a convex set, the application of the convex hull operator results in the original set.

- $\forall \ell^{\#} \in Loc^{\#}$:

$$Flow^{\#}(\ell^{\#})(x, \dot{x}) = \begin{cases} \mathcal{CH}(\bigcup_{\ell \in \alpha^{-1}(\ell^{\#})} F_{\ell}), & |\alpha^{-1}(\ell^{\#})| > 1 \\ Flow(\alpha^{-1}(\ell^{\#}))(x, \dot{x}), & |\alpha^{-1}(\ell^{\#})| = 1 \end{cases}$$

  where $F_{\ell} = \exists x : (Flow(\ell)(x, \dot{x}) \wedge I(\ell)(x))$.
- $Trans^{\#} = \{(\ell^{\#}, g, \xi, \hat{\ell}^{\#}) | \exists \ell \in \alpha^{-1}(\ell^{\#}), \hat{\ell} \in \alpha^{-1}(\hat{\ell}^{\#})$ *s.t.* $(\ell, g, \xi, \hat{\ell}) \in Trans\}$, *i.e., an abstract transition between* $\ell^{\#}$ *and* $\hat{\ell}^{\#}$ *is added when a transition in the concrete state space connecting the corresponding locations exists.*
- $\forall \ell^{\#} \in Loc^{\#} : I^{\#}(\ell^{\#}) = \mathcal{CH}(\bigcup_{\ell \in \alpha^{-1}(\ell^{\#})} I(\ell))$, *i.e., similarly to the initial regions, the invariants are merged and over-approximated by a convex hull.*

Dynamics:
$\dot{x} = 2x + 3y$
$\dot{y} = 4x - 5y$

Invariant:
$0 \leq x \leq 1$
$\wedge\ 0 \leq y \leq 1$

F_1:
$-5\dot{x} - 3\dot{y} \leq 0$
$\wedge\ -22 + 5\dot{x} + 3\dot{y} \leq 0$
$\wedge\ -2\dot{x} + \dot{y} \leq 0$
$\wedge\ -11 + 2\dot{x} - \dot{y} \leq 0$

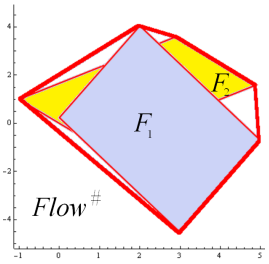
(a) Location ℓ_1.

Dynamics:
$\dot{x} = -x + 3y + 5$
$\dot{y} = \ \ x + 2y$

Invariant:
$1 \leq x \leq 3$
$\wedge\ -1 \leq y \leq 0.3$

F_2:
$-5 + 2\dot{x} - 3\dot{y} \leq 0$
$\wedge\ -5 - 2\dot{x} + 3\dot{y} \leq 0$
$\wedge\ -\dot{x} - \dot{y} \leq 0$
$\wedge\ -6.5 + \dot{x} + \dot{y} \leq 0$

(b) Location ℓ_2.



(c) Convex Hull.

Fig. 4.2: Elimination of unprimed variables before merging of the locations.

In other words, we merge the dynamics of multiple locations in two steps. We first over-approximate the original dynamics in every concrete location by quantifying away unprimed variables, i.e., we obtain a constraint reasoning only about derivatives (see Figure 4.2). Secondly, we define abstract dynamics by constructing a convex hull of the constraints computed in the first step. If an abstract location is a singleton, i.e., $|\alpha^{-1}(\ell^{\#})| = 1$, we just keep its original dynamics.

Algorithm 3 Compositional analysis of $\mathcal{H}_1||\mathcal{H}_2$

Input: Hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$
Output: Is the composed system $\mathcal{H}_1||\mathcal{H}_2$ safe?
1: $\mathcal{H}_2^{\#}$:= ConstructAbstraction ($\mathcal{H}_2$)
2: **while** *true* **do**
3: $\pi^{\#}$:= Analysis ($\mathcal{H}_1||\mathcal{H}_2^{\#}$)
4: **if** $\pi^{\#}$ is empty **then**
5: **return** "System is safe"
6: **else**
7: $\mathcal{SP}$:= SpuriousnessAnalysis ($\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_2^{\#}, \pi^{\#}$)
8: **if** $\mathcal{SP}$ is empty **then**
9: **return** "System is unsafe"
10: **else**
11: $\mathcal{H}_2^{\#}$:= Refinement ($\mathcal{H}_2^{\#}, \mathcal{SP}$)
12: **end if**
13: **end if**
14: **end while**

We observe that by construction the set of reachable states of the abstract automaton $\mathcal{H}^{\#}$ leads to an over-approximation compared to the states reachable by the concrete automaton $\mathcal{H}$. Therefore, the following proposition holds:

Proposition 4.4. *Let $\mathcal{H}^{\#}$ be a location-merging abstraction of the concrete hybrid automaton $\mathcal{H}$. Then the non-reachability of the bad location ℓ_{bad} in $\mathcal{H}^{\#}$ implies its non-reachability also in the concrete automaton $\mathcal{H}$.*

4.1.2 Compositional Analysis

Our compositional analysis is illustrated in Algorithm 3. In order to simplify the presentation we consider a case of a system consisting of two components $\mathcal{H}_1$ and $\mathcal{H}_2$, where $\mathcal{H}_1$ is a plant and $\mathcal{H}_2$ is a controller. However, the scheme is applicable to systems with more than two components [16].

In the following we provide a conceptual description of the algorithm. The algorithm checks whether the bad state S_{bad} can be reached by the system $\mathcal{H}_1||\mathcal{H}_2$.The algorithm starts by computing an abstraction of $\mathcal{H}_2$ in the function ConstructAbstraction (line 1). For more details on the abstraction construction see Section 4.1.1. The algorithm iteratively refines the original abstraction (lines 2–14). Note that in the worst case we will end up with the original system. However, in many cases we will need to refine only a part of the system (see Section 4.3 for the detailed discussion). In every refinement iteration the algorithm proceeds as follows. First, the state space of the abstract system $\mathcal{H}_1||\mathcal{H}_2^{\#}$ is analyzed in the function Analysis (line 3). This function returns an abstract bad path or "empty" if no such path has

Algorithm 4 Spuriousness analysis

Input: Concrete automaton $\mathcal{H}_1$, concrete automaton $\mathcal{H}_2$ and its abstract version $\mathcal{H}_2^\#$ and abstract bad path $\pi^\# = s_0^\#, \ldots, s_{m-1}^\#$ in the state space of $\mathcal{H}_1 || \mathcal{H}_2^\#$.
Output: Information about the possible splitting points store or empty set if the abstract bad path $\pi^\#$ is concretizable

```
1: SP := ∅
2: Push (L_waiting, (α^{-1}(s_0^#) ∩ S_init(H_1||H_2), 0))
3: while L_waiting ≠ ∅ do
4:    (s_curr, i) := GetNext (L_waiting)
5:    s'_curr := ContSuccessors (s_curr)
6:    Push (L_passed, s'_curr)
7:    if i = m − 1 then
8:       if s'_curr is a symbolic error state then
9:          return empty set, i.e., concrete bad state found
10:      else
11:         Store the abstract bad path π^# and the corresponding concrete path π ending
            in s'_curr into SP
12:      end if
13:   end if
14:   S' := DiscreteSuccessors (s'_curr) ∩ α^{-1}(s_{i+1}^#)
15:   if S' is empty then
16:      Store the abstract bad path π^# and the corresponding concrete path π ending in
         s'_curr into SP
17:   else
18:      Push (L_waiting, S' \ L_passed, i + 1)
19:   end if
20: end while
21: return SP
```

been found. If no abstract bad path has been found, we can conclude that also the original system is safe as we consider only over-approximations (line 5). Otherwise, the algorithm proceeds in the function SpuriousnessAnalysis (line 7) with the spuriousness analysis of the found abstract bad path $\pi^\#$. The function SpuriousnessAnalysis returns the information on how to refine $\mathcal{H}_2^\#$ or "empty" if the abstract path $\pi^\#$ can be concretized. In the latter case, we exit with status "System is unsafe" (line 9). Otherwise, $\mathcal{H}_2^\#$ is refined in the function Refinement based on the structure of the abstract bad path gained during the spuriousness analysis.

4.1.3 Spuriousness Check

In this section, we consider the function SpuriousnessAnalysis (see Algorithm 4) in more detail. Given an abstract bad path $\pi^\# = s_0^\#, \ldots, s_{m-1}^\#$, the function enumerates concrete paths corresponding to $\pi^\#$ and looks for the ones which end up in a bad state. The enumeration of concrete paths of

the composed automaton $\mathcal{H}_1||\mathcal{H}_2$ along the abstract path $\pi^\#$ is organized in a breadth-first fashion. In particular, we make use of two lists: $\mathcal{L}_{waiting}$ and $\mathcal{L}_{passed}$. $\mathcal{L}_{waiting}$ stores symbolic states which still have to be considered and $\mathcal{L}_{passed}$ stores symbolic states which have already been considered and thus do not have to be visited again. The data structure $\mathcal{SP}$ stores information relevant for the refinement step. In particular, tuples $(\pi^\#, \pi)$, where π is a path in the concrete state space which does not belong to $\alpha^{-1}(\pi^\#)$, are kept in $\mathcal{SP}$. In other words, in the last symbolic state $s_{|\pi|-1}$ of π we cannot take any discrete transition which would lead to some concrete state represented by an abstract state $s^\#_{|\pi|}$. Therefore, a tuple $(\pi^\#, \pi)$ essentially provides a possible *reason* for the spuriousness of π with respect to $\pi^\#$. We will use this information to refine the abstract component $\mathcal{H}_2^\#$ (see Section 4.1.4).

The algorithm starts by pushing the concrete initial states which correspond to the first abstract symbolic state $s_0^\#$ in $\mathcal{L}_{waiting}$ (line 2). It is important to mention that α^{-1} concretizes only the part of the symbolic state relevant to $\mathcal{H}_2^\#$. This property also holds for the algorithm described in Section 4.1.4. Note that we furthermore store the position of the abstract state which corresponds to the considered concrete symbolic state in the waiting list (we start with $s_0^\#$ and thus the position is 0). We will consequently use this information to compute the discrete symbolic successors of a given symbolic state which correspond to the analyzed bad path $\pi^\#$. In lines 3–20 the concrete state space is iteratively explored in a breadth-first manner. Every iteration consists of the following steps. First, the next tuple (s_{curr}, i) is picked from the waiting list $\mathcal{L}_{waiting}$ (line 4), where s_{curr} is a symbolic state and i shows its position with respect to the abstract path. Afterwards, the continuous successor, i.e., a symbolic state reflecting the states reachable according to the continuous dynamics, is computed and added to the passed list $\mathcal{L}_{passed}$ (lines 5–6). If the end of the abstract path is reached then the intersection with the bad state is checked (lines 8–10). If the end of the abstract path is reached, but no intersection with the bad state is detected, we store both the abstract and concrete paths in $\mathcal{SP}$ in order to use this information in the refinement step. If the algorithm is still in the middle of the abstract bad path, it moves on to the computation of the concrete symbolic states which correspond to the abstract bad path (line 14). We achieve this by computing discrete successors and intersecting them with the concrete states represented by the next symbolic state on the abstract path. Note that the position i allows the algorithm to easily find the next abstract symbolic state on the path with respect to the currently considered concrete state.

If the set of discrete successors is empty, we say that a possible *splitting point* has been found. In other words, we could refine the abstract location $\ell_i^\#$ of $s_i^\# = (\ell_i^\#, R_i^\#)$ by splitting it (see Section 4.1.4). We store the abstract bad path and the concrete path we have considered up to now into $\mathcal{SP}$ (line 16). Otherwise, we add the discrete state into the waiting list $\mathcal{L}_{waiting}$ (line 18). After having analyzed all concrete paths corresponding to $\pi^\#$, the function SpuriousnessAnalysis returns $\mathcal{SP}$. It is only possible to report that the considered abstract bad path is not concretizable after having considered all possible concrete paths corresponding to it. Thus, the algorithm does not stop after discovering a particular splitting point, but just stores it for the later reuse during the refinement.

While mapping an abstract bad path to a concrete one, Algorithm 4 refers to the functions ContSuccessors and DiscreteSuccessors which are applied to concrete symbolic states. Thus, if the function SpuriousnessAnalysis declares some abstract bad path $\pi^\#$ to be genuine by finding its concrete counterpart π, then we can automatically conclude that the standard SpaceEx reachability algorithm would also have reported π to be a bad path. Therefore, our framework provides the same level of precision as the standard SpaceEx reachability algorithm. Finally, we note that the full concretization of a symbolic path is known to be a highly nontrivial problem. Once a concrete symbolic bad path is found with our approach, further concretization to hybrid automaton trajectories can be achieved using techniques from optimal control such as the one proposed in the work by Zutshi et al. [70].

4.1.4 Refinement Algorithm

The refinement algorithm Refinement uses $\mathcal{SP}$ in order to appropriately refine the abstraction $\mathcal{H}_2^\#$ in a compositional way. The data structure $\mathcal{SP}$ contains information about multiple possible splitting points. For the refinement we choose a tuple $(\pi^\#, \pi_{max}) \in \mathcal{SP}$ which *maximizes* the length of the concrete path π over all the elements of $\mathcal{SP}$. Intuitively, by choosing a tuple with this property, we ensure that π_{max} cannot be extended for all concrete paths which correspond to $\pi^\#$. Let the abstract bad path $\pi^\# = s_0^\#, \ldots, s_i^\#, \ldots, s_n^\#$ and the concrete path $\pi_{max} = s_0, \ldots, s_i, \ldots, s_m$ $(m \leq n)$, where $s_i = (\ell_i, R_i)$ and $s_i^\# = (\ell_i^\#, R_i^\#)$. Furthermore, $\ell_i = (\ell_i^{(1)}, \ell_i^{(2)})$, where $\ell_i^{(1)}$ and $\ell_i^{(2)}$ are locations of $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. The location of the abstracted composed automaton $\mathcal{H}_1 || \mathcal{H}_2^\#$ is given by the tuple $\ell_i^\# = (\ell_i^{(1)}, \ell_i^{\#(2)})$. Depending on the location partitioning of $\mathcal{H}_2^\#$ the refinement algorithm distinguishes three cases:

1. $|\alpha^{-1}(\ell_m^{\#(2)})| > 1$, i.e., the abstract location corresponding to the last concrete location can be split:
   The refinement algorithm proceeds by splitting the abstract location $\ell_m^{\#(2)}$ of $\mathcal{H}_2^{\#}$ into two locations: $\alpha^{-1}(\ell_m^{\#(2)}) \setminus \ell_m^{(2)}$ and $\ell_m^{(2)}$, where $\ell_m^{(2)}$ is a location of $\mathcal{H}_2$ corresponding to the concrete symbolic state $s_m = ((\ell_m^{(1)}, \ell_m^{(2)}), R_m)$.
2. $|\alpha^{-1}(\ell_m^{\#(2)})| = 1$ and $|\alpha^{-1}(\ell_{m+1}^{\#(2)})| > 1$, i.e., the abstract location of $\mathcal{H}_2^{\#}$ corresponding to the last concrete location cannot be split, whereas the successor abstract location still comprises multiple locations:
   The refinement algorithm splits $\ell_{m+1}^{\#(2)}$ into $\alpha^{-1}(\ell_{m+1}^{\#(2)}) \setminus \ell'$ and ℓ', where $\ell' = \{\ell | \ell \in \ell_{m+1}^{\#(2)},\ \ell$ is a target location of discrete transition from $\ell_m^{\#(2)}\ \}$.
   In other words, we look for locations in $\ell_{m+1}^{\#(2)}$ which have incoming transitions from $\ell_m^{\#(2)}$ and split them apart. Note that in this case we do not look at the transition guard and any other continuous artifacts.
3. $|\alpha^{-1}(\ell_m^{\#(2)})| = 1$ and $|\alpha^{-1}(\ell_{m+1}^{\#(2)})| = 1$, i.e., neither the abstract location corresponding to the last concrete location nor its successor can be split:
   The algorithm iterates over the abstract path and looks for a abstract state in $\mathcal{H}_2^{\#}$ with a location which still can be split, i.e., we look for i s.t. $i < m \wedge |\alpha^{-1}(\ell_i^{\#(2)})| > 1$. The location $\ell_i^{\#(2)}$ is split into locations $\alpha^{-1}(\ell_i^{\#(2)}) \setminus \ell_i^{(2)}$ and $\ell_i^{(2)}$, where $\ell_i^{(2)}$ is a location of $\mathcal{H}_2$ corresponding to $s_i = ((\ell_i^{(1)}, \ell_i^{(2)}), R_i)$.

Therefore, during the refinement process, we only refer to the locations of the abstracted component $\mathcal{H}_2^{\#}$, i.e., we consider the projection of the found path to $\mathcal{H}_2^{\#}$. The refinement algorithm as described above also has a progress property:

Proposition 4.5 (Progress property). *The size of the location partitioning increases by one location after every application of the refinement algorithm over cases 1–3.*

Proof. By construction, the number of locations in $\mathcal{H}_2^{\#}$ increases by one in cases 1 and 2 after every refinement iteration. In case 3 the refinement can be only done under the assumption that there exists an index i s.t. $i < m \wedge |\alpha^{-1}(\ell_i^{\#(2)})| > 1$ holds. This statement is true as the opposite would mean that the whole abstract bad path $\pi^{\#}$ only consists of *concrete* states. This in turn would lead to the fact that $\pi^{\#}$ is already a concrete path to the bad state. The function REFINEMENT is, however, called only for abstract bad paths which were found to be spurious. □

This proposition lets us conclude that Algorithm 3 terminates after a finite number of iterations after having considered the original system in the

worst case. By combining this result with Proposition 4.4 and rule ASym, we can derive the following soundness and relative completeness results:

Theorem 4.6 (Soundness). *If our compositional framework is able to prove that $\mathcal{H}_1||A$ cannot reach the (abstract) error states, then the composition $\mathcal{H}_1||\mathcal{H}_2$ is safe, that is, it cannot reach the (concrete) error states.*

Theorem 4.7 (Relative Completeness). *If our compositional framework is able to find a symbolic error path in $\mathcal{H}_1||A$ which is not spurious, then there exists a concrete symbolic error path in the composition $\mathcal{H}_1||\mathcal{H}_2$, too.*

The existence of a symbolic error path does not necessarily imply the existence of an error trajectory (due to the undecidability of the reachability problem for affine hybrid automata). This is why we call the above result (for symbolic paths) relative completeness.

4.2 Related Work

The framework developed by Pasareanu et al. [57] enables automated compositional verification using rule ASym. In that work, both assumptions and properties are expressed as finite state automata. The framework uses the L* [9] automata-learning algorithm to iteratively compute assumptions in the form of deterministic finite-state automata. Other learning-based approaches for automating assumption generation for rule ASym have been suggested as well [7]. All these approaches were done in the context of transition systems, not for hybrid automata as we do here.

Several ways to compute abstractions of hybrid automata have been proposed. Alur et al. [2] propose to use a variant of predicate abstraction to construct a hybrid automaton abstraction. In a slightly different setting, Tiwari [67] suggests to use Lie derivatives to generate useful predicates. Both mentioned approaches essentially reduce the analysis of a hybrid automaton to the level of a discrete transition system. Jha et al. [46] partially eliminate continuous variables in the system under consideration. Prabhakar et al. [62] propose the use of CEGAR for initialized rectangular automata (IRA), where the abstractions reduce the complexity of both the continuous and the discrete dynamics. In this thesis, we use a similar idea, but apply it to the more general class of affine hybrid automata, and even more importantly, we extend it to a compositional verification framework. Finally, Doyen et al. [33] take an affine automaton, and, through hybridization, obtain its abstraction in the form of a rectangular automaton with larger discrete space. We do the opposite: we take an affine automaton, and construct a much smaller linear hybrid automaton.

4.3 Evaluation

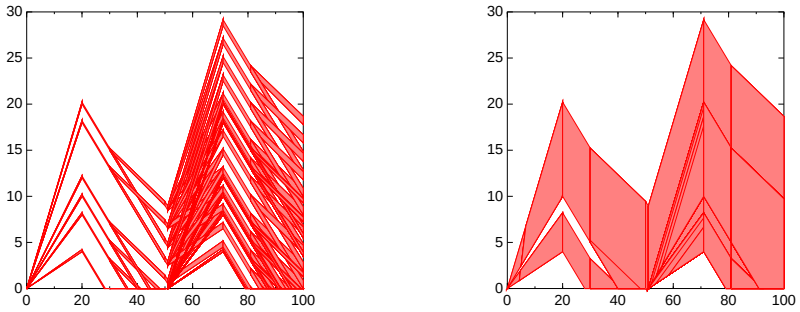
4.3.1 Benchmarks

For the evaluation of our approach we have extended the switched buffer network benchmark [40]. The system under consideration consists of multiple tanks connected by channels. The channels are used to transport the liquid stored in the tanks. There are two special tanks: the liquid enters the network through the *initial* tank and is transported towards the *sink* tank. We consider properties reasoning about the fill level of the sink tank.

The rate of change of the fill level f_T of a tank T, depends on the rates of inflow $v_{in\,i}$ and the rates of outflow $v_{out\,j}$ of the liquid, where $v_{in\,i}$ is the velocity at which the liquid flows into the tank of the i-th input channel, and $v_{out\,j}$ is the velocity at which the liquid flows out of the tank for the j-th output channel. Therefore, the evolution of the fill level of the tank T is described by the differential equation $\dot{f_T} = \sum_i v_{in\,i} - \sum_j v_{out\,j}$, where i and j range over incoming and outgoing channels of T, respectively. Note that due to fine-granular modelling of tanks and channels this benchmark class exhibits a large number of continuous variables. In particular, in our benchmark suite the number of continuous variables is in the range from 17 to 21 for the buffer networks with up to 4 tanks, whereas it is well-known that the analysis complexity of hybrid automata rapidly grows with the number of variables in the system under consideration.

We extend the switched buffer network [40] by the model of a complex *stratified* controller. The controller is organized in a number of phases of some given length, where multiple options (governing the modes of particular channels) are available in every phase. After having finished the last phase the controller returns to the first one. The controller can open/close channels and adjust the throughput values at every step. We consider the following modes of controller operations:

1. Throughput provided by an interval ("No Dynamics"): when the channel is activated, its throughput v is constrained by the inequality $v_{min} \leq v \leq v_{max}$.
2. Throughput evolving at a constant rate ("Constant Dynamics"): the throughput is defined by the differential equation of the form $\dot{v} = c$ for some constant c.
3. Throughput evolving according to affine dynamics $\dot{v} = c(v_{target} - v)$ ("Affine Dynamics"): the controller provides a target throughput velocity v_{target} and some constant factor c. According to this dynamics the channel opens gradually with the opening speed decaying towards the target velocity.



(a) Original system. (b) Initial abstraction.

Fig. 4.3: Fill level of the sink tank for instance 4 vs. time

4.3.2 Experiments

We have implemented our approach in SpaceEx [39]. The experiments were conducted on a machine with an Intel Core i7 3.4 GHz processor and with 16 GB of memory. In the following, we report the results for our compositional analysis implemented in SpaceEx. We compare the analysis results of the original concrete system and the compositional analysis. For both settings, we compare the number of iterations of SpaceEx and the whole analysis runtime in seconds (see Table 4.1). The best results are highlighted in bold. We analyze 12 structurally different benchmark instances. For each of them we vary forbidden states and in this way end up with 36 different benchmark settings. We also vary controller dynamics. In particular, we provide 12 instances for each of the modes "No Dynamics", "Constant Dynamics" and "Affine Dynamics". The number of continuous variables varies in the considered benchmark instances from 17 to 21 variables. The initial abstraction is generated by merging some of the strata in the controller.

We observe that our compositional reasoning algorithm generally boosts the run time compared to the analysis of the original system. For example, in instance 4 (system is safe) the analysis of the concrete system takes around 609 seconds compared to around 158 seconds with the compositional analysis. The speed-up is justified by the smaller branching factor due to location merging. In Figure 4.3a and Figure 4.3b the fill level of sink tank vs. time for the original system and the initial abstraction are plotted. Figure 4.3b particularly shows that multiple "thin" flow-pipes are merged into a couple of "thick" ones, i.e., the system stops differentiating between some options in the controller.

Furthermore, we remark that our compositional algorithm shows promising results also in the falsification setting, i.e., when the bad state is reach-

#	Res.	Tanks	Vars.	Phases	Refs.	It. (u)	It. (m)	Time (u)	Time (m)
No Dynamics									
1	safe	3	17	2 (5,1)	0	4640	**253**	779.754	**14.692**
2	unsafe	3	17	2 (5,1)	0	2555	**191**	299.437	**35.370**
3	safe	3	17	2 (5,1)	1	4640	**1744**	796.218	**191.841**
4	safe	3	17	4 (6,1,2,1)	0	3242	**1115**	608.796	**157.924**
5	unsafe	3	17	4 (6,1,2,1)	0	2410	**756**	196.461	**66.740**
6	safe	3	17	4 (6,1,2,1)	2	3242	**1648**	639.838	**254.653**
7	safe	4	21	2 (5,1)	0	2345	**690**	2162.273	**621.137**
8	unsafe	4	21	2 (5,1)	0	1348	**483**	1139.365	**479.811**
9	safe	4	21	2 (5,1)	1	2345	**1001**	2164.069	**937.064**
10	safe	4	21	4 (4,1,2,1)	0	1361	**394**	1327.062	**406.592**
11	unsafe	4	21	4 (4,1,2,1)	0	1070	**316**	502.992	**303.988**
12	safe	4	21	4 (4,1,2,1)	1	1361	**684**	1174.735	**700.072**
Constant Dynamics									
13	safe	3	17	4 (2,1,5,1)	0	1386	**424**	90.457	**21.484**
14	unsafe	3	17	4 (2,1,5,1)	0	461	**232**	18.773	**10.807**
15	safe	3	17	4 (2,1,5,1)	2	1386	**1261**	81.076	**77.938**
16	safe	3	17	6 (2,1,6,1,2,1)	0	1989	**1027**	146.726	**63.878**
17	unsafe	3	17	6 (2,1,6,1,2,1)	0	809	**352**	32.961	**14.279**
18	safe	3	17	6 (2,1,6,1,2,1)	2	**1989**	2041	**142.385**	250.451
19	safe	4	21	4 (2,1,4,1)	0	1293	**787**	1350.973	**1318.623**
20	unsafe	4	21	4 (2,1,4,1)	0	1080	**682**	1429.120	**1298.147**
21	safe	4	21	4 (2,1,4,1)	1	1293	**814**	1579.792	**1197.098**
22	safe	4	21	6 (2,1,4,1,2,1)	0	903	**563**	1255.978	**1140.114**
23	unsafe	4	21	6 (2,1,4,1,2,1)	0	798	**510**	1230.193	**1141.791**
24	safe	4	21	6 (2,1,4,1,2,1)	1	903	**581**	1365.629	**1318.049**
Affine Dynamics									
25	safe	3	17	4 (2,1,5,1)	0	7747	**1168**	1544.363	**86.046**
26	unsafe	3	17	4 (2,1,5,1)	0	5103	**1042**	939.430	**100.871**
27	safe	3	17	4 (2,1,5,1)	1	7747	**6214**	1669.268	**1240.215**
28	safe	3	17	6 (2,1,6,1,2,1)	0	6129	**2760**	717.462	**231.727**
29	unsafe	3	17	6 (2,1,6,1,2,1)	0	5382	**2397**	639.342	**203.143**
30	safe	3	17	6 (2,1,6,1,2,1)	7	**6129**	15068	**706.960**	2158.671
31	safe	4	21	4 (2,1,4,1)	0	1718	**1451**	3603.238	**3125.016**
32	unsafe	4	21	4 (2,1,4,1)	0	1692	**1392**	3776.840	**3247.464**
33	safe	4	21	4 (2,1,4,1)	1	**1718**	2559	4372.284	**3805.045**
34	safe	4	21	6 (2,1,4,1,2,1)	0	983	**642**	1382.567	**1078.893**
35	unsafe	4	21	6 (2,1,4,1,2,1)	0	922	**611**	1206.011	**1213.798**
36	safe	4	21	6 (2,1,4,1,2,1)	1	983	**755**	1442.506	**1321.658**

Table 4.1: Experimental results for the switched buffer benchmark. Abbreviations: #: benchmark instance number, Res.: result of the system analysis, i.e., whether the bad state can be reached, Tanks: number of tanks in the instance, Vars.: number of continuous variables in the system, Phases: number of phases in the controller and number of options in every phase, Refs.: number of refinement steps, It. (u): number of SpaceEx iterations when analyzing the concrete (unmerged) system, It. (m): number of SpaceEx iterations in scope of the compositional analysis, Time (u): total time in seconds of the analysis of the concrete system, Time (m): total time in seconds of the compositional analysis.

able. In instance 5, our approach reduces the run-time from around 196 seconds for the concrete system to only 67 seconds in scope of the compositional framework.

The necessity to refine the abstraction, in case a spurious abstract bad path has been discovered, can generally be handled efficiently by our framework, e.g., in instance 6 our approach takes around 254 seconds (including two refinement steps) compared to 640 seconds for the concrete system. However, due to an unfortunate choice of the abstract bad path, we might need to refine an excessive number of times (instance 30) which in turn decreases the overall performance.

4.4 Conclusion

In this chapter, we have adapted the idea of compositional analysis to the domain of hybrid automata. We have presented an abstraction based on location merging. The abstract location invariant is computed by taking a convex hull of the concrete locations to be merged. The abstract continuous dynamics are computed by eliminating the state variables and computing a convex hull.

5

Hybrid Planning

Planning in hybrid domains is a challenging problem that has found increasing attention in the planning community. In addition to classical planning, hybrid domains allow for modelling continuous behavior with continuous variables that evolve over time. Such problems frequently occur in practice, e. g., in robotics or embedded systems. Furthermore, real-world scenarios must take into account that exogenous events may happen, as a consequence or independently of the plan actions. PDDL+ [37] is the PDDL extension for modelling such domains through the use of continuous processes and events.

Planning in hybrid domains is challenging because apart from the state explosion caused by discrete state variables, the continuous variables cause the reachability problem undecidable [1]. From a practical point of view, various planning algorithms and tools with different features and limitations have emerged [58, 56, 55, 27, 66, 31]. However, only TM-LPSAT [66] and UPMurphi [31] can deal with the full feature range of PDDL+, and both suffer from scalability issues.

From an abstract point of view, it is well known that hybrid planning domains are related to the formalism of hybrid automata [42] studied in model checking. In the last years, powerful model checking techniques and tools based on, e. g., SMT [23] and symbolic search [38, 39], have been developed for this formalism. Apparently, algorithms based on such techniques can possibly be beneficial for planning in hybrid domains as well, and might particularly help to tackle the limitations of currently available planning systems with respect to the supported PDDL+ feature range. However, despite the relationship of hybrid planning domains and hybrid automata, these techniques have not been applied for planning in hybrid domains so far. The main obstruction to this synergy is the lack of a common modelling language, which makes it difficult to share benchmarks and to foster the cross-fertilization between these two areas.

In this chapter, we make a first step in bridging the gap between these two worlds. We provide a formal translation from PDDL+ to the formalism of hybrid automata [21]. The translation provides an over-approximation of the PDDL+ semantics, which is sufficient to prove *plan non-existence* in unsolvable domains. In addition, we identify a subset of PDDL+ features for which our translation is exact and can be applied for finding hybrid plans. In contrast to the class of hybrid automata that has been used to define the semantics of PDDL+ [37], our translation obeys the *standard* semantics of hybrid automata. A case study with the SpaceEx model checker [39] shows considerable improvements in scalability compared to the Colin and UPMurphi planner, and extends the class of tractable problems. Overall, our translation is supposed to build a solid basis for using hybrid system model checking tools for dealing with hybrid domains, thus extending the planning-as-model-checking paradigm [24] to the domain of hybrid systems.

The remainder of the chapter is organized as follows. We introduce the PDDL+ language in Section 5.1. Afterwards, we move on to the discussion of the semantical issues raised by PDDL+ in Section 5.2. In Section 5.3, we present our translation from PDDL+ to the formalism of hybrid automata. This is followed by the case study in Section 5.4. Finally, we conclude the chapter in Section 5.5.

5.1 The PDDL+ Language

In this section, we provide some additional background and a description of the PDDL+ language we consider throughout the chapter. PDDL+ supports the representation of domains with a mixed discrete-continuous dynamics, providing a flexible model of continuous change. In particular, it allows to model exogenous events to reflect changes that are initiated by the environment. PDDL+ is built on top of PDDL 2.1 and introduces the new constructs of *processes* and *events*.

Definition 5.1 (Planning Instance). *A* planning instance *is a pair* $I = (Dom, Prob)$*, where* $Dom = (Fs, Rs, As, Es, Ps, arity)$ *is a tuple consisting of a finite set of* function symbols Fs*, a finite set of* relation symbols Rs*, a finite set of (durative)* actions As*, a finite set of* events Es*, a finite set of* processes Ps*, and a function arity mapping all symbols in* $Fs \cup Rs$ *to their respective arities.*

The triple $Prob = (Os, Init, G)$ *consists of a finite set of* domain objects Os*, the* initial *state* $Init$*, and the* goal *specification* G*.*

For a given planning instance Π, a *state* of Π consists of a discrete component, described as a set of propositions P, and a numerical component,

described as a vector of real variables $\mathbf{v}$. Instantaneous actions are described through preconditions (which are conjunctions of propositions in P and/or numeric constraints over $\mathbf{v}$, and define when an action can be applied) and effects (which define how the action modifies the current state). *Instantaneous* actions and events are restricted to the expression of discrete change. Events have preconditions as for actions, but they are used to model exogenous change in the world, therefore they are triggered as soon as the preconditions are true. A process is responsible for the continuous change of variables, and is active as long as its preconditions are true.

Durative actions have three sets of preconditions, representing the conditions that must hold when it starts (denoted by $\text{pre}_{\vdash}$ and $\text{pre}_{\vdash}^{num}$ to distinguish between preconditions on propositions and on numeric constraints, respectively), the invariant that must hold throughout its execution (propositional invariant $\text{pre}_{\leftrightarrow}$ and numeric invariant $\text{pre}_{\leftrightarrow}^{num}$), and the conditions that must hold at the end of the action ($\text{pre}_{\dashv}$ and $\text{pre}_{\dashv}^{num}$). Similarly, a durative action has three sets of effects: effects that are applied when the action starts ($\text{eff}_{\vdash}^{+}$, $\text{eff}_{\vdash}^{-}$, $\text{eff}_{\vdash}^{num}$ denoting predicates that are added, deleted, and numeric effects, respectively), effects that are applied when the action ends ($\text{eff}_{\dashv}^{+}$, $\text{eff}_{\dashv}^{-}$, $\text{eff}_{\dashv}^{num}$) and a set of continuous numeric effects $\text{eff}_{\leftrightarrow}^{num}$ which are applied continuously while the action is executing. A graphical representation of a durative action is shown in Figure 5.1.

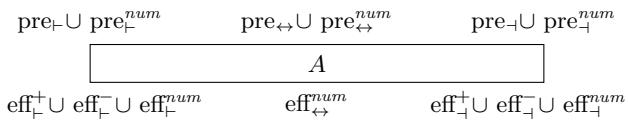


Fig. 5.1: PDDL durative action

A durative action A has a duration $dur(A)$ which can either be fixed in the model or left to the planner decision.

5.2 Semantical Issues Raised by PDDL+

Although parts of PDDL+ are defined in terms of hybrid automata, there are several semantical issues raised by PDDL+ which do not allow to apply the translation given by Fox and Long [37] to common model checking tools – in particular, their translation does not obey the standard semantics of hybrid automata (see Definition 2.1). In the formal definition of PDDL+ [37], assumptions about the class of domains that can be modelled and about plan validity [36] are made. In the following, we briefly recall these assumptions.

No Moving Targets: The *no moving targets* rule states that no two actions are allowed to simultaneously make use of a value if one of the two is accessing the value to update it (i. e., the value is a moving target for the other action to access). As a consequence of this restriction, plans must respect the *ε-separation* requirement, i. e., interfering actions must be separated by at least a time interval of length ε. The planner Colin [27] makes a strictly stronger assumption, extending this requirement also to actions that are not mutex. In our work, we make the same assumption as Colin. We remark that *ε-separation* is not respected in the standard hybrid automata semantics, where transitions can start or end at the same instant and hence can compromise plan validity.

Events: Events are particularly challenging as they could trigger an infinite cascading sequence of events. To address this issue, we make the same restrictions proposed by Fox and Long [37]. Firstly, each event must delete one of its own preconditions and thus avoid self triggering. Secondly, planning instances must be *event-deterministic*: In every state in which two events e_1 and e_2 are applicable, the transition sequences e_1 followed by e_2 and e_2 followed by e_1 are both valid and reach the same resulting state.

Actions and events reveal the key difference between state changes that are deliberately planned (actions), and those that are caused by changes in the world (events). While the planner can decide whether or not to fire an applicable action (actions are *may* transitions), events have to be fired as soon as they become enabled (events are *must* transitions). This distinction complicates the relationship between PDDL+ and standard hybrid automata, where such a distinction is not present and all transitions are *may* transitions.

Concurrent Processes: It is possible that several processes are active at the same time, affecting the value of the same variable. To handle such *concurrent processes*, the continuous effects affecting the rate of change of a variable are combined by simply summing the effects of the processes. Although the handling of concurrent processes is very simple in PDDL+, it is a problematic feature in the standard hybrid automata setting, as each location in a hybrid automaton contains a single flow describing the continuous effects corresponding to that location. Therefore, combining the effects of concurrent processes would generate an explosion of the number of locations in the hybrid automaton.

PDDL+ Semantics: Fox and Long [37] give a formal semantics of PDDL+ providing a mapping between PDDL+ domain and hybrid automata. However, Fox and Long make the key assumption to have hybrid automata where *conditional flows* can be defined. In any given location, instead of having a fixed rate of change for each variable, the rate of change

of each variable depends on which processes are active in the current state. Conditional flows allow for easily modelling concurrent processes by using different rates of change depending on the current state.

Similarly, conditional flows are used to model events. To model events, we must force the corresponding event automaton to leave the current location (and to enter the location corresponding to the event's effect) as soon as it is triggered. Hence, the issue is to model *must* transitions. For this purpose, Fox and Long use the *time slippage* mechanism. A time-slip variable $\mathtt{T}$ is used to measure the amount of time that elapses between the preconditions of an event becoming true and the event triggering. The value of $\mathtt{T}$ must be 0 in any valid planning instance. To this aim, each location contains an invariant enforcing this requirement. Furthermore, the conditional flow is extended with the additional time-slippage flow that sets $\dot{\mathtt{T}} = 1$ whenever the preconditions of any event become true. However, conditional flows are not part of standard hybrid automata semantics. Furthermore, the issue of modelling the *ε-separation* is not addressed.

5.3 Modeling PDDL+ as Hybrid Automata

Based on the discussed semantical issues raised by PDDL+, we provide a formal translation of hybrid planning domains to *standard* hybrid automata to overcome these limitations. For the description of our translation, we assume a grounded planning instance I and use the following naming conventions: Function symbols are denoted with *continuous variables*, whereas (Boolean) grounded predicates are denoted with *discrete variables*. In particular, for the rest of the chapter, we assume the actions in I to be grounded. The translation from I to a network of hybrid automata is based on translating grounded actions, discrete and continuous variables, events and processes to corresponding hybrid automata. This translation is described in the next sections.

5.3.1 Discrete Variable Automata

Common model checkers like SpaceEx do not support discrete variables in their input models. Hence, we represent discrete variables with a variant of their domain-transition graphs [41]. A Boolean variable v is translated to automaton $\mathcal{H}_v$ with two locations that reflect the *true* and *false* values of v. Transitions between locations reflect how values can be changed through actions. More precisely, the synchronization labels reflect the discrete preconditions and effects of actions that have v in their precondition and effect,

respectively. Roughly speaking, labels c that do not occur in all transitions of $\mathcal{H}_v$ possibly require changing $\mathcal{H}_v$'s location in order to be able to synchronize with c (thus representing a precondition). Similarly, labels c that occur in at least one non-self loop transition of $\mathcal{H}_v$ reflect that c can possibly change the value of v (thus representing an effect). We will make the description of synchronization labels more precise when introducing the translation for actions.

5.3.2 Continuous Variable Automata

Continuous variables x are translated to automata $\mathcal{H}_x$ as follows. For all possible flows $\dot{x} = k$ of x, $\mathcal{H}_x$ contains a location annotated with $\dot{x} = k$. There is a transition between two locations if it is possible to change the flow of x accordingly via an action. Furthermore, for all actions that affect the particular flow, there is a self-loop in the corresponding location. As an example, consider the automaton in Figure 5.2.

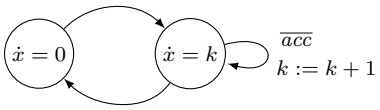


Fig. 5.2: Example variable automaton $\mathcal{H}_x$

The automaton $\mathcal{H}_x$ models the behavior of the acceleration x of an engine. There are two possible flows for x, namely $\dot{x} = 0$ (corresponding to the case that the engine is turned off), and $\dot{x} = k$ (corresponding to the case where the engine is turned on, and the current acceleration is k). In case the engine is turned on, we can apply the action accelerate (represented as label $\overline{acc}$) to increase k.

5.3.3 Durative Action Automata

Grounded durative actions are translated to automata $\mathcal{H}_a$ such that $\mathcal{H}_a$ ensures the ε-separation property, and such that the (propositional and numeric) preconditions, effects and invariants of a are respected when a is starting, running and ending, respectively. For a given action a, $\mathcal{H}_a$ has the overall structure given in Figure 5.3. The guards and updates of $\mathcal{H}_a$'s transitions are denoted with g and ξ according to Definition 2.1. The locations are annotated with the corresponding invariant I. Synchronization labels are annotated with a bar.

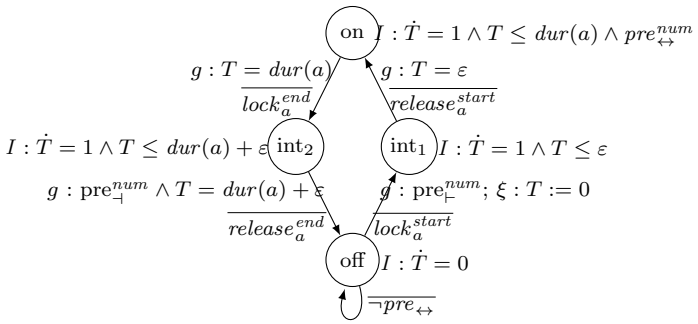


Fig. 5.3: Structure of action automaton $\mathcal{H}_a$

The automaton $\mathcal{H}_a$ uses a local continuous variable T that models a clock to keep track of action a's duration. Based on T, $\mathcal{H}_a$ simulates the execution of a as follows.

1. The *off* location models that a is not running. The invariant $\dot{T} = 0$ reflects that the clock T is stopped as well. (See below for a description of the self-loop.)
2. The int_1 location and the transition from *off* to int_1 model the behavior of a in the time interval $[0, \varepsilon]$ (for brevity, we assume that a is started at time point 0). The invariant of int_1 ensures that T is running, and that int_1 is left after at most ε time units. The guard g of the transition leading to int_1 reflects a's numeric precondition $\text{pre}^{num}_{\vdash}$, and its update ξ resets the clock T to zero. In addition, through synchronization, the label $\overline{lock^{start}_a}$ ensures the ε-separation property during the starting phase of a, as well as the required behavior of a's preconditions and effects:
   - In order to ensure the ε-separation property, $\overline{lock^{start}_a}$ locks the overall system in the sense that no other automaton can start (or end, see below) as long as $\mathcal{H}_a$ is in the int_1 location. To achieve this, a global lock automaton synchronizes with this label, with the property that such a synchronization is no longer possible for starting or ending other actions until the lock is released. To make this more clear, the lock automaton (simplified such that it only contains transitions for the starting phase of a) is depicted in Figure 5.4 (see below for a description of the corresponding release label).
   - $\overline{lock^{start}_a}$ reflects the check for the propositional precondition $\text{pre}_{\vdash}$ as well as the check for the invariant $\text{pre}_{\leftrightarrow}$ (recall that $\text{pre}_{\leftrightarrow}$ must hold during the execution of a, hence it must hold at the start of a). These preconditions are satisfied iff a synchronization with corresponding discrete variable automata is possible. For example, if $\text{pre}_{\vdash}$ requires

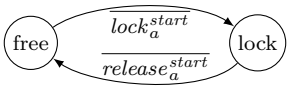


Fig. 5.4: Global lock automaton

a variable v to be true, this is reflected in the corresponding variable automaton $\mathcal{H}_v$ as depicted on the left in Figure 5.5, where we observe that v must be in the *true* location such that a synchronization is possible.

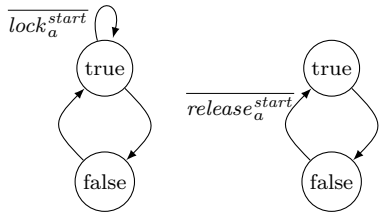


Fig. 5.5: Example variable automata $\mathcal{H}_v$ and $\mathcal{H}_w$

- $\overline{lock_a^{start}}$ reflects the continuous numerical effects described by $\text{eff}_{\leftrightarrow}^{num}$, which affects the flow of its continuous variables by synchronizing with the corresponding continuous variable automata (e. g., by increasing k in Figure 5.2, where $\overline{acc}$ is replaced by the lock label).

3. The transition from int_1 to *on* models the time point ε. According to the guard $T = \varepsilon$, it must be taken after exactly ε time units. Furthermore, its label $\overline{release_a^{start}}$ releases the system via the lock automaton, allowing other actions to start or end again. In addition, $\overline{release_a^{start}}$ reflects the start effects $\text{eff}_{\vdash}^{+}$, $\text{eff}_{\vdash}^{-}$ and $\text{eff}_{\vdash}^{num}$ by synchronizing with the corresponding variable automata. For example, if $\text{eff}_{\vdash}^{+}$ sets a variable w from false to true, this is reflected in $\mathcal{H}_w$ as shown on the right in Figure 5.5.
4. The *on* location models a's behavior in the time interval $[\varepsilon, dur(a)]$. The invariant of *on* reflects the duration of a and the numeric invariant $\text{pre}_{\leftrightarrow}^{num}$. We remark that without further knowledge, doing so would require $\text{pre}_{\leftrightarrow}^{num}$ to hold in the time interval $[\varepsilon, dur(a)]$, whereas the original PDDL+ semantics would only require it to hold in the interval $[\varepsilon, dur(a) - \varepsilon]$. However, due to the ε-separation, the behavior of numeric invariants with strict and non-strict inequalities is identical, and we can hence interpret strict as non-strict inequalities without loss of generality.

5. Propositional invariants of actions a must hold as long as $\mathcal{H}_a$ is in its *on* location. To model this, we include all synchronization labels that possibly violate a's propositional invariant into the synchronization alphabet of $\mathcal{H}_a$ (e. g., for a propositional invariant $p = \mathit{true}$, we include all labels that represent effects of actions that set p to false). This causes all actions that execute an effect that violates a's propositional invariant to synchronize with a transition in $\mathcal{H}_a$. However, as there is no such outgoing synchronization transition of the *on* location, actions cannot violate a's propositional invariant as long as $\mathcal{H}_a$ is running. To be able to synchronize with such actions when a is not running, we introduce self-loops to $\mathcal{H}_a$'s *off* location that allow corresponding synchronization. In more detail, the self-loop with label $\overline{\neg \mathit{pre}_{\leftrightarrow}}$ in the *off* location represents a set of self-loops with labels for actions that violate a constraint in $\mathrm{pre}_{\leftrightarrow}$. Note that we do not need such self-loops for int_1 (and int_2) because int_1 (and int_2) model the locked system where no other action may start or end.
6. The int_2 location and the transition from *on* to int_2 models the behavior of a in the interval $[\mathit{dur}(a), \mathit{dur}(a)+\varepsilon]$ when a is finished. The label $\overline{\mathit{lock}_a^{\mathit{end}}}$ locks the system to ensure the ε-separation during the ending phase of a, and reflects $\mathrm{pre}_{\dashv}$ via synchronization (analogously to the start of a). In addition, it reflects the end of the continuous numeric change reflected by $\mathrm{eff}_{\leftrightarrow}^{\mathit{num}}$. For example, if k has been increased by $\mathrm{eff}_{\leftrightarrow}^{\mathit{num}}$ in Figure 5.2 at the start of a, k is decreased again to reset the flow before a has been started (a corresponding self-loop is omitted in Figure 5.2).
7. The transition from int_2 to *off* models the end of the execution of a. Its guard checks both a's duration $T = \mathit{dur}(a) + \varepsilon$ and precondition $\mathrm{pre}_{\dashv}^{\mathit{num}}$. The label $\overline{\mathit{release}_a^{\mathit{end}}}$ releases the system (indicating that a is finished), and reflects the effect updates $\mathrm{eff}_{\dashv}^{+}$, $\mathrm{eff}_{\dashv}^{-}$, and $\mathrm{eff}_{\dashv}^{\mathit{num}}$.

We observe that, by construction, the PDDL+ semantics of durative actions a is reflected by the hybrid automaton $\mathcal{H}_a$. In particular, $\mathcal{H}_a$ respects the ε-separation property.

5.3.4 Instantaneous Action Automata

Instantaneous actions a are modelled as automata $\mathcal{H}_a$ as follows. Similarly to durative actions, $\mathcal{H}_a$ contains an *off* and *on* location and respects the ε-separation. However, in contrast to durative actions, instantaneous actions do not feature durations (as suggested by the name), and hence, we do not need additional intermediate locations in the automaton model. In more detail, the transition from *off* to *on* is labelled with a corresponding $\overline{\mathit{lock}_a}$ label, which locks the system via the global lock automaton, and reflects the guard

and effects analogously to the $\overline{lock_a^{start}}$ labels for durative actions (we do not need to distinguish between start and end labels because a is instantaneous). In addition, the transition features a numerical guard constraint that reflects the numerical precondition $pre_{\vdash}^{num}$ of a. Finally, $\mathcal{H}_a$ stays for ε time in *on*, and releases the system by returning to *off*.

5.3.5 Event and Process Automata

Events and processes require a *must* semantics, as they trigger as soon as they become enabled. In this thesis, we over-approximate this *must* behavior with the (common) *may* behavior, which allows for more behavior and is hence sufficient for proving plan non-existence. Generally, over-approximations allow for at least the same (and possibly more) behavior as the original model. (Realizing *must* behavior more precisely is an important issue for future work).

Events are essentially instantaneous actions with *must* behavior. Hence, in our translation, we over-approximate events with instantaneous action automata.

Processes p are modelled as automata $\mathcal{H}_p$ that consist of an *off* and *on* location similar to events. There is a transition from *off* to *on* which synchronizes over the propositional precondition constraints $pre_{\vdash}$. This is an over-approximation because the transition is not forced to be taken as soon as possible. Furthermore, the *on* location features an invariant induced by the numeric precondition $pre_{\vdash}^{num}$. Finally, there are transitions from *on* to *off* for each negated constraint in $pre_{\vdash}^{num}$ (reflecting that the numeric invariant gets violated), and a transition that allows for returning in case the propositional invariant is set to false (again yielding an over-approximation). The effects of p are reflected in the same way as for continuous variable automata (e.g., see again Figure 5.2). This translation allows modelling of concurrent processes in the standard hybrid automata semantics without the need of conditional flows.

5.3.6 Overall Translation Scheme

For a given planning instance, the overall translation is defined by a network of hybrid automata which contains a translated automaton for all discrete and continuous variables, durative and instantaneous actions, processes and events. The resulting system of hybrid automata is an over-approximation of the original PDDL+ planning instance.

Proposition 5.2. *Let I be a planning instance, and let $\mathcal{N}$ be the translated network of hybrid automata. Then for all plans π in I, there is a corresponding sequence σ of transitions in $\mathcal{N}$ such that for each time point t, the values of the discrete and continuous variables of π and σ are equal in t.*

Proof. (sketch) By construction, the semantics of variables and actions is reflected exactly by the translated automata. For processes and events, *must* transitions are approximated with *may* transitions, yielding an over-approximation.

The over-approximation is sufficient to prove plan non-existence. In the more simple case where no processes and events are present, the back direction holds as well. For such planning instances I, the translation can also be applied for finding plans because transition sequences in $\mathcal{N}$ are guaranteed to correspond to applicable action sequences in I.

5.4 Case Study

As a case study, we apply our translation with the SpaceEx model checker [39], which is considered as a state-of-the art tool in the area of hybrid systems model checking. The search engine of SpaceEx performs symbolic search, which is suited for effectively proving plan non-existence. Proving plan non-existence has recently found increasing attention for classical planning [11], and becomes even harder for planning in hybrid domains. For a particular class of planning domains, SpaceEx is guaranteed to find valid plans in solvable domains as well (see below). We consider several instances (with growing size) of the generator [45] and the car domains [37], which are standard and challenging benchmarks in the hybrid planning community. We compare our translation for SpaceEx with the state-of-the-art planners Colin [27] and UPMurphi [30]. The experiments are performed on an x64 Linux machine with 6 GB of RAM and an Intel i7 CPU (2.20GHz).

The results for unsolvable instances are reported in Table 5.1. Colin can prove plan non-existence for a restricted class of domains, namely when there is a tight deadline on reaching the goals (which sets a finite horizon for the plan), and each ground action can only be applied a finite number of times. UPMurphi cannot provide any guarantees about plan non-existence as it relies on discretizing the time line and the continuous variables prior to search. In other words, plans might exist for a finer discretization than actually used by UPMurphi. The results for UPMurphi are included in Table 5.1 for the sake of completeness. We observe that our translation with symbolic search is able to scale better than both Colin and UPMurphi. In particular, our

approach is able to effectively prove plan non-existence in the car domain, which is out of scope for both UPMurphi (as discussed) and Colin (as Colin is not able to deal with processes and events, which are present in the car domain).

D	Tool	1	2	3	4	5	6	7	8	9	10
Gen	SpaceEx	0.01	0.09	0.83	4.25	58.61	1214.35	-	-	-	-
Gen	CoLin	0.01	0.1	1.7	32.48	761.28	-	-	-	-	-
Gen	UPMur	0.9	29.42	-	-	-	-	-	-	-	-
Car	SpaceEx	0.98	4.91	9.46	19.65	37.19	59.40	112.43	210.47	350.14	574.71
Car	CoLin	x	x	x	x	x	x	x	x	x	x
Car	UPMur	36.01	445.23	-	-	-	-	-	-	-	-

Table 5.1: Results in seconds for unsolvable instances. Instance numbers correspond to number of tanks (*generator*) and maximum acceleration (*car*). Abbrev.: '-': tool still running after 30 minutes, 'x': tool cannot handle the problem.

The symbolic search performed by SpaceEx induces an over-approximation of the original system, which is suited for effectively proving plan non-existence. In contrast, applying symbolic search to find plans might result in *spurious* plans, i. e., plans that do not correspond to valid plans in the concrete. However, for the subclass of planning problems that do neither feature processes nor events (according to Proposition 5.2) and do only include simple differential equations of the form $\dot{x} = c$, the search algorithm by SpaceEx guarantees that a path to a goal corresponds to a valid plan as well. These requirements are satisfied by the generator, but not by the car domain. The results are depicted in Table 5.2.

Domain	Tool	1	2	3	4	5	6	7	8	9	10
Generator	SpaceEx	0.01	0.03	0.07	0.1	0.19	0.28	0.45	0.65	0.93	1.22
Generator	CoLin	0.01	0.09	0.2	2.52	32.62	600.58	-	-	-	-
Generator	UPMurphi	0.2	18.2	402.34	-	-	-	-	-	-	-
Car	SpaceEx	0.01	0.01	0.01	0.03	0.04	0.05	0.06	0.07	0.08	0.1
Car	CoLin	x	x	x	x	x	x	x	x	x	x
Car	UPMurphi	28.44	386.5	-	-	-	-	-	-	-	-

Table 5.2: Results in seconds for solvable instances

Table 5.2 shows scalability improvements for solvable instances as well. As discussed, the results for the car domain must be taken with care (as the found paths might be spurious), but are included for the sake of completeness. In contrast, for the generator domain, the found paths by SpaceEx are guaranteed to correspond to valid plans. We observe that SpaceEx outper-

forms the other tools by several orders of magnitude in terms of scalability. We remark that our current implementation does not yet extract these plans, but this step is purely technical and efficiently implementable (essentially a call to an SMT solver). Overall, we observe that symbolic search is beneficial for both proving plan non-existence as well as for finding paths to goal states. Generally, symbolic search seems to be well suited for hybrid domains because it handles several paths simultaneously.

5.5 Conclusion

We have presented a formal translation from PDDL+ to the standard formalism of hybrid automata. Our translation forms the basis for bridging the gap between planning in hybrid domains and model checking of hybrid automata. Our experimental evaluation has shown that the translation can be effectively applied to proving plan non-existence in challenging hybrid domains. In particular, our translation extends the class of tractable planning domains for proving plan non-existence as shown for the car domain. For a particular class of hybrid domains, SpaceEx can also be applied for effectively finding plans. For future research, the precise modelling of *must* transitions in order to avoid spurious plans should be addressed. Furthermore, it will be interesting to apply the translation also with other model checking tools in order to exploit their particular strengths. Generally, we hope that our work forms the basis to eventually allow the planning community to systematically benefit from the large body of research in the area of hybrid automata.

6

Conclusion and Future Research

In this thesis, we have presented a number of *abstraction-based* approaches to analyze hybrid automata. Our approaches strongly improve the analysis scalability, which in turn makes many models algorithmically tractable compared to already existing methods. Our abstractions are generic, i.e. they do not rely on any domain-specific information, and can be computed completely automatically.

We have discussed how abstractions can be utilized for guided search in the state space of hybrid automata. We have presented two abstraction-based heuristics. The first one, the box-based heuristic [20], works by estimating the distance between the abstraction of a symbolic state and a bad state. It has shown good results on systems with mainly continuous behavior. In addition, we have presented a further heuristic which combines the ideas of a pattern database and a *coarse-grained space abstraction* [18, 17]. This abstraction makes use of an internal polytope representation in SpaceEx based on the support function representation. The analysis of hybrid automata becomes particularly expensive while reasoning about models consisting of networks of hybrid automata. Therefore, at the next stage, we have considered the compositional analysis of hybrid automata. We have extended an assume guarantee abstraction refinement (AGAR) framework to hybrid automata [19]. This extension enables an efficient decomposition of complex networks consisting of multiple components. An important ingredient of the AGAR framework is the abstraction used to simplify parts of the network. In our work, we have introduced an *abstraction based on location merging*. We approximate the continuous dynamics by using quantifier elimination and hence end up in the class of linear hybrid automata. We have implemented our techniques within the hybrid model checker SpaceEx. In order to evaluate our approaches, we have considered a number of challenging hybrid automata benchmark models. Finally, we have considered the application of

model checking techniques to hybrid planning. We have presented a translation from hybrid planning problems described by the PDDL+ language to networks of hybrid automata [21]. Our translation forms the basis for bridging the gap between planning in hybrid domains and model checking of hybrid automata. Our experimental evaluation on common hybrid planning benchmarks has shown that the translation can be effectively applied to proving plan non-existence in challenging hybrid domains.

The framework of hybrid automata provides a way to model highly complex systems from different domains. We note that a modelling task itself poses a challenge. On the one hand, this task becomes especially hard, e.g., in the biological domain, where a researcher should already have an in-depth knowledge of biology while working on the model. On the other hand, we expect the analysis of the model to greatly benefit from the domain knowledge as biological models typically follow some structural patterns. In our work, we focus on generic approaches which do not make use of any background information. For the future, it will be productive to investigate how domain specific knowledge can be exploited in order to additionally facilitate hybrid automata analysis. In this thesis, we have seen that the knowledge transfer between the two research areas of model checking and automated planning can be very fruitful. Still, there is a lot of work to be done in order to bring together different communities working on similar problems. In particular, we expect synergies to arise at the interface of symbolic model checking and robotics. For example, there is a large potential of using the *symbolic* abstractions to guide the *numerical* simulations in the flavour of RRTs [60]. Finally, it will be instructive to investigate how reachabilty algorithms for systems with affine dynamics might be used as part of reachability analysis of hybrid automata exhibiting non-linear differential equations.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
2. R. Alur, T. Dang, and F. Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In *Hybrid Systems: Computation and Control (HSCC 2002)*, volume 2289 of *LNCS*, pages 35–48. Springer, 2002.
3. R. Alur, T. Dang, and F. Ivancic. Counter-example guided predicate abstraction of hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, pages 250–271. Springer, 2003.
4. R. Alur, T. Dang, and F. Ivancic. Progress on reachability analysis of hybrid systems using predicate abstraction. In *Hybrid Systems: Computation and Control (HSCC 2003)*, volume 2623 of *LNCS*, pages 4–19. Springer, 2003.
5. R. Alur and T. Henzinger. Modularity for timed and hybrid systems. In *Concurrency Theory (CONCUR 1997)*, volume 1243 of *LNCS*, pages 74–88. Springer, 1997.
6. R. Alur, T. Henzinger, and P. H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.
7. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification (CAV 2005)*, volume 3576 of *LNCS*, pages 548–562. Springer, 2005.
8. K. Anderson, R. Holte, and J. Schaeffer. Partial pattern databases. In *Symposium on Abstraction, Reformulation, and Approximation (SARA 2007)*, volume 4612 of *LNCS*, pages 20–34. Springer, 2007.
9. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.
10. E. Asarin, T. Dang, and A. Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Informatica*, 43(7):451–476, 2007.
11. C. Bäckström, P. Jonsson, and S. Ståhlberg. Fast detection of unsolvable planning instances using local consistency. In *Symposium on Combinatorial Search (SoCS 2013)*. AAAI Press, 2013.
12. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 203–213. ACM Press, 2001.
13. E. Bartocci, F. Corradini, M. R. Di Berardini, E. Entcheva, S. Smolka, and R. Grosu. Modeling and simulation of cardiac tissue using hybrid I/O automata. *Theoretical Computer Science*, 410(410):3149–3165, 2009.

14. D. Bertsekas, A. Nedi, A. Ozdaglar, et al. Convex analysis and optimization. 2003.
15. A. Bhatia and E. Frazzoli. Incremental search methods for reachability analysis of continuous and hybrid systems. In *Hybrid Systems: Computation and Control (HSCC 2004)*, volume 2993 of *LNCS*, pages 451–471. Springer, 2004.
16. M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 135–148. Springer, 2008.
17. S. Bogomolov, A. Donzé, G. Frehse, R. Grosu, T. T. Johnson, H. Ladan, A. Podelski, and M. Wehrle. Guided search for hybrid systems based on coarse-grained space abstractions. Submitted to *International Journal on Software Tools for Technology Transfer (STTT)*.
18. S. Bogomolov, A. Donzé, G. Frehse, R. Grosu, T. T. Johnson, H. Ladan, A. Podelski, and M. Wehrle. Abstraction-based guided search for hybrid systems. In *Model Checking Software (SPIN 2013)*, volume 7976 of *LNCS*, pages 117–134. Springer, 2013.
19. S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. S. Pasareanu, A. Podelski, and T. Strump. Assume-guarantee abstraction refinement meets hybrid systems. In *Haifa Verification Conference (HVC 2014)*, volume 8855 of *LNCS*, pages 116–131. Springer, 2014.
20. S. Bogomolov, G. Frehse, R. Grosu, H. Ladan, A. Podelski, and M. Wehrle. A box-based distance between regions for guiding the reachability analysis of SpaceEx. In *Computer Aided Verification (CAV 2012)*, volume 7358 of *LNCS*, pages 479–494. Springer, 2012.
21. S. Bogomolov, D. Magazzeni, A. Podelski, and M. Wehrle. Planning as model checking in hybrid domains. In *AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 2228–2234. AAAI Press, 2014.
22. O. Bournez, O. Maler, and A. Pnueli. Orthogonal polyhedra: Representation and computation. In *Hybrid Systems: Computation and Control (HSCC 1999)*, volume 1569 of *LNCS*, pages 46–60. Springer, 1999.
23. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000.
24. A. Cimatti, M. Roveri, and P. Traverso. Strong planning in non-deterministic domains via model checking. In *Artificial Intelligence Planning Systems (AIPS 1998)*, pages 36–43. AAAI Press, 1998.
25. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
26. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
27. A. J. Coles, A. Coles, M. Fox, and D. Long. COLIN: Planning with continuous linear numeric change. *Journal of Artificial Intelligence Research (JAIR)*, 44:1–96, 2012.
28. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.
29. T. Dang and T. Nahhal. Coverage-guided test generation for continuous and hybrid systems. *Formal Methods in System Design (FMSD)*, 34(2):183–213, 2009.
30. G. Della Penna, D. Magazzeni, and F. Mercorio. A universal planning system for hybrid domains. *Applied Intelligence*, 36(4):932–959, 2012.
31. G. Della Penna, D. Magazzeni, F. Mercorio, and B. Intrigila. UPMurphi: A tool for universal planning on PDDL+ problems. In *International Conference on Automated Planning and Scheduling (ICAPS 2009)*. AAAI Press, 2009.
32. A. Donzé and G. Frehse. Modular, hierarchical models of control systems in SpaceEx. In *European Control Conference (ECC 2013)*, pages 4244 – 4251. IEEE, 2013.

33. L. Doyen, T. A. Henzinger, and J.-F. Raskin. Automatic rectangular refinement of affine hybrid systems. In *Formal Modelling and Analysis of Timed Systems (FORMATS 2005)*, volume 3829 of *LNCS*, pages 144–161. Springer, 2005.
34. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2):247–267, 2004.
35. A. Fehnker and F. Ivančić. Benchmarks for hybrid systems verification. In *Hybrid Systems: Computation and Control (HSCC 2004)*, volume 2993 of *LNCS*, pages 381–397. Springer, 2004.
36. M. Fox, R. Howey, and D. Long. Validating plans in the context of processes and exogenous events. In *AAAI Conference on Artificial Intelligence (AAAI 2005)*, pages 1151–1156. AAAI Press, 2005.
37. M. Fox and D. Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research (JAIR)*, 27:235–297, 2006.
38. G. Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(3):263–279, 2008.
39. G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification (CAV 2011)*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.
40. G. Frehse and O. Maler. Reachability analysis of a switched buffer network. In *Hybrid Systems: Computation and Control (HSCC 2007)*, volume 4416 of *LNCS*, pages 698–701. Springer, 2007.
41. M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
42. T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pages 278–292, 1996.
43. R. C. Holte, J. Grajkowski, and B. Tanner. Hierarchical heuristic search revisited. In *Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, volume 3607 of *LNCS*, pages 121–133. Springer, 2005.
44. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesley, 2006.
45. R. Howey, D. Long, and M. Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 294–301. IEEE, 2004.
46. S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *Hybrid Systems: Computation and Control (HSCC 2007)*, volume 4416 of *LNCS*, pages 287–300. Springer, 2007.
47. K. H. Johansson, M. Egerstedt, J. Lygeros, and S. Sastry. On the regularization of zeno hybrid automata. *Systems & Control Letters*, 38(3):141–150, 1999.
48. T. T. Johnson, J. Green, S. Mitra, R. Dudley, and R. S. Erwin. Satellite rendezvous and conjunction avoidance: Case studies in verification of nonlinear hybrid systems. In *Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 252–266. Springer, 2012.
49. H. K. Khalil. *Nonlinear Systems.* Prentice Hall, 2002.
50. S. Kupferschmid, J. Hoffmann, and K. G. Larsen. Fast directed model checking via Russian doll abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 203–217. Springer, 2008.
51. S. Kupferschmid and M. Wehrle. Abstractions and pattern databases: The quest for succinctness and accuracy. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, volume 6605 of *LNCS*, pages 276–290. Springer, 2011.

52. A. B. Kurzhanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control (HSCC 2000)*, volume 1790 of *LNCS*, pages 202–214. Springer, 2000.
53. B. J. Larsen, E. Burns, W. Ruml, and R. Holte. Searching without a heuristic: Efficient use of abstraction. In *AAAI Conference on Artificial Intelligence (AAAI 2010)*. AAAI Press, 2010.
54. C. Le Guernic and A. Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250–262, 2010.
55. H. X. Li and B. C. Williams. Generative planning for hybrid systems based on flow tubes. In *International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 206–213. AAAI Press, 2008.
56. D. V. McDermott. Reasoning about autonomous processes in an estimated-regression planner. In *International Conference on Automated Planning and Scheduling (ICAPS 2003)*, pages 143–152. AAAI Press, 2003.
57. C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design (FMSD)*, 32(3):175–205, 2008.
58. J. S. Penberthy and D. S. Weld. Temporal planning with continuous change. In *AAAI Conference on Artificial Intelligence (AAAI 1994)*, pages 1010–1015. AAAI Press, 1994.
59. E. Plaku, L. Kavraki, and M. Vardi. Hybrid systems: From verification to falsification. In *Computer Aided Verification (CAV 2007)*, volume 4590 of *LNCS*, pages 463–476. Springer, 2007.
60. E. Plaku, L. E. Kavraki, and M. Y. Vardi. Hybrid systems: from verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design (FMSD)*, 34(2):157–182, Oct. 2008.
61. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1989.
62. P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan. Hybrid automata-based CEGAR for rectangular hybrid systems. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, volume 7737 of *LNCS*, pages 48–67. Springer, 2013.
63. K. Qian and A. Nymeyer. Guided invariant model checking based on abstraction and symbolic pattern databases. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 497–511. Springer, 2004.
64. S. Ratschan and J.-G. Smaus. Finding errors of hybrid systems by optimising an abstraction-based quality estimate. In *Tests and Proofs (TAP 2009)*, volume 5668 of *LNCS*, pages 153–168. Springer, 2009.
65. R. C. Robinson. *An introduction to dynamical systems: continuous and discrete*, volume 19. American Mathematical Society, 2012.
66. J.-A. Shin and E. Davis. Processes and continuous change in a SAT-based planner. *Artificial Intelligence*, 166(1-2):194–253, 2005.
67. A. Tiwari. Abstractions for hybrid systems. *Formal Methods in System Design (FMSD)*, 32(1):57–83, 2008.
68. A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In *Hybrid Systems: Computation and Control (HSCC 2002)*, volume 2289 of *LNCS*, pages 465–478. Springer, 2002.
69. M. Wehrle and S. Kupferschmid. Downward pattern refinement for timed automata. *To appear in International Journal on Software Tools for Technology Transfer (STTT)*, 2014.
70. A. Zutshi, S. Sankaranarayanan, J. Deshmukh, and J. Kapinski. A trajectory splicing approach to concretizing counterexamples for hybrid systems. In *Conference on Decision and Control (CDC 2013)*, pages 3918–3925. IEEE, 2013.