Incremental Computation of Succinct Abstractions For Hybrid Systems*

Tomáš Dzetkulič and Stefan Ratschan

Institute of Computer Science, Academy of Sciences of the Czech Republic

Abstract. In this paper, we introduce a new approach to computing abstractions for hybrid dynamical systems whose continuous behavior is governed by non-linear ordinary differential equations. The abstractions try to capture the reachability information relevant for a given safety property as succinctly as possible. This is achieved by an incremental refinement of the abstractions, simultaneously trying to avoid increases in their size as much as possible. The approach is independent of a concrete technique for computing reachability information, and can hence be combined with whatever technique suitable for the problem class at hand. We illustrate the usefulness of the technique with computational experiments.

1 Introduction

In this paper, we study hybrid (dynamical) systems, that is, systems with both discrete and continuous state and evolution. We address the computation of abstractions of hybrid systems that capture a given safety property as succinctly as possible. In other words, given a hybrid system, a set of initial states, and a set of states considered to be unsafe, we try to characterize the set of trajectories (of unbounded length) from an initial to an unsafe state. In cases where the input system does not have such a trajectory (i.e., it fulfills the safety property), we aim at computing an empty abstraction, which verifies the safety property at hand. In other cases, the computed abstraction can be used to guide testing/falsification of the input system.

The abstractions are computed incrementally. This has the obvious advantages of such incremental computation, which are analogous to the case of discrete systems. However, in addition to that, for systems with (complex) continuous dynamics, incrementality has another important aspect: In such cases, even over bounded time, exact reachability computation is possible only for very special cases, and hence (unlike for discrete systems) one *has to* use over-approximation. It is a-priori not clear, how much and where to over-approximate in order to prove a given property. By incremental computation one can adapt the level of over-approximation to the given problem.

* This work was supported by Czech Science Foundation grants 201/08/J020 and 201/09/H057, MŠMT project number OC10048 and institutional research plan AV0Z100300504.

Our technique is parametric in the method used for computing reachability information. We have an implementation that is based on hyper-rectangles and interval constraint propagation [4]. But instantiations with various alternative techniques are possible.

Moreover, our technique is orthogonal to other techniques for abstraction refinement, especially counter-example guided abstraction refinement (CEGAR) [1, 7]: The essential step of our technique tries to capture information about the given safety verification problem *without* increasing the size of the abstraction. Moreover, this step is not inherently exponential in the problem dimension, hence allowing the computation of abstractions of high dimension. In contrast to that, CEGAR approaches are inherently based on increasing the number of states of the abstraction. Both approaches can be used in combination.

Computational experiments show the usefulness of the approach.

Concerning other related work, some approaches to hybrid systems verification do exploit incrementality in reachability computation [12, 11]. But, reuse of analyses only concerns dropping initial/unsafe states that have been shown not to lie on any error trajectory—no reuse is done concerning the analysis itself.

In the verification of time-unbounded properties of discrete (finite or infinite state) systems, fixpoint techniques similar to the ones used in this paper are ubiquitous. Especially, the idea of abstraction slicing [6] tries to keep abstractions small within a discrete CEGAR approach. What sets our approach apart is the fact that we specifically exploit (partial) system continuity, especially concerning the way how constraint solvers may compute approximate (but conservative) information for such systems (Section 3), and concerning the way how continuity restricts possible system evolutions (Section 4.1).

Our previous approach for hybrid systems verification [16] computes abstractions based on an algorithm that is a special case of the method presented here. However, as our computational experiments will show, variants of our method that are different from that special case, show much better behavior, especially for hybrid systems with cyclic behavior. Moreover, our previous approach was restricted to boxes, and a very specific way of computing reachability information based on the mean-value theorem and interval constraint propagation.

The content of the paper is as follows: in Section 2 we describe the used formalism for modelling hybrid systems, in Section 3 we describe how to incrementally improve an abstraction of a given hybrid system, in Section 4 we show how to further improve the technique, in Section 5 we discuss a concrete implementation of the method, in Section 6 we discuss the behavior of this implementation on some computational experiments, and in Section 7 we conclude the paper.

2 Hybrid Systems

In this section, we briefly recall our formalism for modelling hybrid systems. It captures many relevant classes of hybrid systems, and many other formalisms for hybrid systems in the literature are special cases of it. We use a set S to

denote the discrete modes of a hybrid system, where S is finite and nonempty. $I_1, \ldots, I_k \subseteq \mathbb{R}$ are compact intervals over which the continuous variables of a hybrid system range. Φ denotes the state space of a hybrid system, i.e., $\Phi = S \times I_1 \times \cdots \times I_k$.

Definition 1. *A* hybrid system H *is a tuple* $(\texttt{Flow}, \texttt{Jump}, \texttt{Init}, \texttt{Unsafe})$*, where* $\texttt{Flow} \subseteq \Phi \times \mathbb{R}^k$*,* $\texttt{Jump} \subseteq \Phi \times \Phi$*,* $\texttt{Init} \subseteq \Phi$*, and* $\texttt{Unsafe} \subseteq \Phi$*.*

Informally speaking, the predicate `Init` specifies the initial states of a hybrid system and `Unsafe` the set of unsafe states that should not be reachable from an initial state. The relation `Flow` specifies the possible continuous flow of the system by relating states with corresponding derivatives, and `Jump` specifies the possible discontinuous jumps by relating each state to a successor state. Formally, the behavior of H is defined as follows:

Definition 2. *A* flow *of length* $l \geq 0$ *in a mode* $s \in S$ *is a function* $r : [0, l] \to \Phi$ *such that the projection of* r *to its continuous part is differentiable and for all* $t \in [0, l]$*, the mode of* $r(t)$ *is* s*. A* trajectory *of* H *is a sequence of flows* $r_0, \ldots, r_p$ *of lengths* $l_0, \ldots, l_p$ *such that for all* $i \in \{0, \ldots, p\}$*,*

1. *if* $i > 0$ *then* $(r_{i-1}(l_{i-1}), r_i(0)) \in \texttt{Jump}$*, and*
2. *if* $l_i > 0$ *then* $(r_i(t), \dot{r}_i(t)) \in \texttt{Flow}$*, for all* $t \in [0, l_i]$*, where* $\dot{r}_i$ *is the derivative of the projection of* r_i *to its continuous component.*

A (concrete) error trajectory *of a hybrid system* H *is a trajectory* $r_0, \ldots, r_p$ *of* H *such that* $r_0(0) \in \texttt{Init}$ *and* $r_p(l) \in \texttt{Unsafe}$*, where* l *is the length of* r_p*.* H *is* safe *if it does not have an error trajectory.*

In the rest of the paper we will assume an arbitrary, but fixed hybrid system H. We will denote the set of its error trajectories by $\mathcal{E}$.

Instead of defining some concrete syntax in which hybrid systems are described, we keep this paper independent of concrete syntax, and require the user to provide certain operations on hybrid systems that we will introduce in the next section.

3 Incremental Abstract Forward/Backward Computation

For hybrid systems with complex continuous dynamics even bounded time reach set computation necessarily involves over-approximation. In such cases one would like to first compute approximate information using loose over-approximation, and then incrementally refine this.

Our approach will be based on an incremental refinement of a covering of the hybrid systems state space by connected sets that we will call *regions*. We will form the regions in such a way that no two regions will overlap (i.e., regions are allowed to intersect, but only on their boundaries). The method is independent of the class of regions used. In the instantiation of the method used by our

implementation (see Section 5), the regions will be formed by pairs consisting of a mode and a Cartesian product of intervals (i.e., a *box*), but other classes of regions (e.g., based on polyhedra) are equally conceivable.

Definition 3. *An* abstraction *is a graph whose vertices (which we call* abstract states*) may be labelled with labels* `Init` *or* `Unsafe`*. Moreover, to each abstract state, we assign a region. We call the edges of an abstraction* abstract transitions*.*

By abuse of notation, we will usually use the same notation for an abstract state and the region assigned to it.

We call a sequence of abstract states $a_1, \ldots, a_n$ an *abstract trajectory*. If all abstract states and all transitions between successive abstract states in an abstract trajectory belong to an abstraction $\mathcal{A}$, we call it an $\mathcal{A}$-*abstract trajectory* and we denote it by $a_1 \to \cdots \to a_n$. An ($\mathcal{A}$-)abstract trajectory represents the set of concrete trajectories that begin in the region of a_1, move from one abstract state region to the next only if there is a corresponding concrete transition, and end in the region of a_n. We denote this set by $[\![a_1, \ldots, a_n]\!]$ for a given abstract trajectory or $[\![a_1 \to \cdots \to a_n]\!]$ for some $\mathcal{A}$-abstract trajectory.

This can be formalized as follows: We define a *splitting* of a flow l to be a sequence of flows $s_1, \ldots, s_j$ such that $s_1(0) = l(0)$, $s_j(length(s_j)) = l(length(l))$ and if $i > 1$ then $s_{i-1}(length(s_{i-1})) = s_i(0)$. A *trajectory splitting* is a concatenation of splittings of its individual contained flows. $[\![a_1, \ldots, a_n]\!]$ then is the set of all concrete trajectories $r_1, \ldots, r_p$ that have a trajectory splitting $q_1, \ldots q_n$, such that for all i, the mode of abstract state a_i is the same as the projection of q_i to its discrete part and such that the projection of q_i to its continuous part is in the region of a_i.

An $\mathcal{A}$-*abstract error trajectory* is an $\mathcal{A}$-abstract trajectory $a_1 \to \cdots \to a_n$ such that in $\mathcal{A}$, a_1 is labelled initial, and a_n is labelled unsafe.

An abstraction $\mathcal{A}$ represents the set of all concrete trajectories $[\![a_1 \to \cdots \to a_n]\!]$ for abstract error trajectories $a_1 \to \cdots \to a_n$ in the abstraction $\mathcal{A}$. We denote this set by $[\![\mathcal{A}]\!]$.

The intuition is that, during abstraction refinement, the abstraction stays an over-approximation of the set of error trajectories $\mathcal{E}$ of a given system. We say that an abstraction $\mathcal{A}^*$ is a *refinement* of an abstraction $\mathcal{A}$ iff

- the abstraction $\mathcal{A}^*$ represents less trajectories than $\mathcal{A}$, that is, $[\![\mathcal{A}^*]\!] \subseteq [\![\mathcal{A}]\!]$, and
- the abstraction $\mathcal{A}^*$ does not lose error trajectories that are present in $\mathcal{A}$, that is $[\![\mathcal{A}^*]\!] \supseteq [\![\mathcal{A}]\!] \cap \mathcal{E}$.

Now we will come up with an algorithm that will incrementally improve an abstraction by refining it, *without* increasing the number of abstract states in the abstraction. Note that, in particular, $\mathcal{A}$ is a refinement of $\mathcal{A}$ itself, but in practice we will try to remove as many trajectories from the abstraction as possible.

Given abstract states a and a', we will assume a procedure $InitReach(a)$ that computes an over-approximation of the set of points in a that are reachable from an initial point in a, and a procedure $Reach(a, a')$ that computes an over-approximation of the set of points in a' reachable from a according to the system

dynamics [1]. See Section 5.2 for an example how this can be implemented in practice. We assume that smaller inputs improve the precision of these operations, that is:

- $a_1 \subseteq a_2$ implies $\mathit{InitReach}(a_1) \subseteq \mathit{InitReach}(a_2)$
- $a_1 \subseteq a_2$ and $a_1' \subseteq a_2'$ implies $\mathit{Reach}(a_1, a_1') \subseteq \mathit{Reach}(a_2, a_2')$

Furthermore, we assume that these procedures exploit information about empty inputs, that is:

- $a = \emptyset$ implies $\mathit{InitReach}(a) = \emptyset$
- $a = \emptyset$ implies $\mathit{Reach}(a, a') = \emptyset$
- $a' = \emptyset$ implies $\mathit{Reach}(a, a') = \emptyset$

In the following, we require the existence of operations $\sqsubseteq$ and $\uplus$ on regions, with the following properties.

- $\sqsubseteq$ s.t. if $a^* \sqsubseteq a$, then for all $n \in \mathbb{N}$, $i \in \{1 \dots n\}$ and for all regions $b_1 \dots b_{i-1}, b_{i+1} \dots b_n$: $[\![b_1, \dots, b_{i-1}, a^*, b_{i+1}, \dots, b_n]\!] \subseteq [\![b_1, \dots, b_{i-1}, a, b_{i+1}, \dots, b_n]\!]$ i.e., less concrete trajectories follow a given abstract trajectory after replacing an abstract state by smaller one wrt. $\sqsubseteq$ operation.
- $\uplus$ s.t. for all regions $a_1, a_2, b : a_1 \sqsubseteq b \wedge a_2 \sqsubseteq b$ implies $a_1 \uplus_b a_2 \sqsubseteq b$, $a_1 \sqsubseteq a_1 \uplus_b a_2$ and $a_2 \sqsubseteq a_1 \uplus_b a_2$.

Since in our case abstract states represent sets, this can be ensured by the following:

- $\uplus$ s.t. for all $a_1, a_2 \subseteq b$: $a_1 \cup a_2 \subseteq a_1 \uplus_b a_2$ and $a_1 \uplus_b a_2 \subseteq b$
- $\sqsubseteq$ s.t. $a_1 \sqsubseteq a_2$ iff $a_1 \subseteq a_2$

This is our natural interpretation of $\uplus$ and $\sqsubseteq$. However, different choices are possible, as long as they fulfill the above properties: For certain representations of regions it might be convenient to use a weaker form of $\sqsubseteq$, for efficiency reasons. Also, later (Section 4.1) we will see that for being able to encode more information into abstract states, different interpretations of those symbols are convenient.

In the instantiation of the method with boxes, $a_1 \uplus_b a_2$ is the smallest box that includes both argument boxes a_1 and a_2, but does not exceed b (i.e., box union intersected with bounding box), and $\sqsubseteq$ is the subset operation on boxes. Note that for $a_1, a_2 \subseteq b$ defining $a_1 \sqsubseteq a_2$ iff $a_1 \uplus_b a_2 = a_2$ fulfills the above property.

The following algorithm (which we will call *pruning algorithm*) computes a refinement of a given abstraction $\mathcal{A}$:

[1] Here we do not assume any time bound. However, it is possible to apply our method also for implementations of those procedures that compute reachability over bounded time. This would only require slight modifications of the algorithms.

$\mathcal{A}^* \leftarrow$ copy of $\mathcal{A}$, all regions set to $\emptyset$, no initial labels, no edges
// from now on, for every abstract state a of $\mathcal{A}$,
// we denote by a^* the corresponding abstract state of $\mathcal{A}^*$
for all $a \in \mathcal{A}$, a is initial
    $a^* \leftarrow \mathit{InitReach}(a)$
    if $a^* \neq \emptyset$ **then**
        mark a^* as initial
while there is a pair of abstract states (a_1, a_2) in $\mathcal{A}$ with
    $a_1 \rightarrow a_2$, s.t. $\mathit{Reach}(a_1^*, a_2) \not\sqsubseteq a_2^*$ **do**
        if $a_1^* \not\rightarrow a_2^*$ in $\mathcal{A}^*$ **then** introduce an edge $a_1^* \rightarrow a_2^*$ into $\mathcal{A}^*$
        $a_2^* \leftarrow (a_2^* \uplus_{a_2} \mathit{Reach}(a_1^*, a_2))$
return $\mathcal{A}^*$

Note that the pruning algorithm does *not* increase the size (i.e., the number of nodes) of the abstraction. Still it deduces some some interesting information:

Theorem 1. *The result of the pruning algorithm is a refinement of the input abstraction $\mathcal{A}$.*

Proof. We have to prove two items:

- $[\![\mathcal{A}^*]\!] \subseteq [\![\mathcal{A}]\!]$: This follows from the following:
  - the set of initial/unsafe marks of $\mathcal{A}^*$ is a subset of the set of marks of $\mathcal{A}$
  - the set of edges of $\mathcal{A}^*$ is a subset of the set of edges of $\mathcal{A}$
  - the abstract states of $\mathcal{A}^*$ are subsets of the corresponding abstract states of $\mathcal{A}$ since $\mathit{InitReach}(a) \subseteq a$, and $a_2^* \uplus_{a_2} \mathit{Reach}(a_1^*, a_2) \subseteq a_2$ from the definition of $\uplus$.
- $[\![\mathcal{A}^*]\!] \supseteq [\![\mathcal{A}]\!] \cap \mathcal{E}$: Let T be an $\mathcal{A}$-error trajectory in $[\![\mathcal{A}]\!] \cap \mathcal{E}$. We prove that T is an element of $[\![\mathcal{A}^*]\!]$. Let $a_1 \rightarrow \cdots \rightarrow a_n$ be an $\mathcal{A}$-abstract error trajectory s.t. $T \in [\![a_1 \rightarrow \cdots \rightarrow a_n]\!]$. We prove that for the corresponding $\mathcal{A}^*$-abstract trajectory $a_1^* \rightarrow \cdots \rightarrow a_n^*$, also $T \in [\![a_1^* \rightarrow \cdots \rightarrow a_n^*]\!]$. Let T_i be a fragment of trajectory T before the transition $a_i \rightarrow a_{i+1}$ is made. We will prove by induction that for all i: $T_i \in [\![a_1^* \rightarrow \cdots \rightarrow a_i^*]\!]$ and observation $T = T_n$ concludes the proof.
  - a_1 contains the initial point of T and $\mathit{InitReach}(a_1)$ contains all points of T in a_1. After the first loop of the algorithm, a_1^* is initial in $\mathcal{A}^*$ and it is equal to $\mathit{InitReach}(a_1)$. After that, all regions in $\mathcal{A}^*$ only increase w.r.t. $\sqsubseteq$ operation and that implies that $T_i \in [\![a_1^*]\!]$.
  - We assume that for some $i < n$: $T_i \in [\![a_1^* \rightarrow \cdots \rightarrow a_i^*]\!]$. Since T leaves a_i to a_{i+1}, a_i^* contains all the points of T in a_i, $\mathit{Reach}(a_i^*, a_{i+1})$ contains all the points of T in a_{i+1}. Since $\mathit{Reach}(a_i^*, a_{i+1})$ is non-empty, the abstract transition $a_i^* \rightarrow a_{i+1}^*$ is introduced into $\mathcal{A}^*$ and from the while cycle termination condition, we have: $\mathit{Reach}(a_i^*, a_{i+1}) \sqsubseteq a_{i+1}^*$, thus $T_{i+1} \in [\![a_1^* \rightarrow \cdots \rightarrow a_{i+1}^*]\!]$.

□

Note that the second part of the proof does *not* depend on the definition of $\uplus$. Hence, for ensuring that no error trajectory is lost by the algorithm, $\uplus$ does not necessarily have to fulfill the requirement stated above. This requirement just ensures that all computed reachability information is captured, and hence will not have to be re-computed again.

Note moreover, that it is a-priori not clear, that the pruning algorithm terminates. However, termination can be ensured, for example, by using a representation for which, for given regions a and b, there is not infinite chain $a \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq b$. See Section 5.1 for a more detailed discussion of this issue.

As already mentioned, the pruning algorithm tries to deduce information about a given system without increasing the size of the abstraction. In cases, where it can deduce no more information, we have to fall back to some increase of the size of the abstraction (cf. to a similar approach in constraint programming where one falls back to exponential-time splitting, when polynomial-time deduction does not succeed any more).

We do this by the *Split* operation that chooses an abstract state and splits it into two, copying all the involved edges and introducing edges between the two new states. All the labels and abstract transitions to other abstract states are copied as well. Moreover, two new abstract transitions that connect the original abstract state with its copy are added. The region assigned to the abstract state is equally split among two abstract states. Such an refinement decreases the amount of over-approximation in subsequent calls to the pruning algorithm due to the properties of the *Reach* and *InitReach*. We chose such a simple splitting strategy to show the usefulness of our approach in isolation. However, it is possible to use much more sophisticated splitting strategies, for example, one could use one CEGAR step [1, 7] instead of our simple splitting technique.

It is clear that the pruning algorithm can also be done backward in time (i.e., removing parts of the abstraction not leading to an unsafe state)[12, 11]. We will denote the resulting algorithm by $Prune^{-}(\mathcal{A})$. Now we have to following overall algorithm for computing increasingly fine abstractions:

while there is an $\mathcal{A}$-abstract error trajectory
  $\mathcal{A} \leftarrow Prune(\mathcal{A})$
  $\mathcal{A} \leftarrow Prune^{-}(\mathcal{A})$
  $\mathcal{A} \leftarrow Split(\mathcal{A})$
return "safe"

Since neither pruning nor splitting removes an error trajectory, the absence of an $\mathcal{A}$-abstract error trajectory at the termination of the while loop implies the absence of an error trajectory of the original system. Hence, in such a case, the algorithm correctly returns the information that the input system was safe.

Note that forward pruning may enable further backward pruning and vice versa, hence the algorithm may be extended in such a way that forward and backward pruning are done in a loop until no further improvement occurs. If either

forward or backward pruning is dropped from the algorithm, it will incrementally compute a tighter and tighter over-approximation of the (forward/backward) reach set.

4 Improvements

In this section we introduce three improvements to the basic pruning algorithm. The first improvement exploits the specific structure of hybrid systems (continuous time, discontinuous jumps) and the two other improvements introduce more incrementality into the way the algorithms handles the abstraction.

4.1 Exit Regions

When computing $\mathit{Reach}(a_1^*, a_2)$ we get an over-approximation of the set of points in a_2 reachable from a_1^* according to the system dynamics. However, trajectories can leave a_1^* not arbitrarily, but only at points fulfilling certain conditions:

- flows can leave a_1^* only over its boundary, and
- jumps can leave a_1^* only over parts of a_1^* belonging to the projection of the set `Jump` to its first part corresponding to jump source, that is, over the set $\{(m, x) \in S \times \mathbb{R}^k \mid \exists x' \in \mathbb{R}^k, m' \in S \,.\, ((m, x), (m', x')) \in \texttt{Jump}\}$.

Hence, in the computation of $\mathit{Reach}(a_1^*, a_2)$, instead of the full region a_1^* we can use a subset fulfilling this condition. However, a_1^* already is an over-approximation. Hence, it is better to directly compute an over-approximation of the points fulfilling this condition, as follows:

We assume that the function $\mathit{InitReach}(a)$, in addition to an over-approximation of the set of points in a that are reachable from an initial point in a, also computes an over-approximation of the set of states in a through which a trajectory starting from an initial point in *a leaves* a. Moreover, we assume that $\mathit{Reach}(a, a')$ in addition to an over-approximation of the set of points in a' reachable from a, computes an over-approximation of the set of states in a' through which a trajectory coming from *a leaves* a'.

We store those additional regions with abstract states, calling them *exit regions*. For extending the operations $\uplus$ and $\sqsubseteq$ to abstract states consisting of a region and an exit region we will need the following:

Lemma 1. *Assume a region a that contains a trajectory T, an $a^* \sqsubseteq a$ that also contains T and an exit region a_e containing the point L where T leaves a. Then T also leaves a^* at L.*

Proof. Since a^* contains all points of T in a, it also contains L. Let us assume that T leaves a^* at L^* and that $L^* \neq L$. Since a^* contains T and both L and L^*, it also contains a flow r, that is a part of the trajectory T, such that $r(0) = L \wedge r(\mathit{length}(r)) = L^*$. Since $a^* \sqsubseteq a$, flow r is also in a and that contradicts the assumption that T leaves a at L. □

Hence, T leaves a^* at a point that already was an element of the exit region a_e of the original box a. This motivates us to extend the check $\sqsubseteq$ to abstract states with exit regions in such a way that for abstract states (i.e., region/exit region pairs) (c_r, c_e) and (d_r, d_e), $(c_r, c_e) \sqsubseteq (d_r, d_e)$ iff $c_r \sqsubseteq d_r$ and $c_e \sqsubseteq d_e$. Moreover, $\uplus$ will also apply the corresponding operation on both the region and exit region.

This makes it possible to implement *InitReach* and *Reach* in such a way that they are compatible with $\sqsubseteq$.

Now, we extend the semantics of abstract trajectories in such a way that concrete trajectories have to leave abstract states through their exit regions. The correctness of the pruning algorithm is preserved, if the new operations $\uplus$ and $\sqsubseteq$ still fulfill the necessary properties. This is clearly the case for $\uplus$. But also $\sqsubseteq$ again fulfills the required property:

Theorem 2. *For $\sqsubseteq$ extended with exit regions, for all abstract states $a = (a_r, a_e)$ and $a^* = (a_r^*, a_e^*)$, if $a_r^* \sqsubseteq a_r$ and $a_e^* \sqsubseteq a_e$ then $a^* \sqsubseteq a$.*

Proof. Let T be a trajectory in $[\![b_1, \ldots b_{i-1}, a^*, b_{i+1}, \ldots, b_n]\!]$. We show that this trajectory is also in $[\![b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n]\!]$. Since $a_r^* \sqsubseteq a_r$, T is covered by the region a_r. All we have to prove is that T leaves a at the point in a_e. From Lemma 1, we know that T leaves a and a^* at the same point L. Since T is in $[\![b_1, \ldots b_{i-1}, a^*, b_{i+1}, \ldots, b_n]\!]$, L belongs to a_e^* and since $a_e^* \sqsubseteq a_e$ it also belongs to a_e. □

This implies that Theorem 1 is also valid for abstractions with exit regions:

Corollary 1. *The result of the pruning algorithm extended with exit regions is a refinement of the input abstraction $\mathcal{A}$.*

Exit regions can be computed in both forward (i.e., $Prune(\mathcal{A})$) and backward (i.e., $Prune^-(\mathcal{A})$) computation. We will call such a region computed during backward computation an *enter region*. Exit regions are computed only during forward computation, while enter regions are computed only during backward computation. When doing computation in one direction, the dual exit regions do not change.

Moreover, in the computation of $Reach(a_1^*, a_2)$, one can exploit not only the exit region of a_1^*, where the trajectory leaves a_1^*, but also the enter region of a_2, where the trajectory has to enter the region of a_2. Such an information can then be used by the underlying reachability computation algorithm to constrain the reachable states.

Furthermore, in $Split(\mathcal{A})$, new boundaries are created. A trajectory can now enter and exit the region through this new boundary and we have to create new abstract states in such a way that the new boundary is part of the enter and exit regions. For new abstract state a^*, new boundary b and original exit region a_e, we create a new exit region a_e^* in such a way that $(a_e \cap a^*) \cup b \subseteq a_e^*$. This clearly does not change the set of represented error trajectories.

4.2 Avoided Redundant Edge Checks

One disadvantage of the pruning algorithm is that it may do redundant tests for the condition $Reach(a_1^*, a_2) \not\sqsubseteq a_2^*$ in the update function. Whenever such a test has been made, this can be remembered until the information is not valid any more.

To this purpose we add additional edges to the abstraction that we label with $\sqsubseteq$ (and which we call *consistency edges*). We keep the invariant (that we will call *consistency invariant*) that whenever $a_1^* \rightarrow_{\sqsubseteq} a_2^*$, then $Reach(a_1^*, a_2) \sqsubseteq a_2^*$.

Moreover we use a procedure `propChange`(a) that, for every a' with $a \rightarrow a'$ deletes every edge $a \rightarrow_{\sqsubseteq} a'$. This allows us to change the while loop in the pruning algorithm as follows:

$\mathcal{A}^* \leftarrow$ copy of $\mathcal{A}$, all regions set to $\emptyset$, no initial labels, no edges
// from now on, for every abstract state a of $\mathcal{A}$,
// we denote by a^* the corresponding abstract state of $\mathcal{A}^*$
for all $a \in \mathcal{A}$, a is initial
    $a^* \leftarrow InitReach(a)$
    if $a^* \neq \emptyset$ then
        mark a^* as initial
        propChange(a^*)
while there is a pair of abstract states (a_1, a_2) in $\mathcal{A}$ with
    $a_1 \rightarrow a_2$, s.t. $a_1^* \not\rightarrow_{\sqsubseteq} a_2^*$ and $Reach(a_1^*, a_2) \not\sqsubseteq a_2^*$ **do**
        introduce an edge $a_1^* \rightarrow_{\sqsubseteq} a_2^*$
        if $a_1^* \not\rightarrow a_2^*$ in $\mathcal{A}^*$ **then** introduce an edge $a_1^* \rightarrow a_2^*$ into $\mathcal{A}^*$
        $a_2^* \leftarrow a_2^* \uplus_{a_2} Reach(a_1^*, a_2)$
        propChange(a_2^*)
return $\mathcal{A}^*$

Algorithms of such a type are known in the literature under them name "chaotic iteration" or "worklist algorithms". They have been used and studied mainly in the context of abstract interpretation [8, 5, 13] and constraint satisfaction [2, 3].

Theorem 3. *Independent of the consistency edges of the input $\mathcal{A}$, the improved pruning algorithm computes the same result as the original one.*

Due to space restrictions we omit the proof of this theorem.

4.3 Incremental Refinement of Abstraction

Now observe that splitting, or dual pruning, only changes a part of the abstraction. Still, the pruning algorithms do a complete re-computation. This is not necessary, and in order to avoid it:

– We mark all abstract states for which we know, that a re-computation will not improve, with the mark `Cons` (the *consistency mark*).
– Whenever splitting or dual pruning changes an abstract state, we delete this consistency mark, and all consistency marks of states reachable from it.
– At the beginning of the pruning algorithm for all abstract states we reset the abstract state with the result of *InitReach* only if the consistency mark is not set. Abstract states with the consistency mark, retain the value from the input abstraction $\mathcal{A}$.

Since we do separate forward and backward pruning, we also need separate consistency marks for both cases. Splitting removes both consistency marks at the same time.

5 Implementation

The method introduced in this paper can be instantiated with various techniques for forming and representing abstract states, and computing reachability information. Nonetheless, in order to study the viability of the approach, we created a specific implementation that uses boxes (with floating point endpoints) for representing regions.

In the *Split* operation on boxes, we pick a splitting dimension of the box assigned to the region and we split the box into halves using this dimension. For picking the splitting dimension, a round-robin strategy has proved to be the useful heuristics [16].

5.1 Widening

Termination of the algorithm that uses boxes for region representation is ensured by doing all the computations on the *finite* set of floating point numbers. Hence there are only finitely many possibilities of changing boxes with $\uplus$, until a fixpoint is reached. This may in some cases lead to stuttering (i.e., many small improvements by close floating point numbers) and thus to a slow convergence to a fixpoint.

We designed a widening strategy to avoid stuttering and speed up the convergence of the algorithm. Here, widening is applied to the line $a_2^* \leftarrow (a_2^* \uplus_{a_2} Reach(a_1^*, a_2))$ of the algorithm, that we replace with $a_2^* \leftarrow widening(a_2, a_2^*, a_2^* \uplus_{a_2} Reach(a_1^*, a_2))$. Informally speaking, when a change from a_2^* to $a_2^* \uplus_{a_2} Reach(a_1^*, a_2)$ would be small compared to a_2, widening gives us region a bit bigger than $a_2^* \uplus_{a_2} Reach(a_1^*, a_2)$, but still smaller or equal than a_2.

Formally, the widening operates on each individual dimension separately (recall that in our implementation regions are represented by boxes, that is Cartesian products of intervals). Let b be the box $a_2^* \uplus_{a_2} Reach(a_1^*, a_2)$ and in the considered dimension i let $[\underline{a_2}_i, \overline{a_2}_i]$, $[\underline{a_2^*}_i, \overline{a_2^*}_i]$ and $[\underline{b}_i, \overline{b}_i]$ be the intervals of the boxes a_2, a_2^* and b respectively. We will denote by $w_i := \overline{a_2}_i - \underline{a_2}_i$ the width of the box a_2 in its i-th dimension. For a given, pre-chosen constant $c \in [0, 1]$ (which

we call *widening-constant*), the result of widening in the considered dimension is $[\underline{a_2}_i, \overline{a_2}_i] \cap [min(\underline{b}_i, \underline{a_2^*}_i - cw), max(\overline{b}_i, \overline{a_2^*}_i + cw)]$. Hence, widening depends on the width of the box in the original abstraction. After splitting the region, widening in the next execution of pruning algorithm adds smaller parts of the region. In our implementation, we use the widening constant $c = 1/16$.

Our previous approach for hybrid systems verification [16] computes abstractions based on an algorithm that can be viewed as is a special case of the method presented here that uses some extreme form of widening where $widening(a_2, a_2^*, a_2^* \uplus_{a_2} Reach(a_1^*, a_2)) = Reach(a_1, a_2)$. In other words, the method does not accumulate reachability information at all, and immediately uses the weaker old abstract state a_1 instead of a_1^*. This results in a much simpler algorithm where instead of our while loop, reachability information is computed only once for each neighbor in the abstraction. However, as our computational experiments will show, different variants of the method presented in this paper show much better behavior than that particular instantiation, especially for hybrid systems with cyclic behavior.

5.2 Computation of Reachability Information

For computing reachability information of hybrid systems, as needed by our functions *Reach* and *InitReach*, one has to come up with a way of handling the differentiation operator. Here, one can over-approximate it to a purely polynomial constraint using Taylor expansion. Since *Reach* and *InitReach* only need reachability information in bounded regions, this results in bounds on all derivatives, and hence *Reach* and *InitReach* can be computed in a conservative way, even over unbounded time. Our current implementation only uses Taylor polynomials of degree one (corresponding to the mean-value theorem). This is a very crude over-approximation, which is sufficient for studying the behavior of the algorithm presented in this paper. In order to arrive at an efficient implementation, one would have to use Taylor polynomials of higher degrees, which is an easy, but tedious, implementation exercise.

The resulting constraint contains variables corresponding to derivatives and to the source points of trajectories and jumps. In order to arrive at a description of the set of reachable states, those have to be eliminated. In theory, one could use quantifier elimination procedures for this (cf. the notion of logical interpretation [18]). However, in the case of polynomials and real numbers, those do not scale in the problem dimension at all. Hence we simply project the boxes computed by interval constraint propagation [4] (we also have a generalization of this technique available [14]). We also have investigated an alternative method based on an over-approximation of Fourier-Motzkin elimination [10].

6 Computational Experiments

We studied the behavior of our implementation in two scenarios: First, the scenario where the abstraction refinement method of this paper is used for hybrid

systems verification. And second, where it is used for computing abstractions of high-dimensional systems that can then be used for other purposes (e.g., running simulations for testing). Since we did not want to test the underlying reachability computation algorithms, but the incremental abstract computation algorithm introduced in this paper, we only used the highly over-approximating reachability computation described in Section 5.2. For practical analysis of concrete hybrid systems, one can use much more sophisticated reachability algorithms, adapted to the type of system at hand.

For studying the first scenario, we took benchmarks from our database (available on the web at http://hsolver.sourceforge.net/benchmarks). For each example, the behavior of a full verification cycle is described in Table 1. Here, the three main columns describe the three widening strategies described in Section 5.1, i.e., verification algorithm with (moderate) widening, without widening, and with the extreme widening strategy that corresponds to the previous algorithm from [16]. The column *Refine* represents the number of refinement steps of the overall algorithm for safety verification from Section 3. All timings were measured on PC with Intel Core 2 3.0GHz CPU and 4GB RAM. From the measured results we conclude that an extreme widening strategy does not pay off and can solve less benchmark examples than approaches with less aggressive widening. This is illustrated particularly nicely by the *cycle* benchmark, where verification finishes in one refinement step with the method presented in this paper, while the previous approach is not able to solve this benchmark at all. The reason is that for such benchmarks with cyclic system behavior, widening should not prevent the analysis of full system cycles.

We also conclude that the moderate widening strategy pays off since it causes only negligible run time increase, but removes stuttering in the *mutant* benchmark and does not increase the number of refinement steps in any benchmark.

For studying the second scenario we conducted computational experiments with high dimensional problems. For this purpose, we created two example problems using the parallel composition of several instances of simpler benchmarks. Our first high dimensional example contains 102 clock variables and one mode. The example was not verified in the time limit of one hour, however, the volume of all boxes in the abstraction was pruned from the size of approximately 8×10^{40} down to 1×10^{30}. Hence, our method resulted in an abstraction whose volume is 8×10^{10} times smaller, and that contains additional information about initial and unsafe states, and abstract transitions. The result can be used to guide testing or falsification [17], since we know that the pruned parts of the abstraction do not contain any error trajectory. Our second high dimensional benchmark is a benchmark with 100 variables with non-linear evolution, one clock variable and one mode. It was verified using thirteen refinement steps in 36 minutes. From the measured results in high dimension we conclude, that our technique for abstraction refinement is feasible also in case of high dimensional benchmarks.

The first benchmark we used was the parallel composition of 34, three-variable *1-flow* benchmarks from our database. The second benchmark was a

Table 1. Experimental results

	F/B + widening		F/B reachability		orig. alg.	
Name	Refine	Time	Refine	Time	Refine	Time
1-flow	2	$< 1s$	2	$< 1s$	2	$< 1s$
2-tanks	2	$< 1s$	2	$< 1s$	9	$1s$
car	1	$< 1s$	1	$< 1s$	1	$< 1s$
circuit	101	$752s$	101	$706s$	N/A	
clock	15	$1s$	15	$1s$	16	$2s$
convoi-1	1	$< 1s$	2	$< 1s$	2	$< 1s$
convoi	19	$4s$	20	$4s$	N/A	
cycle	1	$< 1s$	1	$< 1s$	N/A	
eco	24	$18s$	24	$18s$	N/A	
fischer2	1	$< 1s$	1	$< 1s$	6	$40s$
focus	5	$< 1s$	5	$< 1s$	5	$< 1s$
hallstah	134	$463s$	134	$427s$	N/A	
mixing	1	$< 1s$	1	$< 1s$	1	$< 1s$
mutant	2	$< 1s$	N/A		N/A	
sinusoid	118	$121s$	118	$109s$	N/A	
van-der-pole	1	$< 1s$	1	$< 1s$	2	$< 1s$

parallel composition of 50 three-variable *clock* benchmarks, where all the instances share the common third clock variable, but have separate variables for non-linear evolution. The original clock benchmark needs fifteen refinement steps including fifteen splitting steps. Splitting in such a high dimension would create huge abstractions, so we have simplified the instances of the benchmark, changing the constant in the differential equation from $\dot{x_1} = -5.5x_2 + x_2^2$ to $\dot{x_1} = -5.5x_2 + 0.3x_2^2$. We believe, that instantiation of the technique with a reachability computation method that requires less splitting steps allows verification of the benchmark without this simplification.

7 Conclusion

In this paper we introduced a technique for computing abstractions of hybrid systems that can handle arbitrary hybrid systems for which certain reachability computation algorithms are provided. The abstractions are computed in such a way that they are as succinct as possible. Computational experiments confirm the usefulness of the approach. Especially, the approach can compute information for high-dimensional systems that can be used for guiding testing or falsification. In future work we will instantiate this technique with different methods for reachability computation and we will try to create an algorithm that provably terminates for all robust inputs [9, 15].

References

1. R. Alur, T. Dang, and F. Ivančić. Predicate abstraction for reachability analysis of hybrid systems. *Trans. on Embedded Computing Sys.*, 5(1):152–199, 2006.
2. K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, 1999.
3. K. R. Apt. The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems*, 22(6):1002–1036, 2000.
4. F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16, pages 571–603. Elsevier, Amsterdam, 2006.
5. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, M. Broy, and I. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer Berlin / Heidelberg, 1993.
6. I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing abstractions. *Fundamenta Informaticae*, 89(4):369–392, 2008.
7. E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. J. of Foundations of Comp. Sc.*, 14(4):583–604, 2003.
8. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 1–12, 1977.
9. W. Damm, G. Pinto, and S. Ratschan. Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. *International Journal of Foundations of Computer Science (IJFCS)*, 18(1):63–86, 2007.
10. T. Dzetkulič and S. Ratschan. How to capture hybrid systems evolution into slices of parallel hyperplanes. In *ADHS'09: 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 274–279, 2009.
11. G. Frehse, B. H. Krogh, and R. A. Rutenbar. Verifying analog oscillator circuits using forward/backward abstraction refinement. In *DATE 2006: Design, Automation and Test in Europe*, 2006.
12. T. A. Henzinger and P.-H. Ho. A note on abstract interpretation strategies for hybrid automata. volume 999 of *LNCS*, pages 252–264, 1995.
13. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
14. S. Ratschan. Efficient solving of quantified inequality constraints over the real numbers. *ACM Transactions on Computational Logic*, 7(4):723–748, 2006.
15. S. Ratschan. Safety verification of non-linear hybrid systems is quasi-semidecidable. In *TAMC 2010: 7th Annual Conference on Theory and Applications of Models of Computation*, volume 6108 of *LNCS*, pages 397–408. Springer, 2010.
16. S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Transactions in Embedded Computing Systems*, 6(1), 2007.
17. S. Ratschan and J.-G. Smaus. Finding errors of hybrid systems by optimising an abstraction-based quality estimate. In C. Dubois, editor, *Tests and Proofs*, volume 5668 of *LNCS*, pages 153–168. Springer, 2009.
18. A. Tiwari and S. Gulwani. Logical interpretation: Static program analysis using theorem proving. In F. Pfenning, editor, *Automated Deduction CADE-21*, volume 4603 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin / Heidelberg, 2007.