RECURSIVE AND BACKWARD REASONING IN THE VERIFICATION ON HYBRID SYSTEMS

Stefan Ratschan
Institute of Computer Science, Czech Academy of Sciences, Prague, Czech Republic

Zhikun She
LMIB and School of Science, Beihang University, Beijing, China

Keywords: Hybrid Systems, Verification, Constraint Propagation.

Abstract: In this paper we introduce two improvements to the method of verification of hybrid systems by constraint propagation based abstraction refinement that we introduced earlier. The first improvement improves the recursive propagation of reachability information over the regions constituting the abstraction, and the second improvement reasons backward from the set of unsafe states, instead of reasoning forward from the set of initial states. Detailed computational experiments document the usefulness of these improvements.

1 INTRODUCTION

Safety verification of hybrid systems is the problem of verifying that for a given hybrid system no trajectory that starts in an initial state ever reaches an unsafe state. Abstraction refinement approaches this problem by iteratively refining an overapproximation of the hybrid system (the *abstraction*) that is constructed in such a way that the safety of the abstraction implies the safety of the concrete system. In our method of constraint propagation based abstraction refinement (Ratschan and She, 2007) the abstraction is built by decomposing the state-space into hyper-rectangles (*boxes*) and using a constraint solver to test, which of these boxes might contain an initial/unsafe state, and which box might be reachable from another box.

In this paper, we introduce two improvements to the method: recursive reasoning and backward reasoning. Recursive reasoning improves the way the method removes elements from boxes for which it can prove that they are not reachable from an initial state. The original method argues that a point in a box is not reachable from another box, if it is not reachable from a point on the common boundary. In this paper we strengthen this condition using a convenient over-approximation of the requirement that this common point on the boundary again has to be reachable. Backward reasoning uses the observation that we can remove not only elements from boxes for which we can prove that they are not reachable from an initial state, but also elements for which we can prove that they do not lead to an unsafe state.

There are various other methods for the verification of hybrid systems that use a decomposition of the state space into boxes (Preußig et al., 1999; Kloetzer and Belta, 2006). Another paper (Frehse et al., 2006) employs backward reasoning in a more coarse-grained manner than in this paper, computing over-approximations of increasingly precise forward and backward reach sets.

The content of the paper is as follows: In Section 2 we review our hybrid systems formalism, and in Section 3 we review our verification method and discuss properties of the underlying constraint solving technique; in Sections 4 and 5 we introduce the first improvement to our verification method, and in Section 6 our second improvement and the combination of the two improvements; in Section 7 we present some computational experiments, and in Section 8 we conclude the paper.

2 VERIFICATION OF HYBRID SYSTEMS

Hybrid systems are systems with continuous and discrete state variables. In this paper, we briefly recall our formalism for modeling hybrid systems (Ratschan and She, 2007).

We use a set S to denote the modes of a hybrid

Ratschan S. and She Z. (2008).
RECURSIVE AND BACKWARD REASONING IN THE VERIFICATION ON HYBRID SYSTEMS.
In *Proceedings of the Fifth International Conference on Informatics in Control, Automation and Robotics - SPSMC*, pages 65-71
DOI: 10.5220/0001475500650071

system, where S is finite and nonempty. $I_1,\dots,I_k \subseteq \mathbb{R}$ 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 $(Flow, Jump, Init, Unsafe)$, where $Flow \subseteq \Phi \times \mathbb{R}^k$, $Jump \subseteq \Phi \times \Phi$, $I \subseteq \Phi$, and $Unsafe \subseteq \Phi$.*

Informally speaking, the predicate *Init* specifies the initial states of a hybrid system and *Unsafe* the states that should not be reachable from an initial state. The relation *Flow* specifies how the system may develop continuously by relating each state to the possible corresponding derivatives, and *Jump* specifies how H may change states discontinuously by relating each state to its possible successor states. 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 $r_0,\dots,r_p$ of lengths $l_0,\dots,l_p$ such that for all $i \in \{0,\dots,p\}$,*

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

Definition 3. *A (concrete)* counterexample *of a hybrid system H is a trajectory $r_0,\dots,r_p$ of H such that $r_0(0) \in Init$ and $r_p(l) \in Unsafe$, where l is the length of r_p. H is* safe *if it does not have a counterexample.*

We use the following constraint language to describe hybrid systems and corresponding safety verification problems. The variable s ranges over S and the tuple of variables $\vec{x} = (x_1,\dots,x_k)$ ranges over $I_1 \times \cdots \times I_k$, respectively. In addition, to denote the derivatives of $x_1,\dots,x_k$ we use the tuple of variables $\dot{\vec{x}} = (\dot{x}_1,\dots,\dot{x}_k)$ that ranges over $\mathbb{R}^k$, and to denote the targets of jumps, we use the primed variable s' and the tuple of variables $\vec{x}' = (x'_1,\dots,x'_k)$ that range over S and $I_1 \times \cdots \times I_k$, respectively. Constraints are arbitrary Boolean combinations of equalities and inequalities over terms that may contain function symbols, such as $+$, $\times$, exp, sin, and cos.

We assume in the remainder of the text that a hybrid system is described by our constraint language. That means, the flows of a hybrid system are given by a constraint $Flow(s,\vec{x},\dot{\vec{x}})$, the jumps are given by a constraint $Jump(s,\vec{x},s',\vec{x}')$, the initial states are given by a constraint $Init(s,\vec{x})$, and a constraint $Unsafe(s,\vec{x})$ describes the unsafe states. To simplify notation, we do not distinguish between a constraint and the set it represents.

Example 1. *Consider the following simple hybrid system with the modes m_1, m_2 and the continuous variables x_1, x_2 which both range over the interval $[0,2]$, i.e, $\Phi = \{m_1, m_2\} \times [0,2] \times [0,2]$.*

The set of initial states are given by the $Init(s,(x_1,x_2)) = (s = m_1 \wedge x_1 = 0 \wedge x_2 = 0)$. The constraint $Unsafe(s,(x_1,x_2)) = (x_1 > 1.5 \wedge x_2 = 1.5)$ describes the set of unsafe states. The hybrid system can switch modes from m_1 to m_2 if $x_2 = 1$, i.e., $Jump(s,(x_1,x_2),s',(x'_1,x'_2)) = (s = m_1 \wedge x_2 = 1) \to (s' = m_2 \wedge x'_1 = x_1 \wedge x'_2 = x_2)$. The continuous behavior is described by constants. In addition, for a flow in mode m_1, the constraint $0 \leq x_1 \leq 1$ must hold. The corresponding flow constraint is

$$\begin{gathered} Flow(s,(x_1,x_2),(\dot{x}_1,\dot{x}_2)) = \\ (s = m_1 \to (\dot{x}_1 = 1 \wedge \dot{x}_2 = 1 \wedge 0 \leq x_1 \leq 1)) \wedge \\ (s = m_2 \to (\dot{x}_1 = 1 \wedge \dot{x}_2 = -1)) \,. \end{gathered}$$

Note that the constraint $0 \leq x_1 \leq 1$ in flow forces a jump from mode m_1 to m_2 if x_1 becomes 1.

Obviously, this hybrid system is safe. ■

3 FORWARD SEARCH BASED ABSTRACTION REFINEMENT

In this section, we review our previous approach (Ratschan and She, 2007) for verifying safety of hybrid systems using constraint propagation based abstraction refinement.

We abstract to systems of the following form:

Definition 4. *A* discrete system *over a finite set S is a tuple $(Trans, Init, Unsafe)$ where $Trans \subseteq S \times S$ and $Init \subseteq S$, $Unsafe \subseteq S$. We call the set S the* state space *of the system.*

In contrast to Definition 1, here the state space is a parameter. This will allow us to add new states to the state space during abstraction refinement.

Definition 5. *A* trajectory *of a discrete system $(Trans, Init, Unsafe)$ over a set S is a function $r : \{0,\dots,p\} \mapsto S$ such that for all $t \in \{1,\dots,p\}$, $(r(t-1), r(t)) \in Trans$. The system is* safe *if and only if there is no trajectory from an element of Init, to an element of Unsafe.*

When we use abstraction to analyze hybrid systems, the abstraction should over-approximate the concrete system in a conservative way: if the abstraction is safe, then the original system should also be safe. If the current abstraction is not yet safe, we refine the abstraction, that is, we include more information about the concrete system into it. This results in Algorithm 1.

Algorithm 1: Abstraction Refinement.

```
Require: a hybrid system $H$ described by constraints
Ensure: “safe”, if the algorithm terminates
  let $A$ be a discrete abstraction of the hybrid system represented by $H$
  while $A$ is not safe do
    refine the abstraction $A$
  end while
```

In order to implement this algorithm, we need to fix the state space of the abstract system. Here we use pairs (s,B), where s is one of the modes $\{\mathtt{s_1},\ldots,\mathtt{s_n}\}$ and B is a hyper-rectangle (*box*), representing subsets of the concrete state space Φ. Together with an abstract state, we store the information whether it is initial or unsafe and the information from which other states it is reachable. We call such information the *marks* of the state. For the initial abstraction we use the state space $\{(\mathtt{s_i},\{\vec{x} \mid (\mathtt{s_i},\vec{x}) \in \Phi\}) \mid 1 \leq i \leq n\}$, where all states are marked as initial, and unsafe, and all transitions between states are possible.

For refining the abstraction, we split a box into two pieces, replace one abstract state by two, and include more information from the concrete system into the abstract one by removing unreachable elements from the boxes, removing superfluous marks from the new abstract states, and removing unreachable states from the abstraction.

To remove unreachable elements from the boxes representing the abstraction, we use a constraint that formalizes when an element of the concrete state space might be reachable, and then remove elements that do not fulfill this constraint. In order to do this, for a box $B = [\underline{x}_1,\overline{x}_1] \times \cdots \times [\underline{x}_k,\overline{x}_k]$, we let its j-th lower face be $[\underline{x}_1,\overline{x}_1] \times \cdots \times [\underline{x}_j,\underline{x}_j] \times \cdots \times [\underline{x}_k,\overline{x}_k]$ and its j-th upper face be $[\underline{x}_1,\overline{x}_1] \times \cdots \times [\overline{x}_j,\overline{x}_j] \times \cdots \times [\underline{x}_k,\overline{x}_k]$. Two boxes are *non-overlapping* if their interiors are disjoint.

Now observe that a point in a box B is reachable only if it is reachable either from the initial set via a flow in B, from a jump via a flow in B, or from a neighboring box via a flow in B. We will now formulate constraints corresponding to each of these conditions. Then we can remove points from boxes that do not fulfill at least one of these constraints.

The approach can be used with any constraint that describes that $\vec{y}$ can be reachable from $\vec{x}$ via a flow in B and mode s, for example, the one introduced in our previous publications (Ratschan and She, 2006). We denote the used constraint by $Reach_B(s,\vec{x},\vec{y})$. Thus, the above three possibilities for reachability allow us to formulate the following theorem:

Theorem 1. *For a set of abstract states $\mathcal{B}$, a pair $(s',B') \in \mathcal{B}$ and a point $\vec{z} \in B'$, if $(s',\vec{z})$ is reachable and z is not an element of the box of any other abstract state in $\mathcal{B}$, then*

$$Ifl_{B'}(s',\vec{z}) \vee \bigvee_{(s,B)\in\mathcal{B}} Jfl_{B,B'}(s,s',\vec{z}) \vee \bigvee_{(s,B)\in\mathcal{B},s=s',B\neq B'} Bfl_{B,B'}(s',\vec{z})$$

where $Ifl_{B'}(s',\vec{z})$, $Jfl_{B,B'}(s,s',\vec{z})$, and $Bfl_{B,B'}(s',\vec{z})$ denote the following three constraints, respectively:

- $\exists\vec{x} \in B' [Init(s',\vec{x}) \wedge Reach_{B'}(s',\vec{x},\vec{z})]$,
- $\exists\vec{x} \in B \exists\vec{x}' \in B'[Jump(s,\vec{x},s',\vec{x}') \wedge Reach_{B'}(s',\vec{x}',\vec{z})]$
- $\exists\vec{x} \in B \cap B'[[\forall faces\ F\ of\ B'[\vec{x} \in F \Rightarrow in^F_{s',B'}(\vec{x})]] \wedge Reach_{B'}(s',\vec{x},\vec{z})]$.

Here, $in^F_{s',B'}(\vec{x}) = \exists\dot{x}_1,\ldots,\exists\dot{x}_k[F(s',\vec{x},(\dot{x}_1,\ldots,\dot{x}_k)) \wedge \dot{x}_j \geq 0]$, if F is the j-th lower face of B', and if F is the j-th upper face of B', $incoming^F_{s',B'}(\vec{x}) = \exists\dot{x}_1,\ldots,\exists\dot{x}_k[F(s',\vec{x},(\dot{x}_1,\ldots,\dot{x}_k)) \wedge \dot{x}_j \leq 0]$.

We denote the main constraint of Theorem 1 by $reach_{\mathcal{B},B'}(s',\vec{z})$. If we can prove that a certain point does not fulfill this constraint, we know that it is not reachable. For now, we assume that we have an algorithm (a *pruning algorithm*) that takes such a constraint, and an abstract state (s',B') and returns a subbox of B' that still contains all the solutions of the constraint in B'. Since the constraint $reach_{\mathcal{B},B'}(s',\vec{z})$ depends on all current abstract states, a change of B' might allow further pruning of other abstract states. So we can repeat pruning until a fixpoint is reached. Given a set of abstract states $\mathcal{B}$, we denote the resulting fixpoint by $Prune_H(\mathcal{B})$.

Now we remove the initial mark from an abstract state (s',B') if we can disprove $Ifl_{B'}(s',\vec{z}')$ in Theorem 1 (i.e., if the pruning algorithm returned the empty box for this constraint), and we remove the unsafe mark of an abstract state state (s',B') if we can disprove the constraint $\exists\vec{x} \in B\ Unsafe(s,\vec{x})$. Moreover, we remove a transition from (s,B) to (s',B') if we can disprove both $Bfl_{B,B'}(s',\vec{z}')$ and $Jfl_{B,B'}(s,s',\vec{z}')$ from Theorem 1. As already mentioned, after recomputing the marks, we remove all abstract states from the abstraction that are not reachable. It is easy to compute these, since the set of abstract states is finite.

There are several methods for implementing the needed pruning algorithms (Benhamou and Granvilliers, 2006). For the domain of the real numbers, given a constraint c and a floating-point box B, they compute another floating-point box $P(c,B)$ such that $P(c,B) \subseteq B$ (contractance), and such that $P(c,B)$ contains all solutions of c in B. Existential quantifiers and disjunctions can be handled by slight extensions (for disjunctions we take the *box union* $\uplus$).

Such pruning algorithms P usually have the *monotonicity property* that for a constraint c, and boxes B and B' with $B' \subseteq B$, $P(c,B') \subseteq P(c,B)$. Moreover, in practice, if $B' \subseteq B$ then $P(c,B')$ is often much smaller than $P(c,B)$. We will exploit this in the improvement of our method described in the next section. In addition, it pays off to distribute disjunctions over conjunctions:

Lemma 1. *For constraints* $c_1,\dots,c_n,d$ *and a box* B*,*

$$P(\vee_{i\in\{1,\dots,n\}}(c_i \wedge d),B) \subseteq P((\vee_{i\in\{1,\dots,n\}}c_i) \wedge d,B)$$

Proof. For each $i \in \{1,\dots,n\}$, $P(c_i \wedge d,B) \subseteq P((\vee_{i\in\{1,\dots,n\}}c_i) \wedge d,B)$. Thus, $\uplus_{i\in\{1,\dots,n\}}P(c_i \wedge d,B) \subseteq P((\vee_{i\in\{1,\dots,n\}}c_i) \wedge d,B)$. Since $P(\vee_{i\in\{1,\dots,n\}}(c_i \wedge d),B) = \uplus_{i\in\{1,\dots,n\}}P(c_i \wedge d,B)$, the lemma holds. ■

4 RECURSIVE PRUNING

In this section we introduce the first improvement to the verification method described in Section 3. Throughout the rest of the paper we assume an abstraction consisting of a set of abstract states $\mathcal{B}$. The improvement introduced in this section aims at pruning more unreachable states from $\mathcal{B}$ by improving the recursive propagation of reachability information for flows from one box to the next.

We consider the pruning of an abstract state $(s',B') \in \mathcal{B}$. The constraint $\mathit{Bfl}_{B,B'}(s',\vec{z})$ defined within Theorem 1 models the fact that a certain point $\vec{z}$ in the box B' is reached from a neighboring box B of B' via a flow in B'. This flow reaches $\vec{z}$ through a common point $\vec{x} \in B \cap B'$ (see Figure 1).

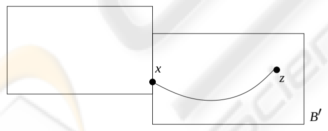

Figure 1: Recursive Pruning.

The basic idea upon which we build in this section is to strengthen this constraint by requiring that also $\vec{x}$ be reachable in the neighboring box B. Naively, this could be done by adding the constraint $\mathit{reach}_{\mathcal{B},B}(s',\vec{x})$ to the constraint $\mathit{Bfl}_{B,B'}(s',\vec{z})$. However, since $\mathit{Bfl}_{B,B'}(s',\vec{z})$ is itself a part of $\mathit{reach}_{\mathcal{B},B'}(s',\vec{x})$, this would result in an infinitely large constraint due to recursion. One could make the constraint finite, by bounding the recursion, but this still would result in a very large constraint. We avoid this, by observing that the neighboring box B is already the result of pruning wrt. $\mathit{reach}_{\mathcal{B},B}(s',\vec{x})$. However, a part of this information is lost because we first prune B and only then take the intersection $B \cap B'$ (i.e., we compute $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B) \cap B'$), and we have:

Lemma 2.

$$P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B') \subseteq P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B) \cap B'$$

Proof. Due to monotonicity of constraint propagation, $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B')$ is a subset of $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B)$. Moreover, $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B') \subseteq P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B')$, and hence $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B') \subseteq B'$. So $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B')$ is also a subset of the intersection of $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B)$ and B'. ■

In practice, the set on the left-hand side might be significantly smaller than the set on the right-hand side (i.e., than the set currently used in the method in Section 3). So it makes sense to compute $P(\mathit{reach}_{\mathcal{B},B'}(s',\vec{x}),B \cap B')$ instead of $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B) \cap B'$. This means that in addition to pruning each box in the abstraction, we could also prune the intersection between each pair of boxes. However, this would need a quadratical number of prunings and stored boxes in memory.

To avoid this, we use an over-approximation of $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B')$ that is still a subset of $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B) \cap B'$. We use the information that the boxes of our abstraction are non-overlapping (i.e., even if two boxes intersect, they only share the boundary but no points of the interior). This implies that the intersection $B \cap B'$ will always be a subset of the boundary of B—independent of the form of the box B'. So one could try to use the boundary $\Box B$ of B instead of the box $B \cap B'$ when computing $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B')$. However, since $\Box B$ is not a box and hence it cannot be an argument to the pruning function, we apply the pruning function to its constituent faces separately. That is, we use the constraint that expresses a disjunction over all faces:

$$\bigvee_{F,\text{face of } B} \left[\vec{x} \in F \wedge \mathit{reach}_{\mathcal{B},B}(s',\vec{x})\right]$$

and call this constraint $\mathit{reachbound}_{\mathcal{B},B}(s',\vec{x})$. Although this over-approximates $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B \cap B')$, Lemma 2 still holds in analogy:

Lemma 3. $P(\bigvee_{F,\mathit{face\ of}\ B}\left[\vec{x} \in F \wedge \mathit{reach}_{\mathcal{B},B}(s',\vec{x})\right],B) \cap B' \subseteq P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B) \cap B'$.

Proof. The disjunction is pruned by taking the box union over the result of pruning each disjunct. Since each face of B is a subset of B, due to monotonicity of constraint propagation, for each face F, $P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),F) \subseteq P(\mathit{reach}_{\mathcal{B},B}(s',\vec{x}),B)$. Hence also the box union over the result of pruning each disjunct is a subset of $P(reach_{\mathcal{B},B}(s',\vec{x}),B)$, which implies the lemma. ■

Since the constraint on the left-hand side only depends on one box, we can compute the corresponding pruning $P(reachbound_{\mathcal{B},B}(s',\vec{x}),B)$ only for one abstract state, and store the resulting box with that abstract state. Since this box encloses the set of states where a flow might leave the abstract state, we call it the *outflow-box* of the abstract state. So, instead of $B \cap B'$ in the constraint $Bfl_{B,B'}$ we can now take the outflow-box of B, and due to Lemma 3 we will arrive at a result that is at least as tight as before.

This is illustrated on an example in Figure 2, where the dotted box is the *outflow-box* resulting from a situation where the upper and left face of box B have been pruned to the empty set, and the outflow-box is the result of taking the union of the result of pruning the two other faces.

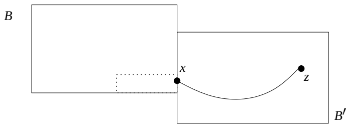

Figure 2: Pruning Faces.

Note that splitting a box B representing a certain abstract state changes its faces. Especially, there might be trajectories that leave the resulting boxes through the new face along which B has been split. Hence the outflow-box of this abstract state becomes invalid. So we simply set the outflow-box to the whole box B and re-compute it, the next time B is pruned.

5 RECURSIVE PRUNING WITH OUTGOING CONDITION

In the previous section we used the fact that within the constraint *Bfl* we can exploit the information that the common point $\vec{x} \in B \cap B'$ itself has to be reachable. In this section we strengthen this information by observing that in order for a trajectory to be able to leave the box B to enter the box B', the vector field at x has to point out of B.

This can be modelled by adding an additional condition in the constraint $reachbound_{\mathcal{B},B}(s',\vec{x})$, arriving at

$$\bigvee_{F,\text{face of }B} \left[\vec{x} \in F \wedge reach_{\mathcal{B},B}(s',\vec{x}) \wedge out^{F}_{s',B}(\vec{x})\right],$$

where $out^{F}_{s',B}(\vec{x})$ is equal to $in^{F}_{s',B}(\vec{x})$ with the inequality sign switched. Now, since $reach_{\mathcal{B},B}(s',\vec{x})$ is a disjunction, Lemma 1 suggests to improve it by pulling out the new conjunction of $reachbound_{\mathcal{B},B}(s',\vec{x})$, arriving at

$$\bigvee_{F,\text{face of}B} \left[\vec{x} \in F \wedge out^{F}_{s',B}(\vec{x}) \wedge Ifl_{B}(s',\vec{x})\right] \vee$$
$$\bigvee_{(s,B')\in\mathcal{B}} \Big[\bigvee_{F,\text{face of}B} [\vec{x} \in F \wedge out^{F}_{s',B}(\vec{x})$$
$$\wedge Jfl_{B',B}(s,s',\vec{x})]\Big] \vee$$
$$\bigvee_{\substack{(s,B')\in\mathcal{B} \\ s=s',B'\neq B}} \Big[\bigvee_{F,\text{face of}B} [\vec{x} \in F \wedge out^{F}_{s',B}(\vec{x})$$
$$\wedge Bfl_{B',B}(s',\vec{x})]\Big].$$

We call the resulting constraint $reachout_{\mathcal{B},B}(s',\vec{x})$, and use this constraint instead to compute the outflow-box of each abstract state.

The following examples illustrates the improvement provided by *reachout* over *reachbound*: Consider the differential equation $(\dot{x}_1,\dot{x}_2) = (1,1)$ with a box $B = [0,1] \times [0,1]$ and an initial point $x_0 = (0,0)$. If we prune a face $[1,1] \times [0,1]$ or $[0,1] \times [1,1]$ wrt. *reachbound*, we will get the point $(1,1)$; and if we prune a face $[0,0] \times [0,1]$ or $[0,1] \times [0,0]$ wrt. *reachbound*, we will get the point $(0,0)$. That is, if we apply the pruning algorithm to *reachbound* and B, we will get the full box $[0,1] \times [0,1]$. Only when adding the outgoing condition, arriving at the constraint *reachout*, we can ignore trajectories moving into the box, arriving at the point $[1,1] \times [1,1]$.

6 BACKWARD REASONING AND COMBINATION

As described in Section 3, in our method we remove elements from the state space for which we can prove that they are not reachable from an initial state. However, the task of safety verification is to prove the absence of a trajectory that starts in an initial state *and reaches an unsafe state*. Hence we can also remove elements from the state space for which we can prove that they do not lead to an unsafe state—without destroying the property that safety of the abstraction implies safety of the concrete system.

For this, observe that a point might lead to an unsafe state only if there is a flow from this point to the unsafe set directly, or a flow from this point to a jump, or a flow from this point to a boundary point. Hence we can formulate an analogous version of Theorem 1:

Theorem 2. *For a set of abstract states $\mathcal{B}$, a pair $(s', B') \in \mathcal{B}$ and a point $\vec{z} \in B'$, if the unsafe set is reachable from $(s', \vec{z})$ and $\vec{z}$ is not an element of the box of any other abstract state in $\mathcal{B}$, then*

$$Urev_{B'}(s', \vec{z}) \vee \bigvee_{(s,B)\in\mathcal{B}} Jrev_{B,B'}(s, s', \vec{z}) \vee \bigvee_{(s,B)\in\mathcal{B}, s=s', B\neq B'} Brev_{B,B'}(s', \vec{z}),$$

where $Urev_{B'}(s', \vec{z})$, $Jrev_{B,B'}(s, s', \vec{z})$, and $Brev_{B,B'}(s', \vec{z})$ denote the following three constraints, respectively:

- $\exists \vec{x} \in B[Reach_B(s, \vec{z}, \vec{x}) \wedge Unsafe(s, \vec{x})]$,
- $\exists \vec{x} \in B \exists \vec{x}' \in B'[Reach_B(s, \vec{z}, \vec{x}) \wedge Jump(s, \vec{x}, s', \vec{x}')]$
- $\exists \vec{x} \in B \cap B'[Reach_B(s, \vec{z}, \vec{x}) \wedge [\forall$ *faces* F *of* $B[\vec{x} \in F \Rightarrow in^F_{s,B'}(x)]]]$

In a similar way as forward reasoning, backward reasoning also allows us to update the initial/unsafe marks and transitions of the abstraction.

Note that by using forward and backward reasoning in Algorithm 1 we might succeed in removing *all* elements from the concrete state space. This results in an empty abstraction which is trivially safe. Hence, Algorithm 1 can report a successful verification in this case. However, the combination of recursive pruning with backward pruning introduces additional difficulties: the outflow box is computed using forward reasoning, and when a box is changed due to backward reasoning, its outflow box is not valid any more. We solve this problem by always, first applying forward pruning and then backward pruning. If backward pruning changes the box, we apply forward pruning again which recomputes a valid outflow box.

7 EXPERIMENTAL RESULTS

We extended our hybrid systems verification package HSOLVER (Ratschan and She, 2004) with the two improvements introduced in this paper. Then we used our problem database[1] of hybrid systems to evaluate our improvements. The experimental results are summarized in Table 1, Table 2 and Table 3 for different versions.

We used an IBM notebook with an Intel Pentium 1.70 GHz CPU with 1024 Mbytes of main memory running Linux. The running times are in seconds and the computations were cancelled when computation did not terminate before three hours or the number of the abstract states exceeded 1000. We used the default splitting strategy of HSOLVER.

Table 1: Experimental Results: I.

	Forward	
Example	time	splits
1-flow	unknown	
2-tanks	1.12	31
car	0.47	0
circuit	53.80	186
clock	1.99	32
convoi-1	1.06	0
convoi	2157.26	374
eco	125.24	223
focus	3.59	57
mixing	296.67	174
mutant	7286.40	742
real-eigen	0.59	2
s-focus	0.54	2
trivial-hard	0.76	26
van-der-pole (VDP)	25.42	64

Table 2: Experimental Results: II.

	Backward		For-Backward	
Example	time	splits	time	splits
1-flow	unknown		0.32	1
2-tanks	0.10	6	0.43	4
car	unknown		1.05	0
circuit	unknown		62.99	188
clock	35.79	327	2.00	43
convoi-1	unknown		1.65	0
convoi	unknown		1847.67	300
eco	unknown		18.32	52
focus	0.62	34	0.89	15
mixing	1.47	7	1.74	0
mutant	unknown		8493.97	618
real-eigen	unknown		0.61	0
s-focus	unknown		0.44	1
trivial-hard	0.03	0	0.05	0
VDP	0.47	1	0.77	1

Comparing the forward version and backward version, there is no clear winner, although the forward version is successful in more cases. The reason seems to lie in the fact that for more examples the set of initial states is smaller than the set of unsafe states.

Moreover, the experimental results show that: (1) For most of the examples, the combined forward and backward version use less splitting steps than both the forward version and backward version. However, for some examples, the CPU time is worse since in the combined version, the constraints are more complex. Note that for some examples (e.g., circuit and clock), the combined version needs slightly more

[1] `http://hsolver.sourceforge.net/benchmarks`

Table 3: Experimental Results: III.

	Recursive		Rec-Backward	
Example	time	splits	time	splits
1-flow	unknown		0.32	1
2-tanks	0.26	3	0.28	1
car	1.29	0	1.07	0
circuit	53.12	171	68.28	192
clock	1.97	16	0.74	14
convoi-1	2.70	0	1.71	0
convoi	2177.75	374	1830.56	300
eco	253.19	290	5.77	22
focus	2.79	48	0.42	8
mixing	104.96	109	1.75	0
mutant	1978.43	195	1850.17	191
real-eigen	0.59	2	0.61	0
s-focus	0.53	2	0.44	1
trivial-hard	0.07	4	0.05	0
VDP	25.81	64	0.77	1

splitting steps. The reason is that although the combined version is more successful in pruning, the splitting heuristics will sometimes choose different boxes which then results—in rare cases—in more necessary splits. (2) For all examples except one, the recursive version needs less splitting steps than the forward version. However, for the eco example, the recursive version needs more splitting steps. This is due to the same reason as above—more successful pruning leads to different box choices. Again, for some example the CPU time is worse since the constraints in the recursive version are more complex. (3) Again with one exception (circuit), the combined recursive and backward version always needs less splitting steps than both the recursive version and combined forward and backward version, often even much less. For most examples also the run-time improved, sometimes over an order of magnitute. Only for two additional, rather easy examples (car, convoi-1), the CPU time slightly increases since the constraints in the combined recursive and backward version are more complex.

Summarizing, the contributions of this paper result in a definite and robust efficiency improvement of the algorithms.

8 CONCLUSIONS

In this paper we have introduced two improvements to a method of safety verification of hybrid systems by constraint propagation based abstraction refinement. The provided computational experiments clearly show the advantage of proposed improvements. We will base future improvements of the method on a detailed study of the behavior of the used algorithms on further benchmark problems.

ACKNOWLEDGEMENTS

The work of the first author has been supported by GAČR grant 201/08/J020 and by the institutional research plan AV0Z100300504. The second author was partly supported by the National Key Basic Research Program of China under Grant No. 2005CB321902 and the Program for Excellent Talents of Beijing under Grant No. 20071D1600600410.

REFERENCES

Benhamou, F. and Granvilliers, L. (2006). Continuous and interval constraints. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, pages 571–603. Elsevier Amsterdam.

Frehse, G., Krogh, B. H., and Rutenbar, R. A. (2006). Verifying analog oscillator circuits using forward/backward abstraction refinement. In *DATE 2006: Design, Automation and Test in Europe*.

Kloetzer, M. and Belta, C. (2006). Reachability analysis of multi-affine systems. In Hespanha, J. and Tiwari, A., editors, *HSCC'06*, volume 3927 of *LNCS*. Springer.

Preußig, J., Stursberg, O., and Kowalewski, S. (1999). Reachability analysis of a class of switched continuous systems by integrating rectangular approximation and rectangular analysis. In Vaandrager, F. and van Schuppen, J., editors, *HSCC'99*, number 1569 in LNCS. Springer.

Ratschan, S. and She, Z. (2004). HSOLVER. http://hsolver.sourceforge.net. Software package.

Ratschan, S. and She, Z. (2006). Constraints for continuous reachability in the verification of hybrid systems. In *Proc. AISC'2006*, number 4120 in LNCS. Springer.

Ratschan, S. and She, Z. (2007). Safety verification of hybrid systems by constraint propagation based abstraction refinement. *ACM Transactions on Embedded Computing Systems*, 6(1).