Safety Verification of Hybrid Systems by Constraint Propagation-Based Abstraction Refinement

STEFAN RATSCHAN
Czech Academy of Sciences, Prague
and
ZHIKUN SHE
Beihang University, Beijing

This paper deals with the problem of safety verification of nonlinear hybrid systems. We start from a classical method that uses interval arithmetic to check whether trajectories can move over the boundaries in a rectangular grid. We put this method into an abstraction refinement framework and improve it by developing an additional refinement step that employs interval-constraint propagation to add information to the abstraction without introducing new grid elements. Moreover, the resulting method allows switching conditions, initial states, and unsafe states to be described by complex constraints, instead of sets that correspond to grid elements. Nevertheless, the method can be easily implemented, since it is based on a well-defined set of constraints, on which one can run any constraint propagation-based solver. Tests of such an implementation are promising.

Categories and Subject Descriptors: I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; C.4 [**Performance of Systems**]: Reliability, Availability, and Serviceability

General Terms: Algorithms, Reliability, Verification

Additional Key Words and Phrases: Hybrid systems, intervals, constraint propagation

ACM Reference Format:

Ratschan, S. and She, Z. 2007. Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Trans. Embedd. Comput. Syst. 6, 1, Article 8 (February 2007), 23 pages. DOI = 10.1145/1210268.1210276 http://doi.acm.org/ 10.1145/1210268.1210276.

This is a revised and extended version of an earlier paper [Ratschan and She 2005]. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

Authors' address: Stefan Ratschan, Institute of Computer Science, Czech Academy of Sciences, Prague, Czech Republic; email: stefan.ratschan@cs.cas.cz; Zhikun She, School of Science, Beihang University, Beijing, China; email: zkshe77@hotmail.com.

1. INTRODUCTION

The environment in which embedded computing systems operate is, in many cases, described by physical laws that are formulated using differential equations. When these are taken into account in verification, the resulting model is a hybrid system. In this paper, we provide a method for verifying that a given nonlinear hybrid system has no trajectory that starts from an initial state and reaches an unsafe state.

Our approach builds upon a known method that decomposes the state space according to a rectangular grid and that uses interval arithmetic to check the flow on the boundary between neighboring grid elements. The reasons for choosing this method as a starting point are: it can do verification instead of verification modulo rounding errors, it can deal with constants that are only known up to intervals, and it uses a check that is less costly than explicit computation of continuous reach sets, or checks based on quantifier elimination. However, this method has the drawback that it may require a very fine grid to provide an affirmative answer. In this paper we provide a remedy to this problem.

In our solution we put the classical interval method into an abstraction refinement framework where the abstract states represent hyperrectangles (*boxes*) in the continuous part of the state space. Here, refinement corresponds to splitting boxes into pieces and recomputing the possible transitions. In order to avoid splitting into too many boxes, we employ an idea that is at the core of the field of constraint programming: instead of a splitting process that is potentially exponential in the dimension of the problem, we try to deduce information without splitting, in an efficient, but possibly incomplete, constraint propagation step. Here we use conditions on the motion of the trajectories within these boxes to construct a constraint without a differentiation operator, whose solution contains the reach set. We then employ an interval constraint-propagation algorithm to remove elements from the boxes that do not fulfill this constraint.

Many algorithms for checking the safety of hybrid systems are based on floating-point computation that involve rounding errors and on algorithms whose correctness can be hampered because of the fact that local instead of global optima are used for intermediate computation. In many cases, this is a perfectly valid approach. However, in some safety critical applications one would like to *verify* safety. Experience shows that just replacing floating-point computation by faithfully rounded interval operations either results in too wide intervals, or—in combination with splitting—in inefficient algorithms. Thus, we tried to develop a genuine interval-based approach here.

Our implementation of the algorithms is publically available [Ratschan and She 2004].

The structure of the paper is as follows: in Section 2, we formalize our safety verification problem; in Section 3, we put the classical interval-based method into an abstraction refinement framework; in Section 4, we improve the method, using interval constraint-propagation techniques; in Section 5, we give a further improvement of the method that employs incremental computation; in Section 6, we discuss our implementation; in Section 7, we illustrate the behavior of the implementation using some benchmarks problems; in Section 8, we discuss the benchmarking results; in Section 9, we cover related work; and in Section 10, we conclude the paper.

2. PROBLEM DEFINITION

We fix a variable s ranging over a finite set of discrete modes $\{\mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{n}\}$ and variables $x_1, \ldots, x_k$ ranging over closed real intervals $I_1, \ldots, I_k$. We denote by Φ the resulting state space $\{\mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{n}\} \times I_1 \times \cdots \times I_k$. In addition, for denoting the derivatives of $x_1, \ldots, x_k$, we assume variables $\dot{x}_1, \ldots, \dot{x}_k$, ranging over $\mathbb{R}$ each,[1] and for denoting the targets of jumps, variables $s', x'_1, \ldots, x'_k$ ranging over $\{\mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{n}\}$ and $I_1, \ldots, I_k$, correspondingly.

In order to describe hybrid systems, we use constraints that are arbitrary Boolean combinations of equalities and inequalities over terms that may contain function symbols like $+$, $\times$, exp, sin, and cos (which further function symbols might be allowed will become clear in Section 6). These constraints are used, on the one hand, to describe the possible flow and jumps and, on the other, to mark certain parts of the state space (e.g., the set of initial states).

Definition 1. A *state space constraint* is a constraint over the variables $s, x_1, \ldots, x_k$. A *flow constraint* is a constraint over the variables s, $x_1, \ldots, x_k$, $\dot{x}_1, \ldots, \dot{x}_k$. A *jump constraint* is a constraint over the variables s, $x_1, \ldots, x_k$ and s', $x'_1, \ldots, x'_k$. A *hybrid system description* (or short: description) is a tuple consisting of a flow constraint, a jump constraint, a state space constraint describing the set of initial states, and a state space constraint, describing the set of unsafe states.

Example of a flow constraint for the case $k = 2$:

$$(\dot{x}_1 = x_1 - x_2 + 3 \wedge \dot{x}_2 = -x_1 - x_2 \wedge x_1 > 1 \wedge x_1 < 3 \wedge x_2 > 1 \wedge x_2 < 3)$$

Example of a flow constraint for the case $n = 2$ and $k = 1$:

$$((s = \mathrm{s}_1 \rightarrow \dot{x} = x) \wedge (s = \mathrm{s}_2 \rightarrow \dot{x} = -x))$$

Example of a jump constraint for the case $n = 2$ and $k = 1$:

$$((s = \mathrm{s}_1 \wedge x \geq 10) \rightarrow (s' = \mathrm{s}_2 \wedge x' = 0))$$

We use these constraints to describe the following:

Definition 2. A *hybrid system* is a tuple (*Flow*, *Jump*, *Init*, *UnSafe*) where *Flow* $\subseteq \Phi \times \mathbb{R}^k$, *Jump* $\subseteq \Phi \times \Phi$, *Init* $\subseteq \Phi$, and *UnSafe* $\subseteq \Phi$.

A hybrid system description gives rise to the hybrid system, for which each constituting set is the solution set of the corresponding constraint of the hybrid system description.

Definition 3. A *flow* of length l in a mode s is a function $r : [0, l] \mapsto \Phi$ 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 a *hybrid system*

[1]The dot does not have any special meaning here, and is just used for distinguishing dotted from undotted variables.

(*Flow*, *Jump*, *Init*, *UnSafe*) is a sequence of flows $r_0, \dots, r_p$ of lengths $l_0, \dots, l_p$ such that for all $i \in \{0, \dots, p\}$,

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

Note that this definition allows us to enforce jumps by formulating a flow constraint that does not allow continuous evolution in a certain region. Note also that our definition of trajectory prohibits the use of flow constraints to describe nondifferentiable evolution. For example, it is not possible to encode if-then-else constructions into the flow constraint as $(\phi \rightarrow \dot{x} = f(x)) \wedge (\neg\phi \rightarrow \dot{x} = g(x))$, because such a constraint can, in general, only be satisfied by nondifferentiable evolutions. However, one can encode the desired behavior using two different modes, for example, $\mathtt{m}_1$ and $\mathtt{m}_2$, a flow constraint $(s = \mathtt{m}_1 \rightarrow \dot{x} = f(x)) \wedge (s = \mathtt{m}_2 \rightarrow \dot{x} = g(x))$, and a jump constraint $(\phi \wedge s = \mathtt{m}_2 \rightarrow s' = \mathtt{m}_1) \wedge (\neg\phi \wedge s = \mathtt{m}_1 \rightarrow s' = \mathtt{m}_2)$ for switching between the two modes based on the condition ϕ.

Definition 4. A hybrid system $H = (\mathit{Flow}, \mathit{Jump}, \mathit{Init}, \mathit{UnSafe})$ is *safe* if, and only if, there is no trajectory $r_0, \dots, r_p$ of H such that $r_0(0)$ is in *Init* and $r_p(l)$ is in *UnSafe*, where l is the length of r_p.

We would like to have an algorithm that, given a hybrid system description, decides whether the corresponding system is safe. However, this is an undecidable problem [Henzinger et al. 1998]. Thus, we aim at an algorithm for which we know that, if it terminates, the hybrid system described by the input is safe. Moreover, we want it to terminate efficiently for problems of practical relevance.

3. AN INTERVAL-BASED METHOD

In this section, we describe an algorithm for verifying safety of hybrid systems. Basically, it is the result of taking a classical method for safety verification and putting it in an abstraction refinement framework. It seems that this classical method is in the folklore of the hybrid systems community and appears for the first time in the literature as a basis for a method that abstracts to timed automata [Stursberg et al. 1997]. It checks the flow at the boundary of boxes using interval arithmetic and requires that switching conditions, initial states, and unsafe states be aligned to the box grid. In contrast, our resulting algorithm allows these sets to be described by complex constraints, as introduced by Definition 1. We assume that we have an algorithm that can test such constraints for falsehood, that is, an algorithm that either returns "false" or "unknown." For details on how to arrive at such an algorithm, see Section 6.

We abstract to systems of the following form:

Definition 5. A *discrete system* over a finite set S is a tuple (*Trans*, *Init*, *UnSafe*) where $\mathit{Trans} \subseteq S \times S$ and $\mathit{Init} \subseteq S$, $\mathit{UnSafe} \subseteq S$. We call the set S the *state space* of the system.

In contrast to Definition 2, here the state space is a parameter. This will allow us to add new states to the state space during abstraction refinement.

Algorithm 1. Abstraction Refinement

```
let $A$ be a discrete abstraction of the hybrid system represented by a description $D$
while  $A$ is not safe
        refine the abstraction $A$
end while
```

Definition 6. 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* iff there is no trajectory from an element of *Init*, to an element of *UnSafe*.

We will abstract the given hybrid system to a discrete system (the *abstraction*) in such a way that if the abstract system is safe, then the original (the *concrete*) system is also 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

In order to implement the above algorithm, we need to fix the state space of the abstract system. Here we use as abstract states pairs (s, B), where s is one of the modes $\{\mathrm{s}_1, \dots, \mathrm{s}_\mathrm{n}\}$ and B is a hyperrectangle (*box*), representing subsets of the concrete state space Φ. For the initial abstraction, we use the state space $\{(\mathrm{s}_\mathrm{i}, \{x \mid (\mathrm{s}_\mathrm{i}, x) \in \Phi\}) \mid 1 \le i \le n\}$. When refining the abstraction, we split a box into two parts, creating two abstract states (s, B_1) and (s, B_2) with $B_1 \cup B_2 = B$, from an abstract state (s, B).

In order to compute a discrete abstraction over this state space, we have to show how to compute the transitions of the resulting abstraction and its set of initial and unsafe states. Here we assume that the input consists of a hybrid system description with flow constraint $Flow(s, x, \dot{x})$, jump constraint $Jump(s, x, s', x')$, initial constraint $Init(s, x)$, and unsafety constraint $UnSafe(s, x)$. Now

- we mark an abstract state (s, B) as initial iff we cannot disprove the constraint $\exists x \in B\ Init(s, x)$, and
- we mark an abstract state (s, B) as unsafe iff we cannot disprove the constraint $\exists x \in B\ UnSafe(s, x)$.

In order to compute the possible transitions between two neighboring boxes in the same mode, we first consider the flow on common boundary points. For a box $B = [\underline{x}_1, \overline{x}_1] \times \dots \times [\underline{x}_k, \overline{x}_k]$, we let its jth lower face be $[\underline{x}_1, \overline{x}_1] \times \dots \times [\underline{x}_j, \underline{x}_j] \times \dots \times [\underline{x}_k, \overline{x}_k]$ and its jth upper face be $[\underline{x}_1, \overline{x}_1] \times \dots \times [\overline{x}_j, \overline{x}_j] \times \dots \times [\underline{x}_k, \overline{x}_k]$. Two boxes are *nonoverlapping* if their interiors are disjoint.

Moreover, we say that *a flow r in a mode s enters the box B at a point $x \in B$* if, and only if, there exist a t_0 and a $\delta > 0$ such that $r(t_0) = (s, x)$ and for all $t \in (t_0, t_0 + \delta)$, $r(t) \in (s, B)$.

Lemma 1. *For a mode s, and a box $B \subseteq \mathbb{R}^k$, if a flow in s enters the box B at a point x, then for all faces F of B such that $x \in F$,*

- $\exists \dot{x}_1, \ldots, \dot{x}_k[Flow(s, x, (\dot{x}_1, \ldots, \dot{x}_k)) \wedge \dot{x}_j \geq 0]$*, if* F *is the* j*th lower face of* B*, and*
- $\exists \dot{x}_1, \ldots, \dot{x}_k[Flow(s, x, (\dot{x}_1, \ldots, \dot{x}_k)) \wedge \dot{x}_j \leq 0]$*, if* F *is the* j*th upper face of* B

We denote the above constraint by $incoming^F_{s,B}(x)$. Using this constraint, we can now construct a constraint for checking the possible transition between two boxes in the same mode.

LEMMA 2. *For a mode* s*, two nonoverlapping boxes* $B, B' \subseteq \mathbb{R}^k$*, if there is a flow in mode* s *that enters* B' *at a common point of* B *and* B'*, then*

$$\exists x[x \in B \wedge x \in B' \wedge [\forall faces\ F\ of\ B'[x \in F \Rightarrow incoming^F_{s,B'}(x)]]]$$

This holds since Lemma 1 can be applied to all the faces of B that contain x. We denote the corresponding constraint by $transition_{s,B,B'}$.

Now we introduce a transition from (s, B) to (s', B') iff

- $s = s'$ and $B = B'$, or
- $s = s'$, $B \neq B'$, and we cannot disprove $transition_{s,B,B'}$ of Lemma 2, or
- we cannot disprove the constraint $\exists x \in B \exists x' \in B' Jump(s, x, s', x')$

Thus, given a hybrid system description D and a set $\mathcal{B}$ of abstract states (i.e., mode/box pairs) such that all boxes corresponding to the same mode are nonoverlapping, we have a method for computing the set of initial states, the set of unsafe states, and the transitions of a corresponding abstraction. We denote the resulting discrete system by $Abstract_D(\mathcal{B})$.

THEOREM 1. *For all hybrid system descriptions* D *and sets of abstract states* $\mathcal{B}$ *covering the whole state space such that all boxes corresponding to the same mode are nonoverlapping, the safety of* $Abstract_D(\mathcal{B})$ *implies the safety of the hybrid system described by* D*.*

This theorem will easily follow from a more general theorem, that we will prove in the next section.

If the differential equations in the flow constraint are in explicit form $\dot{x} = f(x)$ then one can disprove the above constraints using interval arithmetic. According to Lemma 2 one can take all faces F of the common boundary of two boxes B and B', evaluate f on F using interval arithmetic, and check whether the resulting intervals have a sign that does not allow flows over the boundary—as described by Lemma 1. In Section 6, a method will be described that allows the flow constraints also to be in implicit form.

Now a concrete instantiation of Algorithm 1 can maintain the abstract state space $\mathcal{B}$ as described earlier, compute a corresponding abstract system $Abstract_D(\mathcal{B})$, and (since this abstract system is finite) check its safety—either by a brute force algorithm or using a more sophisticated model-checking technology. We can either recompute the abstract system $Abstract_D(\mathcal{B})$ each time we want to check its safety, or we can do this incrementally, just recomputing the elements corresponding to a changed element of the abstract state space (i.e., a box resulting from splitting).

4. A CONSTRAINT PROPAGATION-BASED IMPROVEMENT

The method introduced in the previous section has the problem that splitting can result in a huge number of boxes. This problem is especially virulent for high dimensions, since one needs a number of boxes that is exponential in the number of variables, to arrive at a box with a certain, small side length. In this section, we will try to find a remedy to this problem. The idea is to refine the abstraction without creating more boxes by splitting. Here we can use the observation that the unreachable state space is uninteresting and there is no need to include it in the abstraction.

Thus, we can exclude parts of the state space from the abstraction process, for which we can show that they are not reachable. In order to do this, we 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. We then can remove points from boxes that do not fulfill at least one of these constraints. For this, we first give a constraint describing flows within boxes:

LEMMA 3. *For a box $B \subseteq \mathbb{R}^k$ and a mode s, if a point $y = (y_1, \ldots, y_k) \in B$ is reachable from a point $x = (x_1, \ldots, x_k) \in B$ via a flow in B and s, then*

$$\bigwedge_{1 \le m < n \le k} \exists a_1, \ldots, a_k, \dot{a}_1, \ldots, \dot{a}_k [(a_1, \ldots, a_k) \in B \wedge \\ Flow(s, (a_1, \ldots, a_k), (\dot{a}_1, \ldots, \dot{a}_k)) \wedge \dot{a}_n \cdot (y_m - x_m) = \dot{a}_m \cdot (y_n - x_n)]$$

PROOF. Assume that $r(t) = (r_1(t), \ldots, r_k(t))$ is a flow in B from (s, x) to (s, y). So $r(0) = (s, x)$ and for a certain $t \in \mathbb{R}_{\ge 0}$, $r(t) = (s, y)$. Then, for $i, j \in \{1, \ldots, k\}$ arbitrary, but fixed, by the extended mean value theorem, we have:

$$\exists t' \in [0, t]\, [\dot{r}_j(t')(y_i - x_i) = \dot{r}_i(t')(y_j - x_j)]$$

Now choose such a t' and let $(s, (a_1, \ldots, a_k)) = r(t')$ and $(\dot{a}_1, \ldots, \dot{a}_k) = \dot{r}(t')$. Then, since r is a flow, $Flow(s, (a_1, \ldots, a_k), (\dot{a}_1, \ldots, \dot{a}_k))$ and, hence, the whole constraint holds. □

The intuition behind the above Lemma is shown in Figure 1, which shows that whenever we have a flow from a two-dimensional (2D) point (x_n, x_m) to a 2-dimensional point (y_n, y_m), then there must be a point on the flow where the vector field points exactly in the direction $(y_n - x_n, y_m - x_m)$. Therefore, the box must contain such a point.

We denote the above constraint by $flow_B(s, x, y)$. Now we can write down a constraint describing the first condition—reachability from the initial set:

LEMMA 4. *For a mode s and a box $B \subseteq \mathbb{R}^k$, if $z \in B$ is reachable from the initial set via a flow in s and B, then*

$$\exists y \in B[Init(s, y) \wedge flow_B(s, y, z)]$$

The proof is trivial since it is an immediate consequence of Lemma 3. We denote the above constraint by $initflow_B(s, z)$.

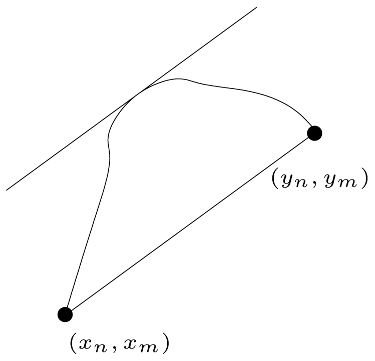

Fig. 1. Flow constraint.

We also have a constraint describing the second condition—reachability from a jump:

LEMMA 5. *For modes s and s', boxes $B, B' \subseteq \mathbb{R}^k$, and $z \in B'$, if (s', z) is reachable from a jump from (s, B) via a flow in B' and s', then*

$$\exists x \in B \exists x' \in B'[Jump(s, x, s', x') \wedge flow_{B'}(s', x', z)]$$

The proof is trivial since it is also a consequence of Lemma 3. We denote the above constraint by $jumpflow_{B,B'}(s, s', z)$.

Finally, we strengthen the condition mentioned in Lemma 2 to a constraint describing the third condition—reachability from a neighboring box.

LEMMA 6. *For a mode s and nonoverlapping boxes $B, B' \subseteq \mathbb{R}^k$, if $z \in B' \setminus B$ is reachable via a flow in s that enters B' at a common point of B and B', then*

$$\exists x\big[x \in B \wedge x \in B' \wedge \big[\forall \text{ faces } F \text{ of } B[x \in F \Rightarrow incoming^F_{s,B'}(x)]\big] \wedge flow_{B'}(s, x, z)\big]$$

This is a consequence of Lemmas 3 and 2. We denote the above constraint by $boundaryflow_{B,B'}(s, z)$.

Now a point that is only covered by one abstract state is reachable only if it is reachable according to Lemmas 4, 5, or 6:

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

$$initflow_{B'}(s', z) \vee \bigvee_{(s,B)\in\mathcal{B}} jumpflow_{B,B'}(s, s', z) \vee \bigvee_{(s,B)\in\mathcal{B}, s=s', B\neq B'} boundaryflow_{B,B'}(s', z)$$

We denote this constraint by $reachable_{\mathcal{B},B'}(s', z)$. Now, if we can prove that a certain point z in B' does not fulfill this constraint, we can remove it from the box B' (if the point is also covered by another abstract state, we can remove it in any case). 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 sub-box of B' that still contains all the solutions of the constraint in B'. (See Section 6 for details on such algorithms.)

Since the constraint $\mathit{reachable}_{\mathcal{B},B'}(s', z)$ depends on all current abstract states, a change of B' might allow further pruning of other abstract states. Thus, we can repeat pruning until a fixpoint is reached. This terminates since we use floating-point computation and there are only finitely many floating point numbers. Given a set of abstract states $\mathcal{B}$, we denote the resulting fixpoint by $\mathit{Prune}_D(\mathcal{B})$.

Now, since we do not need to consider unreachable parts of the state space in the abstraction, we can do the operation $\mathcal{B} \leftarrow \mathit{Prune}_D(\mathcal{B})$ anywhere in Algorithm 1. We do this at the beginning, and each time $\mathcal{B}$ is refined by splitting a box.

Hence, our method can, in some cases, refine the abstraction without splitting, which is a remedy for the problem that splitting may produce a large number of boxes, especially for high dimensions (i.e., for the curse of dimensionality). For doing so, the method considers the flow not only on the boundary but also inside of the boxes.

Now observe that in the computation of $\mathit{Abstract}_D(\mathcal{B})$, we check whether one abstract state is reachable from another one. This information has already been computed by $\mathit{Prune}_D(\mathcal{B})$. More precisely, we get this information from the individual disjuncts of Theorem 2 and we do not need to recompute it.

We get the correctness of our method from the following modification of Theorem 1:

THEOREM 3. *For all hybrid system descriptions D and sets of abstract states $\mathcal{B}$, containing all elements of the state space reachable from the initial set such that all boxes corresponding to the same mode are nonoverlapping, the safety of $\mathit{Abstract}_D(\mathcal{B})$ implies the safety of the hybrid system described by D.*

PROOF. Let D be an arbitrary, but fixed hybrid system description, and let $\mathcal{B}$ be an arbitrary, but fixed set of abstract states containing all elements of the state space reachable from the initial set such that all boxes corresponding to the same mode are nonoverlapping. Let $H = (\mathit{Flow}_1, \mathit{Jump}_1, \mathit{Init}_1, \mathit{UnSafe}_1)$ be the hybrid system described by D and let $(\mathit{Trans}_2, \mathit{Init}_2, \mathit{UnSafe}_2) = \mathit{Abstract}_D(\mathcal{B})$. We will prove that every concrete trajectory from an element of Init_1 to Unsafe_1 has a corresponding abstract trajectory from Init_2 to Unsafe_2.

Now let $r_0, \ldots, r_p$ be a concrete trajectory from an element in Init_1 to UnSafe_1. Let $l_0, \ldots, l_p$ be the corresponding lengths of the flows $r_0, \ldots, r_p$. Let $r_0^\alpha, \ldots, r_p^\alpha$ be such that for all $i \in \{0, \ldots, p\}$, $r_i^\alpha : [0, l_i] \to \mathcal{B}$ and for all $t \in [0, l_i]$, $r_i^\alpha(t)$ is an abstract state (s, B) such that s is equal to the mode of $r_i(t)$ and B contains the continuous part of $r_i^\alpha(t)$. Such an (s, B) always exists, since $\mathcal{B}$ contains the states reachable from the initial set. Moreover, if for a certain t, there is a $\delta > 0$ such that for all $t' \in [t, t+\delta]$, $r_i(t')$ is covered by the same abstract state, then we assign the same abstract state to $r_i^\alpha(t')$.

We construct for every $i \in \{0, \ldots, p\}$ a discrete abstract trajectory from $r_i^\alpha(0)$ to $r_i^\alpha(l_i)$. For this, for a function $f : [0, l_i] \to \mathcal{B}$ and abstract state a, let F^a be such that $F^a(f)$ (the *flattening* of f wrt. a) is again a function in $[0, l_l] \to \mathcal{B}$

and such that for all $t \in [0, l_i]$, $F^a(f)(t) =$

- a, if there exist t^- and t^+ in $[0, l_i]$ such that $t^- < t < t^+$ and such that $f(t^-) = a$ and $f(t^+) = a$;
- $f(t)$, otherwise.

Let ρ_i^α be the result of taking the flattening of r_i^α wrt. every abstract state $a \in \mathcal{B}$, in an arbitrary order, that is, for $\mathcal{B} = \{a_1, \dots, a_q\}$, $\rho_i^\alpha = (F^{a_1} \circ \cdots \circ F^{a_q})(r_i^\alpha)$.

Then ρ_i^α changes its value only finitely often (if it would change its value infinitely often, there would be $t, t^-, t^+ \in [0, l]$ such that $t^- < t < t^+$ and $\rho_i^\alpha(t^-) = \rho_i^\alpha(t^+)$ but $\rho_i^\alpha(t^-) \neq \rho_i^\alpha(t)$, which contradicts the fact that ρ_i^α is flattened.

Hence, every time ρ_i^α changes its value from an abstract state (s, B), to an abstract state (s, B'), there is a corresponding flow that enters B' at a common point of B and B', and because of Lemma 2, $transition_{s,B,B'}$ holds. Hence, by definition of $Abstract_D(\mathcal{B})$, $((s, B), (s, B')) \in Trans_2$. This allows the construction of a finite sequence of abstract states that forms a trajectory of $Abstract_D(\mathcal{B})$ from $r_i^\alpha(0)$ to $r_i^\alpha(l_i)$

Moreover, for every $i \in \{1, \dots, p\}$, $(r_{i-1}(l_{i-1}), r_i(0)) \in Jump_1$. Hence, the jump constraint holds on this pair, and by definition of $Abstract_D(\mathcal{B})$, $(r_{i-1}(l_{i-1}), r_i(0)) \in Trans_2$. So we can concatenate the discrete trajectories from $r_i^\alpha(0)$ to $r_i^\alpha(l_i)$ into a discrete trajectory from $Init_2$ to $Unsafe_2$.

Thus, the safety of $Abstract_D(\mathcal{B})$ implies the safety of D. $\square$

The fact that pruning removes elements from the boxes forming the abstract states has the consequence that in certain cases these boxes do not contain any unsafe states any more. This will result in an abstraction that does not have any unsafe abstract states left. In such a case, the abstract system is trivially safe which immediately implies safety of the concrete system. However, pruning will not be able to remove unsafe states from boxes that intersect with a cycle of the hybrid system. In order to eventually be able to remove the corresponding cycle in the abstraction, we always remove abstract states that are not reachable in the abstraction.

5. INCREMENTAL COMPUTATION

The approach as described up to now applies the algorithm $Prune_D$, from scratch, in each turn of the main loop. In this section, we describe how this can be avoided by reusing information from one call to the next.

First, observe that for a given abstract state (s, B) our solver might prove that one of the disjuncts of the constraint $reachable_{\mathcal{B},B}$ of Theorem 2 has no solution in (s, B). For example, this is trivially the case for *boundaryflow* and nonneighboring boxes. In such a case we can remove the corresponding disjunct from the disjunction and we do not have to check it again for this abstract state or any abstract state that contains a box that results from B by splitting.

We call the resulting method *weakly incremental*. Since our implementation does not keep separate copies of the constraint $reachable_{\mathcal{B},B}(s, z)$ for each abstract state (s, B), we cannot physically remove the disjuncts. Instead we store two types of transitions in the abstraction: boundary transitions and jump transitions. If there is no such transition, we know that the corresponding disjunct is empty.

Algorithm 2. Strongly Incremental Pruning

```
Input: $\mathcal{B}_{in}$, a set of abstract states (i.e., mode/box pairs),
        $Q$ such that
            $Q \subseteq \mathcal{B}_{in}$,
            for all $(s, B) \in \mathcal{B}_{in} \setminus Q$, $prune_{D,\mathcal{B}_{in}}(s, B) = B$
Output: $\mathcal{B}$, such that
            $\mathcal{B}$ contains all reachable elements of $\mathcal{B}_{in}$,
            for all $(s, B) \in \mathcal{B}$, $prune_{D,\mathcal{B}}(s, B) = B$.
$\mathcal{B} \leftarrow \mathcal{B}_{in}$
while $Q \neq \emptyset$
        let $(s, B)$ be an element of $Q$
        $B' \leftarrow prune_{D,\mathcal{B}}(s, B)$
        if $B' \neq B$
            $Q \leftarrow Q \cup \{(s^*, B^*) \in \mathcal{B} \mid (s^*, B^*)$ depends on $(s, B)\}$
            $\mathcal{B} \leftarrow (\mathcal{B} \setminus (s, B)) \cup \{(s, B')\}$
        end if
        $Q \leftarrow Q \setminus (s, B)$
end while
```

Now observe that, after we remove some disjuncts to achieve weak incrementality, the constraint $reachable_{\mathcal{B},B}(s, z)$ only depends on some, but not necessarily all, other abstract states in $\mathcal{B}$ and we only have to recompute it, if one of these changed.

The resulting reimplementation of $Prune_D$, is shown as Algorithm 2. It uses a subalgorithm $prune_{D,\mathcal{B}}(s, B)$, which, given a set of abstract states $\mathcal{B}$ and an element $(s, B) \in \mathcal{B}$, returns the result of pruning a single abstract state (s, B) using the constraint $reachable_{\mathcal{B},B}(s, z)$. We say that (s', B') *depends* on another abstract state (s, B) iff the constraint $reachable_{\mathcal{B},B'}(s', z)$ still contains a disjunct referring to (s, B). The algorithm maintains a set of abstract states for which pruning might succeed and applies the pruning algorithm only to those. If an abstract state is changed, it reconsiders other abstract states that depend on the changed one.

When calling Algorithm 2, we pass as Q all the abstract states that we might be able to prune. These are the abstract states that contain a box that has been split or that depend on another abstract state whose box has been split.

This algorithm can be viewed as an adaption of the constraint propagation algorithm AC-3 [Mackworth 1977; Apt 1999] to our purpose. The main difference is that AC-3 stores constraints in the set Q, while we store abstract states.

We call the resulting method *strongly incremental*. Currently, we implement the set Q by marking the abstract states that are in Q with a Boolean value. This is a simplification from the usual implementations where Q is implemented as a FIFO-queue. This simplification has the following consequences:

- Searching for the next box to prune is not done in constant time, since we have to search for marked abstract states (but, in practice, this might be more efficient than enqueuing/dequeuing).
- The pruning order is different from the usual one. Especially, this order does not take into account the dependencies between the constraints.

As usual in continuous domains, in order to avoid slow convergence, in the implementation of the algorithm, we do not include all the constraints with changed variables into the set Q, but only the ones for which a variable changed more than a certain threshold. Thus, we do not compute a fixpoint, but only an almost fixpoint.

6. CONSTRAINT SOLVING

In this section, we discuss how the algorithm used for pruning the abstract states with respect to constraints (i.e., the subalgorithm $prune_{D,B}(s, B)$) of Algorithm 2) can be implemented. Such pruning algorithms are one of the main topics of the area of constraint programming (for more information see `http://slash.math.unipd.it/cp/`). Usually these work on conjunctions of atomic constraints over a certain domain. For the domain of the real numbers, given a constraint ϕ and a floating-point box B, they compute another floating-point box $N(\phi, B)$, such that $N(\phi, B) \subseteq B$ (contractance) and $N(\phi, B)$ contains all solutions of ϕ in B (cf. the notion of *narrowing operator* [Benhamou et al. 1994; Benhamou 1996], sometimes also called *contractor*).

There are several methods for implementing such a pruning algorithm. The most basic method [Davis 1987; Cleary 1987; Benhamou and Older 1997] decomposes all atomic constraints (i.e., constraints of the form $t \geq 0$ or $t = 0$, where t is a term) into conjunctions of so-called primitive constraints (i.e., constraints such as $x + y = z$, $xy = z$, $z \in [\underline{a}, \overline{a}]$, or $z \geq 0$) by introducing additional auxiliary variables (e.g., decomposing $x + \sin y \geq 0$ to $\sin y = v_1 \wedge x + v_1 = v_2 \wedge v_2 \geq 0$). It then applies a pruning algorithm for these primitive constraints [Hickey et al. 1998] until a fixpoint is reached.

We illustrate pruning of primitive constraints using the example of a primitive constraint $x + y = z$ with the intervals $[1, 4]$, $[2, 3]$, and $[0, 5]$ for x, y, and z, respectively. We can solve the primitive constraint for each of the free variables, arriving at $x = z - y$, $y = z - x$, and $z = x + y$. Each of these forms allows us to prune the interval associated with the variable on the left-hand side of the equation: Using the first solved form, we substract the interval $[2, 3]$ for y from the interval $[0, 5]$ for z, concluding that x can only be in $[-3, 3]$. Intersecting this interval with the original interval $[1, 4]$, we know that x can only be in $[1, 3]$. Proceeding in a similar way for the solved form $y = z - x$ does not change any interval, and, finally, using the solved form $z = x + y$, we can conclude that z can only be in $[3, 5]$.

Pruning for other primitive constraints can be based on interval arithmetic in a similar way. There is extensive literature [Neumaier 1990; Hickey et al. 2001] providing precise formulas for interval arithmetic for addition, subtraction, multiplication, division, and the most common transcendental functions. The floating-point results are always rounded outward, such that the result remains correct also under rounding errors.

There are several variants, alternatives, and improvements of the basic approach described above [Jaulin et al. 2001; Benhamou et al. 1994; Lhomme 1993; Lhomme et al. 1998; Hickey 2001; Lebbah et al. 2002].

For applying such an algorithm to the constraints occurring in our method, we first eliminate the defined predicates by substituting the constraints implied by their definitions. Moreover, the constraints contain variables s and s', ranging over a finite set. These can be easily eliminated by a trivial substitution and simplification.

The constraints also contain existential quantifiers. These can be treated by simply dropping them from the constraints and pruning the Cartesian product of the box corresponding to the free variables and the box bounding the quantified variables [Ratschan 2002]. For disjunctions, one can prune the disjuncts and take the union of the result [Ratschan 2002].

Still, our constraints are not first-order, since they also have box-valued arguments (appearing in the subscripts of the names of constraints) for use with the set-theoretic predicate $\in$. One could eliminate these arguments by substituting the corresponding boxes into the constraints. However, this would require reparsing the constraint each time the corresponding box changed. Instead, we eliminate these boxes as follows: We first rewrite a constraint $v \in B$, with $B = I_1 \times \cdots \times I_k$ to a conjunction $v_1 \in I_1 \wedge \cdots \wedge v_k \in I_k$. We then introduce the meta-constraint $x \subseteq y$ [Older and Benhamou 1993; Hickey 2000] that models the information that the solution set in the variable x is a subset of the solution set in the variable y. Using this constraint, we can now replace a constraint $v_i \in I_i$ by the metaconstraint $v_i \subseteq b_i$, where b_i is a new variable for which we pass the corresponding interval I_i. A solver can then use this meta-constraint to prune the variable v_i to this interval. For example, in the case of a two-dimensional state space, the result for $\mathit{flow}_B(s, (x_1, x_2), (y_1, y_2))$ has the form $\exists a_1, a_2, \dot{a}_1, \dot{a}_2 [a_1 \subseteq b_1 \wedge a_2 \subseteq b_2 \wedge \cdots \wedge \dot{a}_2(y_1 - x_1) = \dot{a}_1(y_2 - x_2)]$, such that for $B = I_1 \times I_2$, we pass to the pruning function the interval I_1 for the new variable b_1, and the interval I_2 for the new variable b_2.

We have implemented the algorithm on top of our RSolver [Ratschan 2004] package that provides pruning and solving of quantified constraints of the real numbers, a graphical user interface, and several other features, and that uses the smathlib library [Hickey] for pruning primitive constraints. The implementation is publically available [Ratschan and She 2004] and we will make the source code open, which will make it easy to extend it or to experiment with changes.

7. COMPUTATION RESULTS

In our opinion, it is essential to test new algorithms on more than just two to three examples. Hence, we devote this section to extensive benchmarking. For this we have to fix a certain splitting strategy in the method. Intuitively—in order to achieve some sort of convergence—one wants to require that the side length of all boxes eventually goes to zero. In fact, this is precisely the condition under which we could prove convergence in the discrete time case in another paper [Damm et al. 2005]. We always split the widest box along the widest side length. However, this strategy has the disadvantage that it depends on the units used for the individual variables. Especially, if the state space is much larger in one variable than in others, then only that variable will be split.

Hence, we also tried a round-robin strategy, where the widest box is split along the variable along, which it has not been split for the longest time. We call these two strategies as *Widest* and *RoundRobin*, respectively.

Example FOCUS:

Flow: $(\dot{x}_1, \dot{x}_2) = (x_1 - x_2, x_1 + x_2)$
Empty jump relation
Init: $2.5 \leq x_1 \leq 3 \wedge x_2 = 0$
Unsafe: $x_1 \leq 2$
The state space: $[0, 4] \times [0, 4]$

Example 2-TANKS: The flow constraint results from setting all the parameters in the two tanks problem [Stursberg et al. 1997] to 1.

Flow: $\big(s = 1 \rightarrow \left(\begin{smallmatrix} \dot{x}_1 \\ \dot{x}_2 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 1-\sqrt{x_1} \\ \sqrt{x_1}-\sqrt{x_2} \end{smallmatrix}\right)\big) \wedge \big(s = 2 \rightarrow \left(\begin{smallmatrix} \dot{x}_1 \\ \dot{x}_2 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 1-\sqrt{x_1-x_2+1} \\ \sqrt{x_1-x_2+1}-\sqrt{x_2} \end{smallmatrix}\right)\big)$
Jump: $(s = 1 \wedge 0.99 \leq x_2 \leq 1) \rightarrow (s' = 2 \wedge x_1' = x_1 \wedge x_2' = 1)$
Init: $s = 1 \wedge (x_1 - 5.5)^2 + (x_2 - 0.25)^2 \leq 0.0625$
Unsafe: $(s = 1 \wedge (x_1 - 4.25)^2 + (x_2 - 0.25)^2 < 0.0625)$
The state space: $(1, [4, 6] \times [0, 1]) \cup (2, [4, 6] \times [1, 2])$

Example ECO: A predator–prey example of ecosystem problems.

Flow: $\big(s = 1 \rightarrow \left(\begin{smallmatrix} \dot{x}_1 \\ \dot{x}_2 \end{smallmatrix}\right) = \left(\begin{smallmatrix} -x_1+x_1x_2 \\ x_2-x_1x_2 \end{smallmatrix}\right)\big) \wedge \big(s = 2 \rightarrow \left(\begin{smallmatrix} \dot{x}_1 \\ \dot{x}_2 \end{smallmatrix}\right) = \left(\begin{smallmatrix} -x_1+x_1x_2 \\ x_2-x_1x_2 \end{smallmatrix}\right)\big)$
Jump: $((s = 1 \wedge 0.875 \leq x_2 \leq 0.9) \rightarrow (s' = 2 \wedge (x_1' - 1.2)^2 + (x_2' - 1.8)^2 \leq 0.01)$
$\vee ((s = 2 \wedge 1.1 \leq x_2 \leq 1.125) \rightarrow (s' = 1 \wedge (x_1' - 0.7)^2 + (x_2' - 0.7)^2 \leq 0.01))$
Init: $s = 1 \wedge (x_1 - 0.8)^2 + (x_2 - 0.2)^2 \leq 0.01$
Unsafe: $(s = 1 \wedge x_1 > 0.8 \wedge x_2 > 0.8 \wedge x_1 <= 0.9 \wedge x_2 \leq 0.9)$
State space: $(1, [0.1, 0.9] \times [0.1, 0.9]) \cup (2, [1.1, 1.9] \times [1.1, 1.9])$

Example 1-FLOW: An example from a paper by J. Preussig [Preussig et al. 1998].

Flow: $\dot{x} = \dot{y} = \dot{t} = 1$
Empty jump relation
Init: $0 \leq x \leq 1 \wedge y = t = 0$
Unsafe: $(0 \leq x \leq 2 \wedge 1 < y \leq 2 \wedge 0 \leq t < 1)$
The state space: $[0, 2] \times [0, 2] \times [0, 4]$

Example CLOCK: A simple example with a clock variable.

Flow: $(\dot{x}, \dot{y}, \dot{t}) = (-5.5y + y^2, 6x - x^2, 1)$
Empty jump relation
Init: $4 \leq x \leq 4.5 \wedge y = 1 \wedge t = 0$
Unsafe: $(1 \leq x < 2 \wedge 2 < y < 3 \wedge 2 \leq t \leq 4)$
The state space: $[1, 5] \times [1, 5] \times [0, 4]$

Example CAR: A three-dimensional and nonlinear example about a simple controller that steers a car along a straight road from a paper by Clarke and others [Clarke et al. 2003a].

The three continuous variables are the position x, the heading angle γ, and the internal timer c. Since we cannot prove the safety property described in the original paper, in this paper we set the unsafe space to $x \leq -4$.

Example HEATING: A linear, three-dimensional example of the room heating problem defined by three rooms and two heaters from a paper by A. Fehnker and F. Ivanćić [2004].

In the original paper, the authors require that the temperature in all rooms always be above a given threshold. However, this given threshold is not specified. In this paper, we specify it to be 14. In addition, we specify the state space to be $[14, 22] \times [14, 22] \times [14, 22]$.

It is currently unknown whether this system is safe or not.

Example CONVOI: A linear collision avoidance example from a part of the car convoi control from a paper by A. Puri and P. Varaiya [1995].

Let gap, v_r, v_l and a_r, respectively, represent the distance between the two cars ($d_{i-1} - d_i$ in the original paper), the velocity of the rear car ($\dot{d}_i$), the velocity of the leading car ($\dot{d}_{i-1}$), and the acceleration of the rear car ($\ddot{d}_i$). By using these variables, and restricting v_l by $-2 \leq \dot{v}_{l \leq} - 0.5$, we transformed the original higher-order differential equation into a four-dimensional differential (in)equation of order one.

We set the state space to $[0, 4] \times [0, 2] \times [0, 2] \times [-2, -0.5]$ and we want to verify that $gap > 0$ when starting from $gap = 1, v_r = 2, v_l = 2$, and $a_r = -0.5$.

Example MIXING: A four-dimensional and nonlinear example about a mixing-tank-system from a paper by O. Stursberg, S. Kowalewski, and S. Engell [Stursberg et al. 2000].

In the original paper, the system is simplified to a two-dimensional system. In this paper, we keep the differential equations $(\dot{V}_1, \dot{V}_2) = (0.008, 0.015)$ in the flow constraint, where V_1 and V_2 are two inlet streams. Then, initially, $V_1(0) = 1, V_2(0) = 1$, and $(h(0), c(0)) \in [1.32, 1.5] \times [1.2, 1.32]$, where h is liquid height and c is concentration. We want to verify that the state $\{(V_1, V_2, h, c) : h \in [1.1, 1.3] \wedge c \in [1.68, 1.80]\}$ is unreachable.

Example CIRCUIT: A two-dimensional and nonlinear example about a tunnel-diode oscillator circuit [Frehse 2005]. It models the voltage drop V and current I.

The original problem was to prove that all trajectories eventually reach a certain set and stay there. We transformed it to a reachability problem, using the state space $[-0.1, 0.6] \times [-0.002, 0.002]$ and the unsafety constraint $V < -0.04 \vee V > 0.54 \vee I < -0.0015 \vee I > 0.00175$.

We tried to verify the above examples by applying our improved method from Section 4 and comparing the two splitting techniques mentioned above, and the basic unincremental (Table I), the weakly incremental (Table II), and the strongly incremental versions (Table III) described in Section 5. The computations were performed on a Pentium IV, 2.60-GHz with 1 GB RAM; and they were canceled in cases when computation did not terminate before 10 hr of computation time. We also list the number of splitting steps (which corresponds to the number of loop turns in Algorithm 1) and the number of calls to

Table I. Direct Computation

	Widest			Round-Robin		
Example	CPU time (s)	Splitting step	Pruning number	CPU time (s)	Splitting steps	Pruning number
FOCUS	11.58	60	145270	4.11	41	48248
2-TANKS	1.43	18	6114	1.42	18	6114
ECO	5233	452	50007339	7007	503	66254349
1-FLOW	0.05	1	15	>10 hr		
CLOCK	62.25	183	83650	15.43	93	21223
CAR	0.52	3	388	2.86	15	4268
HEATING	>10 hr			>10 hr		
CONVOI	65392	415	64446826	41251	368	39798021
MIXING	77798	468	86097290	181.00	54	145204
CIRCUIT	>10 hr			20962	645	105056829

Table II. Weakly Incremental

	Widest			Round-Robin		
Example	CPU time (s)	Splitting step	Pruning number	CPU time (s)	Splitting steps	Pruning number
FOCUS	1.27	60	5581	0.61	41	2763
2-TANKS	0.33	18	775	0.35	18	775
ECO	2.32	72	8971	1.50	58	6002
1-FLOW	0.05	1	15	>10 hr		
CLOCK	62.25	183	83650	15.43	93	21223
CAR	0.38	3	133	1.30	15	1387
HEATING	>10 hr			>10 hr		
CONVOI	3165	415	1182285	2302	368	839944
MIXING	2514	468	1056017	24.82	54	10574
CIRCUIT	> 10 hr			651.19	645	551650

Table III. Strongly Incremental

	Widest			Round-Robin		
Example	CPU time (s)	Splitting step	Pruning number	CPU time (s)	Splitting steps	Pruning number
FOCUS	0.71	107	2443	0.34	71	1572
2-TANKS	0.14	18	344	0.13	18	344
ECO	0.44	72	1590	0.32	58	1295
1-FLOW	0.05	1	12	>10 hr		
CLOCK	7.06	183	8850	2.59	93	3552
CAR	0.34	3	87	0.93	15	907
HEATING	>10 hr			>10 hr		
CONVOI	203.74	415	84733	187.96	369	69158
MIXING	169.68	472	63226	7.68	54	3138
CIRCUIT	>10 hr			319.24	645	16311

the pruning algorithm. In all cases, memory consumption was too small to be of interest.

8. DISCUSSION AND IMPROVEMENT

Note that, to our knowledge, there is no rigorous (i.e., not floating-point error prone) system available that allows similar generality as our method (e.g.,

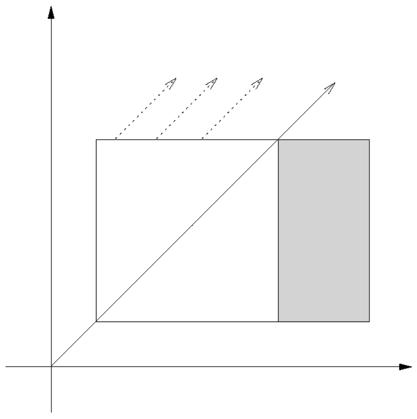

Fig. 2. Transitivity problem.

nonlinear system specification, even with transcendental functions). Moreover, most current systems only work reasonably well with human interaction for specifying certain parameters, whereas we run a full benchmark suite without any interaction. The results clearly show that the efficiency of our method is already similar to systems that do not have the same generality and rigor as our method and are often based on years of development effort. For the example HEATING, whose safety is unknown, our method also does not succeed.

One can conclude that incremental computation definitely pays off in terms of number of calls to the pruning function, where it results in an improvement of orders of magnitude. This is also reflected in the amount of CPU time spent.

Also, the splitting strategy *Widest* behaves as expected—it has problems for examples where the state space of different variables has highly different magnitude, as is the case for the example CIRCUIT. By avoiding this phenomenon, the round-robin splitting strategy is significantly more efficient, on average.

However, the example 1-FLOW shows an anomaly: It can be easily solved by the strategy *Widest*, but the round-robin strategy fails. Here a phenomenon occurs that can be more easily explained on the following simpler example:

Flow: $\dot{x} = \dot{y} = 1$,
Init: $x = y = 0$,
State space: $[0, 4] \times [0, 4]$

Assume that we have an abstract state given by the white box in Figure 2. The neighboring boxes then continue to consider flows from the whole boundary of the white box (the dotted arrows), although there is only one flow from the upper right corner. Here a well-known inherent problem of abstraction refinement techniques occurs: the introduction of additional transitivity along a series of abstract states, that is, given abstract states a, b, and c, if there are (nonspurious) transitions $a \rightarrow b$, and $b \rightarrow c$, then c is declared to be reachable from a, although this is not necessarily the case. Sometimes not even refinement of the abstraction can remove this problem (one would have to check whole counterexample paths to avoid this [Clarke et al. 2003a; Alur et al. 2003]). This is different from the discrete time case, where one can show convergence of a

Table IV. Direct Computation—Multi-splitting

	Widest			Round-Robin		
Example	CPU time (s)	Splitting step	Pruning number	CPU time (s)	Splitting steps	Pruning number
2-TANKS	1.46	11	6372	1.41	11	6372
ECO	52.35	57	466750	23.17	42	197762
CAR	0.50	1	308	0.42	1	188
HEATING	>10 hr			>10 hr		
CIRCUIT	>10 hr			3439	165	16275950

similar abstraction refinement scheme [Damm et al. 2005]. However, in this example, for boxes that are not square, pruning can remove some parts (the grey part in the figure can be removed when starting from the bigger nonsquare box covering both the white and grey area). In the example, 1-FLOW round-robin splitting happens to choose a sequence of variables for splitting, where pruning does not help to avoid this transitivity problem.

Another limitation of the method can be seen on the following example:

Flow: $(\dot{x}, \dot{y}) = (1, \sqrt{2})$
Init: $(x, y) = (0, 0)$,
Unsafe: $y < \sqrt{2}x$,
State space: $[0.2] \times [0, 2]$

Obviously, this system is safe. However, if the flow constraint is perturbed to the constraint Flow: $(\dot{x}, \dot{y}) = (1, 1.414)$ then the resulting system is unsafe. This is because the original system is not robustly safe in the sense that under a small perturbation the perturbed system is unsafe. For such systems, the method will not terminate. However, this behavior is desired! We clearly do not want to declare systems as safe that become unsafe under small perturbations, since the implemented system will never be able to exactly match the verified model [Fränzle 1999; Damm et al. 2005; Ratschan 2002].

The fact that for some examples a high number of splitting steps is necessary, motivated us to experiment with an alternative splitting strategy. Here we use the observation that in multimode systems, very often, there is only a small amount of interaction between the modes and, hence, only a small number of abstract transitions between abstract states in different modes. As a consequence, splitting a certain box in one mode will usually allow only a small amount of pruning in the other modes. Thus, in order to encourage pruning in all modes, instead of splitting only one box in total, we also tried to split one box (again the largest one) in each mode. Clearly this is different from the old strategy only for multimode systems. The result can be seen in Tables IV, V, and VI—a clear improvement over the previous strategy.

9. RELATED WORK

The idea of using abstraction to compute the reach set of hybrid systems is not new. Here the basic choice is, which data structure to use for representing subsets of the continuous part of the state space.

Table V. Weakly Incremental—Multi-splitting

	Widest			Round-Robin		
Example	CPU time (s)	Splitting step	Pruning number	CPU time (s)	Splitting steps	Pruning number
2-TANKS	0.32	11	704	0.32	11	704
ECO	3.27	57	12662	2.01	43	7766
CAR	0.38	1	117	0.36	1	96
HEATING	>10 hr			>10 hr		
CIRCUIT	>10 hr			98.40	165	126731

Table VI. Strongly Incremental—Multi-splitting

	Widest			Round-Robin		
Example	CPU time (s)	Splitting step	Pruning number	CPU time (s)	Splitting steps	Pruning number
2-TANKS	0.17	11	397	0.18	11	397
ECO	0.83	57	3015	0.57	43	2250
CAR	0.34	1	97	0.35	1	88
HEATING	>10 hr			>10 hr		
CIRCUIT	>10 hr			35.77	165	12215

Kowalewski, Stursberg, and co-workers pioneered the use of box representations [Stursberg et al. 1997; Stursberg and Kowalewski 2000; Preussig et al. 1998; Stursberg and Kowalewski 1999; Preussig et al. 1999; Stursberg et al. 2000]. Also, in their method, interval arithmetic is used to check the flow on the boundaries of a rectangular grid. Timing information is then added by checking the flow within these boxes. As a result, one arrives at rectangular or timed automata. All appearing switching conditions, initial states, and unsafe states have to be aligned to the predefined grid, whereas in this paper, we allow complex constraints. Moreover, their method has been designed for a fixed grid and a refinement of the abstraction requires a complete recomputation, whereas in the present work, this can be done incrementally. Their method also does not include a step for refining the abstraction without splitting and it is harder to implement, since it does not build upon an existing constraint solver. However, they generate additional timing information and use additional information on reachable subsets of faces.

Another frequently used technique for representing parts of the state space are polyhedra [Chutinan and Krogh 1999; Alur et al. 2002; Clarke et al. 2003a; Alur et al. 2003; Asarin et al. 2002; Frehse 2005]. This has the advantage of being flexible, but requires involved algorithms for handling these polyhedra and for approximating reachable sets. In contrast to that, boxes are less flexible, but the corresponding operations are simple to implement efficiently, even with validated handling of floating-point rounding errors. It is not clear how one could adapt the pruning mechanism of this paper to polyhedra.

Another method uses semialgebraic sets for representation [Tiwari and Khanna 2002]. This is even more flexible and can produce symbolic output, but requires highly complex quantifier elimination tools [Collins and Hong 1991]. Again, it is not clear how one could employ a pruning mechanism for such a representation.

Other methods try to compute the reach set explicitly, without abstraction. Also, here one has to decide on the representation for this set, for example, zonotopes [Girard 2005], polytopes [Chutinan and Krogh 1999], orthogonal polyhedra [Asarin et al. 2002], ellipsoids [Kurzhanski and Varaiya 2000], or level sets [Mitchell and Tomlin 2000].

There are also methods that use interval arithmetic to compute the reach set explicitly. In one approach [Henzinger et al. 2000], an interval ODE solver is used. In another approach [Hickey and Wittenberg 2004], a constraint logic programming language [Hickey 2000] that allows constraints with differentiation operators is used.

In general, abstraction refinement methods, such as the one described in this paper, have the advantage that they avoid computation of information that is not necessary for the safety checking problem at hand. On the other hand, for cases where this safety checking problem does require a very precise computation of the reach set, a direct computation for this reach set might be more efficient.

10. CONCLUSION

In this paper, we have put a classical method for verifying safety of hybrid systems into an abstraction refinement framework, and we have provided a constraint propagation based remedy for some of the problems of the method. As a result, we need to split into less boxes, we retain information on the flow within boxes, and we can use complex constraints for specifying the hybrid system. Since the method is based on a clear set of constraints, it can be easily implemented using a pruning algorithm based on interval constraint propagation.

Our long-term goal is to arrive at a method for which one can prove termination for all, except numerically, ill-posed cases, in a similar way as can be done for quantified inequality constraints [Ratschan 2002] and hybrid systems in which all trajectories follow polynomials [Fränzle 1999]. Moreover, we will try to exploit the structure of special, for example, linear systems [Tiwari 2003], and analyze whole sequences of abstract states (instead of just pairs) to avoid the transitivity problem.

Interesting further questions are, whether work on constraint propagation in the discrete domain can be useful in a similar way for pruning the discrete state space, and whether similar pruning of the state space can be done for more complex verification tasks, (i.e., for general ACTL queries).

ACKNOWLEDGMENTS

The authors thank Martin Fränzle for carefully removing some of our initial ignorance about the intricacies of hybrid systems.

REFERENCES

Alur, R., Dang, T., and Ivančić, F. 2002. Reachability analysis of hybrid systems via predicate abstraction. See Tomlin and Greenstreet [2002].

Alur, R., Dang, T., and Ivančić, F. 2003. Counter-example guided predicate abstraction of hybrid systems. In *TACAS*, H. Garavel and J. Hatcliff, Eds. LNCS, vol. 2619. Springer, New York. 208–223.

Alur, R. and Pappas, G. J., Eds. 2004. *Hybrid Systems: Computation and Control*. Number 2993 in LNCS. Springer, New York.

Apt, K. R. 1999. The essence of constraint propagation. *Theoretical Computer Science 221*, 1–2, 179–210.

Asarin, E., Dang, T., and Maler, O. 2002. The d/dt tool for verification of hybrid systems. In *CAV'02*. Number 2404 in LNCS. Springer, New York. 365–370.

Benhamou, F. 1996. Heterogeneous constraint solving. In *Proc. of the Fifth International Conference on Algebraic and Logic Programming*. LNCS, vol. 1139. Springer, New York.

Benhamou, F. and Older, W. J. 1997. Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming 32*, 1, 1–24.

Benhamou, F., McAllester, D., and Van Hentenryck, P. 1994. CLP(Intervals) revisited. In *International Symposium on Logic Programming*. MIT Press, Ithaca, NY. 124–138.

Caviness, B. F. and Johnson, J. R., Eds. 1998. *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer, Wien.

Chutinan, A. and Krogh, B. H. 1999. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. See Vaandrager and van Schuppen [1999]. 76–90.

Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., and Theobald, M. 2003a. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *Int. Journal of Foundations of Comp. Science 14*, 4, 583–604.

Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., and Theobald, M. 2003b. Verification of hybrid systems based on counterexample-guided abstraction refinement. In *TACAS 2003*, H. Garavel and J. Hatcliff, Eds. Number 2619 in LNCS. Springer, New York. 192–207.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003c. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM 50*, 5, 752–794.

Cleary, J. G. 1987. Logical arithmetic. *Future Computing Systems 2*, 2, 125–149.

Collins, G. E. and Hong, H. 1991. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation 12*, 299–328. Also in Caviness and Johnson [1998].

Damm, W., Pinto, G., and Ratschan, S. 2005. Guaranteed termination in the verification of LTL properties of nonlinear robust discrete time hybrid systems. In *Proceedings of the Third International Symposium on Automated Technology for Verification and Analysis*, D. A. Peled and Y.-K. Tsay, Eds. Number 3707 in LNCS. Springer, New York. 99–113.

Davis, E. 1987. Constraint propagation with interval labels. *Artificial Intelligence 32*, 3, 281–331.

Fehnker, A. and Ivančić, F. 2004. Benchmarks for hybrid systems verification. See Alur and Pappas [2004].

Fränzle, M. 1999. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In *Computer Science Logic (CSL'99)*, J. Flum and M. Rodriguez-Artalejo, Eds. Number 1683 in LNCS. Springer, New York.

Frehse, G. 2005. PHAVer: Algorithmic verification of hybrid systems past HyTech. See Morari and Thiele [2005].

Girard, A. 2005. Reachability of uncertain linear systems using zonotopes. See Morari and Thiele [2005].

Henzinger, T. A., Kopke, P. W., Puri, A., and Varaiya, P. 1998. What's decidable about hybrid automata. *Journal of Computer and System Sciences 57*, 94–124.

Henzinger, T. A., Horowitz, B., Majumdar, R., and Wong-Toi, H. 2000. Beyond HyTech: hybrid systems analysis using interval numerical methods. See Lynch and Krogh [2000].

Hickey, T. and Wittenberg, D. 2004. Rigorous modeling of hybrid systems using interval arithmetic constraints. See Alur and Pappas [2004].

Hickey, T. J. smathlib. `http://interval.sourceforge.net/interval/prolog/clip/clip/smath/README.%html`.

Hickey, T. J. 2000. Analytic constraint solving and interval arithmetic. In *Proceedings of the 27th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, New York. 338–351.

Hickey, T. J. 2001. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming 2001*, 7 (Oct.).

HICKEY, T. J., VAN EMDEN, M. H., AND WU, H. 1998. A unified framework for interval constraint and interval arithmetic. In *CP'98*, M. Maher and J. Puget, Eds. Number 1520 in LNCS. Springer, New York. 250–264.

HICKEY, T. J., JU, Q., AND VAN EMDEN, M. H. 2001. Interval arithmetic: from principles to implementation. *Journal of the ACM 48*, 5, 1038–1068.

JAULIN, L., KIEFFER, M., DIDRIT, O., AND WALTER, É. 2001. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer, Berlin.

KURZHANSKI, A. AND VARAIYA, P. 2000. Ellipsoidal techniques for reachability analysis. See Lynch and Krogh [2000]. 202–214.

LEBBAH, Y., RUEHER, M., AND MICHEL, C. 2002. A global filtering algorithm for handling systems of quadratic equations and inequations. In *Proc. of Principles and Practice of Constraint Programming (CP 2002)*, P. Van Hentenryck, Ed. Number 2470 in LNCS. Springer, New York.

LHOMME, O. 1993. Consistency techniques for numeric CSPs. In *Proc. 13th Intl. Joint Conf. on Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA.

LHOMME, O., GOTLIEB, A., AND RUEHER, M. 1998. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming 37*, 1–3, 165–183.

LYNCH, N. AND KROGH, B., EDS. 2000. *Proc. HSCC'00*. LNCS, vol. 1790. Springer, New York.

MACKWORTH, A. K. 1977. Consistency in networks of relations. *Artificial Intelligence 8*, 99–118.

MITCHELL, I. AND TOMLIN, C. J. 2000. Level set methods for computation in hybrid systems. See Lynch and Krogh [2000]. 310–323.

MORARI, M. AND THIELE, L., EDS. 2005. *Hybrid Systems: Computation and Control*. LNCS, vol. 3414. Springer, New York.

NEUMAIER, A. 1990. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge.

OLDER, W. AND BENHAMOU, F. 1993. Programming in CLP(BNR). In *1st Workshop on Principles and Practice of Constraint Programming*.

PREUSSIG, J., KOWALEWSKI, S., WONG-TOI, H., AND HENZINGER, T. 1998. An algorithm for the approximative analysis of rectangular automata. In *5th Int. School and Symp. on Formal Techniques in Fault Tolerant and Real Time Systems*. Number 1486 in LNCS. Springer, New York.

PREUSSIG, J., STURSBERG, O., AND KOWALEWSKI, S. 1999. Reachability analysis of a class of switched continuous systems by integrating rectangular approximation and rectangular analysis. See Vaandrager and van Schuppen [1999].

PURI, A. AND VARAIYA, P. 1995. Driving safely in smart cars. In *Proc. of the 1995 American Control Conference*. 3597–3599.

RATSCHAN, S. 2002. Continuous first-order constraint satisfaction. In *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, Eds. Number 2385 in LNCS. Springer, New York. 181–195.

RATSCHAN, S. 2004. RSOLVER. `http://rsolver.sourceforge.net`. Software package.

RATSCHAN, S. AND SHE, Z. 2004. Hsolver. `http://hsolver.sourceforge.net`. Software package.

RATSCHAN, S. AND SHE, Z. 2005. Safety verification of hybrid systems by constraint propagation based abstraction refinement. See Morari and Thiele [2005].

STURSBERG, O. AND KOWALEWSKI, S. 1999. Approximating switched continuous systems by rectangular automata. In *Proc. European Control Conference*. Paper-ID: F1014-4.

STURSBERG, O. AND KOWALEWSKI, S. 2000. Analysis of controlled hybrid processing systems based on approximation by timed automata using interval arithmetic. In *Proceedings of the 8th IEEE Mediterranean Conference on Control and Automation (MED 2000)*.

STURSBERG, O., KOWALEWSKI, S., HOFFMANN, I., AND PREUSSIG, J. 1997. Comparing timed and hybrid automata as approximations of continuous systems. In *Hybrid Systems*, P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, Eds. Number 1273 in LNCS. Springer, New York. 361–377.

STURSBERG, O., KOWALEWSKI, S., AND ENGELL, S. 2000. On the generation of timed discrete approximations for continuous systems. *Mathematical and Computer Models of Dynamical Systems 6*, 51–70.

TIWARI, A. 2003. Approximate reachability for linear systems. In *Hybrid Systems: Computation and Control (HSCC)*, O. Maler and A. Pnueli, Eds. LNCS, vol. 2623. Springer, New York.

TIWARI, A. AND KHANNA, G. 2002. Series of abstractions for hybrid automata. See Tomlin and Greenstreet [2002].

TOMLIN, C. J. AND GREENSTREET, M. R., EDS. 2002. *Hybrid Systems: Computation and Control HSCC*. Number 2289 in LNCS.

VAANDRAGER, F. AND VAN SCHUPPEN, J., EDS. 1999. *Hybrid Systems: Computation and Control—HSCC'99*. Number 1569 in LNCS. Springer, New York.

Received May 2005; revised October 2005; accepted March 2006