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

Stefan Ratschan[1] and Zhikun She[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
`stefan.ratschan@mpi-sb.mpg.de`
`http://www.mpi-sb.mpg.de/~ratschan`
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
`zhikun.she@mpi-sb.mpg.de`
`http://www.mpi-sb.mpg.de/~zhikun`

Abstract. This paper deals with the problem of safety verification of non-linear 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 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. First tests of such an implementation are promising.

1 Introduction

In this paper we provide a method for verifying that a given non-linear 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 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

* 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.

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, and that it ignores the continuous behavior within the grid elements. 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 hyper-rectangles (*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, 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. Then we employ a 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. 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. So we tried to develop a genuine interval based approach here.

Our implementation of the algorithms is publically available [31].

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 constraint propagation techniques; in Section 5 we discuss our implementation; in Section 6 we illustrate the behavior of the implementation using a few example problems; and in Section 8 we conclude the paper.

2 Problem Definition

We fix a variable s ranging over a finite set of discrete modes $\{\mathtt{s_1}, \dots, \mathtt{s_n}\}$ and variables $x_1, \dots, x_k$ ranging over closed real intervals $I_1, \dots, I_k$. We denote by Φ the resulting state space $\{\mathtt{s_1}, \dots, \mathtt{s_n}\} \times I_1 \times \cdots \times I_k$. In addition, for denoting the derivatives of $x_1, \dots, x_k$ we assume variables $\dot{x}_1, \dots, \dot{x}_k$, ranging over $\mathbb{R}$ each, and for denoting the targets of jumps, variables $s', x'_1, \dots, x'_k$ ranging over $\{\mathtt{s_1}, \dots, \mathtt{s_n}\}$ and $I_1, \dots, I_k$, 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 5). These constraints are used, on the one hand, to describe the possible flow and jumps, and on the other hand, 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 $x_1, \ldots x_k$. 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 $n = 2$ and $k = 1$:

$$\Big((s = \mathtt{s}_1 \rightarrow \dot{x} = x) \bigwedge (s = \mathtt{s}_2 \rightarrow \dot{x} = -x)\Big)$$

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. For unrolling such a hybrid system to trajectories we employ the following notation: for a function $r : \mathbb{R}_{\geq 0} \mapsto \Phi$, $\lim_{t' \rightarrow t_-} r(t')$ is the left limit of r at t.

Definition 3. *A* continuous time trajectory *is a function in $\mathbb{R}_{\geq 0} \mapsto \Phi$. A* trajectory of a hybrid system $(Flow, Jump, Init, UnSafe)$ *is a continuous time trajectory r such that*

- *if the real-valued component f of r is differentiable at t, and $\lim_{t' \rightarrow t_-} r(t')$ and $r(t)$ have an equal mode s, then $((s, f(t)), \dot{f}(t)) \in Flow$, and*
- *otherwise, $(\lim_{t' \rightarrow t_-} r(t'), r(t)) \in Jump$.*

A trajectory from a state x to a state y *is a trajectory r such that $r(0) = x$ and there is a $t \in \mathbb{R}_{\geq 0}$ such that $r(t) = y$.*

Note that in this definition we can enforce jumps by formulating a flow constraint that does not allow continuous evolution in a certain region. A *flow* is a trajectory without a jump (i.e., without an evolution according to the relation *Jump*).

Definition 4. *A system is* safe *if and only if there is no trajectory from an initial to an unsafe state.*

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 [16]. So we aim at an algorithm for which we know that, if it terminates, the hybrid system described by the input is safe, and otherwise, we do not know anything.

3 An Interval Based Method

In this section we give an algorithm for verifying safety. 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 the first time in the literature as a basis for a method that abstracts to timed automata [34]. 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". On details how to arrive at such an algorithm see Section 5.

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 $Trans \subseteq S \times S$ and $Init \subseteq S$, $UnSafe \subseteq S$. We call the set S the* state space *of the system.*

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

Trajectories of such systems employ discrete time:

Definition 6. *A* trajectory of a discrete system $(Trans, Init, UnSafe)$ over a set *S is a function $r : \mathbb{N}_0 \mapsto S$ such that for all $t \in \mathbb{N}_0$, $(r(t), r(t+1)) \in Trans$.*

When analyzing discrete systems, we would like to ignore details not relevant to the property we want to verify, that is we would like to use abstractions. This has to be done in a conservative way, that is, if the abstraction is safe, then the original system should also be safe:

Definition 7. *An* abstraction function *between a discrete system $(Trans_1, Init_1, UnSafe_1)$ over S_1 and a discrete system $(Trans_2, Init_2, UnSafe_2)$ over S_2 is a function $\alpha : S_1 \mapsto S_2$ such that for every transition $(x, y) \in Trans_1$, $(\alpha(x), \alpha(y)) \in Trans_2$, for every $q \in Init_1$, $\alpha(q) \in Init_2$, and for every $q \in UnSafe_1$, $\alpha(q) \in UnSafe_2$. A system is an* abstraction *of another one iff there exists an abstraction function between the two.*

Given a system that abstracts another one, we call the former the abstract system and the latter the concrete system.

In our case we want to use abstraction to analyze hybrid instead of discrete systems. Here we have the problem that in a hybrid system we have no notion of transition. Usually, this problem is solved by defining an abstract transition to correspond to either a jump or a (arbitrarily long) flow [2], or to a jump followed by a flow [9]. For both methods one has to follow a flow over potentially unbounded time, which can be extremely costly. Therefore, in our definition, we only require that every concrete trajectory have a corresponding abstract one.

Definition 8. *An* abstraction function *between a hybrid system* $(Flow_1,\ Jump_1,\ Init_1,\ UnSafe_1)$ *and a discrete system* $(Trans_2, Init_2, UnSafe_2)$ *over* S *is a function* $\alpha : \Phi \mapsto S$ *such that:*

– *for all* p, q *in* Φ*, if there is a trajectory from* p *to* q *according to* $Flow_1$ *and* $Jump_1$*, then there is a trajectory from* $\alpha(p)$ *to* $\alpha(q)$ *according to* $Trans_2$*.*
– *for all* $q \in Init_1$*,* $\alpha(q) \in Init_2$*, and for every* $q \in UnSafe_1$*,* $\alpha(q) \in UnSafe_2$*.*

A discrete system is an abstraction of a hybrid system, iff there exists an abstraction function between the two.

Since for every trajectory from Init to Unsafe of the first system there is a corresponding trajectory from Init to Unsafe in the second system, we have:

Property 1. For a system C, for every abstraction C_α of C, if C_α is safe, then C is safe.

Therefore we can prove safety on the abstraction instead of the concrete system. If this does not succeed, 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

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

In order to implement the above 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 $\{\mathtt{s_1}, \dots, \mathtt{s_n}\}$ and B is a hyper-rectangle (*box*), representing subsets of the concrete state space Φ. More specifically, for the initial abstraction we use the state space $\{(\mathtt{s_i}, \{x \mid (\mathtt{s_i}, x) \in \Phi\}) \mid 1 \leq i \leq n\}$. 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 $B = [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_k, \overline{x}_k]$, 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.

Lemma 1. *For a mode s, and two non-overlapping boxes $B \subseteq \mathbb{R}^k$ and $B' \subseteq \mathbb{R}^k$ with $B \cap B' \neq \emptyset$, let F be a face of B s.t. $B \cap B' \subseteq F$. If a flow in s leaves B and enters B' through a point $x \in (B \cap B')$, then*

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

We denote the above constraint by $outgoing^F_{s,B}(x)$. 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 non-overlapping boxes $B, B' \subseteq \mathbb{R}^k$, if there is a flow in mode s that comes from B and enters B' through a common point of B and B', then*

$$\exists a \left[a \in B \wedge a \in B' \wedge \left[\forall F \subseteq B \left[a \in F \Rightarrow outgoing^F_{s,B}(a)\right]\right]\right]$$

This is an immediate result of Lemma 1. We denote the corresponding constraint by $transition_{s,B,B'}$.

Now we compute 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
- there are $x \in B$ and $x' \in B'$ such that $Jump(s, x, s', x')$ holds.

So, 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 non-overlapping, we have a method for computing the set of initial states, set of unsafe states, and 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 non-overlapping, $Abstract_D(\mathcal{B})$ is an abstraction of the hybrid system denoted by D.*

Proof. Let D be an arbitrary, but fixed hybrid system description, and let $\mathcal{B}$ be arbitrary, but fixed set of abstract states covering the whole state space such that all boxes corresponding to the same mode are non-overlapping. Denote the hybrid system described by D by $C_1 = (Flow_1, Jump_1, Init_1, UnSafe_1)$ and $Abstract_D(\mathcal{B})$ by $C_2 = (Trans_2, Init_2, UnSafe_2)$. We first assume that the jump relation $Jump_1$ of the hybrid system C_1 is empty. According to Definition 8 we need to construct an abstraction function $\alpha : \Phi \mapsto \mathcal{B}$ between C_1 and C_2. Let

$\alpha : \Phi \mapsto \mathcal{B}$ be such that $\alpha(s, b) = (s, B)$ where $(s, B) \in \mathcal{B}$ with $b \in B$ (the set $\mathcal{B}$ might contain several such abstract states since the boxes might have common boundaries, in that case one can choose any of these). Clearly, such an α exists since the whole state space is covered by $\mathcal{B}$.

Now we prove that α fulfills the two conditions stated in Definition 8:

– Assume that there is a trajectory from p to q. Since there are no jumps, both p and q have the same mode s. We have to prove that there is an abstract trajectory from $\alpha(p)$ to $\alpha(q)$. Suppose that the trajectory from p to q is covered by abstract states $(s, B_1), \ldots, (s, B_t)$ in the following order according to $Flow_1$: $\alpha(p) = (s, B_1), \ldots, (s, B_t) = \alpha(q)$. By Lemma 2, $transition_{s,B_1,B_2}$, $transition_{s,B_2,B_3}, \ldots$, and $transition_{s,B_{t-1},B_t}$ hold. Therefore, $Abstract_D(\mathcal{B})$ will contain all the transitions over these abstract states. This implies that there is a trajectory from $\alpha(p)$ to $\alpha(q)$ according to $Trans_2$.
– Let $x \in Init_1$ be arbitrary but fixed. Let $(s, B) = \alpha(x)$. Then, since $x \in B$, by the definition of $Abstract_D(\mathcal{B})$, $\alpha(x) \in Init_2$. A similar argument holds for $UnSafe_1$ and $UnSafe_2$.

For a hybrid system with jump relation, we partition the trajectory into parts according to where a jump occurs. Thus, a jump does not occur during each part. Moreover, for each part, its end point and the starting point of the next part satisfy the jump constraint. Therefore, the algorithm will compute these transitions from an abstract state containing the end point to an abstract state containing the starting point of the next part. Combining the above proof, we can deduce that if there is a trajectory from x to y, then there is an abstract trajectory from $\alpha(x)$ to $\alpha(y)$. $\square$

If the differential equations in the flow constraint are in explicit form $\dot{x} = Flow(x)$ 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 $Flow$ 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 5 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 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 two main problems: First, splitting can result in a huge number of boxes, especially for high-dimensional problems; second, the method considers the flow only on the box boundaries and ignores the behavior inside of the boxes. In this section we will try to remove these problems.

In order to remove the first problem, we will refine the abstraction without creating more boxes by splitting. Here we can use the observation that, for safety verification, the unreachable state space is uninteresting, and there is no need to include it into the abstraction. Therefore, instead of requiring an abstraction function (Definition 8), we allow it to be a relation:

Definition 9. *An* abstraction relation *between a hybrid system* ($Flow_1$, $Jump_1$, $Init_1$, $UnSafe_1$) *and a discrete system* ($Trans_2, Init_2, UnSafe_2$) *over* S *is a relation* $\alpha \subseteq \Phi \times S$ *such that:*

- *for all* $q \in \Phi$*, if there is a trajectory from an element of* $Init_1$ *to* q *according to* $Flow_1$ *and* $Jump_1$*, then for all* q_α *with* $\alpha(q, q_\alpha)$ *there is a trajectory from an element of* $Init_2$ *to* q_α *according to* $Trans_2$*,*
- *for all* $q \in Init_1$*, there is a* $q_\alpha \in Init_2$*, with* $\alpha(q, q_\alpha)$ *and*
- *for all* $q \in UnSafe_1$*, if* q *is reachable from* $Init_1$*, then there is a* $q_\alpha \in Unsafe_2$ *with* $\alpha(q, q_\alpha)$*.*

A discrete system is an abstraction *of a hybrid system iff there exists an abstraction relation between the two.*

Clearly Property 1 also holds for this adapted definition.

Note that in the literature there is a similar notion of simulation relation [26, 10]. However, a simulation relation relates transitions between arbitrary states instead of only trajectories that start from the initial set. Now we can modify Theorem 1 as follows:

Theorem 2. *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 non-overlapping,* $Abstract_D(\mathcal{B})$ *is an abstraction of the hybrid system denoted by* D*.*

Proof. We proceed in a similar way as in the proof of Theorem 1. We let D and $\mathcal{B}$ be arbitrary, but fixed, fulfilling the conditions of the theorem, and denote the continuous time hybrid system by $C_1 = (Flow_1, Jump_1, Init_1, UnSafe_1)$ and $Abstract_D(\mathcal{B})$ by $C_2 = (Trans_2, Init_2, UnSafe_2)$. We first assume that the hybrid system has no jump relation. According to Definition 9 we have to construct an abstraction relation $\alpha \subseteq \Phi \times \mathcal{B}$ between C_1 and C_2.

Let α be such that $\alpha((s, x), (s_\alpha, B))$ iff $s = s_\alpha$ and $x \in B$.

- We prove that for every concrete trajectory from a p that is reachable from $Init_1$ to a q, there is a corresponding abstract trajectory (we cannot assume $p \in Init_1$ since, when introducing jumps later, this property has to hold for all jump-less fragments). Since there are no jumps, both p and q have the same mode s. Let p_α and q_α be arbitrary, but fixed, such that $\alpha(p, p_\alpha)$ and $\alpha(q, q_\alpha)$. We prove that there is an abstract trajectory from

p_α to q_α. Now let $(s, B_1), \ldots, (s, B_t)$ be abstract states in $\mathcal{B}$ such that the trajectory from p to q passes them in the following order according to $Flow_1 : p_\alpha = (s, B_1), \ldots, (s, B_t) = q_\alpha$. Such boxes exist, since $\mathcal{B}$ covers the reach set of C_1. By Lemma 2, $transition_{s,B_1,B_2}$, $transition_{s,B_2,B_3}$, $\ldots$, and $transition_{s,B_{t-1},B_t}$ hold. Therefore, $Abstract_D(\mathcal{B})$ will contain all the transitions over these abstract states. This implies that there is a trajectory from p_α to q_α according to $Trans_2$.
- For all $q \in Init_1$, there is a $q_\alpha \in Init_2$, with $\alpha(q, q_\alpha)$ holds by definition of $Abstract_D(\mathcal{B})$ since q is reachable and therefore covered by an element of $\mathcal{B}$. In the same way, for every reachable $q \in UnSafe_1$ there is a $q_\alpha \in UnSafe_2$ with $\alpha(q, q_\alpha)$.

For a hybrid system with jump relation we proceed analogously to Theorem 1. $\square$

So 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. Then we 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*

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

Proof. Assume that $r(t) = (r_1(t), \ldots, r_k(t))$ is a flow in B from x to y. So $r(0) = x$ and for a certain $t \in \mathbb{R}_{\ge 0}$, $r(t) = y$. 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 $a = r(t')$ and $\dot{a} = \dot{r}(t')$. Then, since r is a flow, $Flow(s, a, \dot{a})$ and hence the whole constraint holds. $\square$

The intuition behind the above Lemma is that whenever we have a flow from a 2-dimensional point (x_n, x_m) to a 2-dimensional point (y_n, y_m), then there must be a point on the trajectory, 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 $initflow_B(s, z)$.

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

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

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

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

And 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 boxes $B, B' \subseteq \mathrm{I\!R}^k$, if $z \in B'$ is reachable from a common point of B and B' via a flow in s and B, then*

$$\exists a \left[a \in B \wedge a \in B' \wedge \left[\forall F \subseteq B [a \in F \Rightarrow outgoing^F_{s,B}(a)] \right] \wedge flow_{B'}(s, a, z) \right]$$

This is a consequence of Lemma 3 and Lemma 1. We denote the above constraint by $boundaryflow_{B,B'}(s, z)$.

Now a point in a box is only reachable if it is reachable according to Lemma 4, Lemma 5, or Lemma 6:

Theorem 3. *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, 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 $reachable_{B'}(s', z)$. Now, if we can prove that a certain point does not fulfill this constraint, we know that it is not reachable from the set of initial states. 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 the next section for details on such algorithms.

Since the constraint $reachable_{B'}(s', z)$ 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. This terminates since we use floating point computation here and there are only finitely many floating point numbers. Given a set of abstract states $\mathcal{B}$, we denote the resulting fixpoint by $Prune_D(\mathcal{B})$.

Now, since according to Theorem 2, we do not need to consider unreachable parts of the state space in abstraction, we can do the operation $\mathcal{B} \leftarrow Prune_D(\mathcal{B})$ anywhere in Algorithm 1. So we do this at the beginning, and each time $\mathcal{B}$ is refined by splitting a box.

So our method can in some cases refine the abstraction without splitting, which is a remedy for the first problem identified at the beginning of the section. For doing so, it considers the flow not only on the boundary but also inside of the boxes. Therefore, in addition, the result also provides a remedy for the second problem!

Now observe that in the computation of $\mathit{Abstract}_D(\mathcal{B})$ we check whether one abstract state is reachable from another one. But this information has already been computed by $\mathit{Prune}_D(\mathcal{B})$. More precisely, we get this information from the individual disjuncts of Theorem 3, and we do not need to recompute it. Clearly this does not change the correctness of our abstraction process.

Moreover, we do not need to completely recompute $\mathit{Abstract}_D(\mathcal{B})$ after each refinement step: for this we observe that our solver might prove that one of the disjuncts of the constraint of Theorem 3 has an empty solution. For example, this is trivially the case for $boundaryflow$ and non-neighboring boxes. In such a case we can remove the corresponding disjunct from the disjunction. Afterwards, the constraint only depends on some, but not necessarily all other abstract states in $\mathcal{B}$, and we only have to re-compute it, if one of these changed (cf. the constraint propagation algorithm AC-3 [25]).

5 Implementation

In this section we discuss our implementation of the algorithms introduced in the previous sections.

First, we show how to arrive at a pruning algorithm as required by the previous section. Such 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 such that $N(\phi, B)$ contains all solutions of ϕ in B (cf. the notion of *narrowing operator* [5, 4], sometimes also called *contractor*). There are several methods for implementing such a pruning algorithm. The most basic method [13, 11, 6] 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 $x + 2y \geq 0$ to $2y = v_1 \wedge x + v_1 = v_2 \wedge v_2 \geq 0$). Then it applies a pruning algorithm for these primitive constraints [21] until a fixpoint is reached. Here the floating point results are always rounded outwards, such that the result remains correct also under rounding errors. There are several variants, improvements and alternatives in the literature [22, 5, 23, 24, 20].

The constraints introduced in the previous sections also contain existential quantifiers. These can be treated by simply pruning the Cartesian product of the box corresponding to the free variables and the box bounding the quantified variables [29]. For disjunctions one can prune the disjuncts and take the union of the result [29]. Moreover, the constraints contain variables s and s' ranging over a finite set. These can be easily eliminated by a trivial substitution and simplification.

We have implemented the algorithm on top of our RSolver [30] 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 [18] for pruning primitive constraints. The implementation is publically available [31], and we will make the source code open, which will make it easy to extend it or to experiment with changes.

6 Computation Results

In this section we illustrate the behavior of our implementation on a few examples. Here we use the following splitting strategy: we split several boxes at a time, one box per mode, choosing a box with widest side-length for each mode and then bisecting it along its widest variable. In this way, we can avoid that we keep splitting boxes in the same mode. Of course, one can choose other splitting techniques.

Now we compare the computation results obtained by the basic method of Section 3 and our improved method of Section 4 on some examples. The computations were performed on a Pentium M 1.7 GHz notebook with 512 MB memory. Note that we used the straightforward implementation described in the last section, without any special optimizations whatsoever.

Example 1:

Flow: $\dot{x}_1 = x_1 - x_2, \dot{x}_2 = x_1 + x_2$
Empty jump relation
Init: $2.5 \leq x_1 \leq 3.0 \wedge x_2 = 0$
Unsafe: $x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_2 < -x_1 + 2$
The state space: $[0, 4] \times [0, 4]$

For the basic method, after the 7-th splitting, one can get eight boxes and prove that the region $[0, 1] \times [0, 1]$ can not be reached. After the 15-th splitting, one gets sixteen boxes and prove that the set $\{x_1 \geq 0 \wedge 0 \leq x_2 < -x_1 + 2\}$ cannot be reached. However, for the improved method, after splitting for the 7-th time and calling the pruning method, we get seven boxes and can prove that the set $\{x_1 \geq 0 \wedge 0 \leq x_2 < -x_1 + 2\}$ can not be reached.

The reason is that the improved method not only removes the box $[0, 2] \times [0, 1]$, but also removes part of the box $[0, 2] \times [1, 2]$, both of which do not fulfill the constraint in Theorem 3. The algorithm calls the pruning algorithm for 378 times and costs 0.826 seconds.

Example 2: from a paper by J. Preussig and co-workers [27].

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: $[0, 2] \times [0, 2] \times [0, 4]$

For the basic method, splitting does not improve the abstraction. However, the improved method can prove that the trajectories starting from initial set do not enter the unsafe states. The algorithm only executes the splitting once, calls the pruning algorithm 10 times, gets 2 boxes, and costs 0.339 seconds.

Example 3: The flow constraints are constructed by setting all the parameters in the two tanks problem [34] to 1.

Flow: $\big(s = 1 \rightarrow \binom{\dot{x}_1}{\dot{x}_2} = \big(\begin{smallmatrix} 1-\sqrt{x_1} \\ \sqrt{x_1}-\sqrt{x_2} \end{smallmatrix}\big)\big) \wedge \big(s = 2 \rightarrow \binom{\dot{x}_1}{\dot{x}_2} = \big(\begin{smallmatrix} 1-\sqrt{x_1-x_2+1} \\ \sqrt{x_1-x_2+1}-\sqrt{x_2} \end{smallmatrix}\big)\big)$
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: $\big(s = 1 \wedge (x_1 - 4.5)^2 + (x_2 - 0.25)^2 < 0.0625\big)$
The state space: $(1, [4, 6] \times [0, 1]) \cup (2, [4, 6] \times [1, 2])$

The basic method cannot prove that the trajectories starting from the initial states do not enter the unsafe states. The reason is that splitting does not improve the abstraction. But the improved method can prove that the trajectories starting from initial do not enter the unsafe states. The algorithm does 11 splitting steps, calls the pruning algorithm 5658 times, gets 11 boxes in the first mode and 12 boxes in the second mode, and costs 4.620 seconds.

Example 4: A predator-prey example

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

Again, splitting does not improve the abstraction for the basic method. However, it does in our improved method. The algorithm proves safety, using 61 splitting steps, and 478533 calls to the pruning algorithm, resulting in 56 boxes in the first mode and 61 boxes in the second mode, costing 117 seconds.

The main remaining problem of our improved method is that even in some simple cases it still cannot prove that certain elements of the state space are unreachable. For example, applying our method to the following example: Flow: $\dot{x} = \dot{y} = 1$, Init: $x = y = 0$, State space: $[0, 4] \times [0, 4]$, when we start with splitting along the variable x, we cannot remove parts below the $x = y$ line. If we start with splitting along y, we cannot remove parts above the $x = y$ line. The reason is, that we do not follow trajectories over more than one box.

7 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.

Kowalewski, Stursberg and co-workers pioneered the use of box representations [34, 32, 27, 28, 33]. Also in their method, interval arithmetic is used to check the flow on the boundaries of a rectangular grid. Then timing information is 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 re-computation, whereas in the present work, this can be done incrementally. Also, their method 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 [8, 1, 9, 2, 3]. 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 semi-algebraic sets for representation [35]. This is even more flexible, and can produce symbolic output, but requires highly complex quantifier elimination tools [12]. Again, it is not clear how one could employ a pruning mechanism for such a representation.

There are also methods that use interval arithmetic to compute the reach set explicitly, without abstraction. In one approach [15] an interval ODE solver is used, and in another one [17] a constraint logic programming language [19] that allows constraints with differentiation operators.

8 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 constraint propagation.

Our long term goal is to arrive at a method for which one can prove termination for all, but numerically ill-posed cases, in a similar way as can be done for quantified inequality constraints [29], and hybrid systems in which all trajectories follow polynomials [14]. Moreover, we will use counterexamples to guide the refinement process [9, 2].

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).

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

References

1. R. Alur, T. Dang, and F. Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In Tomlin and Greenstreet [36].
2. R. Alur, T. Dang, and F. Ivančić. Counter-example guided predicate abstraction of hybrid systems. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *LNCS*, pages 208–223. Springer, 2003.
3. E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of hybrid systems. In *CAV'02 - Computer Aided Verification*, number 2404 in LNCS, pages 365–370. Springer, 2002.
4. F. Benhamou. Heterogeneous constraint solving. In *Proc. of the Fifth International Conference on Algebraic and Logic Programming*, 1996.
5. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *International Symposium on Logic Programming*, pages 124–138, Ithaca, NY, USA, 1994. MIT Press.
6. F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.
7. B. F. Caviness and J. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer, Wien, 1998.
8. A. Chutinan and B. H. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In Vaandrager and van Schuppen [37], pages 76–90.
9. E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14(4), 2003.
10. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
11. J. G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.
12. G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12:299–328, 1991. Also in [7].
13. E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, 1987.
14. M. Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In J. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (CSL'99)*, number 1683 in LNCS. Springer, 1999.

15. T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HyTech: hybrid systems analysis using interval numerical methods. In N. Lynch and B. Krogh, editors, *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control (HSCC '00)*, volume 1790 of *LNCS*. Springer, 2000.
16. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata. *Journal of Computer and System Sciences*, 57:94–124, 1998.
17. T. Hickey and D. Wittenberg. Rigorous modeling of hybrid systems using interval arithmetic constraints. In R. Alur and G. J. Pappas, editors, *Hybrid Systems: Computation and Control*, number 2993 in LNCS. Springer, 2004.
18. T. J. Hickey. smathlib. `http://interval.sourceforge.net/interval/C/smathlib/README.html`.
19. T. J. Hickey. Analytic constraint solving and interval arithmetic. In *Proceedings of the 27th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2000.
20. T. J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001.
21. T. J. Hickey, M. H. van Emden, and H. Wu. A unified framework for interval constraint and interval arithmetic. In M. Maher and J. Puget, editors, *CP'98*, number 1520 in LNCS, pages 250–264, 1998.
22. L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics.* Springer, Berlin, 2001.
23. O. Lhomme. Consistency techniques for numeric CSPs. In *Proc. 13th Intl. Joint Conf. on Artificial Intelligence*, 1993.
24. O. Lhomme, A. Gotlieb, and M. Rueher. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming*, 37(1–3):165–183, 1998.
25. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
26. R. Milner. An algebraic definition of simulation between programs. In *Proc. of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, 1971.
27. J. Preußig, S. Kowalewski, H. Wong-Toi, and T. Henzinger. 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, 1998.
28. J. Preußig, O. Stursberg, and S. Kowalewski. Reachability analysis of a class of switched continuous systems by integrating rectangular approximation and rectangular analysis. In Vaandrager and van Schuppen [37].
29. S. Ratschan. Continuous first-order constraint satisfaction. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, number 2385 in LNCS, pages 181–195. Springer, 2002.
30. S. Ratschan. Rsolver. `http://www.mpi-sb.mpg.de/~{}ratschan/rsolver`, 2004. Software package.
31. S. Ratschan and Z. She. Hsolver. `http://www.mpi-sb.mpg.de/~{}ratschan/hsolver`, 2004. Software package.
32. O. Stursberg and S. Kowalewski. 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)*, 2000.

33. O. Stursberg, S. Kowalewski, and S. Engell. On the generation of timed discrete approximations for continuous systems. *Mathematical and Computer Models of Dynamical Systems*, 6:51–70, 2000.
34. O. Stursberg, S. Kowalewski, I. Hoffmann, and J. Preußig. Comparing timed and hybrid automata as approximations of continuous systems. In P. J. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Hybrid Systems*, number 1273 in LNCS, pages 361–377. Springer, 1997.
35. A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In Tomlin and Greenstreet [36].
36. C. J. Tomlin and M. R. Greenstreet, editors. *Hybrid Systems: Computation and Control HSCC*, number 2289 in LNCS, 2002.
37. F. Vaandrager and J. van Schuppen, editors. *Hybrid Systems: Computation and Control – HSCC'99*, number 1569 in LNCS. Springer, 1999.