How to Capture Hybrid Systems Evolution Into Slices of Parallel Hyperplanes

Tomas Dzetkulic * **Stefan Ratschan** **

* *Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodarenskou vezi 2, 182 07 Prague 8, Czech Republic (e-mail: dzetkulic@cs.cas.cz).*
** *Institute of Computer Science, Academy of Sciences of the Czech Republic, Pod Vodarenskou vezi 2, 182 07 Prague 8, Czech Republic (e-mail: ratschan@cs.cas.cz).*

Abstract: In this paper we make a step towards an algorithm for the verification of hybrid systems that, on the one hand allows very general inputs (e.g., with non-linear ordinary differential equations), but on the other hand exploits the structure of those parts of the input that represent special cases (e.g., clocks). We show how to compute slices of parallel hyperplanes separating reachable from unreachable parts of the state space for a given abstraction of the input system, and demonstrate the usefulness of such slices within an abstraction refinement algorithm based on hyper-rectangles.

Keywords: verification, hybrid systems, timed automata

1. INTRODUCTION

Current algorithms for the verification of hybrid systems range from one extreme of techniques that are quite efficient but allow as input only a restricted problem class (e.g., timed automata (Bengtsson and Yi, 2004)), to the other extreme of algorithms that do allow as input a very general problem class, but can hardly verify any examples of the size occurring in practical applications (Ratschan and She, 2007).

The key to arrive at algorithms that combine the advantages of both approaches, lies in the exploitation of the structure of parts of the input that represent special cases.

In this paper we use the observation that differential equalities of the form $\dot{x} = c$ (i.e., clocks of arbitrary speed) and affine switching conditions often result in reach sets that are to a certain extent bounded by hyperplanes. Hence we use pairs of affine inequalities of the form $b_{lo} \leq \boldsymbol{a}x \leq b_{hi}$ (which we call *slices*) to separate the reachable from unreachable parts of the state space. However, in order to allow general applicability, we design an algorithm for generating such slices for hybrid systems that contain non-linear ordinary differential equations, non-linear jumps, etc.

The algorithm computes such slices for a given abstraction of a hybrid system. It sets up a constraint that formalizes reachability of states within an abstract state and over-approximates the solution set of this constraint in the form of a slice.

The presentation in the paper uses abstractions in the form of hyper-rectangles (*boxes*), but also discusses the case where polyhedra are used instead. Moreover, we study the use of such slices in safety verification methods based on abstract refinement, especially constraint propagation based abstraction refinement (Ratschan and She, 2007). Extensive computational experiments show that in fact the efficiency of these methods becomes much more robust when using slices. Especially, the method can now verify the up to now unsolved heating benchmark (Fehnker and Ivančić, 2004).

In contrast to many algorithms in hybrid systems verification, the correctness of our algorithms cannot be hampered by floating-point rounding errors, since we do conservative rounding throughout.

Concerning related work, Sankaranarayanan et al. (2008) describe how to over-approximate the reach set of hybrid systems using hyper-planes with *given* coefficients. Their approach proved successful in *forward* computation of the reach set, repeating flow pipe constructions with a fixed time step. In contrast to that, in our approach we deduce not only the constant of the hyper-plane but also their coefficients. Moreover, we aim at improving *abstractions*, and not at forward computation. Hence we do not employ a potentially unbounded number of flow pipe constructions with a fixed time step, but try to infer single slices as fast as possible, allowing for efficient abstraction refinement.

Gulwani and Tiwari (2008) synthesize inductive invariants of hybrid systems. Whenever such an inductive invariant can be found, this verifies safety of the system. However, the resulting search for an invariant can be expensive: Gulwani and Tiwari (2008) use a translation to Boolean satisfiability modulo bit vectors here. In contrast to that, we concentrate on improvements of abstractions that can be computed in negligible time.

⋆ The work on this paper has been supported by GAČR grants 201/08/J020 and 201/09/H057, and by the institutional research plan AV0Z100300504 of the Czech Republic.

The content of the paper is as follows: In Section 2 we describe the problem of hybrid systems verification, in Section 3 we describe how to formulate constraints that over-approximate the reachable states, in Section 4 we describe our over-all approach of computing slices from the reachability constraint already used for computing abstractions and we estimate complexity of such computation, in Section 5 we evaluate the method using some computation experiments, and in Section 6 we conclude the paper.

2. SAFETY VERIFICATION OF HYBRID SYSTEMS

In this section, we briefly recall our formalism for modeling hybrid systems. It captures many relevant classes of hybrid systems, and many other formalisms for hybrid systems in the literature are special cases of it. We use a set S to denote the modes of a hybrid system, where S is finite and nonempty. $I_1, \ldots, I_k \subseteq \mathbb{R}$ are compact intervals over which the continuous variables of a hybrid system range. Φ denotes the state space of a hybrid system, i.e., $\Phi = S \times I_1 \times \cdots \times I_k$.

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

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

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

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

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

We use the following constraint language to describe hybrid systems and corresponding safety verification problems. The variable s ranges over S and the tuple of variables $\mathbf{x} = (\mathsf{x}_1, \ldots, \mathsf{x}_k)$ ranges over $I_1 \times \cdots \times I_k$, respectively. In addition, to denote the derivatives of $\mathsf{x}_1, \ldots, \mathsf{x}_k$ we use the tuple of variables $\dot{\mathbf{x}} = (\dot{\mathsf{x}}_1, \ldots, \dot{\mathsf{x}}_k)$ that ranges over $\mathbb{R}^k$,[1] and to denote the targets of jumps, we use the variable $\widehat{s}$ and the tuple of variables $\widehat{\mathbf{x}} = (\widehat{\mathsf{x}_1}, \ldots, \widehat{\mathsf{x}_k})$ that range over S and $I_1 \times \cdots \times I_k$, respectively. Constraints are arbitrary Boolean combinations of equalities and inequalities over terms that may contain function symbols, like $+$, $\times$, exp, sin, and cos.

[1] The dot does not have any special meaning here; it is only used to distinguish dotted from undotted variables.

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

Example 1. For illustrating the above definitions, consider the following simple hybrid system with the modes m_1, m_2 and the continuous variables x_1, x_2, where x_1 ranges over the interval $[0, 2]$ and x_2 over $[0, 1]$, i.e, $\Phi = \{m_1, m_2\} \times [0, 2] \times [0, 1]$.

The set of initial states are given by the constraint $\texttt{Init}(s, (x_1, x_2)) \equiv (s = m_1 \wedge x_1 = 0 \wedge x_2 = 0)$. The constraint $\texttt{Unsafe}(s, (x_1, x_2)) \equiv (x_1 > 1 \wedge x_2 = 1)$ describes the set of unsafe states. The hybrid system can switch modes from m_1 to m_2 if $x_2 \geq 1$, i.e., $\texttt{Jump}(s, (x_1, x_2), s', (x_1', x_2')) \equiv (s = m_1 \wedge x_2 \geq 1 \to s' = m_2 \wedge x_1' = x_1 \wedge x_2' = x_2)$, The continuous behavior is very simple: In mode m_1, the values of the variables x_1, x_2 change with slope 1; in mode m_2, the slope of variable x_1 is 1 and variable x_2 has slope -1. For a flow in mode m_1, the constraint $0 \leq x_1 \leq 1$ must hold and in mode m_2, $1 \leq x_1 \leq 2$ must hold. The corresponding flow constraint is $\texttt{Flow}(s, (x_1, x_2), (\dot{x}_1, \dot{x}_2)) \equiv (s = m_1 \to \dot{x}_1 = 1 \wedge \dot{x}_2 = 1 \wedge 0 \leq x_1 \leq 1) \wedge (s = m_2 \to \dot{x}_1 = 1 \wedge \dot{x}_2 = -1 \wedge 1 \leq x_1 \leq 2)$. Note that the constraint $0 \leq x_1 \leq 1$ in `Flow` forces a jump from mode m_1 to m_2 if x_1 becomes 1. In general, an invariant that has to hold in a mode can be modeled by formulating a flow constraint that does not allow a continuous behavior in certain regions.

Obviously, this hybrid system is safe.

A *box abstraction* of a hybrid system H is a set of non-overlapping mode/box pairs (which we call *abstract states*) with transitions between them, such that:

- every point on a counterexample of H is an element of an abstract state
- whenever a counterexample moves from an abstract state a_1 to an abstract state a_2 then there is a corresponding transition $a_1 \to a_2$ from a_1 to a_2 (it is an easy, but rather technical exercise to formally define "moves from").

We assume that we have a method for computing such a box abstraction (Ratschan and She, 2007; Stursberg and Kowalewski, 2000).

3. REACHABILITY CONSTRAINTS

Now we assume that we have a box abstraction for a given hybrid system H, and present a constraint formalizing the fact that a given point in the state space of H may lie on a counterexample.

Observe that a point lies on a counterexample iff it is both reachable from an initial state *and* leads to an unsafe state. We will only formalize the first part of this condition, the second part is dual. Details can be found in an earlier

publication (Ratschan and She, 2008). In the case where an over-approximation of the set of reachable states, or of the set of backward reachable states is desired, one can use only one of the two dual formalizations.

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. These three possibilities refer to the three parts of the conjunction in the following Theorem 1. We will now formulate constraints corresponding to each of these conditions. Then we can remove points from boxes that do not fulfill at least one of these constraints.

The approach can be used with any constraint that describes that $\mathbf{y}$ can be reachable from $\mathbf{x}$ via a flow in B and mode s, for example:

Lemma 1. If a hybrid system $(\mathtt{Flow}, \mathtt{Jump}, \mathtt{Init}, \mathtt{Unsafe})$ has a trajectory in a box $B \subseteq \mathbb{R}^k$ consisting of one single flow from a point $\boldsymbol{x} = (s, x_1, \ldots, x_k)^T \in B$ to a point $\boldsymbol{y} = (s, y_1, \ldots, y_k)^T \in B$, then

$$\exists t \in \mathbb{R}_{\geq 0} \exists \dot{\boldsymbol{x}} \in \mathbb{R}^k \exists \dot{\boldsymbol{y}} \in \mathbb{R}^k \\ [\mathtt{Flow}(s, \boldsymbol{x}, \dot{\boldsymbol{x}}) \wedge \mathit{flow}^*_B(t, \boldsymbol{x}, \boldsymbol{y}) \wedge \mathtt{Flow}(s, \boldsymbol{y}, \dot{\boldsymbol{y}})],$$

where $\mathit{flow}^*_B(t, \boldsymbol{x}, \boldsymbol{y})$ denotes

$$\bigwedge_{1 \leq i \leq k} \exists a_1, \ldots, a_k, \dot{a}_1, \ldots, \dot{a}_k [(a_1, \ldots, a_k) \in B \\ \wedge \mathtt{Flow}(s, (a_1, \ldots, a_k), (\dot{a}_1, \ldots, \dot{a}_k)) \wedge y_i = x_i + \dot{a}_i \cdot t]$$

We denote the above constraint by $\mathsf{Reach}_{\mathsf{B}}(\mathsf{s}, \mathbf{x}, \mathbf{y})$. For formalizing the above three possibilities for reachability, for a box $B = [\underline{x}_1, \overline{x}_1] \times \cdots \times [\underline{x}_k, \overline{x}_k]$, we define its j-th lower face to 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 to be $[\underline{x}_1, \overline{x}_1] \times \cdots \times [\overline{x}_j, \overline{x}_j] \times \cdots \times [\underline{x}_k, \overline{x}_k]$. Now we can formulate the following theorem:

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

$$\mathsf{Ifl}_{\mathsf{B}'}(\mathsf{s}', \mathbf{z}) \vee \bigvee_{(\mathsf{s},\mathsf{B}) \in \mathcal{B}, (\mathsf{s},\mathsf{B}) \to (\mathsf{s}',\mathsf{B}')} \mathsf{Jfl}_{\mathsf{B},\mathsf{B}'}(\mathsf{s}, \mathsf{s}', \mathbf{z}) \\ \vee \bigvee_{(\mathsf{s},\mathsf{B}) \in \mathcal{B}, \mathsf{s}=\mathsf{s}', \mathsf{B} \neq \mathsf{B}', (\mathsf{s},\mathsf{B}) \to (\mathsf{s}',\mathsf{B}')} \mathsf{Bfl}_{\mathsf{B},\mathsf{B}'}(\mathsf{s}', \mathbf{z})$$

where $\mathsf{Ifl}_{\mathsf{B}'}(\mathsf{s}', \mathbf{z})$, $\mathsf{Jfl}_{\mathsf{B},\mathsf{B}'}(\mathsf{s}, \mathsf{s}', \mathbf{z})$, and $\mathsf{Bfl}_{\mathsf{B},\mathsf{B}'}(\mathsf{s}', \mathbf{z})$ denote the following three constraints, respectively:

- $\exists \mathbf{x} \in \mathsf{B}' [\mathtt{Init}(\mathsf{s}', \mathbf{x}) \wedge \mathsf{Reach}_{\mathsf{B}'}(\mathsf{s}', \mathbf{x}, \mathbf{z})]$,
- $\exists \mathbf{x} \in \mathsf{B} \exists \mathbf{x}' \in \mathsf{B}' [\mathtt{Jump}(\mathsf{s}, \mathbf{x}, \mathsf{s}', \mathbf{x}') \wedge \mathsf{Reach}_{\mathsf{B}'}(\mathsf{s}', \mathbf{x}', \mathbf{z})]$,
- $\exists \mathbf{x} \in \mathsf{B} \cap \mathsf{B}' \left[\left[\forall \text{faces } \mathsf{f} \text{ of } \mathsf{B}' [\mathbf{x} \in \mathsf{f} \Rightarrow \mathsf{in}^{\mathsf{f}}_{\mathsf{s}',\mathsf{B}'}(\mathbf{x})] \right] \wedge \mathsf{Reach}_{\mathsf{B}'}(\mathsf{s}', \mathbf{x}, \mathbf{z}) \right]$.

Here,
$\mathsf{in}^{\mathsf{f}}_{\mathsf{s}',\mathsf{B}'}(\mathbf{x}) = \exists \dot{\mathsf{x}}_1, \ldots, \exists \dot{\mathsf{x}}_k [\mathtt{Flow}(\mathsf{s}', \mathbf{x}, (\dot{\mathsf{x}}_1, \ldots, \dot{\mathsf{x}}_k)) \wedge \dot{\mathsf{x}}_j \geq 0]$, if f is the j-th lower face of B', and if f is the j-th upper face of B',
$\mathsf{in}^{\mathsf{f}}_{\mathsf{s}',\mathsf{B}'}(\mathbf{x}) = \exists \dot{\mathsf{x}}_1, \ldots, \exists \dot{\mathsf{x}}_k [\mathtt{Flow}(\mathsf{s}', \mathbf{x}, (\dot{\mathsf{x}}_1, \ldots, \dot{\mathsf{x}}_k)) \wedge \dot{\mathsf{x}}_j \leq 0]$.

We denote the main constraint of Theorem 1 by $\mathsf{reach}_{\mathcal{B},\mathsf{B}'}(\mathsf{s}', \mathbf{z})$. If we can prove that a certain point does not fulfill this constraint, we know that it is not reachable.

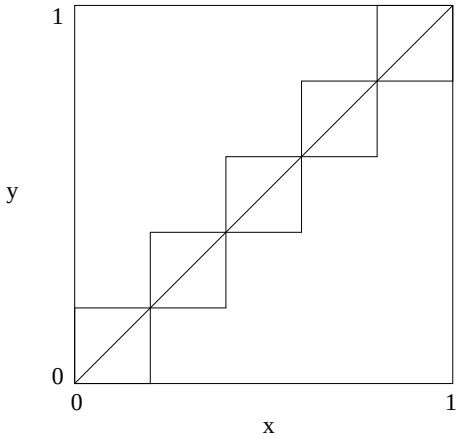

Fig. 1. Constraint $x = y$ and its box enclosure in box $[0, 1]^2$

4. SLICE COMPUTATION

In this section we describe our approach of adding slices of the form $b_{lo} \leq \boldsymbol{a}x \leq b_{hi}$ to abstract states of a given abstraction. Here, b_{lo} may be $-\infty$, and b_{hi} may be ∞. We call a slice without any finite bound *trivial*.

The amount of over-approximation in box abstraction can be huge. See Fig. 1 for graphical representation of a simple linear constraint. With slices, we can remove all over-approximation in such linear cases and also decrease over-approximation in non-linear cases as we will show below

Throughout the rest of the paper we will use interval arithmetic techniques for safely over-approximating global behavior and for safe handling of rounding errors. Whenever one of the arguments to an arithmetic operation is an interval, the corresponding interval operation is used. Here, if the other argument is a single real number, this number is enclosed into an interval before application of the interval operation.

Initially, we add a trivial slice to each abstract state. During computation, non-trivial slices appear that over-approximate the set of states lying on a counter-example.

For this, we start with the reachability constraint described in Theorem 1 with every constraint $x \in B$ in this constraint replaced by $x \in B \wedge \beta_B(x)$, where β_B is the slice that we already have for the box B. Then, we proceed in two steps, described in the corresponding subsections below: First, we replace every non-linear atomic sub-constraint (i.e., equality or inequality) by an over-approximating slice (Subsection 4.1). Then, we compute a slice for the overall constraint using the constraint solving technique described in the next section (Subsection 4.2).

Note that a tighter slice for a given abstract state a results in a tighter reachability constraint for all abstract states a' reachable from a by an abstract transition $a \to a'$. Hence, slice computation can be iterated over the abstraction until no significant slice improvements occur any more.

4.1 Linearization of Atomic Sub-constraints

We compute an over-approximating slice of non-linear atomic sub-constraints by first computing a slice with interval coefficients, and from this, a slice with real coefficients.

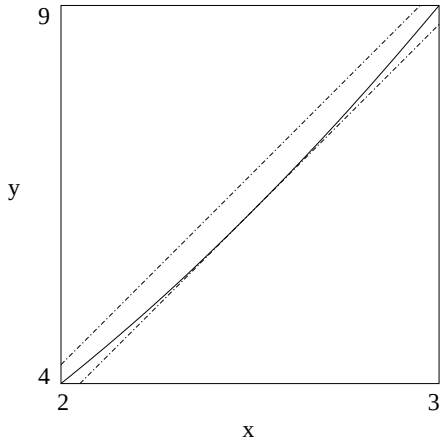

Fig. 2. Constraint $y = x^2$ and a slice over-approximating it's solution in box $[2, 3] \times [4, 9]$

For computing a slice with interval coefficients from a function f in a box B we use a similar method as introduced by Kolev and Nenov (2001): Let us denote by $\mathbf{x}_{mid}$ the center of B and by D an interval vector over-approximating the gradient of ϕ in B (this can be computed by automatic differentiation Griewank (2000)). Let the function $F(\mathbf{x})$ be the linearization $f(\mathbf{x}_{mid})+D(\mathbf{x}-\mathbf{x}_{mid})$. Then, due to the mean value theorem, we have:

Theorem 2. If f and F are as above, then for all $\mathbf{x} \in B$, $f(\mathbf{x}) \in F(\mathbf{x})$.

We compute a slice with constant coefficients from the above linear function with interval coefficients by choosing the midpoint of each interval coefficient. Then we have, for $\mathbf{d}$ being the midpoint of D, $F(\mathbf{x}) =$
$f(\mathbf{x}_{mid}) + D \cdot (\mathbf{x} - \mathbf{x}_{mid}) =$
$f(\mathbf{x}_{mid}) + \mathbf{d} \cdot (\mathbf{x} - \mathbf{x}_{mid}) + (D - \mathbf{d}) \cdot (\mathbf{x} - \mathbf{x}_{mid}) \subseteq$
$\mathbf{d} \cdot \mathbf{x} - \mathbf{d} \cdot \mathbf{x}_{mid} + f(\mathbf{x}_{mid}) + (D - \mathbf{d}) \cdot (B - \mathbf{x}_{mid})$.

The latter expression still contains intervals, but only as constant terms—all coefficients are constants. Depending on whether we have the relation symbol $\leq$, $\geq$ or $=$ we create a slice consisting of a linear inequality in $\mathbf{x}$ with non-trivial upper bound, lower bound, or both bounds, respectively.

The error we make with over-approximation is equal to the width of the interval $(D - \mathbf{d}) \cdot (B - \mathbf{x}_{mid})$, but it goes to zero quadratically with the width of the box B.

Note that the application of the above procedure to an input that is already affine, just returns the input as output.

Example 2. Example of linearization using the above algorithm for constraint $x^2 - y = 0$ in box $(x, y) \in [2, 3] \times [4, 9]$: Automatic derivation gives us $D_x = [4, 6]$ and $D_y = [-1, -1]$. The midpoints $(d_x, d_y) = (5, -1)$ of these intervals are the constant coefficients of the linearization. The corresponding interval constant term $-\mathbf{d} \cdot \mathbf{x}_{mid} + f(\mathbf{x}_{mid}) + (D - \mathbf{d}) \cdot (B - \mathbf{x}_{mid})$ is $-5 * 2.5 + 6.5 + 2.5^2 - 6.5 + [-1, 1] * [-0.5, 0.5] = -6.25 + [-0.5, 0.5] = [-6.75, -5.75]$. The result of linearization of the constraint $x^2 - y = 0$ in the box $[2, 3] \times [4, 9]$ is the slice $5.75 \leq 5x - y \leq 6.75$.

See Fig. 2 for the graphical representation of this example.

4.2 Approximate Solving of Quantified Constraints

In this sub-section we compute, given a quantified constraint and a box, a slice over-approximating the solution set of the constraint in the box.

For atomic constraints (equalities or inequalities) we get the slices using the linearization approach described in the previous section. Then we proceed recursively, bottom-up, according to the logical symbols (existential quantifiers, conjunctions, and disjunctions) occurring in the formula. Due to the fact that $\exists[\phi \vee \psi]$ is equivalent to $[\exists\phi] \vee [\exists\psi]$ we can assume that the logical symbol below an existential quantifier is always a conjunction. Hence we can treat existential quantifiers and conjunctions in one step. We will now describe this step, and at the end of the section we show how we handle disjunctions.

The common way of handling quantifiers, is to use methods for eliminating them. However, quantifier elimination is known to be expensive, since it blows up the number of constraints (Fischer and Rabin, 1974). Our method uses a similar elimination step as Fourier-Motzkin elimination, but only generates a sub-set of the corresponding constraints. From this subset we then pick the constraint that provides most information in the given box.

Due to the correctness of Fourier-Motzkin elimination, and due to the fact that removing constraints from a conjunction over-approximates the solution set of this conjunction, this method is correct. That is, it results in a conservative over-approximation of the solution set of the input constraint in the given box.

In order to be able to choose constraints that provide more information within a given box B, we will define a goodness function $\gamma(\phi, B)$ that assigns a value from the interval $[0, 1]$ to a constraint in such a way, that a higher value means that the constraint is more important (contains more useful information).

Definition 3. For a constraint $f(x) \leq 0$ and box B, let $f(B) = [\underline{a}, \overline{a}]$ be an interval that we obtain from interval arithmetic if we substitute the whole box B for the variables in function f. We define the goodness function $\gamma(f(x) \leq 0, B)$ as follows:

- 0, if $\overline{a} \leq 0$,
- 1, if $\underline{a} > 0$, and
- $\overline{a}/(\overline{a} - \underline{a})$, otherwise.

For a slice $\beta(x)$, we define $\gamma(\beta, B)$ as the sum of $\gamma(\phi, B)$ for both contained inequalities.

We can see that, if $f(B)$ evaluates to an interval containing only negative values, the constraint $f \leq 0$ is true on the whole box and this constraint can be excluded from our system. Such a constraint will be assigned goodness value 0. Constraints with higher values are more important, because they cut off some volume of the box.

In the case where we have abstractions based on polyhedra instead of boxes, computation of a similar goodness function would need the solution of two linear programs computing a lower and upper found of f in the polyhedron.

Recall that Fourier-Motzkin elimination is based on the fact that two linear inequalities ϕ and ψ in which a certain

variable x has coefficients of opposite sign, can be written in the form $f \leq x$ and $x \leq g$, where f and g are linear expressions not containing the variable x. This in turn implies the constraint $f \leq g$, again not containing variable x. For such ϕ and ψ we will now write this implied constraint as $\texttt{elim}(\phi, \psi)$.

The problem is, that for a given conjunction of k linear inequalities, in the worst case there may be $k^2/4$ such pairs. Hence we use the goodness function to only generate those constraints in Fourier-Motzkin elimination that appear to provide promising information. Here we use a 2-step process: First, generate implied constraints using `elim` using only a subset of the $k^2/4$ pairs used in Fourier-Motzkin elimination, and second, choose a sub-sub-set of the implied constraints.

The algorithm has the following specification:

- Input:
  - a set Φ of linear inequalities in variables $x_1, \ldots, x_n$
  - an n-dimensional box $B = I_1 \times \cdots \times I_n$
- Output: a set Φ' of linear inequalities in variables $x_1, \ldots, x_{n-1}$ such that for all $(x_1, \ldots, x_n) \in B$, $\bigwedge \Phi(x_1, \ldots, x_n) \implies \bigwedge \Phi'(x_1, \ldots, x_{n-1})$

For applying it we simply view the slices computed for all sub-constraints as two inequalities.

In the algorithm, we use interval arithmetic to avoid rounding errors. We denote by $\overline{I}$ the upper bound of the interval I and $\underline{I}$ the lower bound of the interval. The algorithm works as follows:

- Add constraints $x_n - \overline{I_n} \leq 0$ and $-x_n + \underline{I_n} \leq 0$ to the input constraint Φ (to enforce the bounds given by the input box)
- Split the constraints in Φ into 3 groups:
  - Φ^+, where x_n has positive coefficient
  - Φ^-, where x_n has negative coefficient
  - Φ^0, where x_n has zero coefficient
- let Φ' be $\{\texttt{elim}(\phi_\gamma^+, \phi^-) \mid \phi^- \in \Phi^-\} \cup \{\texttt{elim}(\phi^+, \phi_\gamma^-) \mid \phi^- \in \Phi^-\} \cup \Phi^0$, where ϕ_γ^+ (ϕ_γ^-, respectively) is the element of Φ^+ (Φ^-, respectively) with maximal goodness.
- remove all constraints from Φ' whose goodness is not good enough according to some heuristics.

Here, to take care of rounding errors, when computing `elim` we first use interval arithmetic and then compute an over-approximating constraint with constant coefficients using the according method from Subsection 4.1.

Our choice of pairs of constraints to combine is based upon the following criteria:

- Use all input constraints in at least one combined pair in order preserve variety of constraints for the second step.
- Combine the best constraints, with respect to goodness function, with as many other constraints as possible.

Note that, in the worst case, Φ' contains $|\Phi|+1$ constraints. Moreover, this set can be made arbitrarily small by using appropriate heuristics in the last step of the algorithm.

The heuristic that we use is the following: We first check whether Φ' contains some constraints with equal coefficients. If yes, we collect all their bounds into one slice. For such a slice we define the goodness function as the sum of goodness of both contained inequalities. Then, we pick the slice with the best goodness function. If we apply the algorithm several times to eliminate a block of several existential quantifiers, we remove constraints only after the last elimination step. Hence, in the worst case, after eliminating the q-th quantifier, we have $|\Phi|+q$ constraints, from which we extract one slice after the last elimination.

Example 3. Let us assume that $x \in [0,1], y \in [0,1], t \in [0,1]$ and that the result of the linearization described in Section 4 is $x = t, y = [0,1] + t$. Before applying the above algorithm, we use a preprocessing step to replace each equality by two inequalities and to eliminate the intervals. The result is $x - t \leq 0, -x + t \leq 0, y - t - 1 \leq 0, -y + t \leq 0$. Goodness of these constraints is 0.5, 0.5, 0 and 0.5 respectively. Now we apply the above algorithm.In the first step, we add constraints $t \leq 1$ and $-t \leq 0$ to the input set. Now we split our constraints: $x - t \leq 0, y - t - 1 \leq 0$ and $-t \leq 0$ form a negative group. Goodness of these constraints is $0.5, 0$ and 0 respectively. We will pick first constraint for combining step. The result of combining step are constraints $0 \leq 0, x - y \leq 0$ and $x \leq 1$. Similarly we do the combining step with positive group with result $-1 \leq 0, 0 - y \leq 0$ and $y \leq 2$. The only constraint with the positive goodness is $x - y \leq 0$. It would be the result of the algorithm.

For handling disjunctions we have to show how, given a constraint $\phi_1 \vee \cdots \vee \phi_n$, boxes $B_1, \ldots, B_n$ and slices $\beta_1, \ldots, \beta_n$, to compute a single slice that over-approximates the solution set of $\phi_1 \vee \cdots \vee \phi_n$ in the box hull of $B_1 \cup \cdots \cup B_n$.

For doing this we exploit the fact that the slices computed for the constraints under the big disjunction in Theorem 1 often point in a similar direction. The algorithm picks multiple promising slice normals (i.e, coefficients $\boldsymbol{a}$), then computes bounds of slices having such normals and in the final step algorithm chooses the slice with the highest goodness. The candidates for good normals are the normals of input slices. In the following, we will explain how to compute upper and lower bound of a slice with a given normal. With each normal $\mathbf{a}$ we do the following: For each box B_i and respective slice β_i in the disjunction we find $m_i = min(\mathbf{ax})$ in $B_i \cap \beta_i$ and $M_i = max(\mathbf{ax})$ in $B_i \cap \beta_i$. The slice $\min_i(m_i) \leq \mathbf{a} \cdot \mathbf{x} \leq \max_i(M_i)$ then over-approximates all the results of the input disjunction. To solve the optimization problems, we use approximate quantifier elimination to eliminate $\mathbf{x}$ from $q = \mathbf{ax} \wedge \beta_i$ in box B_i. The resulting inequality just contains the single variable q, and hence allows us to read off a bounds for q that safely over-approximates m_i and M_i.

4.3 Complexity of slice computation

We now estimate how many operations is needed in the worst case for computing a single slice. For hybrid system with n variables, we have constraints with $O(n)$ inequalities. The algorithm does $O(n)$ elimination steps, while computing goodness functions and combining constraints

takes $O(n^2)$ operations. Hence computing one slice takes $O(n^3)$ arithmetic operations.

5. COMPUTATIONAL EXPERIMENTS

For our experiments, we integrated slice computation into verification by constraint propagation based abstraction refinement (Ratschan and She, 2007). That algorithm computes an abstraction based on boxes by alternating a pruning step that replaces boxes by smaller ones still capturing all counter-examples of the input system, with a step that splits boxes. In our extension of the algorithm, after every pruning of a box, we also compute a slice for that box.

Table 1. Experimental results

	without slices		with slices		
Name	Recomp	Time	Recomp	Slices	Time
1-flow	2	$< 1s$	2	1	$< 1s$
2-tanks	5	$< 1s$	5	1	$< 1s$
car	30	$< 1s$	30	8	$< 1s$
circuit	1478	$9s$	1478	0	$16s$
eco	1230	$4s$	1230	8	$10s$
focus	51	$< 1s$	51	0	$< 1s$
mixing	3	$< 1s$	3	2	$< 1s$
van-der-pole	5	$< 1s$	5	0	$< 1s$
fischer2	N/A	N/A	24	9	$< 1s$
fischer3	N/A	N/A	656	259	$19s$
heat-simple	46028	$1904s$	1963	728	$632s$
heating	N/A	N/A	26862	2804	$3525s$

We took benchmarks from our database (`http://hsolver.sourceforge.net/benchmarks`) and, for each example, describe the behavior of a full verification cycle with integrated slice computation in Table 1. All timings were measured on PC with Intel Core 2 3.0GHz CPU and 2GB RAM. The first two columns refer to abstraction refinement without slice computation, the rest refers to the method with slice computation added. The columns "Recomp" refer to the number of recomputed boxes, and "Slices" refers to the number of times a non-trivial slice was computed in a certain box recomputation. We can see that slices indeed exploit the fact that certain examples partially correspond to special cases (clocks in the case of fischer2 and fischer3, affine switching hyperplanes in the heating example). Note that up to our knowledge, the heating example has been unsolved up to now. On the other hand, in cases where slices are not able to exploit any special structure, computation time may increase due to the additional computational effort (for example, the circular structure of the focus example prohibits any slice computation).

We did additional timings measuring the amount of time needed by slice computation within the algorithms, and in all examined cases it needed approximately halve of computation time. One single slice computation usually takes 12ms in examples of dimension 3. Moreover, due to the cubic worst-case complexity of slice computation, even in the worst case one will be able to compute slices for very high problem dimensions. Also observe that due to sparsity, in practice, slice computation will be much more efficient than in the worst case.

Summing up, we can conclude that the efficiency of the resulting method is more robust in the sense that it never more than approximately doubles computation time, but by exploiting some simple problem structure, it makes the solution of example with such structure much more efficient, and hence allows the solution of many more examples.

6. CONCLUSION

In this paper we introduced a method for the verification of hybrid systems that can handle a very general class of hybrid systems, but still exploits some of the structure of more special cases. Computational experiments confirm a corresponding efficiency improvement. In future work we will further decrease the amount of over-approximation of our slices, while retaining the efficiency of their computation. Moreover, we will try to arrive at an algorithm that provably terminates successfully for all robust inputs (Damm et al., 2007; Ratschan, 2009).

REFERENCES

Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, 87–124. Springer.

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

Damm, W., Pinto, G., and Ratschan, S. (2007). Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems. *Int. J. of Foundations of Computer Science*, 18(1), 63–86.

Fehnker, A. and Ivančić, F. (2004). Benchmarks for hybrid systems verification. In R. Alur and G.J. Pappas (eds.), *HSCC'04*, number 2993 in LNCS. Springer.

Fischer, M.J. and Rabin, M.O. (1974). Super-exponential complexity of presburger arithmetic. *SIAM-AMS Proceedings*, 7, 27–41. Also in Caviness and Johnson (1998).

Griewank, A. (2000). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM.

Gulwani, S. and Tiwari, A. (2008). Constraint-based approach for verification and synthesis of hybrid systems. In *CAV*, number 5123 in LNCS, 190–203. Springer.

Kolev, L.V. and Nenov, I.P. (2001). Cheap and tight bounds on the solution set of perturbed systems of nonlinear equations. *Reliable Computing*, 7(5), 399–408.

Ratschan, S. (2009). Safety verification of non-linear hybrid systems is quasi-decidable. Submitted.

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

Ratschan, S. and She, Z. (2008). Recursive and backward reasoning in the verification on hybrid systems. In *Proc. of the 5th Int. Conf. on Informatics in Control, Automation, and Robotics*. INSTICC Press. Available from `http://www.cs.cas.cz/~ratschan/papers/bwrec.pdf`.

Sankaranarayanan, S., Dang, T., and Ivani, F. (2008). Symbolic model checking of hybrid systems using template polyhedra. In *TACACS*, number 4963 in LNCS, 188–202. Springer.

Stursberg, O. and Kowalewski, S. (2000). Analysis of controlled hybrid processing systems based on approximation by timed automata using interval arithmetic. In *Proc. of the 8th IEEE Mediterranean Conf. on Control and Automation (MED 2000)*.