Verification of Hybrid Systems by Incremental Abstract Forward/Backward Computation

Dzetkulič, Tomáš
2010

Verification of Hybrid Systems by Incremental Abstract Forward/Backward Computation

Post-Graduate Student:
MGR. TOMÁŠ DZETKULIČ
Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ
dzetkulic@cs.cas.cz

Supervisor:
ING. STEFAN RATSCHAN, PH.D.
Institute of Computer Science of the ASCR, v. v. i.
Pod Vodárenskou věží 2
182 07 Prague 8, CZ
ratschan@cs.cas.cz

Field of Study:
Verification of Hybrid Systems

This work has been supported by Czech Science Foundation grants 201/08/J020 and 201/09/H057, and by the institutional research plan AV0Z100300504 of the Czech Republic.
This paper will be presented at the International Workshop on Reachability Problems 2010 and will be published in the internal workshop proceedings.

Abstract

In this paper, we introduce a new approach to unbounded safety verification of hybrid systems with non-linear ordinary differential equations. It incrementally refines an abstraction of the system, but avoids increases in the size of the abstraction as much as possible, in order to avoid the usual blow-up problem of applications of counter-example guided abstraction refinement in a hybrid systems context.

1. Introduction

In this paper, we study hybrid (dynamical) systems, that is systems with both discrete and continuous state and evolution. We address the problem of unbounded safety verification of hybrid systems, that is, the verification that a given hybrid system does not have a trajectory (of unbounded length) from an initial state to a set that is considered unsafe. The traditional approach to solving this problem computes the set of reachable states of the system. If the intersection of this reach set with the set of unsafe states is empty, the safety property holds. This has several disadvantages: (1) When computing the reach set, information about the topology of the set of unsafe states is ignored. (2) Even over bounded time, exact reachability computation is possible only for very special cases, and hence (unlike for discrete systems) one has to use over-approximation. It is a-priori not clear, how much to over-approximate in order to prove a given property.

Hence, it is necessary, to compute several, incrementally tighter reach set over-approximations. However, the current approaches do not exploit information from one over-approximation to the next. Some approaches do exploit dual (forward/backward) or incrementally tighter reachability analyses [6, 8]. But, reuse of analyses only concerns dropping initial/unsafe states that have been shown not to lie on any error trajectory—no reuse is done concerning the analysis itself.

In order to avoid these problems, the hybrid systems community has tried to employ counter-example guided abstraction refinement techniques in the hybrid systems context [4, 5]. However, unlike in the discrete case, this has had only limited success in the hybrid case, since here the removal of one single counter-example at a time often already very early blows up the size of the abstraction.

Our previous approach [1] employs local reachability checking techniques on the hybrid system abstraction. This can be simulated in the method described in this paper by an extremely aggressive widening strategies which does not behave well for hybrid systems with cyclic behavior.

Computational experiments show the efficiency of the approach.

2. 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 discrete modes of a hybrid system, where S is finite and nonempty. $I_1, \ldots, I_k \subseteq \mathbb{R}$ are compact intervals over which the continuous variables

of a hybrid system range. Φ denotes the state space of a hybrid system, i.e., $\Phi = S \times I_1 \times \cdots \times I_k$.

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

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

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

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

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

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

In practice one would also have to define some concrete syntax in which hybrid systems are described. However, this paper will be independent of concrete syntax. Instead, we will later require some operations that will provide information on the hybrid system at hand.

3. Incremental Abstract Forward/Backward Computation

The main shortcoming of the usual hybrid systems reachability algorithms is its lack of incrementality which is an especially pressing problem for systems with complex dynamics, because in that case even bounded time reach set computation necessarily involves over-approximation. In such cases we would like to first compute approximate information using high over-approximation, and incrementally refine this.

Our approach will be based on an incremental refinement of a covering of the hybrid systems state space by connected sets that we will call *regions*. In our case, the regions will be formed by pairs consisting of a mode and a Cartesian product of intervals (i.e., a *box*). Moreover, we will form the regions in such a way that no pair of regions with the same mode will have overlapping boxes. In theory the approach is also applicable to regions that have a different form.

The operations that we require on regions are the following:

- $\uplus$ s.t. $a_1 \cup a_2 \subseteq a_1 \uplus_b a_2$
- $\sqsubseteq$ s.t. $a_1 \sqsubseteq a_2$ implies $a_1 \subseteq a_2$

In our case of boxes, $a_1 \uplus a_2$ is the smallest boxes that includes both argument boxes a_1 and a_2 (i.e., box union), and $\sqsubseteq$ is the subset operation on boxes.

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

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

A given abstraction $\mathcal{A}$ represents the set of trajectories that start in abstract states marked as `Init`, end in abstract states marked as `Unsafe`, never leave the abstraction, and move from one abstract state to the next only if there is a corresponding abstract transition. We denote this set by $[\![\mathcal{A}]\!]$.

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

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

Now we will come up with an algorithm that will incrementally improve an abstraction by making it tighter.

Note that, in particular, $\mathcal{A}$ is tighter than $\mathcal{A}$ itself, but in practice we will try to remove as many trajectories from the abstraction as possible.

Given abstract states a and a', we will assume a procedure $InitReach(a)$ that computes an over-approximation of the set of points in a that are reachable from an initial point in a, and a procedure $Reach(a, a')$ that computes an over-approximation of the set of points in a' reachable from a according to the system dynamics. In our case, we implemented both procedures based on interval constraint propagation [1, 7]. We assume that smaller inputs improve the precision of these operations, that is:

- $a_1 \subseteq a_2$ implies $InitReach(a_1) \subseteq InitReach(a_2)$
- $a_1 \subseteq a_2$ and $a'_1 \subseteq a'_2$ implies $Reach(a_1, a'_1) \subseteq Reach(a_2, a'_2)$

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

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

Now, the following algorithm (which we will call *pruning algorithm*) computes a tighter abstraction for a given abstraction $\mathcal{A}$.

$\mathcal{A}^* \leftarrow$ copy of $\mathcal{A}$
in $\mathcal{A}^*$: set all regions to $\emptyset$, delete initial labels and edges
// from now on, for every abstract state a of $\mathcal{A}$,
// we denote by a^* the corresponding abstract state of $\mathcal{A}^*$
for all $a \in A$, a is initial
  $a^* \leftarrow InitReach(a)$
  if $a^* \neq \emptyset$ **then**
    mark a^* as initial
let $update(a_1, a_2) =$ // defines a function update
  if $a_1^* \not\rightarrow a_2^*$ and $Reach(a_1^*, a_2) \neq \emptyset$ **then**
    introduce an edge $a_1^* \rightarrow a_2^*$
  if $Reach(a_1^*, a_2) \not\subseteq a_2^*$ **then**
    $a_2^* \leftarrow a_2^* \uplus Reach(a_1^*, a_2)$
    return true
  else
    return false in
while $\exists(a_1, a_2)$ such that $a_1 \rightarrow a_2$, $update(a_1, a_2)$
return $\mathcal{A}^*$

Unlike approaches based on counter-example guided abstraction refinement, the pruning algorithm does *not* increase the size (i.e., the number of nodes) of the abstraction. Still it deduces some some interesting information:

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

Proof: We have to prove two items:

- $[\![\mathcal{A}^*]\!] \subseteq [\![\mathcal{A}]\!]$: This follows from the following:
  - the set of initial/unsafe marks of $\mathcal{A}^*$ is a subset of the set of marks of $\mathcal{A}$
  - the set of edges of $\mathcal{A}^*$ is a subset of the set of edges of $\mathcal{A}$
  - the abstract states of $\mathcal{A}^*$ are subsets of the corresponding abstract states of $\mathcal{A}$ since $InitReach(a) \subseteq a$, and $Reach(a^*, a) \subseteq a$.
- $[\![\mathcal{A}^*]\!] \supseteq [\![\mathcal{A}]\!] \cap \mathcal{E}$: Let T be an error trajectory in $[\![\mathcal{A}]\!] \cap \mathcal{E}$. We prove that T is an element of $[\![\mathcal{A}^*]\!]$. Let $a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n$ be the abstract error trajectory corresponding to T in $\mathcal{A}$. We prove that the corresponding abstract trajectory $a_1^* \rightarrow a_2^* \rightarrow \cdots \rightarrow a_n^*$ in $\mathcal{A}^*$ is an abstract error trajectory containing T.
  - a_1^* is initial in $\mathcal{A}^*$ and contains the initial point of T
  - We assume that $a_1^* \rightarrow a_2^* \rightarrow \cdots \rightarrow a_i^*$, with $i < n$ forms an abstract trajectory containing T in $\mathcal{A}^*$, and prove that also $a_1^* \rightarrow a_2^* \rightarrow \cdots \rightarrow a_i^* \rightarrow a_{i+1}^*$ forms an abstract trajectory containing T in $\mathcal{A}^*$.

    To prove that $a_i^* \rightarrow a_{i+1}^*$ in $\mathcal{A}^*$ we observe that T leads from a_i^* to a_{i+1}. Hence $Reach(a_i^*, a_{i+1})$ is non-empty and the abstract transition $a_i^* \rightarrow a_{i+1}^*$ exists. Moreover, $Reach(a_i^*, a_{i+1})$ contains all points of T in a_{i+1}, and since the while loop terminated, $Reach(a_i^*, a_{i+1}) \subseteq a_{i+1}^*$ and hence a_{i+1}^* also contains these points.

■

Note however, that it is a-priori not clear, that the pruning algorithm terminates. In our case, we ensure this by simply doing all computation on *finite* set of floating point numbers (cf. the notion of "widening"). Hence

there are only finitely many possibilities of changing boxes with $\uplus$, until a fixpoint is reached.

Moreover, by using an implementation of $Reach$ that is idempotent we also can avoid stuttering (i.e., many small improvements by close floating point numbers), in most cases.

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

We do this by the *Split* operation that chooses an abstract state and splits it into two, copying all the involved edges and introducing edges between the two new states. All the labels and abstract transitions to other abstract states are copied as well. Moreover, two new abstract transitions that connect the original abstract state with its copy are added. The region assigned to the abstract state is equally split among two abstract states. To do this we pick a splitting dimension of the box assigned to the region and we split the box into halves using this dimension. For picking the splitting dimension, a round-robin strategy has proved to be the useful heuristics [1]. Such an refinement decreases the amount of over-approximation in subsequent calls to the pruning algorithm due to the properties of the $Reach$ and $InitReach$.

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

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

Since neither pruning nor splitting removes an error trajectory, the absence of an abstract error path at the termination of the while loop implies the absence of an error trajectory of the original system. This implies the correctness of the algorithm.

4. Improvements

4.1. Avoided Redundant Edge Checks

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

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

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

$\mathcal{A}^* \leftarrow$ copy of $\mathcal{A}$
in $\mathcal{A}^*$: set all regions to $\emptyset$, delete initial labels and edges
// from now on, for every abstract state a of $\mathcal{A}$,
// we denote by a^* the corresponding abstract state of $\mathcal{A}^*$
for all $a \in A$, a is initial
  $a^* \leftarrow InitReach(a)$
  if $a^* \neq \emptyset$ then
    mark a^* as initial
    propChange(a^*)
let $update(a_1, a_2) =$
  if $a_1^* \rightarrow_{\subseteq} a_2^*$ **then return false**
  introduce an edge $a_1^* \rightarrow_{\subseteq} a_2^*$
  if $a_1^* \not\rightarrow a_2^*$ and $Reach(a_1^*, a_2) \neq \emptyset$ **then**
    introduce an edge $a_1^* \rightarrow a_2^*$
  if $Reach(a_1^*, a_2) \not\subseteq a_2^*$ **then**
    $a_2^* \leftarrow a_2^* \uplus Reach(a_1^*, a_2)$
    propChange(a_2^*)
    return true
  else
    return false in
while $\exists (a_1, a_2)$ such that $a_1 \rightarrow a_2$, $update(a_1, a_2)$
return $\mathcal{A}^*$

Theorem 2 *Independent of the consistency edges of the input $\mathcal{A}$, the improved pruning algorithm computes the same result as the original one.*

Proof: Clearly, at the beginnning of the while loop, in both algorithms, $\mathcal{A}^*$ is the same. We prove that every time the termination condition of the while loop is tes-

ted, the consistency invariant holds, and hence the algorithm produce the same result.

The first time, the termination condition of the while loop is tested, the consistency invariant holds due to the following reasoning: Let a_1^*, a_2^* be such that $a_1^* \rightarrow_{\subseteq} a_2^*$, then $a_1^* = \emptyset$, since otherwise the operation update(a_1^*) would have deleted the consistency edge. Hence $Reach(a_1^*, a_2) = Reach(\emptyset, a_2) \subseteq a_2^*$.

■

4.2. Incremental Refinement of Abstraction

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

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

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

5. Conclusion

In this paper, we have introduced a new approach to unbounded safety verification of hybrid systems with non-linear ordinary differential equations. Currently we are doing detailed computational experiments comparing the algorithm with alternatives and studying various heuristics and implementation choices.

References

[1] S. Ratschan and Z. She, "Safety Verification of Hybrid Systems by Constraint Propagation Based Abstraction Refinement", ACM TECS 2007.

[2] T. Dzetkulič and S. Ratschan, "How to Capture Hybrid Systems Evolution Into Slices of Parallel Hyperplanes", 3rd IFAC Conference on Analysis and Design of Hybrid Systems 2009.

[3] S. Ratschan and T. Dzetkulič, "Verification of Hybrid Systems by Incremental Abstract Forward/Backward Computation" to appear on International Workshop on Reachability Problems 2010.

[4] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine,O. Stursberg, and M. Theobald, "Abstraction and Counterexample-Guided Refinement" in Model Checking of Hybrid Systems Int. Journal of Foundations of Comp. Science, 2003.

[5] R. Alur, T. Dang, and F. Ivančić, "Predicate abstraction for reachability analysis of hybrid systems", ACM TECS 2006.

[6] G. Frehse, B.H. Krogh, and R.A. Rutenbar, "Verifying Analog Oscillator Circuits Using Forward/Backward Abstraction Refinement" DATE 2006: Design, Automation and Test in Europe 2006.

[7] S. Ratschan and Z. She, "Constraints for Continuous Reachability in the Verification of Hybrid Systems", Proc. 8th Int. Conf. on Artif. Intell. and Symb. Comp., 2006.

[8] T.A. Henzinger, "Hybrid Automata with Finite Bisimulations", Proceedings of the 22nd International Colloquium on Automata, Languages, and Programming (ICALP) 1995.