An Algorithm for Formal Safety Verification of Complex Heterogeneous Systems

Stefan Ratschan
Institute of Computer Science, Czech Academy of Science, stefan.ratschan@cs.cas.cz

Abstract. Modern technical systems are heterogeneous in the sense that they tightly integrate computational elements into physical surroundings. Computational elements usually require discrete, and physical systems continuous modeling. In this paper, we present an modeling formalism and safety verification algorithm for such heterogeneous systems.

1. Introduction

Modern technical systems more and more consist of a tight integration of computational devices into physical surroundings. For example, in modern cars, a large part of the development cost goes into software and digital electronics. Moreover, the complexity of such systems is growing rapidly. Hence it is of utmost importance to come up with formalisms for modeling, and algorithms for analyzing such systems.

The notion of a hybrid dynamical system is a current approach for modeling computation in physical surroundings (Lunze and Lamnabhi-Lagarrigue, 2009). Such systems integrate ordinary differential equations with finite state machines, based on a state space that is the Cartesian product of a subset of $\mathbb{R}^n$ and a set of finitely many states. Uncertainty is usually included by also allowing differential inequalities, or by allowing uncertain parameters in the differential equations. However, finite state machines do not suffice for modeling software of the complexity occurring in modern technical systems.

In our work, we will present an extension of the hybrid system model to systems that are parametric in k data types, with k an arbitrary, but fixed, positive integer. Those data types are generic in the sense that they can be chosen arbitrarily as long as they fulfill certain conditions that are met by the most widely-used data types such as integers, arrays, and lists. The state space of the new model is formed by the Cartesian product of a subset of $\mathbb{R}^n$ and the used data types. Again, the dynamics of the continuous part of the states space is given by ordinary differential equations (or inequalities).

Moreover, we provide an algorithm for the formal safety verification of such systems (i.e., the automatic verification that the system state always stays in a certain set of states considered to be safe) based on certain operations that the basic data types are required to provide. The algorithm is an extension of our earlier algorithm for hybrid systems verification (Dzetkulič and Ratschan, 2011).

This work was supported by MŠMT project number OC10048 and long-term financing of the Institute of Computer Science (RVO 67985807).

2. Problem Definition

Let us assume k (not necessarily distinct) data types $D_1, \ldots, D_k$. For each of those data types we assume certain functions and relations. For example for a certain $i \in \{1, \ldots, k\}$, D_i might be the set of integers with the functions addition and multiplication, and relations $=$ and $\leq$. Another examples is the set of lists of integers with the operations `nil` (for describing the empty list), `first` (for taking the first element of a list), `rest` (for taking the result of removing the first element of the list), and `cons` (for constructing a new list from an integer, and an old list). In the case of classical hybrid systems, there is just one data type, consisting of finitely many (but a potentially huge number) of so-called *modes*.

For modeling the physical surroundings we use the n-dimensional real space $\mathbb{R}^n$, with functions such as addition and multiplication, and relations such as equality $=$ and inequality $\leq$.

Now we assume a language $\mathcal{L}$ of constraints whose semantics is built on top of the semantics of the functions and relations of the data types $D_1, \ldots, D_k$, and of the real numbers. For example, having just one data type, the integers, $\mathcal{L}$ might consist of conjunctions of linear equalities and inequalities. Having both integers and lists over integers, we might allow expressions such as

$$x' = x + 1 \wedge l' = \mathtt{cons}(x', l).$$

For us, the specific form of the language will not be important, but it will be essential to have certain constraint solving algorithms on them. We will introduce the specification of those algorithms in Section 3 and describe concrete possibilities for implementing them in Section 4.

Now, in order to describe the behavior of software within physical systems, we will introduce dynamical systems over those data types. The state space Φ will be given by $D_1 \times \ldots \times D_k \times C$, where $C \subseteq \mathbb{R}^n$.

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

Informally speaking, the set `Init` specifies the initial states of a system and `Unsafe` the set of unsafe states that should not be reachable from an initial state. The relation `Flow` specifies the possible continuous behavior of the system by relating states with corresponding derivatives, and `Jump` specifies the possible discontinuous behavior by relating each state to a successor state.

We can describe those sets using the language $\mathcal{L}$. For example, using the constraint above to describe the set `Jump`—assuming that unprimed variables denote the current state and primed variables the successor state—will result in a system that creates a list of successive integers.

Another example, is a system with state space $\{\mathtt{on}, \mathtt{off}\} \times \mathbb{R}^2$, where the set `Flow` could be described by a constraint of the form

$$[\text{mode} = \mathtt{on} \wedge \dot{x} = x + y \wedge \dot{y} = x - y] \vee [\text{mode} = \mathtt{off} \wedge \dot{x} = x + y \wedge \dot{y} = x - 2y].$$

Here we view mode as a variable ranging over $\{\mathtt{on}, \mathtt{off}\}$, and $x, \dot{x}, y, \dot{y}$ as variables ranging over $\mathbb{R}$. Note that the dot in $\dot{x}$ is just used as a way of defining a new variable distinct from x—it is not yet connected to any form of derivation. It will be connected to derivation only now, in the following definition of system behavior:

DEFINITION 2. *For a certain discrete state $s \in D_1 \times \ldots \times D_k$, a* flow *of length $l \geq 0$ in s is a function $r : [0, l] \to \Phi$ such that the projection of r to its continuous part $C \subseteq \mathbb{R}^n$ is differentiable and for all $t \in [0, l]$, the projection of r to its discrete part $D_1 \times \ldots \times D_k$ is s. A* trajectory *of a system 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 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 system H. We will denote the set of its error trajectories by $\mathcal{E}$. In this paper, we study the problem of safety verification. This means that we want to check whether a given system has an error trajectory, that is, whether the set $\mathcal{E}$ is empty.

3. Safety Verification

One method for safety verification (that is used in so-called ”bounded model checking”(Biere et al., 2003; Fränzle et al., 2007)) in the discrete time case is, to take the set of initial states, and compute the set of states reachable in one step, two steps, etc. and to check whether the result intersects the set of unsafe states. This has the drawback that it verifies safety of a given system over a bounded number of steps (at least for infinite state systems, and without additional techniques). Since the number of steps realistic systems can take is often huge, it is often more useful to design methods that check safety over unbounded time.

The straight-forward approach to verify safety over an unbounded number of steps is, to check whether the union of reachable states for subsequent time steps reaches a fix-point (in other words, further times steps do not result in further reachable states). For finite-state systems this is the main topic of the field of unbounded model checking (Clarke et al., 1999). For infinite state (but discrete time) systems in the form of computer programs, this is studied by abstract interpretation (Nielson et al., 1999). Such approaches make it necessary to first choose a representation for sets of system states for which a fix-point check can be easily done, and then to over-approximate the reachable states of the system using that representation.

This technique is also the basis of the first tools for safety verification of hybrid systems (Henzinger et al., 1997). However, for systems with non-trivial continuous evolution, this strategy has one severe drawback: For hybrid systems with non-trivial continuous dynamics even bounded time reach set computation necessarily involves over-approximation. A-priori it is not clear how precisely the reachable sets have to be computed to prove a given safety property. Hence, it may be advantageous to first compute approximate information using loose over-approximation, and then incrementally refine this.

Such techniques are popular in finite state and program verification under the name of ”counter-example guided abstraction refinement” (CEGAR) (Clarke et al., 2003b) that has also been tried for hybrid systems (Alur et al., 2006; Clarke et al., 2003a). However, for systems with a partially continuous state space, this easily results in a behavior where the computed approximate information radically grows in size without representing enough information necessary for proving the safety property at hand.

In some earlier work (Dzetkulič and Ratschan, 2011), we presented a technique for avoiding this behavior for hybrid systems, and here we extend the technique to complex heterogeneous systems of the type described above.

Our approach is be based on an incremental refinement of a covering of the systems state space Φ by connected sets that we will call *regions*. We will form the regions in such a way that no two regions will overlap (i.e., regions are allowed to intersect, but only on their boundaries of the continuous part of the state space). The method is independent of the class of regions used. For example, in the special case of hybrid systems with a state space $M \times \mathbb{R}^n$, where M is finite, the regions can be formed by pairs consisting of an element of M and a Cartesian product of closed intervals (i.e., a *box*). But other classes of regions (e.g., based on polyhedra) are equally conceivable.

DEFINITION 3. *An* abstraction *is a graph whose vertices (which we will also call* abstract states*) are formed by regions that may be labeled with labels* `Init` *or* `Unsafe`*. We call the edges of an abstraction* abstract transitions.

This is the basic form of abstraction. However, an abstraction might be extended with much more information about the concrete system. For example, in our instantiation of this approach to the hybrid systems case (Dzetkulič and Ratschan, 2011), we store additional information on where trajectories might leave the regions.

For example, for a state space $\Phi = \mathbb{R}^2$, and the regions delimited by the black lines in Figure 1, a

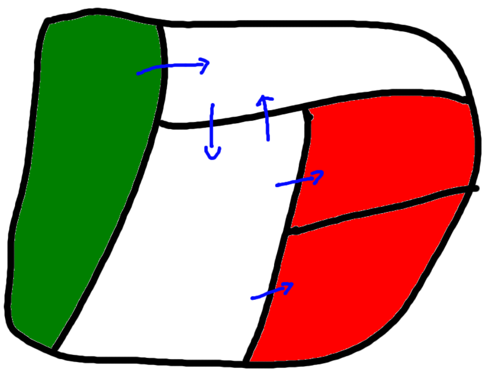

Figure 1. Abstraction Example

corresponding abstraction might be the graph consisting of those regions as nodes, the vertices given by the arrows (colored in blue), and with the region on the left-hand side (colored in green) marked as initial, and two regions on the right-hand side (colored in red) marked as unsafe. Such an abstraction represents the set of all trajectories that start in a region marked as initial (i.e., in a region colored in green in the figure), follows the edges of the graph (i.e.., the blue arrows in the figure), and ends in a region marked as unsafe (i.e., in a region colored in green in the figure). We will formalize this now.

We call a sequence of abstract states $a_1, \ldots, a_l$ an *abstract trajectory*. If all abstract states and all transitions between successive abstract states in an abstract trajectory belong to an abstraction $\mathcal{A}$, we call it an

$\mathcal{A}$-*abstract trajectory* and we denote it by $a_1 \to \ldots \to a_n$. An ($\mathcal{A}$-)abstract trajectory represents the set of concrete trajectories that begin in the region of a_1, move from one abstract state region to the next only if there is a corresponding concrete transition, and end in the region of a_n. We denote this set by $[\![a_1, \ldots, a_n]\!]$ for a given abstract trajectory or $[\![a_1 \to \ldots \to a_n]\!]$ for some $\mathcal{A}$-abstract trajectory.

This can be formalized as follows: We define a *splitting* of a flow l to be a sequence of flows $s_1, \ldots, s_r$ such that for all $i \in \{1, \ldots, r\}$, for all $t \in [0, \mathit{length}(s_i)]$, $s_i(t) = l(\sum_{j\in\{1,\ldots,i-1\}} t + \mathit{length}(s_j))$. A *trajectory splitting* is a concatenation of splittings of its individual contained flows. $[\![a_1, \ldots, a_n]\!]$ then is the set of all concrete trajectories $r_1, \ldots, r_p$ that have a trajectory splitting $q_1, \ldots q_n$, such that for all $i \in \{1, \ldots, n\}$, for all $t \in [0, \mathit{length}(q_i)]$, $q_i(t) \in a_i$.

An $\mathcal{A}$-*abstract error trajectory* is an $\mathcal{A}$-abstract trajectory $a_1 \to \ldots \to a_n$ such that in $\mathcal{A}$, a_1 is labelled initial, and a_n is labelled unsafe.

An abstraction $\mathcal{A}$ represents the set of all concrete trajectories $[\![a_1 \to \ldots \to a_n]\!]$ for abstract error trajectories $a_1 \to \ldots \to a_n$ in the abstraction $\mathcal{A}$. We denote this set by $[\![\mathcal{A}]\!]$.

The intuition is that, during abstraction refinement, the abstraction stays an over-approximation of the set of error trajectories $\mathcal{E}$ of a given system. We say that an abstraction $\mathcal{A}^*$ is a *refinement* of 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 that are present in $\mathcal{A}$, 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 refining it, *without* increasing the number of abstract states in the abstraction. Note that, in particular, $\mathcal{A}$ is a refinement of $\mathcal{A}$ 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 $\mathit{Init}(a)$ that computes an over-approximation of the set of points in a that are initial (i.e., an element of `Init`), and a procedure $\mathit{Reach}(a, a')$ that computes an over-approximation of the set of points in a' reachable from a according to the system dynamics (here we do not assume any time bound, implementations of those procedures that compute reachability over bounded time would only require slight modifications of our algorithms). Our method is independent of the concrete technique used to compute those procedures. Still, in Section 4 we will discuss in detail how this can be implemented in practice. We assume that smaller inputs improve the precision of these operations, that is:

– $a_1 \subseteq a_2$ implies $\mathit{Init}(a_1) \subseteq \mathit{Init}(a_2)$

– $a_1 \subseteq a_2$ and $a'_1 \subseteq a'_2$ implies $\mathit{Reach}(a_1, a'_1) \subseteq \mathit{Reach}(a_2, a'_2)$

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

– $a = \emptyset$ implies $\mathit{Init}(a) = \emptyset$

– $a = \emptyset$ implies $\mathit{Reach}(a, a') = \emptyset$

– $a' = \emptyset$ implies $\mathit{Reach}(a, a') = \emptyset$

In the following, we require the existence of operations $\sqsubseteq$ and $\uplus$ on regions, with the following properties.

– $\sqsubseteq$ such that if $a^* \sqsubseteq a$, then for all $n \in \mathbb{N}$, for all $i \in \{1 \ldots n\}$ and for all regions $b_1 \ldots b_{i-1}, b_{i+1} \ldots b_n$ we have that $[\![b_1, \ldots, b_{i-1}, a^*, b_{i+1}, \ldots, b_n]\!] \subseteq [\![b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n]\!]$ i.e., less concrete trajectories follow a given abstract trajectory after replacing an abstract state by smaller one wrt. $\sqsubseteq$ operation.

– $\uplus$ s.t. for all regions $a_1, a_2, b : a_1 \sqsubseteq b \wedge a_2 \sqsubseteq b$ implies $a_1 \uplus_b a_2 \sqsubseteq b$, $a_1 \sqsubseteq a_1 \uplus_b a_2$ and $a_2 \sqsubseteq a_1 \uplus_b a_2$.

Since in our case abstract states represent sets, this can be ensured by the following:

– $\uplus$ s.t. for all $a_1, a_2 \subseteq b$: $a_1 \cup a_2 \subseteq a_1 \uplus_b a_2$ and $a_1 \uplus_b a_2 \subseteq b$

– $\sqsubseteq$ s.t. $a_1 \sqsubseteq a_2$ iff $a_1 \subseteq a_2$

This is our natural interpretation of $\uplus$ and $\sqsubseteq$. However, different choices are possible, as long as they fulfill the above properties: For certain representations of regions it might be convenient to use a weaker form of $\sqsubseteq$ efficiency reasons. Also, when encoding more information into abstract states (Dzetkulič and Ratschan, 2011), different interpretations of those symbols are often convenient.

In the instantiation of the method with boxes, $a_1 \uplus_b a_2$ is the smallest box that includes both argument boxes a_1 and a_2, but does not exceed b (i.e., box union intersected with bounding box), and $\sqsubseteq$ is the subset operation on boxes. Note that for $a_1, a_2 \subseteq b$ defining $a_1 \sqsubseteq a_2$ iff $a_1 \uplus_b a_2 = a_2$ fulfills the above property.

The following algorithm (which we will call *pruning algorithm*) computes a refinement of a given abstraction $\mathcal{A}$. The intuition is to remove parts from the regions forming the abstraction for which we can prove that they cannot lie on an error trajectory.

```
$\mathcal{A}^* \leftarrow$ copy of $\mathcal{A}$, all regions set to $\emptyset$, no initial labels, no 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** $a \in \mathcal{A}$, $a$ is initial
    $a^* \leftarrow Init(a)$
    **if** $a^* \neq \emptyset$ **then**
        mark $a^*$ as initial
**while** there is a pair of abstract states $(a_1, a_2)$ in $\mathcal{A}$ with
    $a_1 \to a_2$, s.t. $Reach(a_1^*, a_2) \not\sqsubseteq a_2^*$ or $(a_1^* \not\to a_2^*$ and $Reach(a_1^*, a_2) \neq \emptyset)$ **do**
        **if** $a_1^* \not\to a_2^*$ in $\mathcal{A}^*$ **then** introduce an edge $a_1^* \to a_2^*$ into $\mathcal{A}^*$
        **if** $Reach(a_1^*, a_2) \not\sqsubseteq a_2^*$ **then** $a_2^* \leftarrow (a_2^* \uplus_{a_2} Reach(a_1^*, a_2))$
**return** $\mathcal{A}^*$
```

Algorithms of such a type are known in the literature under them name ”chaotic iteration” or ”worklist algorithms” (Cousot and Cousot, 1977; Bourdoncle, 1993; Nielson et al., 1999; Apt, 1999; Apt, 2000).

Like similar algorithms in abstract interpretation, this algorithm computes unbounded reachability based on a fixpoint argument. However, unlike those algorithms, it exploits and refines the knowledge already available in the abstraction $\mathcal{A}$. In contrast to CEGAR approaches, the algorithm does *not* increase the size (i.e., the number of nodes) of the abstraction. Still it deduces some interesting information:

THEOREM 1. *The result of the pruning algorithm is a refinement of the input abstraction $\mathcal{A}$.*

Since the algorithm uses knowledge about the given system only through the operations $Init$ and $Reach$, the correctness proof for the hybrid systems case (Dzetkulič and Ratschan, 2011) also applies here. For a similar approach in a completely discrete context see the notion of abstraction slicing (Brückner et al., 2008).

Note, that it is a-priori not clear, that the pruning algorithm terminates. However, termination can be ensured, for example, by using a representation for which, for given regions a and b, there is not infinite chain $a \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots \sqsubseteq b$.

In the case where the discrete part of the state space is finite, and the regions describing the continuous part are formed by boxes, this holds if the endpoints of the corresponding intervals are formed by floating-point numbers, and the continuous state space is compact. Strategies for ensuring termination of such fixpoint computations are widely studied in the abstract interpretation community under the term ”widening”.

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).

A simple method for doing this is a *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. Such a refinement decreases the amount of over-approximation in subsequent calls to the pruning algorithm due to the properties of $Reach$ and $Init$. For example, Henzinger et. al. (Henzinger et al., 1998) use such a splitting step to reduce the over-approximation of the continuous system dynamics by differential inclusions of the form $\dot{x} \in A$, where A is a polyhedron. It is possible to use much more sophisticated splitting strategies, for example, a splitting step that removes as specific abstract error trajectory (i.e., one CEGAR step) (Alur et al., 2006; Clarke et al., 2003a) instead.

It is clear that the pruning algorithm can also be applied backward in time (i.e., removing parts of the abstraction not leading to an unsafe state) (Henzinger and Ho, 1995; Frehse et al., 2006). We will denote the resulting algorithm by $Prune^{-}(\mathcal{A})$.

Now we have to following overall algorithm for computing increasingly fine abstractions:

```
initialize $\mathcal{A}$ with an arbitrary abstraction such that
    $[\![A]\!]$ contains all error trajectories of the input system
**while** there is an $\mathcal{A}$-abstract error trajectory
    $\mathcal{A} \leftarrow Prune(\mathcal{A})$
    $\mathcal{A} \leftarrow Prune^{-}(\mathcal{A})$
    $\mathcal{A} \leftarrow Split(\mathcal{A})$
**return** ”safe”
```

The most simple way to initialize the abstraction $\mathcal{A}$ in this algorithm is to use the trivial abstraction containing just one vertex for every mode marked as `Init` and `Unsafe`, containing a transition to all other vertices and itself, and a region containing the whole state space of the input system.

Since neither pruning nor splitting removes an error trajectory, the absence of an $\mathcal{A}$-abstract error trajectory at the termination of the while loop implies the absence of an error trajectory of the original system. Hence, in such a case, the algorithm correctly returns the information that the input system was safe. In cases where the input does have an error trajectory, this algorithm does not terminate. However, in such cases, the algorithm maintains an abstraction that, at any time, can be used by other algorithms (Ratschan and Smaus, 2009) for searching for this error trajectory.

Note that forward pruning may enable further backward pruning and vice versa, hence the algorithm may be extended in such a way that forward and backward pruning are done in a loop until no further improvement occurs. If either forward or backward pruning is dropped from the algorithm, it will incrementally compute a tighter and tighter over-approximation of the (forward/backward) reach set.

The improvements to this algorithm introduced in an earlier paper (Dzetkulič and Ratschan, 2011) for the special case of classical hybrid systems can all easily be adapted to the more general case discussed in this paper.

4. Computation of Reachability Information

For applying the techniques in the previous sections to a concrete system one needs to

– choose a class of regions that will be used for representing subsets of the state space,

– instantiate the operations $\sqsubseteq$ and $\uplus$ with concrete algorithms, and

– provide algorithms for computing the reachability operations Reach and Init.

Here we make the observation that for this, techniques from computational logic can be used. First we assume that (as in all examples above) the language $\mathcal{L}$ is based on first-order predicate logic. Moreover we assume that also the regions of the abstraction are formed by predicate logical formulas representing the set of all values that satisfy a given formula. For example, formulas of the form $\bigwedge_{i\in\{1,\dots,n\}} \underline{a}_i \leq x_i \wedge x_i \leq \overline{a}_i$ represent hyper-rectangles (boxes). A concrete implementation may, of course, use a more optimized representation, but the usage of a formula representation in this section helps us to gain more insight into the nature of the problem.

Then, the region operations $\sqsubseteq$ and $\uplus$ can be implemented by (a sound approximation of) logical implication ($a \sqsubseteq b$ is such that $a \sqsubseteq b$ implies $a \Rightarrow b$) and (a conservative approximation of) disjunction ($a_1 \uplus_b a_2$ is such that $(a_1 \vee a_2) \wedge b$ implies $a_1 \uplus_b a_2$). For example, when using boxes, $a \uplus b$ might be the box hull (i.e., the smallest box containing both arguments) which clearly fulfills the above requirement.

Now we turn to the reachability operations Reach and Init. We will write them as first-order predicate logical formulas. The operation $\mathit{Init}(a)$ must be such that the formula

$$a(x) \wedge \texttt{Init}(x)$$

implies $\mathit{Init}(a)(x)$.

For analyzing reachability we provide two separate logical formulas for reachability through jumps $\mathit{Reach}_J(a, a')$ and reachability through flows $\mathit{Reach}_F(a, a')$ which will result in reachability $\mathit{Reach}(a, a')$ being

$$\mathit{Reach}_J(a, a') \vee \mathit{Reach}_F(a, a').$$

The first part, $\mathit{Reach}_J(a, a')$ must be such that

$$\exists x \,.\, a(x) \wedge \texttt{Jump}(x, x') \wedge a'(x')$$

implies $Reach_J(a, a')(x')$, and $Reach_F(a, a')$ must be such that

$$\exists x \exists t \,.\, 0 \leq t \wedge t \leq c \wedge a(x) \wedge T_a(x, x', t) \wedge a'(x')$$

implies $Reach_F(a, a')(x')$, where $T_a(x, x', t)$ models the fact that there is a continuous flow from x to x' in a taking time t (we will later show how to model this as a logical formula), and c is an arbitrary positive real constant, or ∞, in which case the constraint $t \leq c$ can be dropped.

Now, one could just take the above formulas as the implementation of the operations themselves, in which case the implications above are implemented as equivalences. This is, in fact, the approach taken by bounded model checking of finite state systems, where there are techniques for representing and checking satisfiability of *huge* formulas in propositional logic using so-called SAT solvers (Biere et al., 2009). For discrete time systems with other data types/domains this can be extended to so-called satisfiability modulo theory (SMT) solvers (De Moura and Bjørner, 2011; Fränzle et al., 2007).

However, in the unbounded time case this has the disadvantage that when concatenating reachability over several steps (in our case, several applications of the $Reach(a, a')$ operator), more and more quantifiers accumulate, resulting in high-dimensional formulas, on which—in the unbounded time case—a fixpoint check (in our case based on $\sqsubseteq$) has to be done.

Another approach would be, to use the above formulas, but to eliminate the quantifiers in each application by quantifier elimination algorithms (Harrison, 2009), that is, algorithms that that compute equivalent, but quantifier-free formulas. In fact, this is precisely the approach taken in the finite state/propositional case, where practically efficient algorithms based on binary decision diagrams (BDDs) (Drechsler and Becker, 1998) exist. Also, early algorithms for hybrid systems verification took this approach (for very simple continuous dynamics). However, as soon as we leave the purely propositional case, even for quite simple individual theories, quantifier elimination is often not possible (e.g., non-linear integer arithmetic), or highly complex (e.g., linear integer arithmetic).

Hence, it makes sense, to use some form of over-approximation here, not implementing the above implications as equivalences any more. In program verification, the design of such regions with corresponding algorithms is the subject of *abstract domain design* (Filé et al., 1996), and—when using techniques from logic—*logical interpretation* (Tiwari and Gulwani, 2007; Gulwani and Tiwari, 2006). However, those abstract domains cannot directly be applied to systems with continuous time dynamics.

In our instantiation of the method for hybrid systems (Dzetkulič and Ratschan, 2011) we use interval constraint propagation (Benhamou and Granvilliers, 2006) (we also have a generalization of this technique available (Ratschan, 2006)). We also have investigated an alternative method based on an over-approximation of Fourier-Motzkin elimination (Dzetkulič and Ratschan, 2009).

We will now analyze how to handle continuous dynamics in this context. Above we used the expression $T_a(x, x', t)$ to model the fact that there is a continuous flow from x to x' in a taking time t. This could be directly written down in second order predicate logic (i.e., where variable and quantifier are allowed to range over functions and such functions can model system trajectories), however this would bring in additional difficulties for algorithmic analysis.

But, even if we need second order logic to model continuous reachability, we can at least *approximate* continuous reachability in first order logic. Here one get rid of second order quantifiers as follows: For each $v \in \{1, \dots, n\}$, we can do a Taylor expansion at x of the projection of the trajectory to its v-th variable.

From this we get

$$\exists y, d\ .\ a(y) \wedge \mathtt{Flow}^{k+1}(x, x^{(1)}) \wedge \ldots \wedge \mathtt{Flow}(x, x^{(k)}) \wedge \mathtt{Flow}^{k+1}(y, d) \wedge x' = x + \ldots + \frac{x^{(k)}}{k!} t^k + \frac{d|_v}{(k+1)!} t^{k+1},$$

where the notation $d|_v$ denote the projection of d to its v-th variable, and $\mathtt{Flow}^i$ denotes a constraint that assigns to a point y its i-th derivative. If, as in the original Definition 1, only first-order derivatives are available, the formula can be applied only for $k = 0$ (i.e., the case corresponding to the mean-value theorem). The expression $T_a(x, x', t)$ can now be replaced by a conjunction of the above formula over all $v \in \{1, \ldots, n\}$. One can think of many variations of this approach, for example, by applying Taylor expansion backward in time.

5. Conclusion

In this paper we introduced a framework for formal safety verification of systems with both continuous and discrete dynamics, where the discrete part of the state space may include data structures such as lists and arrays. The framework includes an algorithm for safety verification of hybrid systems as an instantiation (Dzetkulič and Ratschan, 2011; Ratschan and She,). Computational experiments with that instantiation confirm the usefulness of the approach.

The remaining challenge is to instantiate the framework to cases with more interesting data structures, to design corresponding reachability analysis algorithms both for the discrete and continuous cases, and their combinations.

References

Alur, R., T. Dang, and F. Ivančić: 2006, 'Predicate abstraction for reachability analysis of hybrid systems'. *Trans. on Embedded Computing Sys.* **5**(1), 152–199.

Apt, K. R.: 1999, 'The Essence of Constraint Propagation'. *Theoretical Computer Science* **221**(1–2), 179–210.

Apt, K. R.: 2000, 'The Role of Commutativity in Constraint Propagation Algorithms'. *ACM Transactions on Programming Languages and Systems* **22**(6), 1002–1036.

Benhamou, F. and L. Granvilliers: 2006, 'Continuous and Interval Constraints'. In: F. Rossi, P. van Beek, and T. Walsh (eds.): *Handbook of Constraint Programming*. Amsterdam: Elsevier, Chapt. 16, pp. 571–603.

Biere, A., A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu: 2003, 'Bounded Model Checking'. *Advances in Computers* **58**, 117 – 148.

Biere, A., M. J. H. Heule, H. van Maaren, and T. Walsh (eds.): 2009, *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bourdoncle, F.: 1993, 'Efficient chaotic iteration strategies with widenings'. In: D. Bjørner, M. Broy, and I. Pottosin (eds.): *Formal Methods in Programming and Their Applications*, Vol. 735 of *LNCS*. Springer, pp. 128–141.

Brückner, I., K. Dräger, B. Finkbeiner, and H. Wehrheim: 2008, 'Slicing Abstractions'. *Fundamenta Informaticae* **89**(4), 369–392.

Clarke, E., A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald: 2003a, 'Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems'. *Int. J. of Foundations of Comp. Sc.* **14**(4), 583–604.

Clarke, E., O. Grumberg, S. Jha, Y. Lu, and H. Veith: 2003b, 'Counterexample-Guided Abstraction Refinement for Symbolic Model Checking'. *Journal of the ACM* **50**(5), 752–794.

Clarke, E. M., O. Grumberg, and D. A. Peled: 1999, *Model Checking*. MIT Press.

Cousot, P. and R. Cousot: 1977, 'Automatic synthesis of optimal invariant assertions: Mathematical foundations'. In: *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*. pp. 1–12.

De Moura, L. and N. Bjørner: 2011, 'Satisfiability modulo theories: introduction and applications'. *Commun. ACM* **54**, 69–77.

Drechsler, R. and B. Becker: 1998, *Binary Decision Diagrams*. Springer.

Dzetkulič, T. and S. Ratschan: 2009, 'How to Capture Hybrid Systems Evolution Into Slices of Parallel Hyperplanes'. In: *ADHS'09: 3rd IFAC Conference on Analysis and Design of Hybrid Systems*. pp. 274–279.

Dzetkulič, T. and S. Ratschan: 2011, 'Incremental Computation of Succinct Abstractions For Hybrid Systems'. In: *FORMATS 2011*, Vol. 6919 of *LNCS*. pp. 271–285, Springer, Heidelberg (2011).

Filé, G., R. Giacobazzi, and F. Ranzato: 1996, 'A Unifying View of Abstract Domain Design'. *ACM Comput. Surv.* **28**, 333–336.

Fränzle, M., C. Herde, S. Ratschan, T. Schubert, and T. Teige: 2007, 'Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure'. *JSAT—Journal on Satisfiability, Boolean Modeling and Computation, Special Issue on SAT/CP Integration* **1**, 209–236.

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

Gulwani, S. and A. Tiwari: 2006, 'Combining abstract interpreters'. In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA, pp. 376–386, ACM.

Harrison, J.: 2009, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

Henzinger, T. A. and P.-H. Ho: 1995, 'A note on abstract interpretation strategies for hybrid automata'. In: *Hybrid Systems II*, Vol. 999 of *LNCS*. pp. 252–264.

Henzinger, T. A., P.-H. Ho, and H. Wong-Toi: 1997, 'HYTECH: a model checker for hybrid systems'. *International Journal on Software Tools for Technology Transfer (STTT)* **1**, 110–122.

Henzinger, T. A., P.-H. Ho, and H. Wong-Toi: 1998, 'Algorithmic Analysis of Nonlinear Hybrid Systems'. *IEEE Transactions on Automatic Control* **43**, 540–554.

Lunze, J. and F. Lamnabhi-Lagarrigue (eds.): 2009, *Handbook of Hybrid Systems Control*. Cambridge University Press.

Nielson, F., H. R. Nielson, and C. Hankin: 1999, *Principles of Program Analysis*. Springer.

Ratschan, S.: 2006, 'Efficient Solving of Quantified Inequality Constraints over the Real Numbers'. *ACM Transactions on Computational Logic* **7**(4), 723–748.

Ratschan, S. and Z. She, 'HSOLVER'. `http://hsolver.sourceforge.net`. Software package.

Ratschan, S. and J.-G. Smaus: 2009, 'Finding Errors of Hybrid Systems by Optimising an Abstraction-Based Quality Estimate'. In: C. Dubois (ed.): *Tests and Proofs*, Vol. 5668 of *LNCS*. pp. 153–168, Springer.

Tiwari, A. and S. Gulwani: 2007, 'Logical Interpretation: Static Program Analysis Using Theorem Proving'. In: F. Pfenning (ed.): *Automated Deduction CADE-21*, Vol. 4603 of *LNCS*. Springer, pp. 147–166.