Design Principles for an Extendable Verification Tool for Hybrid Systems

Goran Frehse * **Rajarshi Ray** *

* *Université Joseph Fourier Grenoble 1 – Verimag, Centre Equation - 2, Avenue de Vignate, 38610 Gieres, France (e-mail: goran.frehse,rajarshi.ray@imag.fr)*

Abstract: The verification of continuous and hybrid systems is known to be hard, and today tools are limited to relatively small problems. Several novel approaches are currently under investigation that exploit various kinds of set representations (polyhedra, zonotopes), improved algorithms (avoiding the wrapping effect) and strategies (such as abstraction refinement). We outline a tool framework that is able to integrate and combine different elements from these approaches. The framework includes implementations for common functionality (hybrid automata, graphical output, basic set operations, etc.) and interfaces that allow us to plug in different implementations, such as a particular kind of set representation or a particular optimization algorithm. This allows us to experimentally evaluate competing ideas, combine promising elements and explore new approaches with relatively little development effort.

Keywords: Hybrid automata, support functions, reachability.

1. INTRODUCTION

The verification of continuous and hybrid systems is a challenging problem, and various approaches are currently being investigated to overcome the complexities of representing and computing with continuous sets of states. Since verification problems are generally undecidable for such systems, experimental results are vital for evaluating and developing new ideas. In this paper, we propose an architecture for such a tool platform designed to facilitate the implementation of algorithms related to reachability and safety verification.

While these methods are based on different representations (polyhedra, zonotopes) and are tailored to different dynamics (piecewise constant, affine, multi-affine, nonlinear), they have several things in common:

- The model is a composition of hybrid automata (including extensions such as hierarchy and templates).
- Basic components of analysis algorithms are post- and pre-operators in various flavors.
- The reachable states are explored using symbolic states.
- They require basic infrastucture such as parsing input and visualizing states.

Our framework provides the common components and developers can substitute components as well as easily add new functionality. We extensively make use of polymorphism to enable the development of heterogenous analysis methods, such as using different set representations in different parts of the state space or at different levels of refinement, or combining symbolic computations with simulation such as in (de Paula and Hu, 2007). In general, our framework allows all set operators (union, intersection, etc.) to return a set of a different type.

The development of the framework was spawned by recent progress in finding efficient data structures and algorithms for reachability computation, see (Asarin et al., 2006). Building on our experience with PHAVer (Frehse, 2008), our goal is to improve and extend a particular type of reachability computation, allowing to substitute different implementations for components where suitable. These restrictions enable us to tune the implementation to suit our specific application. This sets our approach apart from another open tool for hybrid systems, Ariadne (Balluchi et al., 2006), which is designed as an open-source library for computation with hybrid automata and provides scripting capabilities via a Python interface. Various elements of our tool architecture are inspired by Uppaal (Bengtsson et al., 1995), a powerful verification tool for timed automata, whose design has evolved continuously over more than a decade. We present two implementations, one modeled after PHAVer and one after the support function approach of (Girard and Le Guernic, 2008). The latter represents convex continuous sets using tangent hyperplanes of a fixed number and direction, which is similar in spirit to the *template polyhedra* in (Sankaranarayanan et al., 2008) although the computation proceeds differently.

In the following section, we present a basic rechability algorithm for hybrid automata. In Sect. 3, we take stock of several enhancements that our design should be able to integrate. In Sect. 4 we formulate design principles from which we develop our tool architecture. The implementations of two of the considered approaches are presented in Sect. 5 to demonstrate the suitability of that architecture. Some conclusions are drawn in Sect. 6.

⋆ This research was supported in part by the EU project MULTIFORM under grant INFSO-ICT-224249.

2. REACHABILITY OF HYBRID AUTOMATA

The interaction of discrete events and continuous, time-driven dynamics can be efficiently modeled by a so-called *hybrid automaton* Alur et al. (1995). A hybrid automaton $H = (Loc, Var, Lab, Inv, Flow, Trans, Init)$ consists of a graph in which each vertex $l \in Loc$, also called *location* or *mode*, is associated via $Flow(l)$ with a set of differential equations (or inclusions) that defines the time-driven evolution of the continuous variables. A *state* $s \in Loc \times \mathbb{R}^{Var}$ consists of a location and values for all the continuous variables Var. The edges of the graph, also called *discrete transitions* $Trans$, allow the system to jump between locations, thus changing the dynamics, and instantaneously modify the values of continuous variables according to a *jump relation* μ. The jumps may only take place when the values of the variables are within the domain of μ. The system may only remain in a location l as long as the variable values are in a range called *invariant* $Inv(l)$ associated with the location. All behavior originates from the set of *initial states* $Init$.

An *execution* of the automaton is a sequence of discrete jumps and pieces of continuous trajectories according to its dynamics, and originates in one of the initial states. A state is *reachable* if an execution leads to it. In this paper, we are concerned with computing the set of states that are reachable. A related problem is that of *safety*. Given a set of bad states, the system is *safe* if the bad states are not reachable.

2.1 Efficient Reachability using Symbolic States

For a set of states R, let the *discrete post-operator* $\mathrm{Post}_d(R)$ be the set of states reachable by a discrete transition from R, and the *continuous post-operator* $\mathrm{post}_c(R)$ be the set of states reachable from R by letting an arbitrary amount of time elapse. The set of reachable states is the fixed-point of the sequence $R_0 = Init$,

$$R_{k+1} := R_k \cup \mathrm{Post}_d(R_k) \cup \mathrm{post}_c(R_k). \qquad (1)$$

In order to implement the above fixed-point computation, we need to efficiently carry out union, difference, and emptiness tests on sets of states, avoiding redundant computations. A common way to do so is to represent sets of states as sets of *symbolic states*. A symbolic state $s = (D, C)$ represents the cross product of a set of discrete states $D \subseteq Loc$ and a set of continuous states $C \subseteq \mathbb{R}^{Var}$. E.g., D could be a single location and C a polyhedron. Let $\mathbb{S}$ be the set of symbolic states of a given hybrid automaton. The post-operators are extended to symbolic states: given a single symbolic state s, $\mathrm{post}_d(s)$ and $\mathrm{post}_c(s)$ both produce a set of symbolic states.

The sets of states encountered during the computation are represented with a *passed/waiting list* (PWL), see (David et al., 2002) for a detailed discussion. The passed list contains the symbolic states that have been encountered so far. The waiting list contains the symbolic states whose successors still have to be computed. It is implemented as a set of references to elements of the passed list. Formally, a PWL is a pair $(P, W) \subseteq 2^{\mathbb{S}} \times 2^{\mathbb{S}}$ with $W \subseteq P$. We define the following operations for the PWL:

- $(P, W) = init(I)$: Assign a set of symbolic states $I \subseteq S$ to P and W.

Procedure 1 Reachability using Symbolic States

```
(P, W) := init(post_c(Init))
while W ≠ ∅ do
   s := pop(P, W)
   for all s' ∈ post_d(s) do
      for all s'' ∈ post_c(s') do
         (P, W, S) := add(P, W, s'')
         (P, W) := compact(P, W, S)
      end for
   end for
end while
```

- $S = diff(s, s')$: Given symbolic states $s = (C, D)$ and $s' = (C', D')$, produces the set of symbolic states $s \setminus s' = \{(D \setminus D', C), (D \cap D', C \setminus C')\}$, or an overapproximation that is efficient to compute. Our default implementation for convex sets C, C' is

$$diff(s, s') = \begin{cases} \{(D \setminus D', C)\} & \text{if } C \subseteq C', \\ \{(D, C)\} & \text{otherwise.} \end{cases}$$

  If C or C' are nonconvex sets represented as a set of convex sets, we extend this operation pairwise. Let $diff(s, P)$ be the result of applying *diff* consecutively for all $s' \in P$.
- $(P', W', S) = add(P, W, s)$: Add $s' = diff(s, P)$ to P and W.
- $(P', W') = compact(P, W, S)$: Compact P and W by replacing all $s' \in P, W$ by $diff(s', S)$, eliminating symbolic states where D or C is empty.
- $(s, W') = pop(P, W)$: Select a symbolic state $s \in W$, remove in from W and return it for further processing (post computation).

This leads us to Procedure 1, which works as follows:

(1) Intialize the PWL with the time-post of the initial states.
(2) Pick a symbolic state from the PWL.
(3) Apply discrete-post (generating possibly more than one symbolic state).
(4) Apply continuous-post to every generated symbolic state.
(5) Throw away the symb. states (or parts of them) that are already on the passed list – this involves testing for inclusion and emptiness. Put the remaining ones onto the PWL.
(6) Compact the PWL by removing redundant states (this is not always the best reduction; one could also compact first and then add).
(7) If the waiting list is not empty, go to 2.

Note that reachability for hybrid automata is undecidable, and Proc. 1 is not guaranteed to terminate.

The order in which states are popped off the waiting list determines the order of computation (breath first/depth first). This may influence the speed of the computation and may have implications on the interpretation of results. E.g., if a forbidden state is encountered during breath-first exploration, it is the state with the shortest counter example. If overapproximations are used, the resulting set can differ according to which ordering is used, since over-approximation and post operators might not commute.

3. ENHANCING RECHABILITY COMPUTATIONS

Our goal is to enable the implementation of a number of different approaches to computing the set of reachable states using Proc. 1, as well as enabling their eventual combination and further enhancements. We consider the following approaches for computing reachability and safety, which we find amenable to Proc. 1:

- Constant continuous and affine discrete dynamics
  - A HyTech (Henzinger et al., 1997)
  - B PHAVer (Frehse, 2008)
- Affine continuous and discrete dynamics
  - C d/dt (Asarin et al., 2001)
  - D Using zonotopes (Girard, 2005)
  - E Using support functions (Girard and Le Guernic, 2008)
  - F Algorithmic improvements to compute post_c for D,E (Girard et al., 2006)
- Nonlinear dynamics and abstraction refinement
  - G Approximating nonlinear dynamics by hybridization (Asarin et al., 2003)
  - H Forward/backward refinement (Frehse et al., 2006)
  - I CEGAR-type approaches (Frehse et al., 2008)

The reachability techniques in A,B are for piecewise constant derivatives, exact as well as overapproximative over an infinite time horizon. In C, affine continuous and discrete dynamics are overapproximated by discretizing time over a finite time horizon. In D and E, this technique is improved by exploiting the advantages of a particular representation of continuous sets, plus some low-level algorithmic improvements. In G, the techniques for affine dynamics are extended to nonlinear dynamics by overapproximation based on partitioning the state space. In H, a very simple abstraction/refinement technique is used for deciding safety, and more sophisticated ones based on *counter example guided abstraction refinement* (CEGAR) can be found in I. Approaches A–E constitute low-level algorithms that deal with computing post-images for particular dynamics, while G–I are high-level techniques that use low-level reachability algorithms as an intermediate step.

3.1 Common Elements

An analysis of common elements and differences shall provide us with the basis for our design. The system under examination is described as a network of interacting automata. The specification consists of the set of initial and (for safety) forbidden states. In addition, the user has to provide analysis parameters such as discretization time steps or partition sizes. For the analysis, a parallel composition operator transforms the automaton network into a single automaton, possibly on the fly. The set of reachable states is computed using some variant of Proc. 1. The resulting set of states undergoes some basic processing (intersection with forbidden states, projection onto variables of interest), and is output to a file or visualized.

3.2 Differences

The approaches we consider differ along the following lines:

Set Representations Polyhedra (A,B), zonotopes (D), and support functions (E) have each various advantages and disadvantages on fundamental set operations. For example, for polyhedra in constraint form computing intersection is cheap and Minkowski sum is expensive, while for zonotopes Minkowski sum is cheap and the intersection of two zonotopes is not generally a zonotope.

Discrete post-computations Various exact as well as overapproximative techniques for computing the image of discrete transitions are available (such as taking the convex hull), depending on the set representation. Most techniques apply to discrete dynamics in the form of affine maps (resets). For example, the image of an affine map is cheap for zonotopes and polyhedra in generator form, but not for polyhedra in constraint form.

Continuous post-computations Computing the image of a set after time elapse generally necessitates an overapproximations. Different techniques are applicable according to the type of continuous dynamics as well as the set representation. For A,B the image is over infinite time, while C,D,E,F discretize time and compute it over a bounded interval. Even for just linear dynamics, variations abound. For example, F avoids the wrapping effect by essentially reordering the computation and its approach is applicable to D,E.

State exploration Most approaches are defined for forward reachability, but can equally be applied as backward reachability by reversing the system dynamics. One direction may work better than another depending on the characteristics of the system (Mitchell, 2007), and H combines both. I requires keeping track of the dependency graph between symbolic states, i.e., which are the successor states of which. The explored states need to be stored in some form of passed/waiting list, and at each iteration the explored states need to be separated into those that are new and those that already been explored, which involves some form of difference operation (exact, overapproximative, see A). Similar, and more advanced, exploration algorithms are widely used in program analysis (worklist algorithms), and could be adapted to our applications.

Model transformations Hybridization (G) and abstraction/refinement techniques (B,H,I) involve duplicating (splitting) locations, adding and removing transitions, and modifying dynamics and invariants. Such changes in the model must be compatible with the state exploration if they are to be carried out on the fly, or if state representations are to be compatible with different variants of the same model.

High level algorithms In abstraction/refinement schemes like H or I, computing the reachable states is just one step in a larger process. They require certain low-level information like the dependency graph and counter examples to be accessible, and entail model transformations.

Automaton composition Composition operators differ in the type of communication (synchronization) and how variables are shared (A versus B).

4. DESIGN PRINCIPLES

4.1 Scenario Elements

Based on the survey and Proc. 1, we define the following *scenario elements* and their operations:

- automaton representation : add locations, transitions
- automaton network representation (controls composition; itself an automaton) : add automata
- discrete and continuous set representations : inclusion and emptiness tests, transforms (intersection, affine maps, etc.)
- adapt: convert sets and dynamics to the right form (if possible)
- PWL : add, pop symbolic states
- continuous-post :transform a symbolic state into a set of symbolic states
- discrete-post : transform a symbolic state into a finite set of symbolic states

Implementation choices depend on each other. E.g., a specific continuous-post operator might only apply to affine dynamics and require polyhedra as set representations. At the same time, we would like to keep the concrete classes encapsulated as much as possible; whoever writes the polyhedron class shouldn't need to know anything about hybrid automata. We note the following guidelines:

- Implementations for the scenario elements should be interchangeable.
- The scenario elements should be used in Proc. 1 whenever possible (instead of creating new algorithms); this shall guarantee that implementations from different sources remain interchangeable and as compatible as possible, avoiding divergence between different implementations.
- Compatability between scenario elements is optional. We assume that anyone selecting a set of scenario elements to create a scenario has expert knowledge. It suffices that an exception is created when an incompatability is detected during execution.
- Advanced algorithms modify the system model (automaton network) on the fly or between re-runs of the reachability algorithm. Set representations need to be compatible with corresponding changes in locations and transitions, e.g., using keys to refer to previous versions of the location or transition.

4.2 Tool architecture and execution

We define for each of the scenario elements an abstract base class, from which implementations must be derived. We call a set of implementations for the scenario elements a *scenario implementation*, and define a scenario class to hold references to them, similar to the strategy design pattern. Given a scenario object, our implementation of Proc. 1 uses these references to instantiate automaton and set representation, and carry out operations on symbolic states and the PWL.

A run of the tool (assuming the model has already been generated possibly in a graphical editor) consists of the following steps, as shown in Fig. 1:

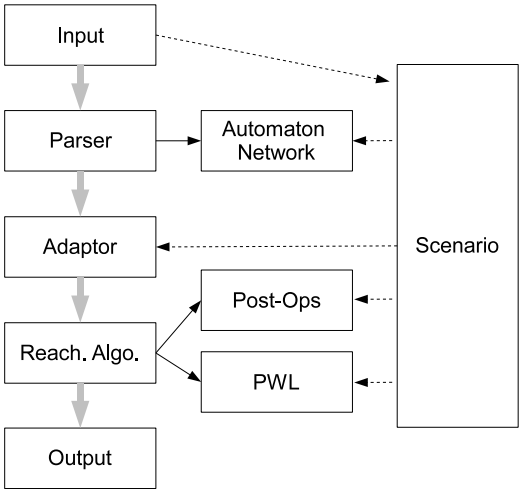


Fig. 1. Schematic of the tool architecture (solid arrows represent acquaintance between objects, dashed arrows represent instantiation). Grey arrows indicate in which order the different components are executed

(1) The user provides the input : models (XML), user commands, scenario selection, output selection.
(2) The input file is parsed to generate a general representation of the automata (transitions/locations) and sets (initial states, bad states).
(3) The general automata are adapted to the right set representation and dynamics according to the scenario (adapt).
(4) The automaton network is instantiated according to the scenario.
(5) The user selected algorithm (reachability, safety) is executed, using the elements provided by the scenario (PWL, post).
(6) The output is created : visualization, file export (model, states).

User options are used to select the scenario, additional options can be passed directly to the scenario.

5. SCENARIO IMPLEMENTATIONS

Our platform provides default implementations for discrete sets, automata, automata networks and the PWL (linked list) – for reasons of space we omit details and refer to a similar implementation in (Frehse, 2008). Scenario implementations must only provide the remaining elements: representations of sets, dynamics, its adaptors, and post-operators. In the following we briefly present two such scenarios, one based on linear hybrid automata (similar to approach B), and one based on support functions (similar to E).

5.1 Linear Hybrid Automaton Scenario

A *linear hybrid automaton* (LHA) is a hybrid automaton whose continuous sets and relations are given by convex linear constraints over, respectively, the variables (invariant, initial states), the derivatives (flow), and the variables distinguishing before and after a jump (jump relation). This means that the continuous dynamics are nondeterministic with constant bounds, e.g., $1 \leq \dot{x} \leq 2$ or $\dot{x}+\dot{y} = 0$.

The discrete dynamics are nondeterministic affine, e.g., $x' = 0$ or $x' = a * x + b$.

We represent continuous sets as polyhedra and provide a straightforward implementation based on linear programming to decide containment and emptiness. Fourier-Motzkin elimination is used for existential quantification. A generic lp-solver interface allows us to use different linear programming solvers, such as the GLPK (Makhorin, 2009).

The continuous dynamics are modeled as the continuous set of derivatives for each location. The discrete dynamics (jump relations) are modeled as a continuous set over primed variables (after the jump) and unprimed variables (before the jump).

For LHA, the post-operators are first-order predicates whose solutions can be computed using the above standard operations on polyhedra.

5.2 Support Function Scenario

For affine continuous and discrete dynamics, an efficient approach to compute the reachable states has been proposed in (Girard and Le Guernic, 2008). Given a set of directions, it uses polyhedral over-approximations, where each face of the polyhedron is a tight bound on the original set in one of the given directions. In this section, we present our implementation of the post-operators using support functions.

We consider the discrete dynamics to be given by a polyhedral *guard set* G and an affine reset map $x' = Cx + D$, which correspond to the jump relation $\mu = \{(x, x') \mid x \in G \wedge x' = Cx + D\}$.

Continuous-Post Operator We compute $post_c(loc, X_0)$ for dynamics

$$\dot{x} = Ax + b, x_0 \in X_0,\ b \in U, \tag{2}$$

where $X_0 \subseteq \mathbb{R}^n$ and $U \subseteq \mathbb{R}^n$ are compact convex sets. The set U can be used to model disturbances on the model or approximation errors, e.g., when approximating a nonlinear system. Given a discretization time step δ, we overapproximate the set of reachable states by a sequence of convex sets Ω_k, each of which covers the states reachable in the time interval $[k\delta, (k+1)\delta]$:

$$\Omega_{k+1} = \Phi\Omega_k \oplus V, \tag{3}$$

where $\Phi = e^{A\delta}$ and $\oplus$ denotes the Minkowski sum. The initial set Ω_0 is obtained by taking the convex hull of X_0 and $e^{A\delta}X_0$, and bloating it such that it covers all trajectories in the time interval $[0, \delta]$. Given a time horizon N, $post_c$ is then covered by the union of $\Omega_0, \ldots, \Omega_N$. (Girard and Le Guernic, 2008) provides an algorithm to compute a polyhedral overapproximation of $post_c$ of a symbolic state, i.e., $\overline{\Omega}_0, \ldots, \overline{\Omega}_N$ for a given time horizon N, where $\overline{\Omega}_k$ is a tight polyhedral overapproximation of Ω_k.

For a nonempty convex set $\Omega \subset \mathbb{R}^n$ and a direction $l \in \mathbb{R}^n$, the *support function* is

$$\rho_\Omega(l) = sup\{\langle l, x\rangle |\ x \in \Omega\} \tag{4}$$

Given a set of r directions $L = \{l_1, \ldots, l_r\}$ and a time horizon N, $\overline{\Omega}_1, \ldots, \overline{\Omega}_N$ is represented as a $r \times N$ matrix

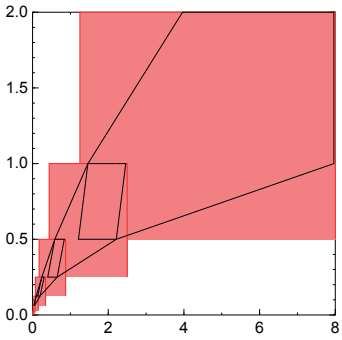


Fig. 2. Polyhedral overapproximation of post_c using support functions (shaded), and actual Ω_k for comparison (outlined)

called *Support Function Matrix* with $(i, j)^{th}$ entry denoting the support function sample of Ω_j in the direction l_i. For a given SFM M and directions L, we denote the outer polyhedral approximation of the jth set as

$$PO(L, M_j) = \bigcap_{i=1}^{r} \langle l_i, x\rangle \leq M_{i,j}.$$

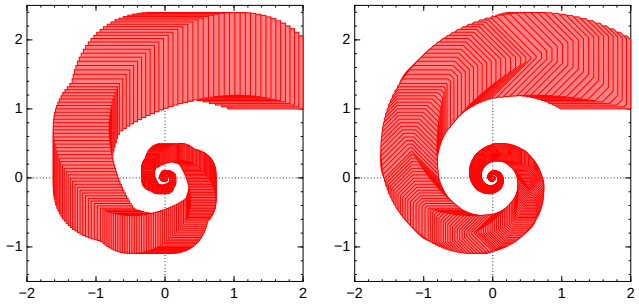
The reachable continuous set resulting from time elapse in a location, say d, is now represented in the form of a SFM.

Example 1. Figure 2 illustrates the polyhedral overapproximation of our support function implementation of post_d. The sequence of Ω_k is shown outlined. Given the axes as directions, the support function approximation finds the bounding boxes of Ω_k shown shaded.

Example 2. To illustrate how the approximation quality depends on the chosen directions, consider the flowpipe approximations of a 5-dimensional system shown in Fig. 3. Computing post_c using support functions for $2n$ box constraints of the form $\pm x_i \leq c$ yields the set shown in Fig 3(a), in 0.4s using 6.4MB RAM on an i386 processor with 3.2GHz. Computing with $2n^2$ octagonal constraints of the form $\pm x_i \pm x_j \leq c$ yields the set shown in Fig 3(b), in 1.9s using 6.7MB RAM.

Discrete-Post Operator The $post_c$ operator gives us an SFM, a matrix representation of $\overline{\Omega}_0, \ldots, \overline{\Omega}_N$ To compute the intersection of the time elapse set with a guard G, we would like to identify the relevant Ω's to consider for computing the intersection with the guard or in other words, we would like to filter out the irrelevant Ω's before proceeding the intersection computation. By relevant, we mean the ones which can possibly intersect with a guard G. One possibility to identify the relevant Ωs' is to compute their distance from G and check if it is less than or equal to the diameter of Ω. If the distance is greater than the diameter, we know that $\Omega \cap G = \emptyset$ and hence can discard the Ω. The identified Ω's can be filtered by simply dropping the corresponding columns from SFM and the result after filtering is a smaller SFM in terms of columns.

To compute the intersection with G, we manipulate our SFM M so as to get a new SFM M_I representing the intersection set. Semantically interpreting M as its outer polyhedral approximation, the intersection is the SFM given by adding the normal vectors of the constraints



(a) Box constraints (b) Octagonal constraints

Fig. 3. Polyhedral overapproximation of post_c using support functions of 5-dimensional system projected to x and y

(faces) of G to the set of directions, and computing the SFM for the new set of directions.

The final part of the $post_d$ operator is to compute the linear transform of the intersection set. Support functions have the convenient property that given compact convex sets $X, D \subseteq \mathbb{R}^n$, a direction $l \in \mathbb{R}^n$, and a $n \times n$ transformation matrix C, $\rho_{CX \oplus D}(l) = \rho_X(C^T l) + \rho_D(l)$. Using this property, we overapproximate the transformed set with an SFM M_T defined by

$$M_{i,j} = \rho_{PO(L,M_j)}(C^T l_i) + \rho_D(l_i).$$

The complexity of this construction is $O(kN^2 + kN(P_{k,n}))$, where k is the number of directions in D, N is the time horizon and $P_{k,n}$ denotes the time complexity of computing the support function of a polyhedron with k constraints and n variables.

6. CONCLUSION

A tool platform on which several competing reachability algorithms can be implemented brings economies of scale to the development of new approaches as common elements can be shared. Comparing different algorithms on the same platform may allow more insight than having implementations differ in speed or precision simply due to different programming languages or libraries. Most importantly, the use and combination of various set representations may open up new perspectives for further research. In this vein, the support function scenario presented in this paper can handle any convex continuous set representation on which one can compute the supremum of a linear function. E.g., one can run this algorithm on an instance where the initial states are given by polyhedra and disturbances are given by ellipsoids (a natural model in many applications), which is significantly more accurate than if one were to overapproximate the ellipsoid by a polyhedron or a zonotope. [1] Exploiting polymorphism, the set representation may even change during the course of the analysis in order to increase precision where necessary.

REFERENCES

Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., and Yovine, S. (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1), 3–34.

Asarin, E., Dang, T., and Maler, O. (2001). d/dt: a tool for reachability analysis of continuous and hybrid systems. In *IFAC Symp. Nonlinear Control Systems*.

Asarin, E., Dang, T., Frehse, G., Girard, A., Le Guernic, C., and Maler, O. (2006). Recent progress in continuous and hybrid reachability analysis. In *IEEE Int. Symp. Computer-Aided Control Systems Design*.

Asarin, E., Dang, T., and Girard, A. (2003). Reachability analysis of nonlinear systems using conservative approximation. In *HSCC'03*, volume 2623 in LNCS, 20–35. Springer.

Balluchi, A., Casagrande, A., Collins, P., Ferrari, A., Villa, T., and Sangiovanni-Vincentelli, A.L. (2006). Ariadne: a framework for reachability analysis of hybrid automata. In *MTNS'06*.

Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., and Yi, W. (1995). UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. Workshop on Verification and Control of Hybrid Systems III*, LNCS. Springer.

David, A., Behrmann, G., Larsen, K.G., and Yi, W. (2002). A tool architecture for the next generation of uppaal. In B.K. Aichernig and T.S.E. Maibaum (eds.), *10th Anniv. Colloq. UNU/IIST*, volume 2757 of *LNCS*, 352–366. Springer.

de Paula, F.M. and Hu, A.J. (2007). An effective guidance strategy for abstraction-guided simulation. In *Design Automation Conference*, 63–68. ACM/IEEE.

Egerstedt, M. and Mishra, B. (eds.) (2008). *HSCC'08*, volume 4981 of *LNCS*. Springer.

Frehse, G. (2008). PHAVer: Algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3), 263–279.

Frehse, G., Jha, S.K., and Krogh, B.H. (2008). A counterexample-guided approach to parameter synthesis for linear hybrid automata. In Egerstedt and Mishra (2008), 187–200.

Frehse, G., Krogh, B.H., and Rutenbar, R.A. (2006). Verifying analog oscillator circuits using forward/backward abstraction refinement. In G.G.E. Gielen (ed.), *DATE*, 257–262.

Girard, A. (2005). Reachability of uncertain linear systems using zonotopes. In *HSCC'05*, volume 3414 in LNCS, 291–305. Springer.

Girard, A., Guernic, C.L., and Maler, O. (2006). Efficient computation of reachable sets of linear time-invariant systems with inputs. In J.P. Hespanha and A. Tiwari (eds.), *HSCC*, volume 3927 of *LNCS*, 257–271. Springer.

Girard, A. and Le Guernic, C. (2008). Efficient reachability analysis for linear systems using support functions. In *Proc. IFAC World Congress*.

Henzinger, T.A., Ho, P.H., and Wong-Toi, H. (1997). HYTECH: A model checker for hybrid systems. *STTT*, 1(1–2), 110–122.

Makhorin, A. (2009). GNU Linear Programming Kit, v.4.37. `http://www.gnu.org/software/glpk`.

Mitchell, I.M. (2007). Comparing forward and backward reachability as tools for safety analysis. In A. Bemporad, A. Bicchi, and G.C. Buttazzo (eds.), *HSCC*, volume 4416 of *LNCS*, 428–443. Springer.

Sankaranarayanan, S., Dang, T., and Ivancic, F. (2008). A policy iteration technique for time elapse over template polyhedra. In Egerstedt and Mishra (2008), 654–657.

[1] We thank Colas Le Guernic for pointing this out.