An Introduction to SpaceEx v0.8

Goran Frehse
Université Grenoble 1 Joseph Fourier - Verimag
Centre Équation, 2 av. de Vignate, 38610 Gières, France
goran.frehse@imag.fr

December 2, 2010

Abstract

This document provides a brief overview of the capabilities of SpaceEx and the concepts behind it. We present how hybrid systems are modeled in SpaceEx using hybrid automata that can be composed in nested networks. We give an overview of the analysis algorithms implemented in SpaceEx, and the different options that are available to the user. The algorithms are described strictly from a user perspective, mentioning mathematical details and algorithmic trickery only when necessary. The reader is assumed to be familiar with the basic concepts of hybrid systems and reachability analysis. For brevity, the presentation is informal, but references to more detailed descriptions are given in the text.

1 Introduction

SpaceEx is a verification platform for hybrid systems. The goal is to verify (ensure beyond reasonable doubt) that a given mathematical model of a hybrid system satisfies the desired safety properties. The basic functionality is to compute the sets of reachable states of the system. We assume that the reader is familiar with hybrid systems and reachability computations. For a gentle introduction to the topic, see [7, 4].
SpaceEx consists of three components, see Fig. 1:

- The *model editor* is a graphical editor for creating models of complex hybrid systems out of nested components. It produces model files in SpaceEx's `sx` format.
- The *analysis core* is a command line program that takes a model file in `sx` format, a configuration file that specifies the initial states, the scenario and other options, analyzes the system and produces a series of output files.
- The *web interface* is a graphical user interface with which one can comfortably specify initial states and other analysis parameters, run the analysis core, and visualize the output graphically. The web interface is browser based, and accesses the analysis core via a web server, which may be running remotely or locally on a virtual machine.

Strictly speaking, the SpaceEx analysis core is not a single tool but a development platform on which several different verification algorithms are implemented. Each algorithm may come with its own set representation, apply to its own class of models,

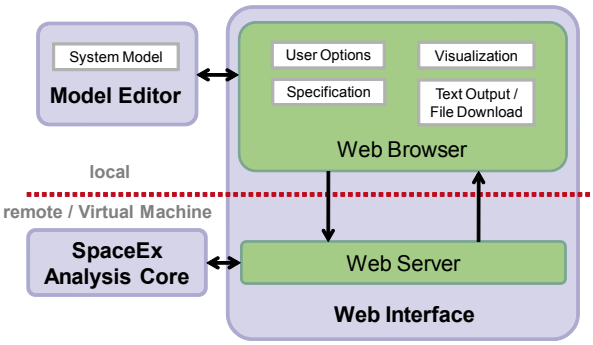


Figure 1: Software architecture of the SpaceEx platform

and produce a different kind of output, so we refer to such a bundle as a *scenario*. Currently, two scenarios are implemented:

- The PHAVer scenario implements the basic algorithm from the tool PHAVer [3]. It applies to Linear Hybrid Automata, which are hybrid systems with piecewise constant bounds on the derivatives.
- The LGG Support Function scenario implements a variant of the Le Guernic-Girard (LGG) algorithm from [5]. It applies to hybrid systems with piecewise affine dynamics with nondeterministic inputs.

The examples used in this paper are included in the Virtual Machine Server and available on the SpaceEx website [11].
The paper is structured as follows. In Sect. 2 we briefly present how systems are modeled in SpaceEx using base and network components and the notation used for describing dynamics and sets of states. In Sect. 3 we present the analysis core of SpaceEx, outline the LGG Support Function scenario and present the main options available to the user. Section 4 summarizes the available output formats.

2 Modeling Hybrid Systems in SpaceEx

SpaceEx models are stored in the `sx` format, an XML based format for which there is a graphical model editor. Please refer to the examples provided on the download page and on the virtual machine.
`sx` models are similar to the hybrid automata known in literature, except that they provide a richer mechanism of hierarchy, templates and instantiations. An `sx` model consists of one or more components. When SpaceEx reads an `sx` file, it translates the components into either an automaton or into a network of automata in parallel composition.

2.1 Components

A model is made up of one or several *components*. There are two types of components: A *base component* corresponds to a single hybrid automaton, and is defined by its locations and transitions. A *network component* consists of one or more instantiations

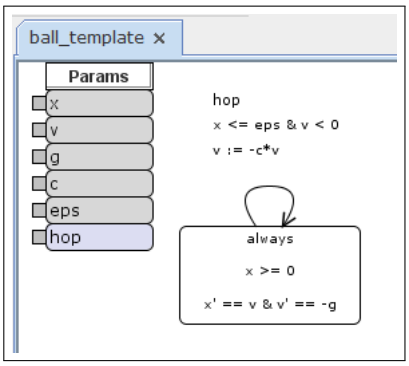


Figure 2: A bouncing ball model with its formal parameters: state variables x and v, constants g, c, and eps, and label hop

of other components (base or network) and corresponds to a set of hybrid automata in parallel composition. Every component has a set of *formal parameters*. A formal parameter may be:

- a continuous variable with arbitrary dynamics,
- a continuous variable with constant dynamics, i.e., it does not change its value over time.[1] It may be assigned a value, like 3.1415, when it is instantiated in a network component. Then it becomes what is commonly referred to as a "constant".
- a synchronization label. Every transition is associated with a label.

A formal parameter is part of the *interface* of a component, unless it is declared as *local* to the component.

> **Note:** All symbols (variables, constants or labels) that are used in describing locations or transitions must be declared as formal parameters of the corresponding component. Symbols that are not of interest outside of the component can be declared as local.

Example 1 The interface of a bouncing ball model is shown in Fig. 2. The continuous state variables x and v are declared as variables with arbitrary ("any") dynamics. The constants g, c, and eps are declared as variables with constant dynamics. They will be assigned values when the template is instantiated inside a network component. The last parameter is the synchronization label hop. □

[1] A variable with constant dynamics is commonly referred to as a "parameter" of a hybrid automaton, but these should not be confused with the formal parameters of components.

2.2 Base Components and Semantics

A base component in the model is translated by SpaceEx into a *hybrid automaton* [1]. It consists of a graph in which each vertex, called *location*, is associated with a *flow*, which is a set of differential equations (or inclusions) that defines the time-driven evolution of the continuous variables. A *state* consists of a location and a value for each of the continuous variables. The edges of the graph, called *transitions*, allow the system to jump between locations, thus changing the dynamics, and instantaneously modify the values of continuous variables according to an *assignment*. The jumps may only take place when the values of the variables satisfy the constraints of the *guard* of the transition. The system may only remain in a location as long it satisfies the constraints of the *invariant* associated with the location. All behavior originates from a given set of *initial states*.
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. Given a set of forbidden states, the system is *safe* if the bad states are not reachable.

Dynamics and Constraints The flow of a location is a set of differential equations. It depends on the scenario which types of constraints are allowed. The LGG scenario accepts affine dynamics of the form

$$\frac{d}{dt}x = Ax + Bu + b_0,$$

where u is a set of nondeterministic inputs. Constraints on u can be included in the invariant of the location. The `sx` format does not currently support vector/matrix notation. A system of the above form is described by the expression

```
x1' == a11*x1 + ... + a1n*xn + b11*u1 + ... + b1m*um &
x2' == a21*x1 + ... + a2n*xn + b21*u1 + ... + b2m*um &
...
xn' == an1*x1 + ... + ann*xn + bn1*u1 + ... + bnm*um
```

where the prime behind a variable denotes its derivative.
Similarly, a transition may modify the variables with an assignment of the form

$$x := Ax + Bu + b_0,$$

where u is a set of nondeterministic inputs that is constrained only by the invariant. The assignment can be described in two forms: either with primes

```
x1' == a11*x1 + ... + a1n*xn + b11*u1 + ... + b1m*um &
x2' == a21*x1 + ... + a2n*xn + b21*u1 + ... + b2m*um &
...
xn' == an1*x1 + ... + ann*xn + bn1*u1 + ... + bnm*um
```

where the prime denotes the value after the transition, or by the equivalent expression

```
x1 := a11*x1 + ... + a1n*xn + b11*u1 + ... + b1m*um &
x2 := a21*x1 + ... + a2n*xn + b21*u1 + ... + b2m*um &
...
xn := an1*x1 + ... + ann*xn + bn1*u1 + ... + bnm*um
```

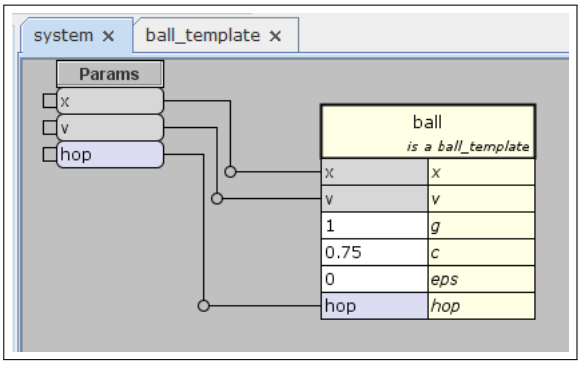


Figure 3: The bouncing ball template instantiated in the network component *system*. The constants are bound to the desired values, the other formal parameters are passed on to the interface of system.

Variables that are not assigned explicitly are supposed to remain constant during the transition.
Guards and invariants can be arbitrary convex linear constraints on the variables, where conjunctions are denoted by an ampersand (&). For example:

```
a*x+b*y == 1 & c <= z <= d
```

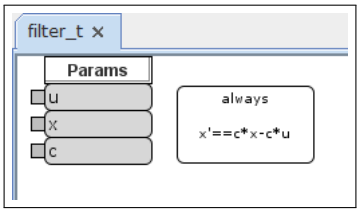
A universal constraint can be denoted with `true`, and an unsatisfiable constraint by `false`.

2.3 Network Components

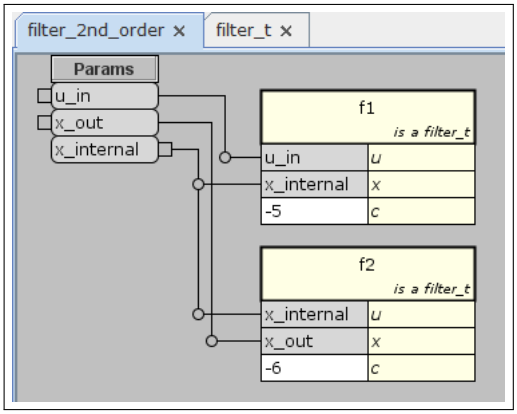
A network component allows one to instantiate one or more components (base or other network components), connect them via their variables and labels, and assign values to their constants.

Instantiating Components When a component A is instantiated inside a network component B, a copy of A is created and added to the contents of network B. The copy must have a name that is unique within B, say A1. It is a copy by reference, so that any later modifications to A will also apply to A1. Component A can be instantiated several times inside B, say as A1, A2, A3, etc.
When the `sx` model is parsed by SpaceEx, each base component is translated into a corresponding hybrid automaton, and each network component is translated into the parallel composition of its subcomponents. Semantically, the parallel composition of hybrid automata is itself a hybrid automaton. In SpaceEx, this composition is carried out on the fly, so that only the reachable parts of that automaton are actually created in memory.
When instantiating a component A in network B, we must specify what happens to each of the formal parameters in its interface. This is called a *bind*. Every formal parameter of A must be bound to either a formal parameter of B or to a numeric value.

Example 2 Figure 3 shows the network component *system*, which is made up of an instantiation of the base component *ball_template*. The instantiation is called *ball*. The



(a) First order filter



(b) Second order filter

Figure 4: A second order filter constructed using two instantiations of a first order filter

formal parameters x, v, and hop are bound to formal parameters x, v, and hop of *system*. The constants g, c, and eps are bound to numeric values. □

Connecting Components Components inside a network can be connected by binding their variables or labels to the same symbols in B. Whether this connection is in parallel or in series is not fixed by the interface. It depends entirely on the restrictions the components make on the variables.

Example 3 Figure 4(a) shows a simple first-order low-pass filter. It has u as input and x as output, c is a value determining the cut-off frequency. The network component *filter_2nd_order* shown in Fig. 4(b) consists of two instantiations of this filter put in series. The network has the input u_{in}, the output x_{out}, and the internal variable $x_{internal}$. The filters are put in series by binding the output of the first filter and the input of the second filter to the same variable $x_{internal}$. When this model is read in SpaceEx, it creates a hybrid automaton with three variables: u_{in}, x_{out}, and $x_{internal}$. Note that $x_{internal}$ was chosen to be local to *filter_2nd_order*. This means that it does not figure in its interface, and should *filter_2nd_order* be used in a network together with other components, they will not have access to that variable. □

Note: Binding two or more variables to the same variable in a network component means that they become literally the same variable. Their dynamics can be defined in one or several of the components as long as this creates no contradictions.

2.4 Causality and Nondeterminism

SpaceEx models have nondeterministic, acausal semantics: Any component can declare any variable as one its formal parameters, and impose constraints (including

equalities) on the variable and its derivative. Different components can impose constraints on the same variable, at the same time or in alternation. In certain application areas, like mechanics or electric circuits, this allows one to decompose models into reusable building blocks, or build models directly from first principles. For example, one component could impose $\dot{x} \leq 0$ and another $\dot{x} \geq 0$. The resulting behaviour has to satisfy both, so $\dot{x} = 0$. If the constraints contradict each other, resulting in, e.g., $\dot{x} \in \emptyset$, there is no solution to the differential equations (or inclusions) and thus there might not be any trajectories after a certain point in time. We say that "time stops" in the model. This may or may not be a modeling error. Usually, there should be at least one trajectory over infinite time for every initial state. But in a nondeterministic system, this doesn't mean that all trajectories from the initial states range over infinite time. Especially when overapproximations are used to simplify a model, states in which time stops may be reachable. As long as this is an artifact of the overapproximation, this poses no problem.

Note: Modeling errors may "stop time" when contradicting constraints render differential inclusions or transition assignments unsatisfiable. The model may then be trivially safe as an artifact of that modeling error. Care should be taken to ensure time passes as intended by the modeler.

We recommend that the modeling process is accompanied by steps that ensure that time indeed passes up to the desired point. While a formal verification of this liveness property is hard and beyond the capabilities of SpaceEx, the following informal procedure can help to detect at least crude modeling errors:

1. Pick a time point T that is large relative to the time constants in the system.
2. Create a monitor automaton with a variable d, flow $\dot{d} = 1$ and invariant $d \leq T$.
3. Put the system in parallel with the monitor and verify that from the initial state $d = 0$ the state $d = T$ is reachable.

2.5 Inputs, Outputs and Controlled Variables

Because of the acausal semantics, there are no inputs and outputs in SpaceEx models per se. For compositional reasoning, there is the notion of a *controlled variable*, which is closely related, and SpaceEx enforces compositional semantics for this purpose.

Note: Users who do not care about compositional reasoning can simply declare all variables as controlled in all components.

We now give a brief description of compositional reasoning and how it employs controlled variables. The following notions about compositional reasoning apply to continuous variables as well as to events (synchronization labels), but for brevity we limit the discussion to the continuous part. Recall that a component H with a variable x restricts the behavior of x over time. This restriction is expressed in the constraints that invariants, flows, guards and assignments of H pose on x and $\dot{x}$. A compositional framework like the one in [2] guarantees the following: The behavior of x in a network

component that contains H is a subset of the behavior of x in H by itself. So if the behavior of x in H is safe, then it is safe also in any other component that includes H. To make the above guarantee, one needs to forbid that other components modify x beyond what is possible in H itself. The framework in [2] achieves this by declaring x *controlled* in components that have as a subcomponent H and *uncontrolled* everywhere else. In addition, the following semantic restrictions are imposed on all components G in which x is uncontrolled:

- When G takes a transition that does not synchronize with a transition of H, it must not change the value of x in that transition. To ensure this SpaceEx adds the constraint $x' == x$ to such transitions in G.
- When G remains in a location and time passes, x must be able to take on any value that satisfies the invariant of G. To ensure this SpaceEx adds self-loops to G that assign x nondeterministically to the allowed values.

Note that no restrictions are imposed on components in which x is controlled. For compositional reasoning, there can be at most one base component that controls a variable, and it must be uncontrolled in all other base components. For network components, the model editor deduces automatically whether a variable is controlled or not.
In the semantics that result from the above restrictions, uncontrolled variables can be loosely understood as *inputs*, and controlled variables as *outputs*.

> **Note:** Users who wish to apply compositional reasoning need to declare every variable as controlled in at most one base component, and as uncontrolled in all other base components. SpaceEx enforces compositional semantics by adding the appropriate constraints and self-loop transitions for every variable that is declared as uncontrolled.

2.6 Initial and Forbidden States

The specification of a reachability problem includes the set of initial states, from which all behavior of the system originates. A set of states can be specified in SpaceEx as a boolean combination of *location constraints* and linear constraints over the variables (see invariants and guards). Conjunctions are denoted by an ampersand (&) and disjunctions by a vertical bar (|). A location constraint is of the form

$$\texttt{loc(} < \textit{base component name} > \texttt{)==} < \textit{location name} >,$$
$$\texttt{loc(} < \textit{base component name} > \texttt{)!=} < \textit{location name} >,$$

where the first form denotes a single location in the given component, and the second form denotes all other locations of that component.

Dot notation and context dependent lookup Component and variable identifiers are instantiated in SpaceEx such that they are unique for each instantiation of a base component. This is achieved by prepending their names with the names of the network components that contain the base component, separated by a period (.). To avoid unnecessarily long names, SpaceEx uses context dependent lookup with respect to the component that is selected for analysis. Component names do not need to be prepended if the identifier is unique within that component.

Example 4 Suppose a base component A has a local variable x and locations a and b. A network component B consists of two instantiations of A called $A1$ and $A2$. To specify that A1 is in location a with $x = 1$ and A2 is in location b with $x = 2$, one can use the constraints

```
loc(B.A1)==a & B.A1.x==1 & loc(B.A2)==b & B.A2.x==2
```

Suppose C is a network component containing an instantiation $A3$ of base component A, and D is a network component containing and instantiation $C1$ of C. We can define initial states for $A3$ with `loc(D.C.A3)==a & D.C.x==1`. If D is the component to be analyzed, we can omit the prefix D. and write `loc(C.A3)==a & C.x==1`. If $A3$ is the only component inside C that has a variable called x, we can omit the prefix for the variable and write `loc(C.A3)==a & x==1`. If D contains only one component with the name $A3$, we can omit its prefix as well, which leaves us with `loc(A3)==a & x==1`. □

If the component to be analyzed is a base component, there is no need to specify the component name, and one can just write `loc()==a` instead.

3 Analyzing Systems using SpaceEx

We now give a brief overview of the LGG scenario and the options available in SpaceEx. Options can be set via the command line or via the web interface. The configuration files saved by the web interface can also be read directly by the command line tool. Note that it is possible to display the command line generated by the web interface, which may be useful for creating scripts etc.

3.1 Reachability Algorithm

The reachability algorithm operates on *symbolic states*. A symbolic state is the cross product of a set of discrete states (locations) and continuous states (variable valuations). The algorithm uses a *passed list* of states found so far and a *waiting list* of states whose successors are yet to be computed. The algorithm proceeds as follows:

1. Intialize the waiting list with the time-elapse successors of the initial states.
2. Pick a symbolic state from the waiting list.
3. Compute the transition successors (generating possibly more than one symbolic state).
4. Compute the time-elapse successors to every generated symbolic state.
5. Throw away the symbolic states (or parts of them) that are already on the passed list – this involves testing for containment and emptiness. Put the remaining ones onto the passed list and the waiting list.
6. Compact the passed list by removing redundant states.
7. If the waiting list is not empty, go to 2.

Note that reachability for hybrid automata is undecidable, and this procedure is not guaranteed to terminate. Upon termination, the result is an overapproximation of the reachable states.
The following options are available to control the reachability algorithm:

- **Max. iterations**: Maximum number of iterations for the reachability algorithm, which is the total number of discrete post computations on symbolic states. If negative, the algorithm terminates only when a fixed point is reached.
- **Relative and absolute error**: These values are used when comparing floating point values and deciding whether they are considered equal. This impacts mainly tests for containment and emptiness of objects.
- **Merging passed with waiting list**: When a new state A contains a state B already on the passed list, B is by default replaced by A on the passed list. This merging process incurs the cost of containment checking and can be disabled.

3.2 The LGG Support Function Scenario

The LGG Support Function scenario is a variant of the support function algorithm proposed in [5]. In the LGG Support Function scenario, the set of reachable states is overapproximated by a set of polyhedra. To make the approach scalable, it uses a combination of operations on implicit set representations (support functions) and overapproximation steps.
Two operators are necessary to compute the reachable states: computing the states reachable by time elapse and computing the image of a set of states that take a transition.

3.3 Computing time elapse

In the following we consider what happens to a convex set of states in a single location if we let time elapse. The set of states reachable from an initial set X_0 is called a *flowpipe*. Starting from the initial set, the LGG scenario computes a series of convex sets that cover the flowpipe. Each convex set covers a chunk of δ time out of the flowpipe, so that after having computed k of these sets we have covered the states that are reachable from X_0 up to time $k\delta$. We call δ the *sampling time*. Since we can't compute sets up to infinity, we define an upper bound on the time span we consider for each flowpipe, called the *local time horizon*.

Example 5 Consider the system shown in Fig. 5. We consider location p, with dynamics

$$\begin{aligned} \dot{x} &= -y, \\ \dot{y} &= x, \end{aligned}$$

which makes its states move around the origin in circular trajectories. We consider as initial set the state $(x = 1, y = 0)$, which gives rise to the circular flowpipe shown in bold in Fig. 6(a). The LGG time elapse algorithm with sampling time $\delta = 0.5$ and local time horizon 1.5 produces the three convex sets shown in Fig. 6(a), which cover the flowpipe. For smaller sampling times, the accuracy increases, as shown in Fig. 6(b) and Fig. 6(c). □

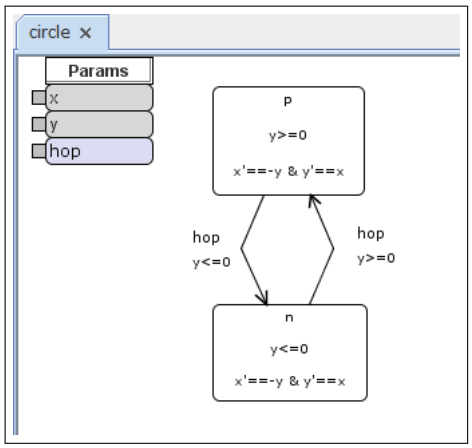
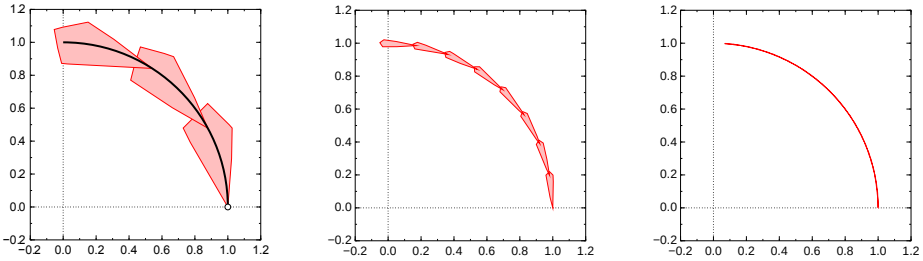


Figure 5: A two-dimensional system moving in circles around the origin

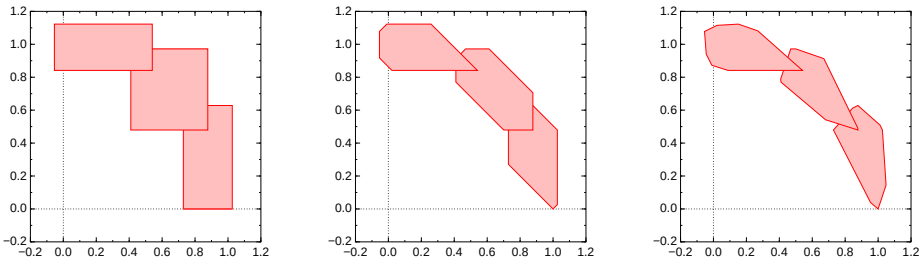


(a) $\delta = 0.5$ (b) $\delta = 0.2$ (c) $\delta = 0.05$

Figure 6: A flowpipe (bold in black) and the convex sets generated by SpaceEx to overapproximate it, for different values of the sampling time δ

From looking at the example, it seems that by reducing the sampling time, we might get arbitrarily close to approximating the flowpipe. But there is another source of overapproximation: The final result of the LGG algorithm are template polyhedra, i.e., polyhedra whose faces have a direction that is given a priori. The use of template polyhedra allows one to avoid costly operations on polyhedra such as convex hull and existential quantification, which can be exponential in the number of variables. The price for this scalability is the degree of overapproximation that such an a-priori choice incurs. As the number of provided directions goes to infinity (assuming they are evenly distributed), the error goes to zero. In the worst case, the number of directions needed to fall under a given error bound is exponential in the number of variables. Experiments have shown that in practise a low number of directions may suffice, but this depends on the system and the property at hand.
The LGG scenario provides three options to choose the template directions for an n-dimensional system:

- *box* directions, i.e., $2n$ directions aligned with the axes, i.e., $x_i = \pm 1$, $x_k = 0$ for $k \neq i$;



(a) box (b) octagonal (c) 16 uniform

Figure 7: Flowpipe overapproximation for different choices of template directions

- *octagonal* directions, i.e., $2n^2$ directions, consisting of all combinations of $x_i = \pm 1$, $x_j = \pm 1$, $x_k = 0$ for $k \neq i, j$;
- *uniform* directions, i.e., a set of m directions that are (as well as possible) uniformly distributed.

Example 6 Figure 7 shows the flowpipe of the initial state $(x = 1, y = 0)$ with sampling time $\delta = 0.5$ and local time horizon 1.5 for box, octagonal and 16 uniformly distributed directions. □

Intersection with the invariant All states that are reachable within a location must satisfy the location's invariant. For the flowpipe computation, this is achieved by intersecting the polyhedra that cover the flowpipe with the invariant.

3.4 Computing successors of transitions

Each flowpipe that is created by the time elapse step is passed separately to the computation of transition successors. To compute the successor states we compute the states that satisfy the guard, and then map them according to the assignment of the transition. The states that satisfy the invariant of the target location are the successor states.

Assignment If the assignment is invertible and deterministic, i.e., of the form $x := Ax + b_0$ with A being an invertible square matrix, the mapped states are computed exactly by mapping the polyhedron. Otherwise (non-invertible A or nondeterministic inputs), the mapped states are computing using a template overapproximation with the same template direction as used for time elapse.

Clustering Each flowpipe consists of a possibly large number of convex sets that cover the actual trajectories. When computing the states that can take a transition, clustering reduces this number. It iteratively replaces a group of sets of the flowpipe with a single convex set, their template hull. An option called *clustering percentage* determines how many sets come out of this process: A percentage of 0 means no reduction in the number of sets, all sets are passed to the aggregation step outlined

below. A percentage of 100 means that all sets are combined into a single set (no aggregation necessary). A value between 0 and 100 groups the convex sets such that the relative distance (Hausdorff) to the original is below the given value (smaller values indicate higher accuracy). The sets coming out of the clustering then go through the aggregation step.

Aggregation The clustering step creates a certain number of convex sets, each one spawning its own flowpipe in the next time elapse computation. This may multiply the number of sets with each iteration, leading to an explosion in the number of sets and slowing the analysis to a halt. To avoid this effect and speed up the analysis, these sets can optionally be overapproximated by their convex hull. A faster but more coarse alternative is to set the clustering percentage to 100, which results in only one convex set (the template hull).

4 SpaceEx Output

The reachable states computed by SpaceEx can be output in a number of formats for further processing and visualization:

Textual (TXT) States are written as text that can be parsed by SpaceEx. The output consists of a series of constraints on locations and variables in DNF and can be used, e.g., as initial states for further analysis.

Vertice List (GEN) Only the vertices of the objects are output as floating point numbers. This format can be passed to tools like `graph` from the Plotutils package [8] for visualization. Note that no location information is included, and that empty as well as universe sets can not be represented. SpaceEx generates a warning when non-representable sets occur in this format. 2D plots are generated using an efficient overapproximation algorithm that samples high-dimensional sets using their support function representation. A bound on the approximation error (Hausdorff distance) can be specified by the option *output error*, which is by default zero. To speed up the output, a larger output error maybe chosen.

JVX format The JavaView JVX file format [6] can represent polyhedral continuous sets in 2D or 3D. It can be visualized by tools like JavaView [10] and JReality [12]. Note that the JReality viewer features an interface to the photo-realistic Sunflow render engine. Using the JReality Virtual Reality Viewer [9], the JVX files can be converted to other formats.

Since we can only plot concrete sets, the plots of implicit set representations used in the LGG scenario are based on overapproximations. Thus plots may be less accurate than the implicit sets used by the algorithm.
For GEN and JVX format, one can specify a list of variables onto which the output is projected. The order of the list corresponds to the order of the output values.

References

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

[2] Goran Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. PhD thesis, Radboud Universiteit Nijmegen, October 2005.

[3] Goran Frehse. PHAVer: algorithmic verification of hybrid systems past HyTech. *STTT*, 10(3):263–279, 2008.

[4] Goran Frehse. Tools for the verification of linear hybrid automata models. In Jan Lunze and Françoise Lamnabhi-Lagarrigue, editors, *Handbook of Hybrid Systems Control, Theory – Tools – Applications*. Cambridge University Press, 2009.

[5] Colas Le Guernic and Antoine Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250 – 262, 2010. IFAC World Congress 2008.

[6] Michael Joswig and Konrad Polthier. Javaview jvx format. `http://www.eg-models.de/formats/Format_Jvx.html`, 2000.

[7] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. Continuous modeling of real-time and hybrid systems: From concepts to tools. *STTT*, 1(1-2):64–85, 1997.

[8] Robert Maier and Nick Tufillaro. Gnu plotutils. `http://www.gnu.org/software/plotutils/`, 2000.

[9] Ulrich Pinkall and Stefan Sechelmann et al. Jreality virtual reality viewer. `http://www3.math.tu-berlin.de/jreality/index.php?article_id=21`, 2009.

[10] Konrad Polthier. Javaview. `http://www.javaview.de`, 2006.

[11] SpaceEx Web Site. `http://spaceex.imag.fr/`, 2010.

[12] Steffen Weißmann, Charles Gunn, Peter Brinkmann, Tim Hoffmann, and Ulrich Pinkall. jreality: a java library for real-time interactive 3d graphics and audio. In *Proceedings of the seventeen ACM international conference on Multimedia*, MM '09, pages 927–928, New York, NY, USA, 2009. ACM.