Computing Flowpipe of Nonlinear Hybrid Systems with Numerical Methods

Olivier Bouissou, Alexandre Chapoutot, Samuel Mimram

▶ **To cite this version:**

Olivier Bouissou, Alexandre Chapoutot, Samuel Mimram. Computing Flowpipe of Nonlinear Hybrid Systems with Numerical Methods. 2013. hal-00831438

HAL Id: hal-00831438
https://hal.science/hal-00831438

Submitted on 7 Jun 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Computing Flowpipe of Nonlinear Hybrid Systems with Numerical Methods

Olivier Bouissou and Samuel Mimram
CEA Saclay Nano-INNOV Institut CARNOT, Gif-sur-Yvette France

Alexandre Chapoutot
U2IS – ENSTA ParisTech, Palaiseau France

January 2013

Abstract

Modern control-command systems often include controllers that perform nonlinear computations to control a physical system, which can typically be described by an hybrid automaton containing high-dimensional systems of nonlinear differential equations. To prove safety of such systems, one must compute all the reachable sets from a given initial position, which might be uncertain (its value is not precisely known). On linear hybrid systems, efficient and precise techniques exist, but they fail to handle nonlinear flows or jump conditions. In this article, we present a new tool name HySon which computes the flowpipes of both linear and nonlinear hybrid systems using guaranteed generalization of classical efficient numerical simulation methods, including with variable integration step-size. In particular, we present an algorithm for detecting discrete events based on guaranteed interpolation polynomials that turns out to be both precise and efficient. Illustrations of the techniques developed in this article are given on representative examples.

1 Introduction

Modern control-command software for industrial systems are becoming more and more complex to design. On the one side, the description of the physical system that must be controlled, a power plant for instance, is frequently done using partial differential equations or nonlinear ordinary differential equations, whose number can grow very fast when one tries to have a precise model. On the other side, the complexity of the controller also increases when one wants it to be precise and efficient. In particular, adaptive controllers (which embed information on the plant dynamics) are more and more used: for such systems, the controller may need to compute approximations of the plant evolution using a look-up table or a simple approximation scheme as in [5]. As an extreme example, consider a controller for an air conditioning device in a car. In order to correctly and pleasantly regulate the temperature in the car, the controller takes information from the temperature of the engine but also from the outside temperature and the sunshine on the car. Based on these data, it acts on a cooling device, which is usually made of a hot and a cold fluid circuit, and is thus described using usual equations in fluid dynamics, which are given by high dimensional nonlinear differential equations.

In an industrial context, the design of such control-command systems is generally *validated* by performing numerical simulations of a high level description of the system using a Simulink like formalism. Usually, some input scenarios are defined and numerical simulation tools are used to observe the reaction of the system to these inputs and check that they are in accordance with the specifications. This methodology is widespread, because the methods for numerical simulation used nowadays are very powerful and efficient to approximate the behavior of complex dynamical systems, and scale very well w.r.t. to both complexity and dimension.

Simulation algorithms are mainly based on two parts: algorithms to compute approximations of the continuous evolutions of the system [23], and algorithms to compute switching times [25]. Matlab/Simulink is the de facto standard for the modeling and simulation of hybrid systems; we recall its basics principles in Section 2.2, and refer the reader to [2] for a complete formalization of its numerical engine.

The main drawback of simulation is that it cannot give strong guarantees on the behavior of a system, since it merely produces approximations of it for a finite subset of the possible inputs. To overcome this problem, verification techniques have been proposed on slightly different models of hybrid systems. The most popular and used technique is bounded model checking of hybrid automata [16, 18, 12, 11] that computes over-approximations of the set of reachable states of a hybrid system over a finite horizon. To apply such techniques on Simulink industrial systems, one must first translate it into the hybrid automata formalism (for example using techniques from [1]), and then apply some simplifications and linearizations to the model in order to obtain a linear hybrid automaton for which the good techniques exist [12]. This process of *linearization* can be performed automatically [7], but increase largely the number of discrete states (exponentially w.r.t. dimension), so that we believe that it is not applicable for large and highly nonlinear systems with stiff dynamics.

Contribution. In this article, we propose a new method to compute bounded horizon over-approximations of the trajectories of hybrid systems. This method improves our previous work [4] as it modifies numerical simulation algorithms to make them compute *guaranteed* bounds of the trajectories. Our algorithm is general enough to handle both nonlinear continuous dynamics and nonlinear jump conditions (also named zero-crossing events in Simulink). In short, our algorithm relies on two guaranteed methods: the continuous evolution is over-approximated using guaranteed integration of differential equations, using a generalization of [6], and the discrete jumps are solved using a new method (presented in Section 3.3) that can be seen as a guaranteed version of the zero-crossing algorithm of Simulink.

Related work. We already mentioned the work on reachability analysis in hybrid automata, either for the linear case [20, 12], or in the nonlinear case where a hybridization is used to construct an over-approximated linear automata [7]. Our approach is quite different as the algorithms we propose do not suppose anything about the differential equations and the jump conditions except their continuity w.r.t. state space variables. Previous works also used guaranteed numerical methods for reachability analysis of hybrid systems [17, 9]. These methods mainly use intervals as representation of sets, such as in the library VNODE [22], to compute guaranteed bounds on the continuous trajectories, and interval methods or a SAT solver to safely over-approximate the discrete jumps. Our method uses a more expressive domain for representing sets (affine forms [14, 4]) and polynomial interpolation for discrete jumps, which offers an efficient bisection method. Finally, the work closest to our is [24], in which a flowpipe for nonlinear hybrid systems is computed using a Taylor model to enclose the continuous behavior, and the discrete jumps are handled by doing the intersection of elements of the Taylor model and polyhedric guards. Compared to our approach, this work only allows for polynomial dynamics and polyhedral guards, while we have no such restrictions (as exemplified in Section 5). Beside, as will be clear from our benchmarks, the use of affine forms and numerical methods is generally more efficient than Taylor models.

Outline of the paper. The rest of this article is organized as follows. In Section 2, we present our formalism for hybrid systems and recall traditional method for their numerical simulation. Then, in Section 3, we explain how we could turn these methods into guaranteed methods that compute enclosures rather than approximations. In Section 4, we present our main algorithm for computing safe bounds on the trajectories of hybrid systems, and Section 5 presents some benchmarks that include both nonlinear dynamics and nonlinear jump conditions.

2 Preliminaries

2.1 Hybrid Automata

In this article, in order to facilitate the understanding of our method, we consider hybrid systems described as hybrid automata (HA). However, our tool HySon uses a slightly different representation as in our previous work [2, 4]. This state-space representation, comparable to the one used in [13], can encode both HA and Simulink models, as shown in [2]. We denote by $\mathbb{R}$ the set of real numbers, and by $\mathbb{B}$ the set of booleans (containing two elements, $\top$ meaning true and $\bot$ meaning false). Given a function $x : \mathbb{R} \to \mathbb{R}^n$, we denote by $x^-(t)$ its left-limit.

Definition 1 (Hybrid automaton, [16])**.** *An n-dimensional* hybrid automaton $\mathcal{H} = (L, F, E, G, R)$ *is a tuple such that L is a finite set of* locations*, the function $F : L \to (\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n)$ associates a* flow equation *to each location, $E \subseteq L \times L$ is a finite set of* edges*, $G : E \to (\mathbb{R}^n \to \mathbb{B})$ maps edges to* guards *and $R : E \to (\mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n)$ maps edges to* reset maps*.*

Notice that to simplify the presentation of our approach, we consider HA without invariants in each location, we will discuss this point in the conclusion. Besides, we assume that a transition $e = (l, l')$ is taken *as soon as* $G(e)$ is true.

Example 1. *We consider a modification of the classical bouncing-ball system that we call the* windy ball*: the ball is falling but there is in addition an horizontal wind which varies with time. So, the dynamics of the horizontal position x and height y of the ball are given by*

$$\dot{x}(t) = 10(1 + 1.5\sin(10t)) \qquad \dot{y}(t) = v_y(t) \qquad \dot{v_y}(t) = -g$$

The HA *thus has only one location l such that $F(l)$ is the above flow. There is also one edge $e = (l, l)$ for when the ball bounces on the floor, with a guard $G(e) = y \leq 0$ and a reset $R(e) = (x, y, v_y) \mapsto (x, y, -0.8v_y)$.*

The operational semantics [16] of an HA is a transition system with two kinds of transitions for the time elapse and the discrete jumps. From this operational semantics we can define the trajectories of the HA, as in [13].

Definition 2 (Trajectory of an hybrid automaton)**.** *Suppose fixed an* HA $\mathcal{H} = (L, F, E, G, R)$*. A* state *of $\mathcal{H}$ is a couple (x, l) with $x \in \mathbb{R}^n$ and $l \in L$. A* trajectory *of $\mathcal{H}$, on the time interval $[t_0, t_f]$, starting from an initial state (x_0, l_0), is a pair of functions (x, l) with $x : [t_0, t_f] \to \mathbb{R}^n$ and $l : [t_0, t_f] \to L$, such that there exists time instants $t_0 \leq t_1 \leq \ldots \leq t_n = t_f$ satisfying, for every index i,*

1. *x is continuous and l is constant on $[t_i, t_{i+1}[$,*
2. *$x(0) = x_0$, $l(0) = l_0$,*
3. *$\forall t \in [t_i, t_{i+1}[$, $\dot{x}(t) = F(l(t))(t, x(t))$,*
4. *$\forall t \in]t_i, t_{i+1}[$, $\forall e = (l(t), l') \in E$, $G(e)(x(t)) = \bot$,*
5. *$G(e)(x^-(t_i)) = \top$ with $e = (l^-(t_i), l(t_i))$ and $x(t_i) = R(e)(t_i, x^-(t_i))$.*

In the above definition, the equations constraint the function x so that it conforms to the flow and jump conditions of $\mathcal{H}$. Equation 2 ensures that x satisfies the initial conditions, Eq. 3 specifies that the dynamics of $x(t)$ is the flow at location $l(t)$, Eq. 4 and 5 ensure that the t_i are the instants where jumping conditions occur and that x evolves as described by reset maps when the corresponding guard is satisfied. Notice that we do not consider Zeno phenomena here as we assume that there are finitely many jumps between t_0 and t_f. Also, we do not discuss conditions ensuring existence and unicity of trajectories as this is beyond the scope of this paper [15], but implicitly suppose that these are granted. We suppose fixed initial and terminal simulation times t_0 and t_f. Given an HA $\mathcal{H}$ and an initial state (x_0, l_0), we denote by $Reach_{\mathcal{H}}(x_0, l_0)$ the continuous trajectory on $[t_0, t_f]$ as defined above, and given $X_0 \subseteq \mathbb{R}^n$ and $L_0 \subseteq L$, we define $Reach_{\mathcal{H}}(S_0, L_0) = \bigcup_{x_0 \in X_0, l_0 \in L_0} Reach_{\mathcal{H}}(x_0, l_0)$.

Computing the set $Reach_{\mathcal{H}}(X_0, L_0)$ for an HA $\mathcal{H}$ is sufficient in order to decide the reachability of some region in the state space, and thus often to prove its safety (for bounded time). As trajectories are in general not computable, over-approximations must be performed: this is the goal of our algorithm presented in Sections 3 and 4. In Section 2.2, we present numerical algorithms, used for example by Simulink, that allow to compute *approximations* of the set $Reach_{\mathcal{H}}(x_0, l_0)$ for some initial state $(x_0, l_0) \in \mathbb{R}^n \times L$. In Section 3 we present how we can adapt these methods in order to be safe w.r.t. the exact trajectories of $\mathcal{H}$.

2.2 Numerical Simulation

Numerical simulation aims at producing discrete approximations of the trajectories of an hybrid system $\mathcal{H}$ on the time interval $[t_0, t_f]$. We described in details in [2] how the simulation engine of Simulink operates, and briefly adapt here this simulation engine to HA.

Suppose that $\mathcal{H}$ is an HA, (x_0, l_0) an initial state, and $(x(t), l(t))$ a trajectory of $\mathcal{H}$ starting from (x_0, l_0). A numerical simulation algorithm computes a sequence $(t_k, x_k, l_k)_{k\in[0,N]}$ of time instants, variables values and locations such that $\forall k \in [0, N],\ x_k \approx x(t_k)$. Most of the difficulty lies in approximating the discrete jumps (instants where a guard becomes true), which are called *zero-crossings* in the numerical simulation community. In order to compute (t_k, x_k, l_k), the following simulation loop is used, where h_k is the simulation step-size (that can be modified to a smaller value in order to maintain a good precision):

1: **repeat**
2: $x_{k+1} \leftarrow \text{SolveODE}(F(l(t_k), x_k, h_k)$ ▷*Solver step 1*
3: $(x_{k+1}, l_{k+1}) \leftarrow \text{SolveZC}(x_k, x_{k+1})$ ▷*Solver step 2*
4: compute h_{k+1}
5: $k \leftarrow k + 1$
6: **until** $t_k \geq t_f$

In this simulation loop, the solver first makes a continuous transition between instants t_k and $t_k + h_k$ under the assumption that no jump occurs (solver step 1), and then it verifies this assumption (solver step 2). If it turns out that there was a jump between t_k and $t_k + h_k$, the solver approximates as precisely as possible the time $t \in [t_k, t_k + h_k]$ at which this jump occurred. We briefly detail both steps in the rest of this section.

Solver step 1. The continuous evolution of x between t_k and $t_k + h_k$ is described by $\dot{x}(t) = F(l(t_k))\big(t, x(t)\big)$ and $x(t_k) = x_k$. So, we want to compute an approximation of the solution at $t_k + h_k$ of the initial value problem (IVP), with $f = F(l(t_k))$:

$$\dot{x}(t) = f(t, x(t)) \qquad\qquad x(t_k) = x_k \tag{1}$$

(we assume classical hypotheses on f ensuring existence and uniqueness of a solution of IVP). Usually, precise simulation algorithms often rely on a variable step solver, for which (h_k) is not constant. The simplest is probably the Bogacki-Shampine method [23], also named ODE23. It computes x_{k+1} by

$$k_1 = f(t_k, x_k) \qquad k_2 = f(t_k + \frac{h_k}{2}, x_k + \frac{h_k}{2}k_1) \qquad k_3 = f(t_k + \frac{3h_k}{4}, x_k + \frac{3h_k}{4}k_2) \tag{2a}$$

$$x_{k+1} = x_k + \frac{h_k}{9}\left(2k_1 + 3k_2 + 4k_3\right) \tag{2b}$$

$$k_4 = f(t_k + h_k, x_{k+1}) \qquad\qquad z_{k+1} = x_k + \frac{h_k}{24}\left(7k_1 + 6k_2 + 8k_3 + 3k_4\right) \tag{2c}$$

The value z_{k+1} defined in (2c) is a third order approximation of $x(t_k + h_k)$, whereas x_{k+1} is a second order approximation of this value, and is used to estimate the error $\text{err} = |x_{k+1} - z_{k+1}|$. This error is compared to a given *tolerance* tol and the step-size is changed accordingly: if the error is smaller then the step is *validated* and the step-size increased in order to speed up computations (in ODE23, next step-size is computed with $h_{k+1} = h_k \sqrt[3]{\text{tol}/\text{err}}$), if the error is greater then the step is *rejected* and the computation is tried again with the smaller step-size $h_k/2$. We refer to [15, p. 167] for a complete description on such numerical methods.

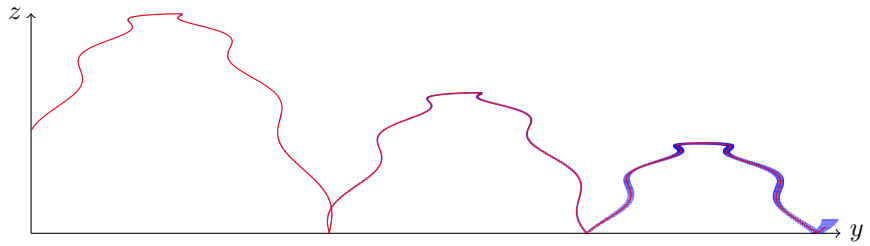
Solver step 2. Once x_k and x_{k+1} computed, the solver checks if there were a jump in the time interval $[t_k, t_{k+1}]$. In order to do so, it tests for each edge e starting from l_k whether $G(e)(x_k)$ is false and $G(e)(x_{k+1})$ is true. If there is no such edge, then it is considered that no jump occurred, we set $l_{k+1} = l_k$ and continue the simulation. Notice this technique does not guarantee the detection of all events occurring between $[t_k, t_{k+1}]$ as explained in [25] or [10].

If the solver finds such an edge, this means that there was a jump on $[t_k, t_{k+1}]$ and we must approximate the first time instant $\xi \in [t_k, t_k + h_k]$ such that $G(e)(x(\xi))$ is true. To do so, the solver encloses ξ in an interval $[t_l, t_r]$ starting with $t_l = t_k$ and $t_r = t_k + h_k$, and reduces this interval until the time precision $|t_l - t_r|$ is smaller than some parameter. To reduce the width of the interval, the solver first makes a guess for ξ using a linear extrapolation and then computes an approximation of $x(\xi)$ using a polynomial interpolation of x on $[t_k, t_k + h_k]$. Depending on $G(e)(x(\xi))$, it then sets $t_l = \xi$ or $t_r = \xi$ and starts over. In the case of Hermite interpolation (which is the method used together with the ODE23 solver), the polynomial interpolation is given, for $t \in [t_k, t_k + h_k]$, by

$$x(t) \approx (2\tau^3 - 3\tau^2 + 1)x_k + (\tau^3 - 2\tau^2 + \tau)h_k\dot{x}_k + (-2\tau^3 + 3\tau^2)x_{k+1} + (\tau^3 - \tau^2)h_k\dot{x}_{k+1} \tag{3}$$

where $\tau = (t - t_k)/h_k$, and $\dot{x}_k$, $\dot{x}_{k+1}$ are approximations of the derivative of x at t_k, t_{k+1}. For more details on zero-crossing algorithms, we refer to [2, 25].

Example 2. *Consider the windy ball again (Example 1). The red curve below is the result of the simulation for $t \in [0, 13]$ using Simulink. In blue is the flowpipe computed by HySon whose computation is going to be described in next sections.*



3 Guaranteed Simulation Methods

The elaboration of our algorithm consisted essentially in adapting simulation algorithms such as the one described in Section 2 in order to (i) compute with *sets of values* instead of values, and (ii) ensure that the resulting algorithm is *guaranteed* in the sense that the set $\hat{x}_k$ of values computed for x at instant t_k always contains the value of the mathematical solution at instant t_k. This means that we have to take in account numerical errors due to the integration method and the use of floats (see Section 3), and design an algorithm computing an over-approximation of jump times (Section 4). In this section, we first briefly present our encoding of sets using affine arithmetic (Section 3.1) and show how explicit Runge-Kutta like numerical integration methods (Section 3.2) and the polynomial interpolation (Section 3.3) can be turned into guaranteed algorithms.

3.1 Computing with Sets

The simplest and most common way to represent and manipulate sets of values is *interval arithmetic* [21]. Nevertheless, this representation usually produces too much over-approximated results, because it cannot take dependencies between variables in account: for instance, if $x = [0, 1]$, then $x - x = [-1, 1] \neq 0$. More generally, it can be shown for most integration schemes that the width of the result can only grow if we interpret sets of values as intervals.

To avoid this problem we use an improvement over interval arithmetic named *affine arithmetic* [8] which can track linear correlations between program variables. A set of values in this domain is represented by an

affine form $\hat{x}$ (also called a *zonotope*), which is a formal expression of the form $\hat{x} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ where the coefficients α_i are real numbers, α_0 being called the *center* of the affine form, and the ε_i are formal variables ranging over the interval $[-1, 1]$. Obviously, an interval $a = [a_1, a_2]$ can be seen as the affine form $\hat{x} = \alpha_0 + \alpha_1 \varepsilon$ with $\alpha_0 = (a_1 + a_2)/2$ and $\alpha_1 = (a_2 - a_1)/2$. Moreover, affine forms encode linear dependencies between variables: if $x \in [a_1, a_2]$ and y is such that $y = 2x$, then x will be represented by the affine form $\hat{x}$ above and y will be represented as $\hat{y} = 2\alpha_0 + 2\alpha_1 \varepsilon$.

Usual operations on real numbers extend to affine arithmetic in the expected way. For instance, if $\hat{x} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ and $\hat{y} = \beta_0 + \sum_{i=1}^{n} \beta_i \varepsilon_i$, then with $a, b, c \in \mathbb{R}$ we have $a\hat{x} + b\hat{y} + c = (a\alpha_0 + b\beta_0 + c) + \sum_{i=1}^{n} (a\alpha_i + b\beta_i)\varepsilon_i$. However, unlike the addition, most operations create new noise symbols. Multiplication for example is defined by $\hat{x} \times \hat{y} = \alpha_0\alpha_1 + \sum_{i=1}^{n} (\alpha_i\beta_0 + \alpha_0\beta_i)\varepsilon_i + \nu\varepsilon_{n+1}$, where $\nu = \left(\sum_{i=1}^{n} |\alpha_i|\right) \times \left(\sum_{i=1}^{n} |\beta_i|\right)$ over-approximates the error between the linear approximation of multiplication and multiplication itself. Other operations, like sin, exp, are evaluated using their Taylor expansions. The set-based evaluation of an expression only consists in interpreting all the mathematical operators (such as + or sin) by their counterpart in affine arithmetic. We will denote by $\text{Aff}(e)$ the evaluation of the expression e using affine arithmetic, see [4] for practical implementation details.

3.2 Guaranteed Numerical Integration

Recall from Section 2 that a numerical integration method computes a sequence of approximations (t_n, x_n) of the solution $x(t; x_0)$ of the IVP defined in (1) such that $x_n \approx x(t_n; x_0)$. Every numerical method member of the Runge-Kutta family follows the *condition order* [15, Chap. II.2, Thm. 2.13]. This condition states that a method is of order p if and only if the $p + 1$ first coefficients of the Taylor expansion of the true solution and the Taylor expansion of the numerical method are equal. The *truncation error* measures the distance between the true solution and the numerical solution and it is defined by $x(t_n; x_0) - x_n$. Using the condition order, it can be shown that this truncation error is proportional to the Lagrange remainders. We now briefly recall our approach to make any explicit Runge-Kutta method guaranteed, which is based on this observation, see [3] for a detailed presentation.

The general form of an explicit *s-stage Runge-Kutta formula* (using s evaluations of f) is

$$x_{n+1} = x_n + h \sum_{i=1}^{s} b_i k_i \qquad \text{with} \qquad k_i = f\left(t_n + c_i h,\, x_n + h \sum_{j=1}^{i-1} a_{ij} k_j\right)$$

for $1 \leq i \leq s$. The coefficients c_i, a_{ij} and b_i are usually summarized in a Butcher table (see [15]) which fully characterizes a Runge-Kutta method. We denote by $\phi(t) = x_n + h_t \sum_{i=1}^{s} b_i k_i(t)$, where $k_i(t)$ is defined as previously with h replaced by $h_t = t - t_n$. Hence the truncation error is defined by

$$x(t_n; x_0) - x_n \quad = \quad \frac{h_n^{p+1}}{(p+1)!} \left(f^{(p)}\left(\xi, x(\xi)\right) - \frac{\mathrm{d}^{p+1}\phi}{\mathrm{d}t^{p+1}}(\eta) \right) \tag{4}$$

for some $\xi \in]t_k, t_{k+1}[$ and $\eta \in]t_n, t_{n+1}[$. In (4), $f^{(p)}$ stands for the p-th derivative of function f w.r.t. time t, and $h_n = t_{n+1} - t_n$ is the step size. In (4), the Lagrange remainder of the exact solution is $f^{(p)}\left(\xi, x(\xi; x_0)\right)$ and the Lagrange remainder of the numerical solution is $\frac{d^{p+1}\phi}{dt^{p+1}}(\eta)$.

The challenge to make Runge-Kutta integration schemes safe w.r.t. the exact solution of IVP amounts to bounding the result of (4). The remainder $\frac{d^{p+1}\phi}{dt^{p+1}}(\eta)$ is straightforward to bound because the function ϕ only depends on the value of the step size h, and so does its $(p+1)$-th derivative:

$$\frac{\mathrm{d}^{p+1}\phi}{\mathrm{d}t^{p+1}}(\eta) \in \text{Aff}\left(\frac{\mathrm{d}^{p+1}\phi}{\mathrm{d}t^{p+1}}([t_n, t_{n+1}]) \right) \tag{5}$$

However, the expression $f^{(p)}\left(\xi, x(\xi; x_0)\right)$ is not so easy to bound as it requires to evaluate f for a particular value of the IVP solution $x(\xi; x_0)$ at a unknown time $\xi \in]t_n, t_{n+1}[$. The solution we used is similar to the one found in [22, 6]: we first compute an a priori enclosure of the IVP on the interval $[t_n, t_{n+1}]$. To do so, we use the

Banach fixed-point theorem on the Picard-Lindelöf operator P, defined by $P(x, t_n, x_n) = t \mapsto x_n + \int_{t_n}^{t} f(s, x(s))\mathrm{d}s$. Notice that this operator is the integral form of (1), so a fixpoint of this operator is also a solution of (1).

Now, to get an a priori enclosure of the solution over $[t_n, t_{n+1}]$, we prove that the operator P (which is an operator on functions) is contracting and use Banach theorem to deduce that it has a fixpoint. To find the enclosure $\hat{z}$ on the solution, we thus iteratively solve using affine arithmetic the equation $P(\hat{z}, t_n, x_n)([t_n, t_{n+1}]) \subseteq \hat{z}$. Then, we know that the set of functions $[t_n, t_{n+1}] \to \hat{z}$ contains the solution of the IVP, so $\hat{z}$ can be used an enclosure of the solution of IVP over the time interval $[t_n, t_{n+1}]$. We can hence bound the Lagrange remainder of the true solution with $\hat{z}$ such that

$$f^{(p)}\left(\xi, x(\xi; x_0)\right) \in \mathrm{Aff}\left(f^{(p)}\left([t_n, t_{n+1}], \hat{z}\right)\right) \tag{6}$$

Finally, using (5) and (6) we can prove Theorem 1 and thus bound the distance between the approximations point of any explicit Runge-Kutta method and any solution of the IVP.

Theorem 1. *Suppose that Φ is a numerical integration scheme and Φ_{Aff} is the evaluation of Φ using affine arithmetic. Given a set $S_0 \subseteq \mathbb{R}^n$ of initial states, and an affine form $\hat{x}_0$ such that $S_0 \subseteq \hat{x}_0$, let $(t_n, \hat{x}_n)$ be a sequence of time instants and affine forms defined by $\hat{x}_{n+1} = \hat{x}'_{n+1} + \hat{e}_{n+1}$ where $(t_{n+1}, \hat{x}'_{n+1}) = \Phi_{\mathrm{Aff}}(t_n, \hat{x}_n)$ and $\hat{e}_{n+1}$ is the truncation error as defined by (4) and is evaluated using (5) and (6). Then, for any $x \in S_0$ and $n \in \mathbb{N}$ we have $x(t_n; x) \in \hat{x}_n$.*

3.3 Guaranteed Polynomial Interpolation

From two (guaranteed) solutions x_n, x_{n+1} at times t_n, t_{n+1} of an IVP, one would like to deduce by interpolation all the solutions $x(t)$ with $t \in [t_n, t_{n+1}]$. This question has motivated a series of work on polynomial approximations of solutions, a.k.a. *continuous extension*, see [15, Chap. 6]. We briefly recall the polynomial interpolation method based on Hermite-Birkhoff which is the main method used for continuous extension. Furthermore, we present a new extension of this method allowing us to compute a guaranteed polynomial interpolation using the result of the Picard-Lindelöf operator.

Suppose given a sequence $(t_i, x_i^{(k)})$ of $n+1$ computed values of the solution of an IVP and its derivative at instants t_i, with $0 \leq i \leq n$ and $k = 0, 1$. Remark that these values are those produced by numerical integration methods. The goal of Hermite-Birkhoff polynomial interpolation is to build a polynomial function $p(t) = \sum_{i=0}^{n} \left(x_i A_i(t) + x_i^{(1)} B_i(t)\right)$ of degree $N = 2n+1$ from these values such that $A_i(t) = \left(1 - 2(t - t_i)\ell_i'(t_i)\right)\ell_i^2(t)$, $B_i(t) = (t - t_i)\ell_i^2(t)$, $\ell_i(t) = \prod_{j=0, j\neq i}^{n} \frac{t - t_j}{t_i - t_j}$, and $\ell_i'(t_i) = \sum_{k=0, k\neq i}^{n} \frac{1}{t_i - t_k}$: the functions $\ell_i(t)$ are the Lagrange polynomials and this interpolation generalizes the Lagrange interpolation. Under the assumption that all the t_i are distinct, we know that the polynomial interpolation is unique. For instance, Eq. (3) is associated to the Hermite-Birkhoff polynomial with $n = 1$. We know that interpolation error $x(t; x_0) - p(t)$ is defined by $\frac{x^{(N+1)}(\xi)}{(N+1)!} \prod_{i=0}^{n} (t - t_i)^2$ with $\xi \in [t_0, t_n]$, which can be reformulated as

$$x(t; x_0) - p(t) \quad = \quad \frac{f^{(N)}(\xi, x(\xi))}{(N+1)!} \prod_{i=0}^{n} (t - t_i)^2 \qquad \text{with } \xi \in [t_k, t_{k+1}]$$

In consequence, to guarantee the polynomial interpolation, it is enough to know an enclosure of the solution $x(t)$ of IVP on the interval $[t_k, t_{k+1}]$. And fortunately, we can reuse the result of the Picard-Lindelöf operator in that context. In next section, this guaranteed polynomial interpolation will be used to approximate the solution of an IVP in order to compute jump times.

Theorem 2. *Let $p_{\mathrm{Aff}}(t)$ be the interpolation polynomial based on $n+1$ guaranteed solutions $\hat{x}_i$ of an* IVP (1) *and $n+1$ evaluations $\hat{x}_i^{(1)}$ of f with affine arithmetic, and let $\hat{z}$ be the result of the Picard-Lindelöf operator. We have,*

$$\forall t \in [t_k, t_{k+1}], \quad x(t) \in \mathrm{Aff}\left(p_{\mathrm{Aff}}(t) + \frac{f^{(N)}([t_k, t_{k+1}], \hat{z})}{(N+1)!} \prod_{i=0}^{n} (t - t_i)^2\right)$$

Algorithm 1 Guaranteed simulation algorithm

Require: $\mathcal{H} = (L, F, E, G, R)$, a, hybrid automaton

Require: $\hat{x}_0$, l_0, h_0, t_f ▷*Initial state, step-size and final time*

1: $n \leftarrow 0$
2: $\hat{t}_n \leftarrow 0$
3: **while** $\inf(\hat{t}_n) \leq t_f$ **do**
4: $\quad (\hat{x}_{n+1}, \hat{x}^h_n) \leftarrow \text{GSolveODE}(F(l_n), \hat{x}_n, h_n)$
5: $\quad (\hat{x}_{n+1}, \hat{t}_{n+1}, l_{n+1}) \leftarrow \text{GSolveZC}(l_n, \hat{x}_n, \hat{x}_{n+1}, \hat{x}^h_n, \hat{t}_n, h_n)$
6: $\quad n \leftarrow n + 1$
7: **end while**

4 Reachability Algorithm

We present in this section our main algorithm to compute an over-approximation of the set of reachable states of linear or nonlinear hybrid systems (Algorithm 1), which is based on the guaranteed numerical methods presented in Section 3. In a nutshell, it works as follows. It produces a sequence of values $(\hat{t}_n, \hat{x}^h_n, \hat{x}_n, l_n)$ such that l_n is the current location, $\hat{t}_n$ is a time interval, $\hat{x}_n$ is an over-approximation of $x(t)$ for every $t \in \hat{t}_n$, and $\hat{x}^h_n$ is an over-approximation of $x(t)$ for every $t \in [\hat{t}_n, \hat{t}_{n+1}]$, i.e. an over-approximation of the trajectory between two discrete instants (here $[\hat{t}_n, \hat{t}_{n+1}]$ designates the convex hull of the union of the two affine forms $\hat{t}_n$ and $\hat{t}_{n+1}$). Our method uses the guaranteed ODE solver described in Section 3.2 to compute $\hat{x}_{n+1}$ and $\hat{x}^h_n$, and the guaranteed polynomial interpolation of Section 3.3 to precisely and safely enclose the potential jumping times between t_n and t_{n+1}, and thus refine t_{n+1} and $\hat{x}_{n+1}$.

Trivalent Logic. First, notice that since we are working with sets of values, the evaluation of a boolean condition, such as $x \geq 0$, is not necessarily false or true, but can also be false for some elements and true for some other elements in the set $\hat{x}$ (for instance when $\hat{x} = [-1, 1]$ in the preceding example). In order to take this in account, boolean conditions are evaluated in the domain of *trivalent logic* instead of usual booleans $\mathbb{B}$. This logic is the natural extension of boolean algebra to the three following values: $\bot$ (false), $\top$ (true) and $\mathtt{I}$ (unknown). We denote this set by $\mathbb{B}^*$. Notice that a function $g : \mathbb{R}^n \to \mathbb{B}$ naturally extends to a function $\text{Aff}(g) : \mathcal{P}(\mathbb{R}^n) \to \mathbb{B}^*$ using affine arithmetic and trivalent logic. In particular, the guards of the discrete jumps will be evaluated in $\mathbb{B}^*$, which brings subtleties in the zero-crossing detection algorithm (when such a guard evaluates to $\mathtt{I}$), as we will see in next section. In the following, we shall write g for $\text{Aff}(g)$ when it is clear from the context.

Main Algorithm. Let $\mathcal{H}$ be an HA as defined in Definition 1. Our method computes a sequence of values $(\hat{t}_n, \hat{x}_n, \hat{x}^h_n, l_n)$ such that l_n is the current mode of the HA, t_n is a time interval and $\hat{x}_n$ and $\hat{x}^h_n$ are affine forms such that we have

$$\forall t \in \hat{t}_n \;\; x(t) \in \hat{x}_n \quad \forall t \in [\hat{t}_n, \hat{t}_{n+1}], \qquad x(t) \in \hat{x}^h_n$$

for all trajectories of $\mathcal{H}$. To compute this sequence, we start from $\hat{t}_0 = 0$ and iterate until the lower bound of $\hat{t}_n$ (denoted $\inf(\hat{t}_n)$) is lower than t_f. The guaranteed simulation loop is given in Algorithm 1, where GSolveODE is the guaranteed solver of ODE presented in Section 3.2 and GSolveZC is the procedure described below. Notice that the function GSolveODE outputs both $\hat{x}_{n+1}$, the tight over-approximation of x at $\hat{t}_n + h_n$, and $\hat{x}^h_n$, the result of Picard iteration (see Section 3.2) since we reuse it in GSolveZC.

Detecting Jumps. We now present our algorithm (GSolveZC) for detecting and handling discrete jumps. Let $\mathcal{H} = (L, F, E, G, R)$ be an HA, and let l_n, $\hat{x}_n$, $\hat{x}_{n+1}$ and $\hat{x}^h_n$ be the states computed with GSolveODE. Let us denote $l^\bullet_n$ the set of all transitions originating from l_n, i.e. $l^\bullet_n = \{e \in E \,|\, \exists l \in L,\ e = (l_n, l)\}$. A transition $e \in l^\bullet_n$ was *surely* activated between t_n and $t_n + h_n$ if $G(e)(\hat{x}_n) = \bot$ and $G(e)(\hat{x}_{n+1}) = \top$. The transition e was *maybe* activated if if $G(e)(\hat{x}_n) = \bot$ and $G(e)(\hat{x}_{n+1}) = \mathtt{I}$. Note that in both cases we have $G(e)(\hat{x}^h_n) = \mathtt{I}$. In this section, we present our algorithm in the simple (but most common) case where we have only one transition activated

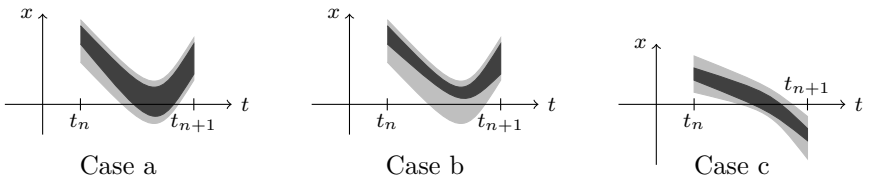


Figure 1: Three cases for discrete transitions. Exact trajectories are depicted in dark gray, the over-approximated flow pipes in light gray.

at a given time, and where we are not in the situation of $G(e)(\hat{x}_n) = G(e)(\hat{x}_{n+1}) = \bot$ with $G(e)(\hat{x}_n^h) = \mathbb{I}$; we discuss these two cases later.

The function GSolveZC is described in Algorithm 2 and runs as follows. First, if for all edges $e \in l_n^\bullet$, $G(e)(\hat{x}_n^h) = \bot$, then no transition was activated between t_n and t_{n+1}, and we do nothing (lines 2–4). Otherwise, if there is $e \in l_n^\bullet$ that may have been activated, then we make sure that we have $G(e)(\hat{x}_{n+}) = \top$, i.e. that the event really occurred between $\hat{t}_n$ and $\hat{t}_{n+1}$ (this is the case (c) in Figure 1, other cases are handled as "special cases" below), which is achieved by continuing the guaranteed integration of $F(l_n)$ until we have $G(e)(\hat{x}_{n+1})$. This is the role of the while loop (lines 6–10), in which we also compute the hull of all Picard over-approximations computed during this process. Then, we are sure that e occurred between $\hat{x}_n$ and $\hat{x}_{n+1}$. We then reduce the time interval $[\hat{t}_n, \hat{t}_{n+1}]$ in order to precisely enclose the time $\hat{t}_{zc}$ at which the condition $G(e)$ became true (line 11). To do so, we use the guaranteed polynomial extrapolation p of Section 3.3 to approximate the value of x between $\hat{t}_n$ and $\hat{t}_{n+1}$ without having to call GSolveODE, and use a bisection algorithm to find the lower and upper limits of $\hat{t}_{zc}$.

To get the lower limit (the upper limit is obtained similarly), the bisection algorithm perform as follows. We start with a working list containing $[\hat{t}_n, \hat{t}_{n+1}]$, the convex hull of both time stamps. Then, we pick the first element $\hat{t}$ of the working list and evaluate p on it. If $p(\hat{t}) = \mathbb{I}$ and the width of $\hat{t}$ is larger than the desired precision, we split $\hat{t}$ into $\hat{t}_1$ and $\hat{t}_2$ and add them to the working list. If the width $\hat{t}$ is smaller than the precision, we return $\hat{t}$. If $p(\hat{t}) = \bot$, we discard $\hat{t}$ and continue with the rest of the working list. Note that we cannot have $p(\hat{t}) = \top$. The method to find the upper limit is the same, except that we discard $\hat{t}$ if $p(\hat{t}) = \top$.

Finally, once we have $\hat{t}_{zc}$, we use the guaranteed polynomial again to compute the zero-crossing state $\hat{x}_{zc} = p(\hat{t}_{zc})$ and set $\hat{x}_{n+1} = R(e)(\hat{x}_{zc})$, i.e. we apply the reset map.

Notice that our algorithm needs to maintain the invariant $G(e)(\hat{x}_n) = \bot$ for all $e \in l_n^\bullet$. This imposes that we sometimes have a particular formulation for zero-crossing conditions. For instance, the guard and reset functions of the windy ball of example 1 should be reformulated as $G(e) = x < 0$ and $R(e)(x, y, v) = (x, 0, -0.8v)$. Under this new formulation, just after the zero-crossing action has been performed, we have $x = 0$ and therefore the zero-crossing condition $x < 0$ is not true. Otherwise, with the first formulation, the simulation will fail at first zero-crossing. The transformation is performed automatically for usual conditions in HySon. Note also that it may be the case that there exist $e' \in l_{n+1}^\bullet$ such that $G(e')(\hat{x}_{n+1}) \neq \bot$, i.e. a transition starting from l_{n+1} may be activated by $\hat{x}_{n+1}$. In this case, we execute the transition immediately after e, and continue until we arrive in a location l such that no transition starting from l is activated. We assume that such l exists, which is true if the HA $\mathcal{H}$ does not have Zeno behavior.

Special Cases. If there is more than one transition activated during the step from t_k to t_{k+1}, we first reject the step and continue with a reduced step-size. This way, we shall eventually reach a step-size where only one condition is activated and not the other. If we cannot separate both transitions before reaching a minimal step-size, we use our previous algorithm on both transitions separately, apply both reset maps and then we follow both possible trajectories, i.e. we have a disjunctive analysis when we are not sure of the location.

Finally, we shall discuss the case when the state at times t_k and t_{k+1} do not verify the guard of a transition e but the hull computed by Picard iteration does (see Figure 1, cases a and b). Then, either the trajectories between t_k and t_{k+1} cross twice the guard boundary and we missed a zero-crossing, (case a) or it is the

Algorithm 2 Guaranteed Zero-crossing algorithm

Require: $\mathcal{H} = (L, F, E, G, R)$, a hybrid automaton

function GSolveZC$(\hat{x}_n, \hat{x}_{n+1}, \hat{x}_n^h, t_n, h_n, l_n)$
  if $\forall e \in l_n^\bullet,\ G(e)(\hat{x}_n^h) = \bot$ **then**
    return $\hat{x}_{n+1}, l_n, t_n + h_n$ ▷*No jumps*
  end if
  Let $e = (l_n, l_{n+1}) \in l_n^\bullet$ be such that $G(e)(\hat{x}_n^h) = \mathbb{I}$
  while $G(e)(\hat{x}_{n+1}) \neq \top$ **do**
    $(\hat{x}_{n+1}, \hat{x}^h) \leftarrow$ GSolveODE$(F(l_n), \hat{x}_{n+1}, h_n)$
    $x_n^h \leftarrow x_n^h \cup x^h$
    $h_n \leftarrow h_n + h_n$
  end while ▷*Now* $G(e)(\hat{x}_n) = \bot$ *and* $G(e)(\hat{x}_{n+1}) = \top$
  $t_{zc} \leftarrow$ tightInterval$(\hat{x}_n, \hat{x}_{n+1}, t_n, t_{n+1})$
  $x_{zc} \leftarrow$ GPolyODE$(\hat{x}_n, \hat{x}_{n+1}, \hat{x}_n^h, t_{zc})$
  return $\big(R(e)(\hat{x}_{zc}), l_{n+1}, t_{zc}\big)$
end function

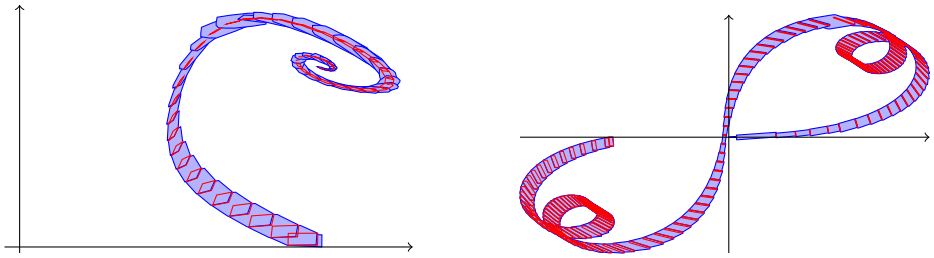


Figure 2: Over-approximation of the trajectories of the Brusselator (left) and Car (right) systems. The blue sets are the over-approximations for all t (given by Picard iteration) and the red sets are the tight enclosures at the discretization time stamps.

over-approximation due to Picard iteration which makes the guard validated (case b). We use again our bisection algorithm to distinguish between these two cases and perform a disjunctive analysis if we cannot differentiate between them.

5 Experimentation

We implemented our method in a tool named HySon. It is written in OCaml and takes as input a representation of a hybrid system either using a set of equations similar to the ones defined in [2] or a Simulink model (for now without stateflow support). We first present the output of HySon on some continuous or hybrid systems, and then we compare the performances of HySon with other tools.

5.1 Continuous Systems

Brusselator. We consider the following system, also used in [24]:

$$\dot{x} = 1 + x^2 y - 2.5x \qquad \dot{y} = 1.5x - x^2 y \qquad x(0) \in [0.9, 1] \qquad y(0) \in [0, 0.1]$$

HySon computes the flowpipe up to $t = 15$ in 14.3s, see Figure 2, left.

Car. We consider the initial value problem given by:

$$\dot{x} = v\cos(0.2t)\cos(\theta) \qquad \dot{y} = v\cos(0.2t)\sin(\theta) \qquad x(0) = 0 \qquad y(0) = 0$$
$$\dot{\theta} = v\sin(0.2t)/5 \qquad \theta(0) = [0, 0.1]$$

HySon computes the flowpipe up to $t = 30$ in 55.9s, see Figure 2, right.

5.2 Hybrid Systems

We now present two hybrid systems: a ball bouncing on a sinusoidal floor and a non-linear system with a polynomial jump condition.

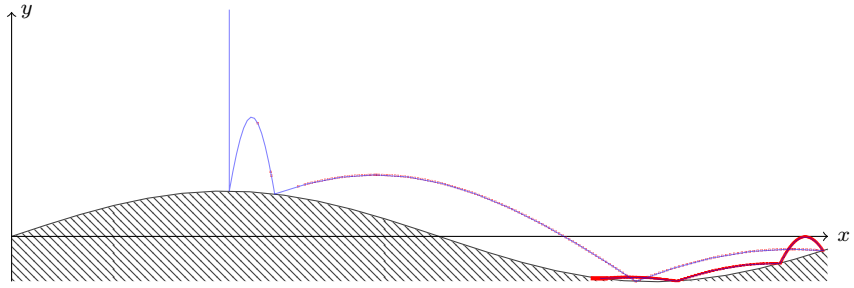
Ball bouncing on a sinusoidal floor. A ball is falling on a sinusoidal floor, and we consider a dynamics with non-linear wind friction for the ball. The dynamics of the system is given by

$$\dot{v}_x = 0 \qquad \dot{x} = v_x \qquad \dot{v}_y = -g + kv_y^2 \qquad \dot{y} = v_y$$

starting from the initial conditions $x(0) = 1.6$, $v_x(0) = 0$, $y(0) = 5$ and $v_y(0) = -5$. The bouncing of the ball is given by the transition:

$$\begin{pmatrix} v_x = e(v_d - v_x) \\ v_y = e(v_d\cos(x) - v_y) \\ y = \sin(x) \end{pmatrix} \qquad \text{when } y < \sin(x)$$

with $v_d = (v_x + v_y\cos(x))/(1 + \cos(x)^2)$, where $g = 9.8$, $k = 0.3$ and $e = 0.8$. Note that the exact dynamics of this system is almost chaotic. HySon is able to compute flow-pipe for this system, as shown on the following figure.



Wolfgram. We study the following system, with $a = 2$:

$$\dot{x}(t) = \begin{cases} t^2 + 2x & \text{if } (x + 3/20)^2 + (t + 1/20)^2 < 1 \\ 2t^2 + 3x^2 - a & \text{otherwise} \end{cases} \qquad x(0) \in [0.3, 0.31]$$

The dynamics of the system is relatively simple, however the jump condition is a polynomial and is thus not well suited for classical intersection techniques as in [24, 12]. Our bisection algorithm for computing the zero-crossing time encloses precisely the jumping time. To precisely enclose the value of x, we insert a reset in the discrete transition and set $x = \sqrt{1 - (t + 1/20)^2} - 3/20$. This transformation allows us to obtain a tight enclosure of x as well. Note however that we performed this transformation manually for now except for polynomial guard, our future work will include the automatization of this task for more expressions.

5.3 Comparison with other Tools

We now compare the performance of HySon with other tools for reachability analysis of non-linear hybrid systems: Flow* as in [24] and HydLogic [19]. We downloaded both tools from the web and run them on various examples included in the Flow* distribution (we could not compile HydLogic). We run HySon on the same examples and present the execution time for both in Table 1. We see that HySon outperforms Flow*

Table 1: Experimental results. LOC is the number of locations, VAR the number of variables and T the final time of simulation. TT is the computation time, in seconds.

Benchmark	LOC	VAR	T	TT (HySon)	TT (Flow*)
Brusselator	1	2	15	14.3	49.97
Van-der-Pol	1	2	6	16.2	49.17
Lorenz	1	3	1	13.32	119.94
WaterTank	2	5	30	4.35	316.72
Hybrid3D	2	3	2.0	26.65	237.4
Pendulum	1	2	3.8	26.75	N/A
Diode oscillator [11]	3	2	20	29.56	42.65

on all these examples, whether they are purely continuous systems (VanDerPol, Brusselator or Lorenz) or hybrid systems (Watertank). Note that for the Lorenz system, we set a fixed step-size of 0.02 to achieve a good precision, which explains the large computation time. For all other examples, we used a variable step-size and an order 3 for the Taylor models used in Flow*. Let us remark however that some examples work well on Flow* but not in HySon, especially the examples with many transitions that may happen simultaneously. We also want to point out that our tool performs well on linear examples. We compared it with SpaceEx [12] on simple examples where HySon and SpaceEx produced very similar results in terms of precision and computation time (Appendix B).

6 Conclusion

We presented a new approach to compute the flowpipes of nonlinear hybrid systems using guaranteed version of numerical methods. Our method is based on guaranteed explicit Runge-Kutta integration methods and on a new guaranteed polynomial interpolation based on the well-known Hermite-Birkoff method. This interpolation is cheap and precise to over-approximate continuous state values. Using both methods, we can precisely compute flowpipes of nonlinear hybrid systems, with a few number of restrictions on the nature of flows and jumps. Remark that with guaranteed polynomial interpolation, we can accurately and soundly handle nonlinear jumps in hybrid systems without using an intersection operator which is usually costly to define. Note also that we can handle in the same manner invariants in hybrid automaton using our algorithm for zero-crossing events. More precisely, we would add a new step in the simulation loop to check that the invariant is fulfilled at each integration step. Finally, the experiments showed that our approach is efficient and precise on a set of representative case studies: we showed that our approach outperforms existing techniques on the flowpipe computation of nonlinear systems.

As future work, we plan to handle multiple zero-crossing events involving trajectories associated to different system behaviors. As a result, to keep the flowpipe computation sharp we must handle disjunctive futures efficiently. We also want to extend our parser of Simulink models, presented in [4], to handle Stateflow and thus apply our tool on more realistic examples.

References

[1] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using GReAT. In *GT-VMT*, ENTCS, 2004.

[2] O. Bouissou and A. Chapoutot. An operational semantics for Simulink's simulation engine. In *LCTES*. ACM, 2012.

[3] O. Bouissou, A. Chapoutot, and A. Djoudi. Enclosing temporal evolution of dynamical systems using numerical methods. under submission, 2013.

[4] O. Bouissou, A. Chapoutot, and S. Mimram. HySon: Precise simulation of hybrid systems with imprecise inputs. In *RSP*. IEEE, 2012.

[5] O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Vedrine. HybridFluctuat: A static analyzer of numerical programs within a continuous environment. In *CAV*, volume 5643 of *LNCS*, pages 620–626. Springer, 2009.

[6] O. Bouissou and M. Martel. GRKLib: a Guaranteed Runge Kutta Library. In *Scientific Computing, Computer Arithmetic and Validated Numerics*, 2006.

[7] T. Dang and R. Testylier. Hybridization domain construction using curvature estimation. In *HSCC*, pages 123–132. ACM, 2011.

[8] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. IMPA/CNPq, 1997.

[9] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle. Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In *SEFM*, volume 7041 of *LNCS*, pages 172–187. Springer, 2011.

[10] J. M. Esposito, V. Kumar, and G. J. Pappas. Accurate event detection for simulating hybrid systems. In *HSCC*, volume 2034 of *LNCS*, pages 204–217. Springer, 2001.

[11] G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC'05*, volume 3414 of *LNCS*, pages 258–273. Springer, 2005.

[12] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *CAV*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011.

[13] R. Goebel, J. Hespanha, A. R. Teel, C. Cai, and R. Sanfelice. Hybrid systems: Generalized solutions and robust stability. In *IFAC NOLCOS*, pages 1–12, 2004.

[14] E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI*, volume 6538 of *LNCS*, pages 232–247. Springer, 2011.

[15] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 1993.

[16] T. A. Henzinger. The theory of hybrid automata. In *Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[17] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond HYTECH: Hybrid systems analysis using interval numerical methods. In *HSCC*, volume 1790 of *LNCS*, pages 130–144. Springer, 2000.

[18] T. A. Henzinger and V. Rusu. Reachability verification for hybrid automata. In *HSCC'98*, volume 1386 of *LNCS*, pages 190–204. Springer-Verlag, 1998.

[19] D. Ishii, K. Ueda, H. Hosobe, and A. Goldsztejn. Interval-based solving of hybrid constraint systems. In *IFAC ADHS*, pages 144–149, 2009.

[20] C. Le Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In *CAV*, volume 5643 of *LNCS*, pages 540–554. Springer, 2009.

[21] R. Moore. *Interval Analysis*. Prentice Hall, 1966.

[22] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. Validated solutions of IVPs for ordinary differential equations. *App. Math. and Comp.*, 105(1):21 – 68, 1999.

[23] L. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge Univ. Press, 2003.

[24] E. A. Xin Chen and S. Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. In *IEEE Real-Time Systems Symposium*, 2012.

[25] F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *IFAC W. Cong.*, pages 7967–7972, 2008.

A Other examples

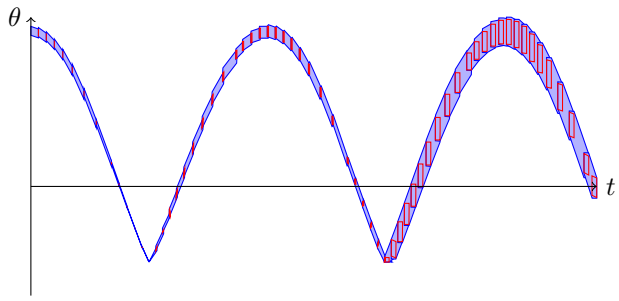
Because of space constraints, we did not include the description of some examples in the article, they can be found below.

A.1 The Bouncing Pendulum

This hybrid system describes pendulum attached to a rope of length $l = 1.2$ falling under a gravity of $g = 9.81$. The angle θ of the pendulum (w.r.t. vertical) is described by the flow equation

$$\ddot{\theta} = -\frac{g}{l}\sin(\theta) \qquad\qquad \theta(0) = [1, 1.05]$$

The pendulum bounces on a wall when $\theta = -0.5$, in which case the reset condition is $\dot{\theta} = -\dot{\theta}$. The guaranteed simulation of the system produces:



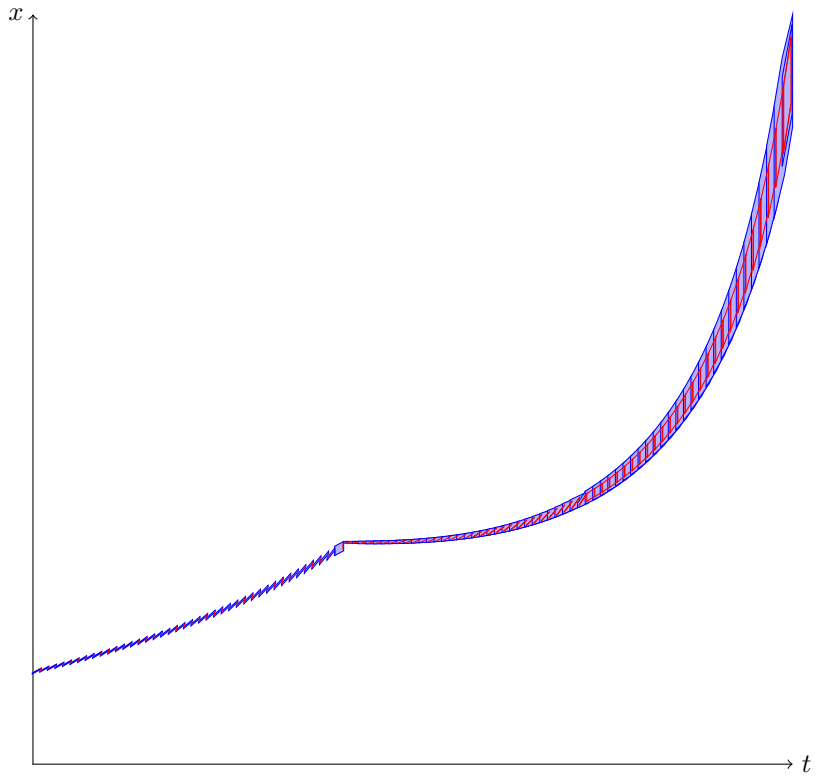
As illustration, we give here the description of the system given as input to HySon:

```
set duration = 3.8;
set dt = 0.05;
set max_dt = 0.1;
set scope_xy = true;

init theta = [1.,1.05];
init dtheta = 0.;
init t = 0;

l = 1.2;
g = 9.81;
theta' = dtheta;
dtheta' = -g/l*sin(theta);
t' = 1;

on sin(theta) <= -0.5 do { print("Bouncing!\n"); dtheta = -dtheta };

output(t,theta);
```

Notice that the dynamics of the system is nonlinear (because of the presence $\sin(\theta)$ in the flow equation) and the guard is also non linear, which makes that it cannot be simulated with Flow*.

A.2 Wolfgram

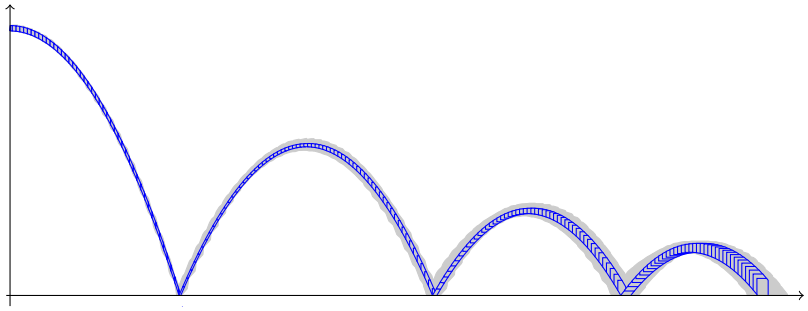
The simulation produced on the Wolfgram example is



B Comparison with SpaceEx

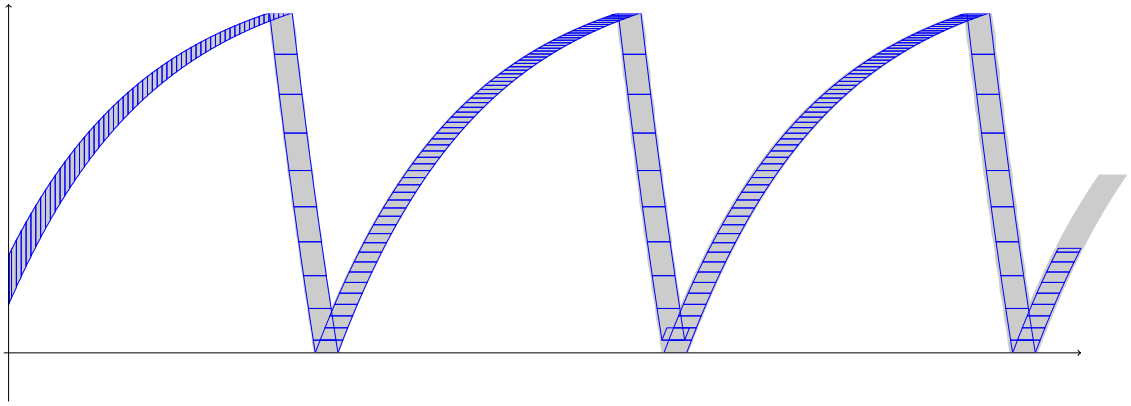
Since the main novelty of HySon is to handle efficiently non-linear systems, we did not detail experiments on linear ones. However, performances are comparable with the state-of-the-art guaranteed simulators dedicated to linear systems. As illustration, we compare here HySon with SpaceEx [12] on two examples.

B.1 Bouncing Ball



The above figure shows the flowpipe computed by SpaceEx (in gray) and by HySon (blue polygons) for the classical bouncing-ball example, up to $t_f = 20$. The computation times were 1.031s for HySon and 1.15s for SpaceEx (we used the support-function representation of sets using 50 directions). Notice that the flowpipe computed by HySon is within the flowpipe of SpaceEx; we could get a more precise results with SpaceEx by increasing the number of directions, but at the cost of higher computation times (8.65s for 200 directions for example).

B.2 Thermostat



The above figure shows the flowpipe computed by SpaceEx (in gray) and by HySon (blue sets) for the classical thermostat example, up to $t_f = 15$. The computation times were 0.89s for HySon and 0.91s for SpaceEx (we used the support-function representation of sets using 50 directions). Notice that both flowpipes are almost identical.