HySon: Set-based Simulation of Hybrid Systems

Olivier Bouissou and Samuel Mimram
CEA, LIST
olivier.bouissou@cea.fr and samuel.mimram@cea.fr

Alexandre Chapoutot
UEI, ENSTA ParisTech
alexandre.chapoutot@ensta-paristech.fr

***Abstract*—Hybrid systems are a widely used model to represent and reason about control-command systems. In an industrial context, these are often implemented in Simulink and their validity is checked by performing many numerical simulations in order to test their behavior with various possible inputs. In this article, we present a tool named HySon which performs set-based simulation of hybrid systems with uncertain parameters, expressed in Simulink. Our tool handles advanced features such as non-linear operations, zero-crossing events or discrete sampling. It is based on well-known efficient numerical algorithms that were adapted to handle set-based domains. We demonstrate the performance of our method on various examples.**

I. Introduction

Hybrid systems are a widely used model to describe embedded systems which manipulate both continuous- and discrete-time varying values. Industrial software tools to design, study and simulate such models have been developed, such as Simulink, LabVIEW, Scilab/Xcos or Modelica. As a typical example of such systems, the reader could think of a cruise controller which maintains the speed of a car at a desired value. This program has inputs (the speed of the car measured periodically) and outputs (the intensity of the engine): if the measured speed is below the desired speed then the controller will increase the intensity of the engine, and conversely. Moreover, the designers of the controller might want to take actions making the streams non-smooth: for instance, the intensity of the engine should be limited in order not to be greater than a given maximal value. The mixture of discrete and continuous-time in these systems provides them with a very useful expressive power, which enables one to model both discrete controllers and their continuous environment, but also makes them very difficult to study.

Since they are often used to design critical systems, new verification tools are more and more needed. On the one hand, we need tools to perform exhaustive testing: given a specification of the range of the inputs, we want to simulate the program on all possible inputs. On the other hand, we need tools to have guaranteed simulations, in which the errors due to floating-point computations are known. In this article, we describe a software, named HySon, that simulates Simulink programs with both these criteria in mind. Namely, HySon is able to simulate a program given "*imprecise*" or "*uncertain*" inputs (for which we do not know the precise value but only an interval in which they lie), and computes a good approximation of the set of all possible Simulink executions, and this approximation takes in account errors due to the use of floating-point values to evaluate numerical formulas. The simulation is thus *set-based* since it computes all the executions w.r.t. various possible input values *at once*, which is much more efficient and safer than the traditional approach which consists in simulating the program with randomly chosen inputs in their specification interval.

Our simulator for Simulink supports various state of the art integration methods (with both fixed and variable step-size), sampling rates and zero-crossing events (generated when a value changes of sign). This simulator is very close to the Simulink engine, except that it manipulates set of values instead of floating-point numbers, being encoded as intervals or as zonotopes. The adaptation is far from straightforward, thus justifying the present article: apart from technical implementation issues, comparisons (used for instance to detect zero-crossings) are subtle to handle when manipulating intervals because it might be true for some of the elements of the set and false for the other, thus requiring significant changes in the simulation algorithm.

We chose to analyze directly Simulink code and not the generated C code for various reasons. First, the code produced by Simulink has to be linked with a library implementing the integration methods for which we do not have the source code. Of course we could think of implementing our own library, with the same interface and functionalities as the Simulink one. However, as we explain in Section III-B, the integration algorithms which are used are quite involved and its code would be complicated, relying on tight invariants for the manipulated variables, and manipulate pointers and complex structures (in order to implement the rollback for small steps for instance). So, there is little hope that an existing analyzer (like Astree [8], Fluctuat [10] or Polyspace) could handle it out of the box, or even at all. Secondly, the advantage of having a dedicated tool working directly on Simulink programs is that it can benefit from the high-level semantics of Simulink thus allowing it to be much more concise to implement, more efficient and more precise.

Related work. We believe that our software is the first of its kind, being able to perform set-based simulation on a realistic language (a reasonably large subset of Simulink) expressing non-linear continuous-time dynamics and non-linear event functions. Our method uses well-known nu-

978-1-4673-2789-3/12/$31.00

merical integration schemes that are already used by designers of control-command software. So, we believe that our method will be easily adopted and understood by these designers. Some other works [12] are very close to ours in the study of Simulink, but are mainly focused on linear systems without zero-crossing events. The reachability problem in hybrid automata is also a field close to our work: for instance, the tool SpaceEx [13] uses zonotopes to compute guaranteed set of trajectories, i.e. bounding the truncation error of the numerical integration, in the more restricted formalism of linear hybrid automata. More recently, similar methods have been used to handle time-varying linear systems [1] and polynomial differential equations [2]. Finally, our work differ from [11] as we adapt numerical methods in order to handle sets of values instead of using them to compute reachable sets, hence reducing the number of simulations to be performed and also taking account of rounding errors.

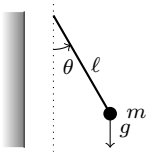
II. Hybrid programs and their simulation

Simulink is a graphical language that allows designers to write dynamical systems as block diagrams where operations are represented by *blocks* connected with *wires*. Each block has inputs and outputs, named *ports*, and *signals* are exchanged between blocks via wires; a signal is a function $\ell : \mathbb{R}_+ \to \mathbb{R}$ mapping time instants to values (for conciseness we only consider real valued signals).

Following [5], a Simulink model can be easily translated into a state-space representation that handles continuous and discrete variables. Continuous variables are attached to integrator (or state-space) blocks, and discrete variables are attached to sampled and unit-delay blocks. Writing n_x (resp. n_d) for the number of continuous (resp. discrete) variables, every Simulink model can be translated into equations of the form of Equation (1): the *continuous-time state function* $f_x : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_x}$ expresses the dynamics of the continuous variables, the *discrete-time state function* $f_d : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_d}$ modifies the discrete variables at sampling instants, while the *output function* $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \to \mathbb{R}^{n_m}$ updates the values of the output variables. We also introduce the zero-crossing events upon which continuous variables $\mathbf{x}$ can be updated: this is formalized by the *zero-crossing function* $f_z : \mathbb{R} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_x}$, while the *zero-crossing events* are defined by the function $z : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \to \{0, 1\}$ which returns 1 if the event must be activated and 0 otherwise. An example of such equations is given in Example 1.

$$\begin{array}{ll} \dot{\mathbf{x}} = f_x(t, \mathbf{x}, \mathbf{d}) & \mathbf{d} = f_d(t, \mathbf{x}, \mathbf{d}) \\ \mathbf{x} = f_z(t, \mathbf{x}, \mathbf{d}) \text{ when } z(t, \mathbf{x}, \mathbf{d}) & \mathbf{y} = g(t, \mathbf{x}, \mathbf{d}) \end{array} \quad (1)$$

Example 1. To illustrate our method, we use a simple yet representative example, the *uncertain bouncing pendulum*. This system consists of a pendulum that bounces against a wall located at angle $\theta = 0.5$, see Figure 1: the wall creates a zero-crossing events during the simulation. The system has two continuous variables (the angle θ, the angular velocity ω) and no discrete variables. The evolution of the



$$f_x(t, \mathbf{x}, \mathbf{d}) = \begin{pmatrix} \omega \\ -g/\ell \sin\theta \end{pmatrix}$$

$$f_z(t, \mathbf{x}, \mathbf{d}) = \begin{pmatrix} \theta \\ -0.7\omega \end{pmatrix} \quad \text{when } \theta = -0.5$$

$$y = \theta$$

Figure 1. A pendulum system and the state-space equations.

continuous variables follow the equation $\ell\ddot{\theta} + g \sin\theta = 0$. The output is the couple (θ, ω) and the sate-space representation is given in Figure 1, with $\mathbf{x} = (\theta, \omega)^t$. ■

The *semantics* of Simulink programs is the mathematical solution of the Equations (1). This solution is generally not computable, and we consider instead the simulation, as performed by the Simulink solver, as a semantics. This simulation process starts at time t_0 with a discretization step h_0 and updates the values of $\mathbf{x}$, $\mathbf{d}$ and $\mathbf{y}$ according to the following simulation loop:

Input: $\mathbf{x}_0$, $\mathbf{d}_0$, t_0, h_0;
$n = 0$;
loop until $t_n \geq t_{\text{end}}$
  evaluate $g(t_n, \mathbf{x}_n, \mathbf{d}_n)$
  compute $\mathbf{d}' = f_d(t_n, \mathbf{x}_n, \mathbf{d}_n)$
  solve $\dot{\mathbf{x}}(t) = f_x(t, \mathbf{x}(t), \mathbf{d}_n)$ over interval $[t_n, t_n + h_n]$
  find_zero_crossing
  compute h_{n+1} ; compute t_{n+1} ; $\mathbf{d}_{n+1} = \mathbf{d}'$; $n = n + 1$

The first two steps of the simulation loop are simple evaluation of f_d and g to update the value of the output and the discrete states. The most important steps are the solve and find_zero_crossing steps that, respectively, compute an approximation of the solution of the differential equation of the state-space representation in Equation (1) at t_{n+1}, and detect any zero-crossing event between t_n and t_{n+1}. We briefly detail these two steps in the rest of this section.

A. Solving differential equations

The simulation engine computes discretized approximations of the solution of differential equations. Given a value $\mathbf{x}_n$ that approximates $\mathbf{x}$ at time t_n, the solve procedure computes an approximation $\mathbf{x}_{n+1}$ at time t_{n+1}. For example, using Euler integration scheme, the value $\mathbf{x}_{n+1}$ is defined as a linear interpolation from $\mathbf{x}_n$ following the slope given by $f_x(t_n, \mathbf{x}_n, \mathbf{d}_n)$: $\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot f_x(t_n, \mathbf{x}_n, \mathbf{d}_n)$, $\mathbf{x}_0$ being the initial values specified by the user.

This idea of linear interpolation to obtain $\mathbf{x}_{n+1}$ can be generalized to any Runge-Kutta like method (solvers `ode4` or `ode23` in Simulink for example). These solvers are described by their Butcher tables, i.e. three matrices A, b and c such that A and b encodes the intermediate steps at which the function f_x must be evaluated and c encodes the final linear interpolation. For example, for the Boagacki-Shampine (BS) method [3] (`ode23` in Simulink), x_{n+1} is

defined by:

$$k_1 = f_x(t_n, \mathbf{x}_n, \mathbf{d}_n) \quad k_2 = f_x\big(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{hk_1}{2}, \mathbf{d}_n\big)$$
$$k_3 = f_x\big(t_n + \frac{3h}{4}, \mathbf{x}_n + \frac{3hk_2}{4}, \mathbf{d}_n\big)$$
$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\big(\frac{2k_1}{9} + \frac{k_2}{3} + \frac{4k_3}{9}\big)$$

and so we have $A = \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0 & 0 \\ 0 & 0.75 & 0 \end{pmatrix}$, $b = \begin{pmatrix} 0 & 0.5 & 0.75 \end{pmatrix}^T$ and $c = \begin{pmatrix} 2/9 & 1/3 & 4/9 \end{pmatrix}$. In the following, we assume given the Butcher table of the numerical scheme.

To improve precision and performance, most integration methods dynamically choose the integration step $h_n = t_{n+1} - t_n$ to keep it small when the values are varying rapidly and big otherwise. Let us show for example the implementation of the `ode23` method (that we already partially presented). This method performs, using the same intermediate computation, another linear interpolation leading to a second approximation point:

$$\mathbf{x}'_{n+1} = \mathbf{x_n} + h\big(\frac{7}{24}k_1 + \frac{2}{4}k_2 + \frac{1}{3}k_3 + \frac{1}{8}f_x(t_{n+1}, \mathbf{x}_{n+1}, \mathbf{d}_n)\big)$$

The Butcher table is extended by $c' = (7/24, 2/4, 1/3, 1/8)$ that performs this second evaluation. The value $\mathbf{x}_{n+1}$ is a second-order approximation of the value $\mathbf{x}(t_{n+1})$ whereas $\mathbf{x}'_{n+1}$ is a third-order approximation (thus the name `ode23`).

The points $\mathbf{x}_{n+1}$ and $\mathbf{x}'_{n+1}$ give an estimation of the numerical error due to the solver. In practice, we check that $\text{err} \leq \text{err\_max}$ where the approximation error is $\text{err} = \|\mathbf{x}'_{n+1} - \mathbf{x}_{n+1}\|_\infty$ and $\text{err\_max} = \max(\text{atol}, \text{rtol} \times \max(\|\mathbf{x}_n\|_\infty, \|\mathbf{x}_{n+1}\|_\infty))$ is the maximal tolerated error, which depends on two user-chosen constants atol and rtol, respectively expressing the *absolute* and *relative tolerance*. If the inequality is not satisfied, the step is *rejected*: the solver tries again with a step size $h_n/2$. If the inequality is verified, the step is *accepted* and the next step h_{n+1} is computed by

$$h_{n+1} = h_n \times (\text{err\_max}/\text{err})^{1/(q+1)} \tag{2}$$

where q is the order of the method ($q = 2$ in case of `ode23`).

B. Handling zero-crossing events

The next step of the simulation loop must detect and solve special events that could have occurred between t_k and t_{k+1}. Actually, the main assumptions that make the numerical solvers stable is that the function f_x is (at least) continuous. However, in many systems, this is not the case: in the bouncing pendulum, the angular velocity changes from ω to $-\omega$ when the pendulum hits the wall. In the simulation loop, the find_zero_crossing step is responsible of detecting and handling such events. The main idea is that the ODE solver is used to compute the approximation point $\mathbf{x}_{k+1}$, assuming no zero-crossing events occurred between t_k and t_{k+1} (i.e. assuming the dynamics of $\mathbf{x}$ is continuous), and then we check if a zero-crossing event occurred. If yes, this event is localized and the simulation continues from it.

The detection of zero-crossing events is realized by comparing the sign of $z(t_k, \mathbf{x}_k, \mathbf{d})$ and $z(t_{k+1}, \mathbf{x}_{k+1}, \mathbf{d})$. If the sign changed, a zero-crossing event occurred (it is a consequence of the intermediate value theorem on z).

Once the event is detected, we approximate the instant t_{zc} and the value $\mathbf{x}_{zc}$ at which $z(t_{zc}, \mathbf{x}_{zc}, \mathbf{d}) = 0$ using a bisection method on the time interval $[t_k, t_{k+1}]$ and a *dense approximation function* to estimate $\mathbf{x}$ at various instants $t \in [t_k, t_{k+1}]$. The dense approximation of the solution of an ODE between t_k and t_{k+1} is a (solver-dependent) function $\phi : t \to \mathbb{R}^{n_x}$ that computes approximations of the solution without using the numerical solver. For example, for the `ode23` solver, it is given by:

$$\phi(t) = (2\tau^3 - 3\tau^2 + 1)\mathbf{x}_k + (\tau^3 - 2\tau^2 + \tau)(t_{k+1} - t_k)\dot{\mathbf{x}}_k + (-2\tau^3 + 3\tau^2)\mathbf{x}_{k+1} + (\tau^3 - \tau^2)(t_{k+1} - t_k)\dot{\mathbf{x}}_{k+1} \tag{3}$$

with $\tau = (t - t_k)/(t_{k+1} - t_k)$. Using this function, we can precisely bracket the zero-crossing event using a bisection method. Once the time t_{zc} is precisely computed, the simulation starts again from t_{zc} and $f_z(t_{zc}, \mathbf{x}_{zc}, \mathbf{d})$. For more details on this process, we refer to [5].

C. Simulating a system

If we sum up all the techniques described in this section, we obtain the algorithm used by Simulink to simulate hybrid and dynamical systems. Given a hybrid program P in state-space representation, we denote by $[\![P]\!]$ its simulation as given by this algorithm and write $[\![P]\!](t)$ for the values of the variables of P at some time t. So $[\![P]\!](t) \in \mathbb{R}^m$, $m = n_x + n_d + n_m$ being the total number of program variables.

In many real-life applications, the systems have uncertainties in inputs, in parameters (e.g. the mass of the pendulum), or in the initial value (e.g. the initial angle). In such situations, one numerical simulation is of little help to study the behavior of the system and the classical approach is to perform many simulations for random values chosen in the uncertainty sets. Consider the bouncing pendulum of Example 1. If the initial state is uncertain and given by $\theta(0) \in [1.0, 1.1]$, $\omega(0) = 0$, then we can perform 8 random simulations in Simulink and we obtain the black lines of Figure 2 for the trajectories of θ. Our goal is to efficiently compute approximations of all these simulations at once: the thick squares represent the output of HySon on this example.

In the rest of this article, we describe our method to compute set-based simulations that automatically compute enclosures of all the possible simulations for any value of the uncertain parameters.

III. Set-based simulation of Simulink

From now on, we consider hybrid programs with uncertainties in their initial states and in their parameters (i.e. we only know that they lie in a given interval and not their precise value). In Section III-A, we first describe

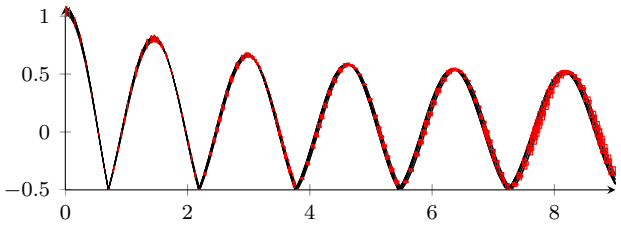


Figure 2. Ten random executions of the bouncing pendulum and one set-based simulation (red boxes).

the representation we chose for encoding sets of values (representing "uncertain" values): we use two well-known domains, intervals for the uncertainty on time and zonotopes for the uncertainty on values. We then describe how we implemented the set-based simulation in Section III-B.

A. Domains for representing sets

The simplest and most common way to represent and manipulate sets of values is interval arithmetic [14]. Nevertheless, this representation usually produces too much over-approximated results in particular because of the *dependency problem*. This problem arises for example when evaluating the expression $e(x) = x - x$, which can produce the interval $e([0,1]) = [-1,1] \neq 0$. This problem is central in the numerical integration as for example in the Euler method with have a computation of $\mathbf{x}_{k+1}$ involves $\mathbf{x}_k$ and $f_x(\mathbf{x}_k, \mathbf{d}_k, t_k)$ which could be $-\mathbf{x}_k$.

To avoid this problem we use an improvement over interval arithmetic named *affine arithmetic* [9] which can track linear correlation between program variables. A set of values in this domain is represented by an *affine form* $\hat{x}$ (also called a *zonotope*), i.e. 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 $\hat{y} = 2\alpha_0 + 2\alpha_1 \varepsilon$.

Affine arithmetic extends usual operations on real numbers in the expected way. For instance, the affine combination of two affine forms $\hat{x} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ and $\hat{y} = \beta_0 + \sum_{i=1}^{n} \beta_i \varepsilon_i$ with $a, b, c \in \mathbb{R}$, is given by:

$$a\hat{x} + b\hat{y} + c \quad = \quad (a\alpha_0 + b\beta_0 + c) + \sum_{i=1}^{n} (a\alpha_i + b\beta_i)\varepsilon_i \ . \qquad (4)$$

However, unlike the addition, most operations create new noise symbols. Multiplication for example is defined by:

$$\hat{x} \times \hat{y} \quad = \quad \alpha_0\alpha_1 + \sum_{i=1}^{n} (\alpha_i\beta_0 + \alpha_0\beta_i)\varepsilon_i + \nu\varepsilon_{n+1} \qquad (5)$$

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.

Algorithm 1 Main simulation algorithm.

Require: approximation $(\widetilde{t}_k, \widetilde{x}_k)$, step-size h_k
  repeat
    $\widetilde{y}_k$ =`butcher`$(\widetilde{x}_k, h_k)$; $\widetilde{e}_k$ =`error`$(\widetilde{y}_k, \widetilde{x}_k)$;
    if not `validate`$(\widetilde{e}_k)$ **then**
      $h_k = h_k/2$;
    end if
  until `validate`$(\widetilde{e}_k)$;
  h_{k+1} =`nextstepsize`$(\widetilde{e}_k, h_k)$;
  if `zerocross`$(\widetilde{y}_k, \widetilde{x}_k)$ **then**
    Handle zc-events;
  else
    return $(\widetilde{t}_k + h_k, \widetilde{y}_k)$
  end if

One of the main difficulties when implementing affine arithmetic using floating-point numbers is to take in account the unavoidable numerical errors due to the use of finite-precision representations for values (and thus rounding on operations). We use an approach based on computations of floating-point arithmetic named *error free transformations*: the round-off error can be represented by a floating-point number and hence it is possible to exactly compute it (we refer to [15] for more details on such methods). For instance, in the case of addition, the round-off error e generated by the sum $s = a + b$ is given by

$$e = (a - (s - (s - a))) + (b - (s - a)) \ .$$

A second comment on the implementation is that an affine form $\hat{x}$ could be represented as an array of floats encoding the coefficients α_i. However, since in practice most of those coefficients are null, it is much more efficient to adopt a sparse representation and encode it as a list of pairs (i, α_i), sorted w.r.t. the first component, containing only coefficients $\alpha_i \neq 0$.

B. Simulation algorithm

We explain how the simulation algorithms used in Simulink were adapted to take into account the peculiarities of the affine domain. Our goal was to use as much as we could the same algorithms as for a floating-point simulation in order to achieve the best efficiency, both in precision and computation time. We shall use intervals to represent uncertain times $\widetilde{t}_k$ and affine arithmetic to represent uncertain states $\widetilde{x}_k$. Given an approximation $(\widetilde{t}_k, \widetilde{x}_k)$ of the values of the simulations at a time $t \in \widetilde{t}_k$, we compute a new approximation $(\widetilde{t}_{k+1}, \widetilde{x}_{k+1})$ using Algorithm 1, very similar to the Simulink simulation loop described in Section II. In the rest of this section, we explain how the involved functions work.

1) Computing and validating the next state: The role of the `butcher` function is to estimate the state $\widetilde{x}_{k+1}$ at time $\widetilde{t}_k + h_k$ using the chosen numerical integration method. To do so, we use the Butcher table representation of Runge-Kutta methods [7] and perform the operations using affine sets arithmetic. For example, if we use the Bogacki-Shampine method (see Section II), this sums up to computing three linear extrapolations and three

Figure 3. First iterations of the `butcher` function.

evaluations of f_x in affine arithmetic. The extrapolations (of the form $\widetilde{x}_k + \frac{1}{2}h_k\widetilde{c}_k^0$ for example) are very precise and the set-based evaluation of f_x can be made precise using good linearization methods (as described in Section III-A). In this way, the `butcher` function can be made very precise and produces a good approximation of the state $\widetilde{x}_{k+1}$ at time $\widetilde{t}_k + h_k$. For example, we show in Figure 3, the first ten iterations of the `butcher` function on the uncertain pendulum, when using the RK4 numerical integration algorithm (the thin lines represent floating-point simulation for randomly chosen initial points).

Next, the approximation error must be estimated in order to best adapt the step-size and achieve a good trade-off between precision and computation time. This is the role of the `error` function. As in Section II, we define the error as $\text{err} = \|\widetilde{\mathbf{y}}'_{k+1} - \widetilde{\mathbf{y}}_{k+1}\|_\infty$, where $\widetilde{\mathbf{y}}'_{k+1}$ and $\widetilde{\mathbf{y}}_{k+1}$ are the two approximations at time $\widetilde{t}_k + h_k$. To compute the norm $\|\widetilde{\mathbf{y}}\|_\infty$ of a vector of affine forms $\widetilde{\mathbf{y}}$, we compute the component-wise absolute values using a Tchebychev approximation as in [9] and then compute the maximum of all these values using $\max(\widetilde{x}, \widetilde{y}) = (\widetilde{x} + \widetilde{y})/2 + \text{abs}(\widetilde{x} - \widetilde{y})/2$.

2) Computing next step-size: Once the error is computed, the next approximation $\widetilde{x}_{k+1}$ must be validated: the step is validated only if the error is smaller than the user-defined tolerance. We use again the equation $\widetilde{\text{err}} \leq \text{err\_max}$ except that now, $\widetilde{\text{err}}$ is now an affine form (as computed by `error`) and err_max is a floating-point value. We shall accept the step if and only if the supremum of $\widetilde{\text{err}}$ is smaller than err_max, that is only if the maximal computed error is smaller than err_max. In this way, we shall reject steps that produced an error which contains err_max, i.e. we may reject steps that would not have been rejected if we performed floating-point simulations. However, this method is conservative in the following sense: a step is accepted if and only if all the (approximated) trajectories it represents are accepted. Thus, we chose precision over performance, another choice could be to reject a step if and only if all the trajectories it contains are rejected. For sufficiently stable systems, our choice is preferable as the variation of the step-size prevents the simulation to reject too many steps. So, the `validate` function is defined as

$$\texttt{validate}(\widetilde{x}_{k+1}, \widetilde{x}_k) \quad = \quad \begin{cases} 1 & \text{if } \sup(\widetilde{\text{err}}_k) \leq \text{err}_m \\ 0 & \text{otherwise} \end{cases} \qquad (6)$$

with $\text{err}_m = \max(\text{atol}, \text{rtol} \times \max(\|\mathbf{x}_k\|_\infty, \|\mathbf{x}_{k+1}\|_\infty))$. If the step-size is not validated, we shall divide the step-size by two and compute a new approximation. If the step is validated, we define the new step-size h_{k+1} by:

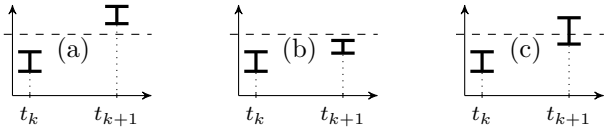
$$h_{k+1} \quad = \quad \text{m}(h_k \times (\text{err\_max}/\widetilde{\text{err}}))^{1/q+1}$$

where $m(\widetilde{y})$ is the center of the set $\widetilde{y}$. Note that this is very close to Equation (2), the main difference is that, as $\widetilde{\text{err}}$ is an affine form, we take the mean of all step-sizes as the next step-size. The `nextH` function computes this h_{k+1}.

3) Handling zero-crossing events: Once we have an approximation $\widetilde{x}_{k+1}$ at time $\widetilde{t}_{k+1}$, we must check if the zero-crossing signal crossed zero between t_k and t_{k+1}, and if so localize precisely the time at which it crossed zero. Of course as $\widetilde{x}_k$ and $\widetilde{x}_{k+1}$ are sets encoded in affine arithmetic, we will compute a time interval $[t_l, t_r]$ that contains all the possible zero-crossing instants for any approximate trajectory going from $\widetilde{x}_k$ to $\widetilde{x}_{k+1}$. Our method for handling zero-crossing events works in two steps: first we localize the zero-crossing instants, then we compute an approximation of the state of the system at these instants. Without loss of generality, we assume that the zero-crossing signal is $\widetilde{x}_k$ (in the pendulum example, it is $\widetilde{x}_k + 0.5$) and that $\widetilde{x}_k$ does not contain zero and is negative.

To compute t_l, we use an algorithm close to the one used by Simulink for floating-point simulation. We describe in the sequel this method, t_r being computed in a similar way. The method uses three functions: the `detect` function that detects a zero-crossing event, the `extrapolate` function that computes an interval of potential zero-crossing events and the `approximate` function that computes an approximation of the state of the system at some time t between $\widetilde{t}_k$ and $\widetilde{t}_{k+1}$.

The detection of a zero-crossing event is simple: we compare the signs of $\widetilde{x}_{x+1}$ and $\widetilde{x}_k$, and if they have different sign we know that there is a zero-crossing. Since $\widetilde{x}_{k+1}$ may contain zero, we now have three possibilities represented by the three following cases:



In case (a), we have $\inf(\widetilde{x}_{k+1}) > 0$, so we are sure there is a zero-crossing between $\widetilde{t}_k$ and $\widetilde{t}_{k+1}$. We thus have $\texttt{detect}(\widetilde{x}_k, \widetilde{x}_{k+1}) = 1$. In case (b), we have $\sup(\widetilde{x}_{k+1}) < 0$, so there is no zero-crossing and we have $\texttt{detect}(\widetilde{x}_k, \widetilde{x}_{k+1}) = 0$. In case (c), we have $0 \in \widetilde{x}_{k+1}$, so we do not know if there is a zero-crossing between $\widetilde{t}_k$ and $\widetilde{t}_{k+1}$: we have $\texttt{detect}(\widetilde{x}_k, \widetilde{x}_{k+1}) = \top$, i.e. we do not know its value.

The extrapolation function is a simple linear interpolation between $\widetilde{x}_k$ and $\widetilde{x}_{k+1}$ and we compute the time interval $[t_1, t_2]$ such that this interpolation crosses zero. Formally,

$$\texttt{extrapolate}(\widetilde{x}_k, \widetilde{t}_k, \widetilde{x}_{k+1}, \widetilde{t}_{k+1}) \quad = \quad \widetilde{t}_k - \frac{\widetilde{x}_k \times h_k}{\widetilde{x}_{k+1} - \widetilde{x}_k} \ .$$

If $\texttt{extrapolate}(\widetilde{x}_k, \widetilde{t}_k, \widetilde{x}_{k+1}, \widetilde{t}_{k+1})$ is the interval $[t_1, t_2]$, then we set the next guess for zero-crossing $t_m = (t_1 + t_2)/2$ and compute an approximation of the signal at time t_m

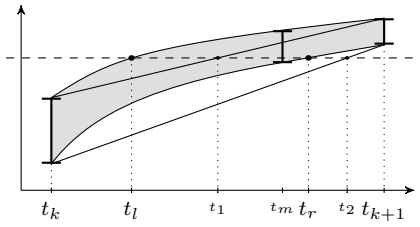


Figure 4. Zero-crossing localization.

using the `approximate` function. This function is solver-dependent and is a *continuous extension* of the numerical integration method [16]; for example, as in Section II, the approximation function for the `ode23` method is the function $\phi(t)$ given by Equation (3), with $\tau = (t - \inf(\widetilde{t}_k))/h_k$. This function is the natural interpretation of the floating-point continuous extension for affine arithmetic, except that we replaced the term $t_{k+1} - t_k$ with h_k. This is important as $\widetilde{t}_{k+1}$ and $\widetilde{t}_k$ are intervals defined by $\widetilde{t}_{k+1} = \widetilde{t}_k + h_k$, but interval arithmetic gives $\widetilde{t}_{k+1} - \widetilde{t}_k \neq h_k$.

With these three functions, we can define our algorithm for computing t_l, the lower value of the zero-crossing instants interval, see Algorithm 2. This algorithm is a loop that repeatedly uses the `extrapolate` and `approximate` functions to compute a new approximation set $\widetilde{y}_m$. Then, if there is a potential zero-crossing between $\widetilde{x}_k$ and $\widetilde{y}_m$ (i.e. if $\texttt{detect}(\widetilde{x}_k, \widetilde{y}_m)$ is not 0), we look for t_l between $\widetilde{t}_k$ and t_m, otherwise we look for t_l between t_m and $\widetilde{t}_{k+1}$. To compute t_r, the only change is that we look left if $\texttt{detect}(\widetilde{x}_k, \widetilde{y}_m) = 1$ and right otherwise. This algorithm for computing $[t_l, t_r]$ is represented graphically below. The gray region is the (set-based) continuous extension function and the black lines between $\widetilde{x}_k$ and $\widetilde{x}_{k+1}$ are the linear extrapolation which produces t_1, t_2 and t_m. The result of the algorithm will be t_l and t_r, as depicted on the figure. This algorithm is very precise: on the pendulum example, we get the interval $[0.752668, 0.771273]$ for the first zero-crossing and if we perform 1000 floating-point simulations, we obtain zero-crossing instants within the interval $[0.753208, 0.771114]$.

Once the zero-crossing time $\widetilde{t}_{zc} = [t_l, t_r]$ is computed, we define the zero-crossing set $\widetilde{x}_{zc}$ as:

$$\widetilde{x}_{zc} \quad = \quad \texttt{approximate}(t_l) \cup \texttt{approximate}(t_r),$$

Algorithm 2 Zero-crossing localization algorithm.

Require: approximations $(\widetilde{t}_k, \widetilde{x}_k)$ and $(\widetilde{t}_{k+1}, \widetilde{x}_{k+1})$
  while $|t_k - t_{k+1}| > \varepsilon$ **do**
    $[t_1, t_2]$ =`extrapolate`$(\widetilde{x}_k, \widetilde{x}_{k+1})$
    $t_m = (t_1 + t_2)/2$
    $\widetilde{y}_m$ =`approximate`(t_m);
    if `detect`$(\widetilde{x}_k, \widetilde{y}_m) \neq 0$ **then**
      `zcfind`$((\widetilde{t}_k, \widetilde{x}_k), (\widetilde{t}_m, \widetilde{y}_m))$;
    else
      `zcfind`$((\widetilde{t}_m, \widetilde{y}_m), (\widetilde{t}_{k+1}, \widetilde{x}_{k+1}))$;
    end if
  end while
  return $(\widetilde{t}_{k+1}, \widetilde{x}_{k+1})$

i.e. we compute approximations of the state at time t_l and t_r and perform the union of both. This is not the optimal value for the zero-crossing state: ideally, we should compute the intersection of all states between t_l and t_r with the hyperplane $g(\widetilde{x}) = 0$ where g is the zero-crossing function which may contain complex, non-linear operations. This intersection is costly and hard to compute precisely in the general case, which is why we chose to start from $\widetilde{x}_{zc}$. Note that in some cases, we can refine this by inverting the zero-crossing function g and setting $\widetilde{x}_{zc}$ to $g^{-1}(0)$. For example, in the pendulum system, we know that when there is a zero-crossing, then $\theta = -0.5$. So we set θ to -0.5 after the zero-crossing, so that we have a very precise value for θ, while the value for $\dot{\theta}$ remains uncertain.

There is one last case when handling zero-crossings, namely when initially we have $0 \in \widetilde{x}_{k+1}$. In this case, there are trajectories between $\widetilde{x}_k$ and $\widetilde{x}_{k+1}$ that did not cross 0, and we handle them in a special way. We compute the zero-crossing interval $[t_l, t_r]$ as before and return two states: the state $([t_l, t_r], \widetilde{x}_{zc})$ as in the standard case and the state $(\widetilde{t}_{k+1}, \widetilde{x}'_{k+1})$ where $\widetilde{x}'_{k+1}$ is the restriction of $\widetilde{x}_{k+1}$ that is below 0. The simulation then continues from these two states: we perform disjunctive simulations and computes sets of approximations states.

IV. Benchmarks

In this section, we present a few benchmarks which are summarized in the Table 5 which shows, along with their dimension, the simulation duration (in seconds) when simulated in the affine domain using Bogacki-Shampine integration method. We also give the computation time for 1000 random Simulink simulations.

Ball is the bouncing ball with $x_0 = [10, 10.1]$, $v_0 = 15$ simulated during 15s. The *Van der Pol* oscillator is defined by the ODE $\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$ with $\mu = 0.2$, $x_0 = [0.8, 1]$ and $\dot{x}_0 = 0$ simulated during 50s. *Mass-spring* is the well-known mass-spring physical system with a mass of $[0.5, 0.6]$, a damping of 0.35 and initial position $x_0 = [-2.1, -2]$ simulated during 40s. *B. pendulum* is the pendulum described in Example 1, simulated for 50s. We also tested HySon on some other examples that we cannot detail here for the lack of space: *Brusselator* (a model of autocatalytic reaction), *Car* (a non-linear model of a car), and *Helicopter* (a 28-dimensional model of a Westland Lynx helicopter [17] used as a benchmark for SpaceEx [13]).

All the examples (excepting *Helicopter*) use error-free transformations in order to encompass floating-point errors. The *Helicopter* dynamics directly generated by the

Name	Dim.	Dur.	Simu.$\times 10^3$
Ball	2	0.068	15
Van der Pol	2	5.97	12.24
Brusselator	2	1.84	13.49
Mass-spring	2	0.134	12.93
B. pendulum	2	13.58	34.34
Car	5	43.84	91.62
Helicopter	28	16.685	438

Figure 5. Experimental results

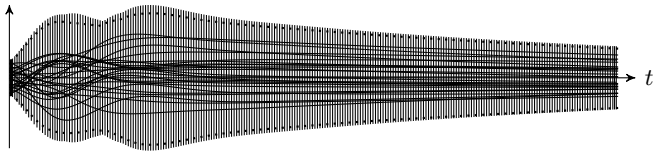


Figure 6. Set based simulation of the Helicopter system (vertical bars) and some random simulations (black lines). The dotted lines are the minimum and maximum of 100.000 simulations at each time.

Tikz output of HySon is presented in Figure 6. We only plot the interval computed for the values at the integration times t_k without interpolation between them. In particular, we can notice that vertical bars are not evenly spaced at the beginning of *Helicopter*: this is due to a change of integration step.

We also report a comparison of the results produce by HySon to the state of the art of reachability software for linear hybrid automata SpaceEx [13]. To be fair, the non-linear examples are not considered. SpaceEx analyses: the *Ball* model in 0.48s, the *Mass-Spring* model in 0.59s; the *Helicopter* model in 23.3s. Note that HySon has similar performance against a specialized tool for linear systems.

Finally, we show the precision of our set-based simulation compared to randomly chosen simulations. To do so, we computed the relative distance between the bounds computed by HySon and the bounds obtained for 1000 runs of Simulink on some problems. We present in the table given in Figure 7 the maximum and mean distances, each time with the fixed step-size solver RK4. Computing this distance for a variable step-size solver would be more complicated as Simulink and HySon do not chose the same step-sizes. We believe however, as shown by the Figures 2 and 3 that HySon also works very well for variable step-size solvers. The results we obtain are very good: even if the maximum relative distance may be high (this happens usually when the simulations are close to zero), the mean distance is small, indicating that the overhead due to the use of set-based algorithms compared to floating point simulations is small in the long run.

Name	Max. distance	Mean distance
Van der Pol	187.26%	9.4%
Mass-spring	5.61%	0.11%
Car	185.3%	6.6%

Figure 7. Set-based simulation precision

V. Conclusion and future works

In this article, we presented a novel approach for simulating hybrid systems with uncertainties. We adapted the simulation engine of Simulink in order to perform *set-based simulation*, which is able to handle uncertainties on some parameters and initial values. The main challenge was to adapt the zero-crossing algorithm to the affine domain which, due to non-linear operations, was not straightforward. We developed a tool that parses Simulink programs and automatically computes good approximations of all the trajectories. We believe that such a tool can be very useful for the design of control-command systems for which some physical parameters are not precisely known.

This work can be extended in many directions to provide a better simulation of uncertain systems. First, since the zero-crossing algorithm can produce two different states (one that zero-crossed and one that did not), our simulation can be sometimes slow when there are many zero-crossing events, since the number of states from which we must perform the simulation may increase. To solve this problem, we can merge these states when they are close enough: a way to do that is to adapt the method from [4]. Another improvement is to improve the computation of zero-crossing states: we believe that a backward analysis will be useful to compute $g^{-1}(0)$ where g is the zero-crossing function. Finally, we want to refine our tool in order to have guaranteed integration of the differential equations, in the spirit of [6]. In this way, we would compute reachable sets of the Simulink programs instead of simulations. This would help to measure the quality of the numerical simulations: the divergence between both is an estimation of the error due to the use of numerical integration tools.

References

[1] M. Althoff, C. Le Guernic, and B. H. Krogh. Reachable set computation for uncertain time-varying linear systems. In *HSCC*, 2011.

[2] E. Asarin, T. Dang, and A. Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Inf.*, 43(7):451–476, 2007.

[3] P. Bogacki and L.F. Shampine. A 3(2) pair of Runge-Kutta formulas. *Applied Mathematics Letters*, 2(4):321–325, 1989.

[4] O. Bouissou. Proving the correctness of the implementation of a control-command algorithm. In *SAS*, volume 5673 of *LNCS*. Springer, 2009.

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

[6] O. Bouissou and M. Martel. GRKlib: a guaranteed Runge Kutta library. In *SCAN*. IEEE, 2006.

[7] J.C. Butcher. *Numerical methods for ordinary differential equations*. Wiley, 2008.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, volume 3444 of *LNCS*. Springer, 2005.

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

[10] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of Fluctuat on safety-critical avionics software. In *FMICS*, volume 5825. Springer, 2009.

[11] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, volume 6174 of *LNCS*. Springer, 2010.

[12] G. E. Fainekos, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Robustness of model-based simulations. In *IEEE RTSS*, 2009.

[13] 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*. Springer, 2011.

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

[15] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhauser, 2009.

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

[17] S. Skogestad and I. Postlethwaite. *Multivariable feedback control: analysis and design*, volume 2. Wiley, 2007.