Set-based Simulation for Design and Verification of Simulink Models

Olivier Bouissou, Samuel Mimram, Baptiste Strazzulla
CEA, LIST, Gif-sur-Yvette, France
Email: firstname.lastname@cea.fr

Alexandre Chapoutot
ENSTA ParisTech, Palaiseau, France
Email: alexandre.chapoutot@ensta-paristech.fr

***Abstract*—Model-based design is a widely used methodology for the development of embedded critical software, such as a discrete controller for a continuous plant. In this setting, numerical simulation of both the plant and the controller plays a crucial role, since it is used to validate the design choices in the early stages of development. However, classical numerical simulation has inherent limitations: it is of limited precision and cannot deal with the intrinsic non-determinism present in complex systems. In this article, we present a tool named HySon that overcomes these drawbacks. It takes as input a Simulink model of a control-command system with non-deterministic uncertainties and automatically computes flow-pipes that contain all possible trajectories of the system. We show on some examples how HySon can be used to improve the quality of model-based design.**

I. Introduction

The typical workflow for embedded software development generally involves at least two teams of engineers, respectively developing the hardware system and the software. For example, during the elaboration of a control-command software, the first team designs the controller, while the second one implements the resulting algorithm on a specific hardware target. We have presented in Figure 1 a simplified workflow model as it can be found in DO178 compliant aeronautics development [14]. In order to reduce the time-to-market and to increase the confidence in the developments both teams use tools to help them. Usually, at system level, Matlab/Simulink is used to describe and simulate both the physical plant and the controller in a uniform model. At the software level, the SCADE Suite is often used because its code generator is qualified for software up to DAL-A level. The development process for the software is thus generally much more rigorous than the development process for the system, while verification and validation of the software with respect to system requirements are needed.

A large number of bugs in software is caused by bad specification of requirements, which occurs early in the development process. It is therefore crucial to ensure that all corner cases have been correctly addressed at the model level. This is usually done by performing many simulations, with the hope of covering the majority of typical behaviors of the system depending on its parameters: those parameters are generally *uncertain*, i.e. the specification does not give their precise value but only a range in which they should lie. However, it cannot be ensured that all the cases have been considered since parameters usually lie in an infinite domain. Starting from this observation, we have developed a tool named HySon, whose aim is to analyze Simulink models used by system engineers to discover bugs or to validate the control algorithms

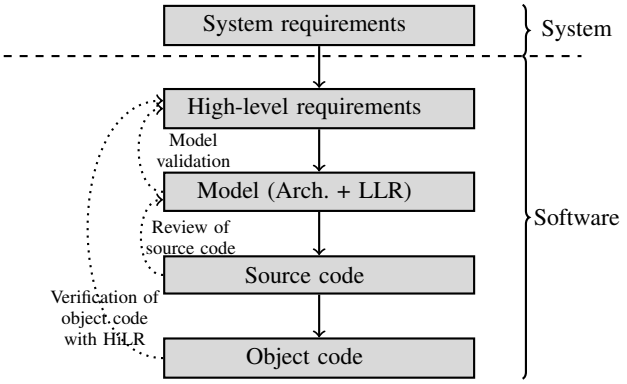


Figure 1: Simplified life-cycle in model-based development.

early in the development process: instead of performing lots of simulations, it can compute an over-approximation of *all* the possible behaviors resulting from (possibly infinite) sets of parameters in *one* simulation pass. The correct-by-design simulation process has been achieved by an in-depth adaptation of classical simulation algorithms, such as those implemented in Simulink, in order to convert them to *set-based simulation*, which is able to manipulate sets of trajectories.

The rest of this article is organized as follows: in Section II we present our method on a simple yet realistic example, in Section III we describe our tool and in Section IV we give some details on the mathematical foundation of our method. Finally, Section V shows the various usages one can make of HySon on some examples.

II. A Motivating Example

We motivate our approach on a simple yet representative example of a controller and show how HySon can be used to increase the confidence in the controller design. We consider a cruise-control system. The *plant* is a simple model of a vehicle moving horizontally on a road with some friction. The dynamics of the speed v is given by $\dot{v} = (u - b\ v)/m$ where u is the power given by the engine, m is the mass of the vehicle, b is the friction of the road and $\dot{v}$ is the derivative of v with respect to time t. The *controller* acts on the power u in order for the vehicle to reach a user-specified speed v_r. We chose to use a discrete PI (proportional-integrator) controller given by: $u(t_k) = k_p \cdot (v_r - v(t_k)) + k_i \cdot h \cdot \sum_{i=0}^{k}(v_r - v(t_i))$ where k_p and k_i are respectively the proportional and integral gains, and the

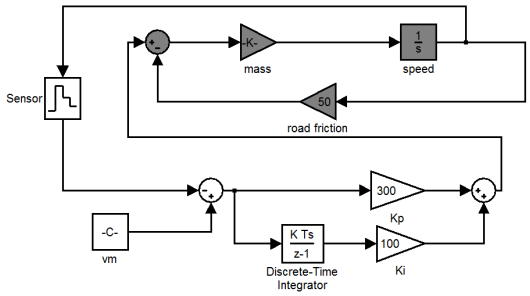


Figure 2: Simulink representation of the cruise-control model. The gray blocks represent the continuous plant system.

t_k are the sampling instants given by $t_k = k\ h$ where h is the sampling rate. Figure 2 shows the Simulink implementation of this system. It should be noted that the sensor block on the left converts a continuous signal into a discrete one.

In a classical model-based design approach, the design choices for the controller (the choice of a PI controller, and of the parameters k_p and k_i) are validated by running some simulations of the system for various values of v_r, and verifying that in each case the high level specifications (maximum overshoot, settling time, etc.) are verified. We see three limitations to this method. First, in this setting, only a finite number of input values v_r can be tested. However, the driver may choose any value for v_r within a known range. Classically, the extreme values of v_r are tested plus some random values in the full range, but this does not ensure a sufficient coverage for the tests cases, as pointed out in [14, Chap 9.2]. Second, when the plant is designed, it is assumed that the mass of the vehicle, for example, is perfectly known. This is often not the case, especially when such physical components are supplied by an external partner: such physical parameters are generally only known up to some precision, and a small variation of a parameter may greatly change the dynamics of the plant (especially for complex non-linear dynamics). The simulation of the plant with *one* value m for the mass of the vehicle may thus not be significant if the real mass is $1.01 \times m$ (i.e. there was a 1% error). Finally, the `sensor` block in this system samples its continuous input signal at a constant rate. It models a perfect sensor that does not introduce any measurement noise, which is not realistic since sensors tend to add random noise, thus perturbing the controller scheme.

When we face only one such uncertainties, Monte-Carlo simulations may offer a good level of confidence in a reasonable time. However, when various sources of uncertainties are combined, the number of simulations required to get a good coverage of all cases is exponential, and random simulations can no longer be used. The solution we are advocating for in this paper is *set-based simulation*. To wit, it is a way to perform exhaustive testing of the system by adapting the simulation algorithm to propagate *sets* of values instead of simply values (represented as floating-point numbers). For example, consider an uncertain mass for the vehicle, and assume that the mass lies within the range $[990, 1010]$. Using HySon, we can run the simulation of the system without having to randomly chose one value of the mass in this range: we automatically propagate the whole interval. The result of this set-based simulation is then

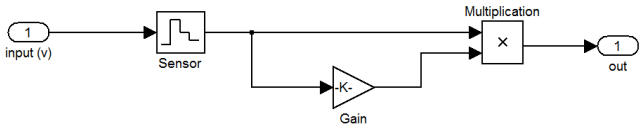


Figure 3: Blocks introducing a uniform noise in a sensor.

a sequence of sets (in details, they are encoded *affine forms*, see Section IV) that contain all the possible values for one random simulation at each simulation time.

On our cruise-control example, we run HySon with two uncertain parameters: the input value v_r is within the range $[12, 15]$ and the mass of the vehicle is within the range $[990, 1010]$. We also modified the sensor block to introduce a 10% random noise: at each sampling time, the measured value v_d is $v_d = (1 + \alpha)v$ where $\alpha \in [-0.1, 0.1]$. To do this in Simulink, we only have to replace the `Sensor` block of Figure 2 with the blocks of Figure 3, and set the parameter of the `Gain` block to the interval $[-0.1, 0.1]$. Note there is a difference about what uncertainty means for the parameters m or v_r, and the noise: the first are uncertain but constant during a whole simulation, while the noise takes a new value at each simulation step. This kind of difference is taken into account and handled by HySon. Figure 4 shows the upper and lower limits of the sets obtained for one simulation in HySon, as well as 3000 random simulations in Simulink. As shown by this example, the set-based simulation proposed by HySon is both complete (it runs all simulations at once) and precise (it does not introduce much pessimism, i.e. over-approximations in the set of possible trajectories). Remark that using HySon, we could analyze the effect of sensor noise on the controller without any other assumptions on the value of this noise than its bounds. In particular, we did not assume any probability distributions on the noise as such assumptions are difficult to prove.

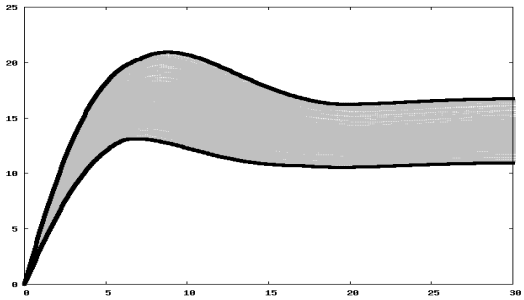


Figure 4: Result of HySon on the uncertain cruise-control system: the dark thick lines are the bounds computed by HySon while the gray dots are random simulations in Simulink.

III. Software Architecture

The overall architecture of HySon is depicted in Figure 5. The tool takes as input two kinds of programs: either a Simulink model or a program written in the special hybrid language named "ODE". The source program is parsed in order

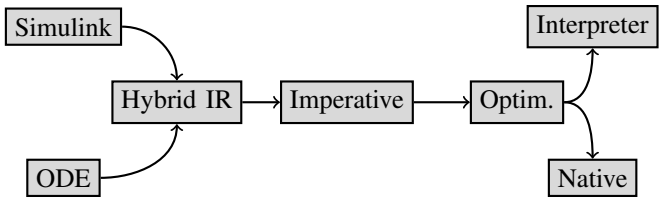


Figure 5: HySon architecture: both languages Simulink and ODE are compiled into a single intermediate representation, optimized, and executed.

```
init v = 15.;
init z = [10.,10.2];
# Initial value of z is unknown in [10,10.2]
gravity = -9.81;
elasticity = 0.8;
v' = gravity;
z' = v;
on z < 0 do { v = -elasticity*v; z = 0; };
output (z);
```

Figure 6: ODE implementation of the bouncing ball.

to generate an intermediate representation, which roughly consists of a sequence of equations. This transformation follows the operational semantics of Simulink, as described in [3]. Currently, the parser handles many of the commonly-used blocks, both continuous and discrete and has been extended since [3] with a fully-fledged integrator (reset and saturation), subsystems and signal routing. HySon handling uncertain parameters, the parser supports ranges in addition to usual floating-point parameters. Those improvements required a fixpoint equation to determine the dimension of a signal and a mix of two different orders of evaluation to handle the reset and external initialization. This representation is then transformed into an imperative program resulting from the implementation of the integration scheme and zero-crossing algorithm, as described in Section IV. Various optimization passes are then performed on the resulting program such as constants propagation, dead code elimination, etc. Finally, the program can be executed directly using a built-in interpreter, or native code can be emitted (currently by compiling a generated OCaml program).

We now quickly present the ODE language, as well as the Simulink parser and the difficulties that had to be overcome during its elaboration.

ODE language. A dynamical system is easily described in our ODE language, which offers a way to provide a textual description of dynamical systems. A program in this language consists of a list of equations of the form `v=e` where `v` is a variable name and `e` is an arithmetic equation. Each variable represents a signal, i.e. a (discrete or continuous) function of time, whose values can either be reals or booleans. For example, the meaning of `z=x+y` is that `x`, `y` and `z` are signals such that $\mathtt{z}(t) = \mathtt{x}(t) + \mathtt{y}(t)$ holds at each instant t. Special kind of equations can be used in order to define the dynamics of the system. First, differential equations are expressed by defining the derivative of a variable over time as the value of some expression: we write it `v'=e` where `v` is a variable and `e` is an arithmetic expression. In this case, the initial value of the variable `v` must be specified using the syntax `init v = v0`. Second, discrete events are defined using the syntax `on(b) do {s}` where `b` is a boolean expression and `s` is a sequence of simple equations. For example, the event `on(z<=0) do {v=-v}` represents the bouncing of a ball on the floor: as soon as the ball hits the floor (its altitude `z` is 0), its speed gets reversed. An example of ODE program is given in Figure 6, which describes the classical *bouncing-ball* system [20].

Simulink parser. If the ODE language is useful for experimenting with small examples, real-world applications are often coded in Simulink. We thus implemented a frontend which translates Simulink models into our intermediate representation. In order to do so, the blocks are sorted according to the links given between the blocks, in order to determine the order in which they are going to be evaluated. When cycles occur in the diagram, those must contain integrator blocks (algebraic loops are not allowed in HySon) which indicate where they should be broken, as explained below. Potentially, external initial conditions must be evaluated first and the state ports require to have the integrator block evaluated twice. In addition, we also consider the *Initial Condition* block which behaves as a source during initialization and as a link by just conveying its input otherwise. To handle all these special cases, two orders actually co-exist: one for the initialization phase and one for the simulation phase.

Using these orders, we can also propagate dimensions through each links to handle *mux/demux* blocks. To do that, we create a special fixpoint equation that determines the dimension of each link and solve it using a Kleene iteration. Note that we could have used Matlab to obtain the evaluation order and dimensions, but we chose to re-implement the sorting and the dimension algorithms so that HySon can be used independently from Matlab. Using the order of evaluation and the dimension of links, we can then translate the model into the formalism specific to HySon and then simulate the model.

The evaluation of the *integrator*s (continuous and discrete) is among the most important part of the simulation process, and its implementation is quite subtle since those blocks are highly configurable. Two main features of the integrator blocks are *saturation* (which allows to not take the input in account as long as the output is above or below a certain threshold), and the *reset* of the internal state of the integrator. The saturation mechanism is implemented by internally adding a zero-crossing event (see Section IV): when the output is above the threshold and the input is positive, the integrator integrates zero instead of the input, until the input becomes negative (the situation where the output is below the threshold is similar). The reset operation in Simulink uses the state port, which is a second output whose value is equal to the output of the integrator except when a reset occurs. It is mainly used to break algebraic loops, since the condition to reset the integration can potentially depend on its own output. For continuous integration, the zero-crossing detection handles that case, but for discrete-time integration, which is sequential, the integrator has to first emit its result through the *state port* and then check, once every blocks it depends on are evaluated, if the condition is met before outputting an appropriate result.

A large effort has been made to handle most of the options of integrator blocks (including different kinds of integration scheme for discrete-time integrators) so that HySon can be used on many models without requiring modifications. Note that other parsers of Simulink models exist, in particular the Gene-auto parser [22]. Our tool handles a larger set of Simulink blocks, including continuous blocks such as the integrator block while Gene-auto focuses on discrete time models.

In the rest of this article, we only present examples in the ODE language or as differential equations, since those are more readable on paper, but our tool could of course takes as input the same systems written in Simulink.

IV. Guaranteed Set-Based Simulation

The foundations of the tool HySon are based on a series of previous works by the authors. First, an operational semantics of a realistic subset of Simulink language, involving both continuous- and discrete-time blocks, was rigorously defined [3]. This semantics was then extended in order to deal with uncertainties, modeled as parameters lying in interval values, hence showing that the simulation engine of Simulink could be adapted in order to compute with sets of values [5]. Finally, classical numerical integration methods (RK4, ODE23, etc.) were adapted in order to compute a safe enclosure of the solution of an ordinary differential equation [4]. We recall here the main ingredients on which the set-based simulation used in HySon is based.

Problem to solve. A Simulink model is associated to a hybrid dynamical system of the simplified form:

$$\dot{\mathbf{y}} = f_c(\mathbf{y}, \mathbf{x}_n, u) \quad \text{continuous-time function} \tag{1a}$$

$$\mathbf{x}_{n+1} = f_d(\mathbf{y}, \mathbf{x}_n, u) \quad \text{discrete-time function} \tag{1b}$$

$$z = g(\mathbf{y}, \mathbf{x}_n, u) \quad \text{output function} \tag{1c}$$

with $\mathbf{x}$ (resp. $\mathbf{y}$) the discrete (resp. continuous) states of the model, u being the input, and y the output. The time derivative of y is denoted by $\dot{y} = \frac{dy}{dt}$. The simulation, i.e. the numerical resolution of Equations (1), uses an iterative method which successively evaluates Equations (1c), (1b) and (1a). The main idea of the simulation algorithm, see [3] for further details, is: at iteration n, from the states $\mathbf{y}_n$, $\mathbf{x}_n$ and the input u_n the algorithm computes an output z_n and updates the internal states $\mathbf{y}$ and $\mathbf{x}$ which will be used in the next iteration. In case of Equation (1a), the value for $\mathbf{y}$ at next considered instant is computed using a numerical integration algorithm. It is often the case for complex, industrial systems that some parameters are uncertain (for example the mass of the vehicle in the introductory example). We model these uncertainties by allowing the functions f_c and f_d to depend on some uncertain parameters. In other words, we consider the system

$$\dot{\mathbf{y}} = f_c(\mathbf{y}, \mathbf{x}_n, u, p_c) \tag{2a}$$

$$\mathbf{x}_{n+1} = f_d(\mathbf{y}, \mathbf{x}_n, u, p_d) \tag{2b}$$

$$z = g(\mathbf{y}, \mathbf{x}_n, u) \tag{2c}$$

where p_c and p_d are formal, unknown but bounded parameters. We assume that we are given sets of possible values for these parameters: $p_c \in \mathbf{P_c}$ and $p_d \in \mathbf{P_d}$. Each choice of the unknown parameters p_c and p_d gives rise to a new dynamical system which, when simulated, produces an output signal z. The goal of HySon is to compute efficiently and precisely an over-approximation of all the signals z for every possible value of the unknown parameters in $\mathbf{P_c}$ and $\mathbf{P_d}$. This is what we call *set-based simulation*, that uses the same algorithms as for numerical simulation but the functions in Equation (2) and the numerical methods are extended to manipulate sets of values. The second use of HySon is to compute *guaranteed set-based simulation*, in which the tool computes an over-approximation of the mathematical solution of Equations (2) rather than of their numerical simulation. This can be used to safely compute bounds on the trajectories of the physical system represented by the equations. In the rest of this section, we detail the methods used to achieve these goals. First, we present our encoding for sets of values, based on the notion of affine forms [15].

Computing with sets of values. In order to represent sets of values, we use *affine arithmetic* [9] which can be seen as a much improved interval arithmetic which is able to track linear correlations between the variables of the model. We chose this domain because it can represent sets of values quite accurately, in a compact way, and usual operations can reasonably be computed on those. A set of values in this domain is represented by an *affine form* $\hat{a}$, which is a formal expression of the form $\hat{a} = \alpha_0 + \sum_{i=1}^{n} \alpha_i \varepsilon_i$ 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 $\hat{a} = \alpha_0 + \alpha_1 \varepsilon$ with $\alpha_0 = (a_1 + a_2)/2$ and $\alpha_1 = (a_2 - a_1)/2$. All classical operations (addition, subtraction, multiplication, ...) are defined over affine forms. For instance, given $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i$ and $\hat{y} = y_0 + \sum_{i=1}^{n} y_i \varepsilon_i$, their sum and product are computed using the following formulas:

$$\hat{x} + \hat{y} = x_0 + y_0 + \sum_{i=1}^{n} (x_i + y_i)\varepsilon_i$$

$$\hat{x} \times \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\varepsilon_i + R\varepsilon_{n+1}$$

where ε_{n+1} is a new noise symbol introduced to represent the over-approximation performed by the linearisation of the multiplication, and $R = \left(\sum_{i=1}^{n} |x_i|\right) \times \left(\sum_{i=1}^{n} |y_i|\right)$. This arithmetic reduces the impact of the *dependency problem* of interval arithmetic, e.g. if $a = [1, 2]$ and we compute $a - a$ the result in interval arithmetic is $[-1, 1]$ while with affine arithmetic is 0, see [5] for more details. We use this arithmetic to both simulate and safely solve Equations (2). These operations rely on a rather simple algorithm that we detail next.

Set-based numerical simulation. Given a system given by Equations (2) with an initial state given by two affine forms $\hat{y}_0$ and $\hat{x}_0$, the following algorithm computes a sequence of affine forms $\hat{y}_n$ and $\hat{x}_n$:

Input: $\hat{x}_0$, $\hat{y}_0$, t_0, h;
$n = 0$;
loop **until** $t_n \geq t_{\text{end}}$
  evaluate $g(\hat{y}_n, \hat{x}_n, u)$
  compute $\hat{x}_{n+1} = f_d(\hat{y}_n, \hat{x}_n, u, \mathbf{P_d})$
  $\hat{y}_{n+1} =$ **solve_ode**$\big(\dot{y}(t) = f_c(y(t), \hat{x}_n, \mathbf{P_c}), \hat{y}_n, h\big)$
  $\hat{y}_{n+1} =$ **solve_zc**$(\hat{y}_n, \hat{y}_{n+1})$
  $n = n + 1$; $t_{n+1} = t_n + h$

This algorithm depends on two functions: solve_ode and solve_zc. solve_ode computes the next value of the continuous variables given the current value $\hat{y}_n$, the ODE and the chosen step-size h. Note that efficient algorithms also modify the step-size according to user-defined tolerances, but we here only give a simplified version that keeps the step-size constant, see [5] for further details on variable step-size. The second function, solve_zc, detects and applies special events that occur between two discretization instants. Depending on the chosen implementation for these two functions (guaranteed or not), we either obtain a set-based simulation or a guaranteed set-based simulation algorithm.

Set-based numerical integration. The main algorithms used to perform numerical integration of ODEs are explicit Runge-Kutta methods [21]. For example, Heun method defines $\hat{y}_{n+1}$ from $\hat{x}_n$ and $\hat{y}_n$ as:

$$\begin{aligned} \hat{k}_1 &= f_c(\hat{y}_n, \hat{x}_n, \mathbf{P_c}) \\ \hat{k}_2 &= f_c(\hat{y}_n + h\hat{k}_1, \hat{x}_n, \mathbf{P_c}) \\ \hat{y}_{n+1} &= \hat{y}_n + \tfrac{h}{2}(\hat{k}_1 + \hat{k}_2) \end{aligned}$$

The evaluation of such scheme using affine form arithmetic produces a tight enclosure of all the simulations one would perform for any uncertain parameter, even for stiff non-linear dynamics.

EXAMPLE 1. *Let us consider the evolution of a quadrotor (some form of helicopter lifting system with four rotors), which is a continuous system with 12 variables with a highly non-linear dynamics (due to the presence of numerous trigonometric operations) given by the following equations [2]:*

$$\begin{cases} \dot{\phi}_1 = \phi_2 & \dot{\phi}_2 = \theta_2\psi_2 a_1 + \theta_2 a_2 \Omega_r + b_1 u_2 \\ \dot{\theta}_1 = \theta_2 & \dot{\theta}_2 = \phi_2\psi_2 a_3 - \phi_2 a_4 \Omega_r + b_2 u_3 \\ \dot{\psi}_1 = \psi_2 & \dot{\psi}_2 = \theta_2\phi_2 a_5 + b_3 u_4 \\ \dot{z}_1 = z_2 & \\ \dot{z}_2 = g - u_1 \cos(\theta)\cos(\phi)/m & \\ \dot{x}_1 = x_2 & \\ \dot{x}_2 = u_1 \left(\cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi)\right)/m & \\ \dot{y}_1 = y_2 & \\ \dot{y}_2 = u_1 \left(\cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi)\right)/m & \end{cases} \tag{3}$$

with

$$\begin{aligned} \Omega_1 &= \alpha_z c_z + \alpha_\theta c_\theta - \alpha_\psi c_\psi & \Omega_2 &= \alpha_z c_z - \alpha_\phi c_\phi + \alpha_\psi c_\psi \\ \Omega_3 &= \alpha_z c_z - \alpha_\theta c_\theta - \alpha_\psi c_\psi & \Omega_4 &= \alpha_z c_z + \alpha_\phi c_\phi + \alpha_\psi c_\psi \\ \Omega_r &= \beta\left(\Omega_2 + \Omega_4 - \Omega_1 - \Omega_3\right) & & \\ u_1 &= \gamma\left(\Omega_1 + \Omega_2 + \Omega_3 + \Omega_4\right) & u_2 &= \gamma\left(\Omega_4 - \Omega_2\right) \\ u_3 &= \gamma\left(\Omega_1 - \Omega_3\right) & u_4 &= \beta\left(\Omega_2 + \Omega_4 - \Omega_1 - \Omega_3\right) \\ a_1 &= (I_{yy} - I_{zz})/I_{xx} & a_2 &= J_r/I_{xx} \\ a_3 &= (I_{zz} - I_{xx})/I_{yy} & a_4 &= J_r/I_{yy} \\ a_5 &= (I_{xx} - I_{yy})/I_{zz} & b_1 &= l/I_{xx} \\ b_2 &= l/I_{yy} & b_3 &= 1.0/I_{zz} \end{aligned} \tag{4}$$

with constants

$$\begin{aligned} \alpha_z &= 7987.2 & \beta &= 7.5 \cdot 10^{-7} \\ \alpha_\theta = \alpha_\phi &= 34727.0 & \gamma &= 3.13 \cdot 10^{-5} \\ \alpha_\psi &= 333333.3 & & \end{aligned}$$

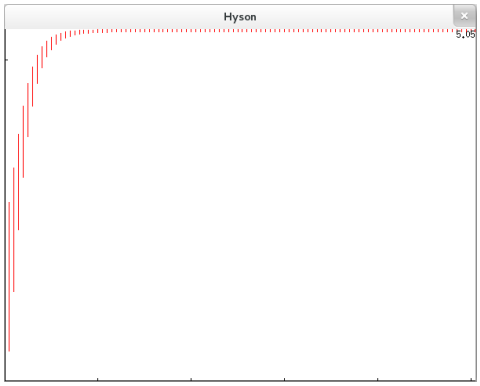


Figure 7: Altitude z in the quadrotor example 1.

where I_{xx}, I_{yy}, I_{zz} (inertia moments), J_r (rotor inertia) and l (distance to center) are physical constants of the engine, and the continuous controller is given by

$$\begin{aligned} c_z &= 1.0 - 0.23\left(100(z_r - z) - 20z_2\right)/\left(\cos(\phi)\cos(\theta)\right) \\ c_\phi &= 0.8\left(\phi_r - \phi\right) - 0.4\phi_2 \\ c_\theta &= 1.2\left(\theta_r - \theta\right) - 0.4\theta_2 \\ c_\psi &= \left(\psi_r - \psi\right) - 0.4\psi_2 \end{aligned} \tag{5}$$

We consider here an instance of the system with the following uncertain parameters:

- *the initial value for z is in $[0, 0.1]$,*
- *the reference value z_r for z is in $[1, 2]$,*
- *the physical parameters of the system are uncertain: the inertial moments I_{xx} and I_{yy} are in $[0.007, 0.012]$, $I_{zz} \in [0.01, 0.02]$ and $l \in [0.2, 0.3]$.*

We simulated this system using HySon up to 5 seconds, and at the end of the simulation we got $z \in [1.09445, 2.0945]$ (the output of the simulation is shown in Figure 7). The simulation took $24.64s$. We also ran 1000 Monte Carlo simulations of the system and got, in $99.79s$, an interval of $[1.09446, 2.0918]$. This demonstrates both the efficiency and the precision of our simulation engine, even for highly non-linear systems.

Guaranteed set-based numerical integration. The previous Heun method computes a value $\hat{y}_{n+1}$ from $\hat{y}_n$ that approximates the set of values

$$Y_h = \left\{ y(h) \,|\, \dot{y}(t) = f_c(y(t), x_n, p_c),\ y(0) \in \hat{y}_n,\ x_n \in \hat{x}_n,\ p_c \in \mathbf{P_c} \right\}$$

Y_h is the set of values of the solution of the ODE (2a) for any initial value in $\hat{y}_n$, any constant value for $x_n \in \hat{x}_n$ and any parameter $p_c \in \mathbf{P_c}$. We thus have $\hat{y}_{n+1} \approx Y_h$. The goal of a guaranteed set-based numerical integration is to solve Equation (2a) such that we obtain an affine form $\hat{Y}_{n+1}$ that encloses the solution Y_h: we want $Y_h \subseteq \hat{Y}_{n+1}$. To do so, we have to bound the error introduced by numerical integration method and we showed, in [4] for the class of explicit Runge-Kutta integration methods, that it can be done by bounding the remainder of a suitable Taylor series. Based on [4], HySon can solve Equation (1) with different guaranteed numerical integration methods which makes it more adaptable to the different kinds of differential equations used to model plants.

For example, if we consider Heun solver again, we may write $\hat{y}_{n+1}$ as $\hat{y}_{n+1} = \Phi(\hat{y}_n, h)$ where Φ is the function:

$$\Phi : \left\{ \begin{array}{rcl} \mathbb{R}^m \times \mathbb{R}_+ & \rightarrow & \mathbb{R}^m \\ (y,t) & \mapsto & y + \frac{t}{2}\Big(f_c(y, \hat{x}_n, \mathbf{P_c}) \\ & & \quad + f_c(y + t \cdot f_c(y, \hat{x}_n, \mathbf{P_c}), \hat{x}_n, \mathbf{P_c}) \Big) \end{array} \right.$$

Then, if we write $\phi(t) = \Phi(\hat{y}_n, t)$, we have that:

$$\hat{y}_{n+1} - Y_h \subseteq \frac{h^3}{6}\left(\frac{d\phi^3}{dt^3}([0,h]) - \frac{df_c^2}{dt^2}(\hat{y}, \hat{x}_n, \mathbf{P_c})\right) \qquad (6)$$

where $\hat{y}$ is an affine form over-approximating the set of values of the solution to the differential equation between time 0 and h. This set is computed using Picard iteration as in [19], [4], and an evaluation of (6) then gives a precise and sound bound on Y_h.

Set-based zero-crossing events. In some cases, the continuous-time dynamics is made of several modes. As a simple example, consider a mass-spring system with two different stiffness coefficients depending on the position of the mass as depicted in Figure 8: the stiffness of the spring is k_1 when the mass is close to the wall and k_2 otherwise.

The switching between the two modes is usually detected by looking at the sign of a function, named *zero-crossing function*. In the case of the mass-spring system this function could be positive if the position of the mass is between the wall and the length ℓ, negative otherwise, and the switching instant is given when the function is zero. A challenge for the simulation engine is to detect this kind of event, named *state-event* or *zero-crossing event*, since the time when they occur is not known a priori, contrarily to the *time-events* associated to the sampling rate of discrete-time sub-systems which is fixed throughout the simulation.

To handle state-events in our simulation framework, we extend Equations (1) to take zero-crossing functions in account. More precisely, we extend the mathematical model of the continuous-time dynamic, with a zero-crossing function v, such that

$$\dot{\mathbf{y}} = \begin{cases} f_{c,1}(\mathbf{y}, \mathbf{x}_n, u) & \text{if } v(\mathbf{y}) \leq 0 \\ f_{c,2}(\mathbf{y}, \mathbf{x}_n, u) & \text{otherwise} \end{cases} \quad \text{with } \mathbf{y}(0) = \mathbf{y}_0, v(\mathbf{y}_0) < 0$$

The dynamic of the continuous-time systems follows $f_{c,1}$ until v equals to zero, and then it follows $f_{c,2}$ (we have described systems with only two modes for simplicity but the general case with possibly more modes is similar). In this case, the simulation algorithm must detect and locate the exact time when the function v is zero. Note that to respect all the hypotheses under which the solution of the continuous dynamics exists, the simulation treats this problem in two steps: first numerically integrate the ODE and then solve the equation $v(\mathbf{y}(t)) = 0$ that is find the time instant where v is zero.

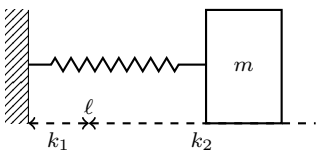


Figure 8: A mass-spring system with two different stiffness.

The classical numerical approach to deal with this kind of equations is to use an interpolation polynomial p in order to approximate the solution $\mathbf{y}$ and find the zeros of v. In consequence, we need to solve $v(p(t)) = 0$ instead of $v(\mathbf{y}(t)) = 0$ with a bisection-like algorithm. Note that the polynomial approximation avoids the use of numerical integration methods, which are computationally expensive, to find zeros of v, which depends on $\mathbf{y}$, and then increases the performance of the simulation. We recall the main steps of this method to facilitate the understanding of our contribution on this particular feature in the set-based simulation framework. Assume we follow the dynamic $\dot{\mathbf{y}} = f_{c,1}(\mathbf{y}, \mathbf{x}_n, u)$ and we want to detect an event associated to the zero-crossing function v. After a successful numerical integration between instants t_n and t_{n+1} we have the available information: at instant t_n we know the solution $\mathbf{y}_n$ and at instant t_{n+1} we know y_{n+1}. To detect an event we compute $v(\mathbf{y}_n)$ and $v(\mathbf{y}_{n+1})$ and we compare the signs of these two results. If the sign is different then an event is detected. Note that this approach is not safe as for example the function v may change its sign twice during the integration step, see [23] for further explanations. If an event is detected then the simulator looks for the precise time when the event occurs that is when v is zero. To do so, it uses an Hermite-Birkhoff approximation which from $\mathbf{y}_n$, $\mathbf{y}_{n+1}$ and $f_{c,1}(\mathbf{y}_n)$ and $f_{c,1}(\mathbf{y}_{n+1})$ defines the polynomial function:

$$\begin{array}{rcl} p(t) & = & (2\tau^3 - 3\tau^2 + 1)\mathbf{y}_n + (\tau^3 - 2\tau^2 + \tau)h f_{c,1}(\mathbf{y}_n) \\ & & + (-2\tau^3 + 3\tau^2)\mathbf{y}_{n+1} + (\tau^3 - \tau^2)h f_{c,1}(\mathbf{y}_{n+1}) \end{array} \qquad (7)$$

with $\tau = (t - t_n)/h$ and $h = t_{n+1} - t_n$. This polynomial function is then used in a iterative algorithm to solve $v(p(t)) = 0$.

We can adapt the previous algorithm to deal with uncertainties using affine arithmetic for evaluating the function v and the interpolation polynomial p, hence we have a set-based detection of zero-crossing event. The detection of the event is translated into a comparison of signs between the affine evaluations of v with $\hat{\mathbf{y}}_n$ at time t_n and $\hat{\mathbf{y}}_{n+1}$ at time t_{n+1}, note this approach is still not safe. The enclosure of the time instants of the zero-crossing events still uses a bisection-like algorithm using the affine evaluations of p, see [5]. We now deal with a continuum of zero-crossing events instead of one single event. This induces different situations in the detection of zero-crossing event depicted in Figure 9. In Case a, the whole set of trajectories generates a set of zero-crossing event then we have to enclose an interval of times associated to it. In Case b, a part of the trajectories generates a zero-crossing event while the other part not. In that case, we can split the set of trajectories in two subsets and continue the simulation with each subset. An other solution is to continue the simulation by gathering all the values generated by the integration until all the trajectories stop generating an zero-crossing event and then post-process the list of values to enclose the time associated to the zero-crossing event. In Case c, due to the pessimism of the affine arithmetic, we may detect some spurious events that we detect events that are not in the real system. In that case, the algorithm used to compute the enclosure of the time of the zero-crossing event should detect the absence of event.

We propose in this article to adapt this approach in order

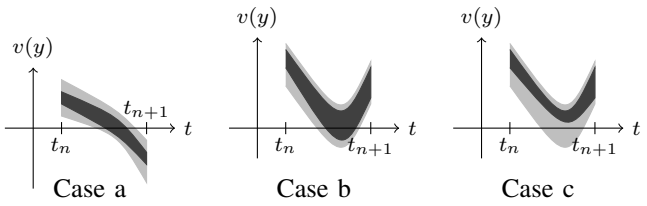


Figure 9: Three cases for zero-crossing events. Exact trajectories are depicted in dark gray, the over-approximated flow pipes in light gray.

to guarantee the detection of event and the computation of the time when zero-crossing event occurs.

Guaranteed set-based zero-crossing events. The challenge in defining a guaranteed detection of zero-crossing event is twofold, the detection of the event must be guaranteed and the computation of the enclosure of time instants must also be guaranteed. We explain in the sequel our contribution on these two challenges.

We guarantee the detection of the occurrence of a zero-crossing event by using the result of the Picard iteration. Recall that the Picard iteration produces an enclosure of the solution $\hat{\mathbf{y}}$ of $\dot{\mathbf{y}} = f(\mathbf{y}, \mathbf{x}_n, u)$ on the time interval $[t_n, t_{n+1}]$. So evaluating with affine arithmetic v with $\hat{\mathbf{y}}$ produces a safe enclosure of the behavior of v on the interval $[t_n, t_{n+1}]$ hence if zero is in the result of this evaluation then we guarantee the presence of at least one event. Note also that due to the pessimism of affine evaluation, we can detect spurious events but the method is safe.

We guarantee the computation of the zeroes of v by defining a safe interpolation polynomial. The idea is to bound the interpolation error in order to safely enclose the solution $\mathbf{y}$ between to integration time instants and use this safe interpolation to find an enclosure of the time at which the dynamics changes. Suppose given $\hat{\mathbf{y}}_n$, $\hat{\mathbf{y}}_{n+1}$, $f_{c,1}(\hat{\mathbf{y}}_n)$ and $f_{c,1}(\hat{\mathbf{y}}_{n+1})$ as affine forms, we can define the polynomial function as in Equation (7). We know that the interpolation error, that is the maximal distance between the interpolation polynomial function and the exact function, is

$$\mathbf{y}(t;\mathbf{y}_0) - p(t) \quad = \quad \frac{\mathbf{y}^{(4)}(\xi)}{4!}(t-t_n)^2(t-t_{n+1})^2 \quad \text{with } \xi \in [t_0, t_n] \ ,$$

which can be reformulated as

$$\mathbf{y}(t;\mathbf{y}_0) - p(t) \quad = \quad \frac{f^{(3)}(\xi, \mathbf{y}(\xi))}{4!}(t-t_n)^2(t-t_{n+1})^2$$

with $\xi \in [t_k, t_{k+1}]$. In other words, we can express the interpolation error in terms of the derivatives of f the function defining the continuous dynamics. In consequence, to guarantee the polynomial interpolation, it is enough to know an enclosure $\hat{\mathbf{y}}$ of the solution $\mathbf{y}(t)$ of IVP on the interval $[t_k, t_{k+1}]$. And fortunately, we can reuse the result of the Picard operator in that context, that is the meaning of Theorem 1.

THEOREM 1. *Let $\hat{p}(t)$ be the interpolation polynomial based on guaranteed solutions $\hat{\mathbf{y}}_n$ and $\hat{\mathbf{y}}_{n+1}$ of an* IVP (1a) *and affine forms of $f_{c,1}(\hat{\mathbf{y}}_n)$ and $f_{c,1}(\hat{\mathbf{y}}_{n+1})$, and let $\hat{\mathbf{y}}$ be the result of the Picard operator. We have, $\forall t \in [t_k, t_{k+1}]$,*

$$\mathbf{y}(t) \in \text{Aff}\left(\hat{p}(t) + \frac{f^{(3)}([t_k, t_{k+1}], \hat{\mathbf{y}})}{4!}(t-t_n)^2(t-t_{n+1})^2\right) \ .$$

Using the guaranteed interpolation polynomial function of the solution of the IVP between time instants t_n and t_{n+1}, we can safely enclose by iterative methods the time when zero-crossing events occur.

Guaranteed floating-point computations. Due to the finite representation of floating-point numbers, most operations on those (even simple ones such as addition) give rise to approximation errors. In the context of a very unstable or stiff system, this can lead to large accumulated errors during a simulation. This is why those are considered in HySon: the ideal theoretical simulation of the system using infinite precision real numbers is guaranteed to be contained in the interval of values provided by the simulation.

V. HYSON FEATURES

We now present in more details the kinds of systems HySon can handle and the information one can obtain with this tool.

Plant design. HySon can handle continuous blocks in Simulink (such as the integrator block) as well as non-linear operations. It also supports zero-crossing detection and advanced Simulink blocks such as resets, subsystems, mux/demux or signal routing. In this way, HySon can be used to validate the design of complex plants with non-linear behaviors. In particular, it can be used to check the robustness of a plant design with respect to perturbations expressed as uncertain parameters.

EXAMPLE 2. *We consider a train model*[1] *and check its behavior with a step input. A simulation of this system shows that the response time is small (around 30s) and that the oscillations before entering the steady state are small (see Figure 10, dashed line). We introduced a* 10% *uncertainty on the mass of the train and run set-based simulations in HySon. Result is shown on Figure 10, full lines. We could prove that the uncertain system had the same settling time and that the oscillations remained bounded. Note that, although this model is linear, it must be simulated on a long time (up to $t = 1000s$). HySon is stable enough to handle this long simulation time.*

Open loop controller. HySon also treats discrete systems. In particular, we handle various sampling times in different blocks, and discrete blocks such as the *Unit-Delay*, the *Zero-Order Hold* or the *Discrete-Time Integrator*. It also handles logic operations and mathematical functions. It can thus be used to perform formal verification of discrete systems in Simulink, as it is done by abstract interpretation tools like [8], [10] on software. Beside an earlier detection of bugs, this can be useful to easily define representative inputs of the system, using the Signal Builder block of Simulink for example.

EXAMPLE 3. *Consider a dynamical system given by $\ddot{y} + 0.6\dot{y} + y + y^3 = u$, that we call the Duffing system, and we consider u to be the signal defined by Figure 11a, in dotted. To handle more representative inputs, we introduce uncertainties*

[1] http://bit.ly/16T5QWF

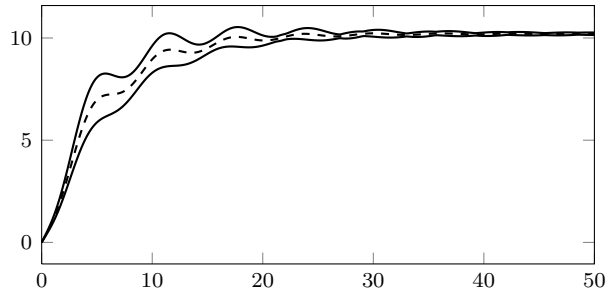
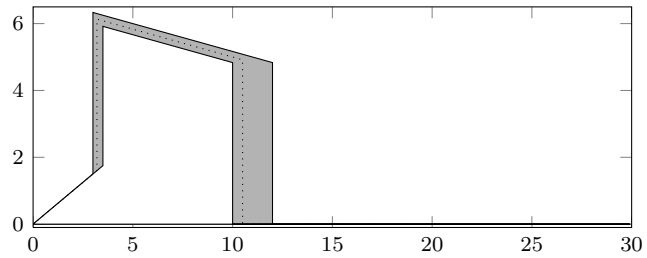
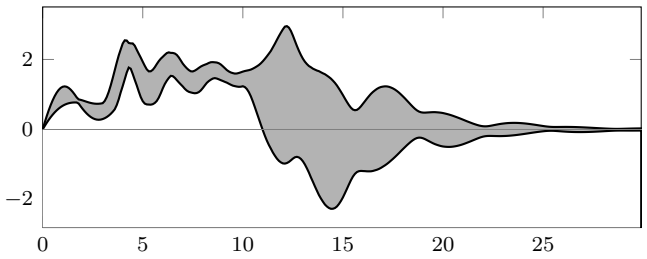


Figure 10: Simulations of the train system: Simulink plot in dashed lines and HySon bounds, zooming on the first 50 seconds.



(a) Inputs of the system. In dotted, the initial input. In gray, the set of all inputs considered with the uncertainties.



(b) Outputs of the system with the uncertain inputs.

Figure 11: Inputs and outputs for the Duffing system.

on the switching times t_1 and t_2 of u and on the derivative of u between t_1 and t_2. This gives a set of signals contained in the gray region of Figure 11a. We also show how the dynamical system responds to these inputs in Figure 11b. We see that during the uncertain switching phase, the upper and lower bounds of the output seem to diverge, but the output remains stable as all trajectories finally converge towards 0. The reason of the local instability is that we don't know during the switch phase the value of u so that all possible values between the upper and lower bounds. This explain the local divergence on the output y as the value of u is then very large.

Closed-loop controller. Finally, the main advantage of HySon is that it can analyze closed-loop systems as shown by the cruise-control system of Section II. Using the set-based simulation algorithms, we can estimate the properties of the controller (overshoot, settling time, ...) even in presence of uncertainties. To do that, we mainly add observer to the system that infer the requested property. For example, an observer that should compute the overshoot of a certain variable v will detect, using a zero-crossing event, when the derivative $\dot{v}$ becomes 0, indicating a local maximum or minimum for v. We then can compute the maximal value v_m of v during the simulation and thus the overshoot, defined as $(v_m - v_r)/v_r$ if v_r is the reference value.Using the guaranteed version of the simulation algorithms, we can *prove* maximal and minimal bounds on these properties. In this way, HySon can be used for robust control design and can help reduce the safety margins taken when designing a controller as it can consider *realistic* models of sensors and uncertainties and it *precisely* computes the propagation of these uncertainties through the system.

EXAMPLE 4. *We consider a vehicle whose dynamics is given by the non-linear equations*

$$m\dot{v} \quad = \quad f(u,v) - f_a(v) - f_g(v) - f_d(v) \tag{8}$$

where

$$f(u,v) \quad = \quad 2280u\left(1 - 0.4\left(\frac{12v}{420} - 1\right)^2\right)$$

is the engine force with u is the control input (the desired speed is noted v_r), v the speed of the car, m its mass and g the gravitational constant. The force $f_a = 0.01mg$ is the rolling friction, the force $f_g = 0.4992v^2$ is the aerodynamic drag and and the force $f_d = mg\sin(\theta)$ is the road slope force where θ is the slope angle.

The PI control is defined by

$$\begin{aligned} u &= k_p\,(v_r - v) + e_i \\ \dot{e}_i &= k_i\,(v_r - v) \end{aligned} \tag{9}$$

The uncertain parameters are the slope $\theta \in [0.2, 0.5]$ of the road, and in the initial condition of the speed of the car: $v(0) \in [15, 20]$.

To compute the over-shoot of the system, we add the following zero-crossing event:

$$w_o = max(w_o, v) \text{ when } \dot{v} = 0$$

In this way, the final value of w_o is the maximal value of v, allowing to compute the overshoot. The table below gives the value of the overshoot inferred by HySon and by 100 and 1000 Monte-Carlo simulations.

HySon		Monte Carlo			
		100		1000	
w_o	Time (s)	w_o	Time (s)	w_o	Time (s)
[30.18,31.89]	10.14	[30.54,31.76]	3.4	[30.48,31.83]	33.92

VI. CONCLUSION AND FUTURE WORK

We have presented HySon, a guaranteed set-based simulator for Simulink, illustrating on simple examples some of its potential applications in an industrial context. Starting from a Simulink model that may contain both discrete and continuous blocks as well as uncertainties in some parameters, HySon computes the set of all possible simulation traces for any value of the uncertain parameters. In this way, HySon can be use to validate the design choices by providing a more reliable simulation process than Monte-Carlo simulations. HySon also provides a *guaranteed* simulation process in which the numerical algorithms used for simulation are turned

into *guaranteed* algorithms that compute over-approximations rather than approximations. HySon can be used to formally prove the safety of a closed-loop controller.

Other tools exist for formal verification or test case generation of Simulink models [12], [18], [1], but they are mostly concerned with discrete systems. Hybrid systems model checking [13], [7] aims at computing over-approximations of hybrid dynamical systems. Most tools however are based on the hybrid automata framework that cannot express the full complexity of Simulink models, in particular discrete blocks with timing. Finally, the work that is closer to ours is the BREACH toolbox and its recent applications [11], [16] that infers possible characteristics of a closed-loop system from a finite set of simulations. Our approach can be used to speed-up this process as it can run infinitely many simulations at once, precisely and efficiently.

There are clearly many ways to develop this technique, let us now list the future challenges that we see for the model-based verification of control systems. First, we want to investigate the Software In the Loop phase to analyze a software implementation of the controller interacting with the Simulink model of the plant. This should not be too hard as we share with the Fluctuat tool the same abstract domain. Second, we want to extend HySon to provide the same set-based simulations for models written in other languages, such as Modelica or AMESim. In this way, we should also be able to verify systems developed using different models (Simulink and Modelica for example). The main challenge here is that these systems are generally based on a different mathematical model: for example, Modelica is based on Differential Algebraic Equations (DAEs) while Simulink is based on Ordinary Differential Equations (ODEs). Thus, handling Modelica imply that we integrate DAE solvers such as [19], [17] into our framework. We also want to continue improving our technique for formal verification of Simulink closed-loop models. To do so, we need to infer invariants on the trajectories of the system that remain true for unbounded time. For example, we want to infer Lyapunov-like information about the plant trajectories. Finally, one promising direction of improvements is to use variants of the affine form domain that also handle probabilistic information, such as the probabilistic affine forms of [6]. In this way, we may provide guaranteed probabilistic information about the output of a system, i.e. we compute bounds on the probability distribution of the output of the system. This makes much sense to analyze systems with probabilistic noise for example, and we believe that such a probabilistic information can be very useful for engineers developing embedded systems.

REFERENCES

[1] D. Bahrami, A. Faivre, and A. Lapitre. Diversity - tg, automatic test case generation from matlab/simulink models. In *ERTS*, 2012.

[2] S. Bouabdallah. *Design and control of quadrotors with application to autonomous flying*. PhD thesis, 2007.

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

[4] O. Bouissou, A. Chapoutot, and A. Djoudi. Enclosing temporal evolution of dynamical systems using numerical methods. In *NASA Formal Methods*, number 7871 in LNCS. Springer, 2013.

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

[6] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94:189–201, 2012.

[7] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2013.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. M. D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[9] L. H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications*. Brazilian Mathematics Colloquium monographs. 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 of *LNCS*, pages 53–69. Springer, 2009.

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

[12] J.-F. Étienne, S. Fechter, and E. Juppeaux. Using Simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain. In *CSDM*, 2010.

[13] G. Frehse, C. L. Guernic, A. Donzé, R. Ray, and al. et. Spaceex: Scalable verification of hybrid systems. In *CAV*, 2011.

[14] GMV Company. SOMCA: Safety implications in performing SOftware Model Coverage Analysis. Research project, EASA, 2010.

[15] E. Goubault and S. Putot. A zonotopic framework for functional abstractions. *CoRR*, abs/0910.1763, 2009.

[16] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia. Mining requirements from closed-loop control models. In *HSCC*, 2013.

[17] P. Kunkel and V. Mehrmann. *Differential-algebraic Equations: Analysis and Numerical Solution*. EMS textbooks in mathematics. European Mathematical Society, 2006.

[18] S. Mohalik et al. Automatic test case generation from simulink/stateflow models using model checking. *STVR journal*, 2013.

[19] N. Nedialkov. Interval tools for ODEs and DAEs. Technical Report CAS 06-09-NN, Dept. of Computing and Software, McMaster University, 2006.

[20] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Heidelberg, 2010.

[21] L. Shampine and M. Reichelt. The MATLAB ODE Suite. *Journal on Sci. Comput.*, 18(1):1–22, 1997.

[22] A. Toom, T. Naks, M. Pantel, M. Gandriau, and Indrawati. Gene-Auto: an Automatic Code Generator for a safe subset ofSimulink/Stateflow and Scicos. In *ERTS*, 2008.

[23] F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *IFAC*, 2008.