

An Operational Semantics for Simulink's Simulation Engine

Olivier Bouissou

CEA LIST, DILS/LMeASI – Point Courrier 174,
F91191 Gif sur Yvette CEDEX, France
olivier.bouissou@cea.fr

Alexandre Chapoutot

ENSTA ParisTech, UEI – 32 boulevard Victor 75739
Paris cedex 15, France
alexandre.chapoutot@ensta-paristech.fr

Abstract

The industrial tool Matlab/Simulink is widely used in the design of embedded systems. The main feature of this tool is its ability to model in a common formalism the software and its physical environment. This makes it very useful for validating the design of embedded software using numerical simulation. However, the formal verification of such models is still problematic as Simulink is a programming language for which no formal semantics exists. In this article, we present an operational semantics of a representative subset of Simulink which includes both continuous-time and discrete-time blocks. We believe that this work gives a better understanding of Simulink and it defines the foundations of a general framework to apply formal methods on Simulink's high level descriptions of embedded systems.

Categories and Subject Descriptors D.3.1 [Formal Definition and Theory]: Semantics; D.3.2 [Language Classifications]: Data-flow languages; I.6.4 [Simulation and Modeling]: Model validation and analysis

General Terms Languages, Reliability, Verification

Keywords Hybrid systems, Operational semantics

1. Introduction

Design of embedded systems is based on tools coming from control theory. Such tools use a mathematical model of the plant, generally with differential equations, which is then used to define a mathematical model of a controller. This model may contain both continuous- and discrete-time parts. An exact solution of this mathematical model is in general not computable and numerical simulation methods are used to study the behavior of the closed-loop system made of the controller and the plant. This is even more obvious when the model involves complex non-linear computations.

Matlab/SimulinkTM is the de facto standard tool used for the design of embedded control systems. The cycle of development based on Simulink, especially in automotive industry, runs as follows. First, the Model in the Loop (MIL) step concerns the definition of a mathematical model of the plant and the control law. Numerical simulation is used to validate that the control law fulfills the specifications. Moreover, the simulations results will be used as an "oracle" for the validation of the next steps. Then, the Software in

the Loop (SIL) step concerns the implementation of the control algorithm (automatically or manually) in a low-level language such as C. Results of the numerical simulation of the plant model and the controller implementation are compared to those obtained at the MIL step. Finally, the Processor in the Loop (PIL) step concerns the compilation of the controller implementation into an executable running on a particular hardware. Results of the simulation of the plant model and the execution of the executable (on an emulator or on a board) are compared to the results obtained at the MIL step. In summary, we can see that Simulink is used in all the steps of the cycle of development and that the numerical simulation plays a crucial role in the design and implementation of control-command software. Even more, the designer accepts or rejects the design of embedded software based on the results of numerical simulations.

As a widely used tool, Simulink is a good target to apply formal verification methods on the earlier stage of the design of control systems. One of the goals of the formal verification of Simulink models is to guarantee that the mathematical model of the controller fulfills the specification [2, 5, 6, 12, 20]. Presently, this is done by numerical simulations of the model and checking if the simulations fulfill the specifications. Our long term goal is to formally verify the correctness of the numerical simulation process w.r.t. the mathematical behavior. That is, we want to verify that the approximations introduced in the different steps of the numerical simulation are small enough to ensure that the run of the real system will not violate the specification. The first challenge in this task is to understand how Simulink simulation engine works because no formal semantics of it exists. This article focuses on this task.

Simulink lacks a formal semantics. But contrary to the other approaches, we do not consider the solver as a black box. We think that hiding the mechanism of the solver prevents the comparison between the results of the numerical simulation and the mathematical behavior. Our main contribution is a formal definition of a structural operational semantics [15] of the Simulink solver that emphasizes how the numerical simulation works. We consider a core subset of Simulink with both discrete and continuous blocks, as well as the zero-crossing detection process.

The rest of the article is organized as follows. In Section 2, we present an example of a Simulink program and explain in more details why we believe it is important to apply formal methods on it. We then briefly present the main tools used in numerical simulation in Section 3. In Section 4, we show how to transform a Simulink program as a sequence of equations and define some notations. Finally, we define our operational semantics in Section 5 and discuss its correctness in Section 6. We then conclude by comparing our approach to related work in Section 7.

2. Motivating example

The plant We consider a system that controls the speed of a vehicle. The physical system (also named the plant) is a vehicle whose speed v and position x is given by the differential equations

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LC TES 2012 June 12–13, 2012, Beijing, China.

Copyright © 2012 ACM 978-1-4503-1212-7...\$10.00

of Equation (1) ($\dot{x} = dx/dt$ stands for the derivative of x w.r.t. time t) where b is the friction coefficient (reducing the speed of the vehicle), m is the mass and $u(t)$ is the power given by the engine over time.

$$m\dot{v}(t) = -b \times v(t) + u(t) \quad \text{and} \quad \dot{x}(t) = v(t) . \quad (1)$$

The controller The controller defines the function $u(t)$ such that the speed $v(t)$ stabilizes at $10m.s^{-1}$, and the speed reaches $9m.s^{-1}$ in less than 3 seconds. To achieve these requirements, we use a Proportional-Integral controller (PI controller). So, the function $u(t)$ is given by Equations (2a) to (2c).

$$\forall t \in \mathbb{R}, u(t) = 900 \times (v_m - v(t)) + 40 \times u_i(\lfloor t \rfloor_k) \quad (2a)$$

$$\forall k \in \mathbb{N}^+, u_i(\lfloor t \rfloor_k) = \sum_{j=0}^k 0.1 \times (v_m - v(0.1 \times j)) \quad (2b)$$

$$\lfloor t \rfloor_k = \max\{0.1 \times k \mid k \in \mathbb{N} \wedge 0.1 \times k \leq t\} \quad (2c)$$

Equation (2b) is the implementation of the integral part of the PI controller using Euler method and sampling rate of 0.1. We see that this control is *discrete-time* as it only outputs new values at specific time instants, i.e., at $t = k \times 0.1$ for some $k \in \mathbb{R}$. Note that the proportional term (see Equation (2a)) is implemented using an analog controller to ensure a fast response (second requirement).

Moreover, we added a safety mechanism to the system: we assume that, as soon as the vehicle reaches 7.1 meters, it tries to stop, i.e., to reach the speed $v_m = 0$. This would be, for example, a device that continuously monitors the position of the vehicle and sends a signal as soon as this position passes 7.1 meters. So in the motion equations (Equations (2a) and (2b)), the term v_m is actually given by: if $x(t) < 7.1$ then $v_m(t) = 10$, otherwise $v_m(t) = 0$. This will be useful to illustrate zero-crossing methods.

Figure 1 shows a Simulink implementation of the whole system made of the plant, the discrete and the continuous-time controller. We will describe with more details in Section 4 the Simulink language. On the right of the system, circled with dashed lines, there is the plant where the *integrator* blocks (labeled with dv and dx) express differential equations. On top of the system, circled with dotted lines, is the continuous-time part of the controller, while the discrete-time controller is circled with full lines. In the discrete-time controller, the *Unit Delay* block is a one-buffer memory and serves for the discrete integration. Remark that nothing in this block diagram indicates that some part of it is discrete; it is only the options set for the *Gain-h* block (which multiplies its input by h) that makes the subsystem discrete: we specified that this block has a sampling rate of 0.1s. The safety mechanism is modeled by the *Switch* block on the left of the diagram: it changes the desired speed to 0 as soon as the position x is such that $x \geq 7.1$.

From simulation based validation to formal verification We want to stress two main points: first the physical environment plays an important role in the design of the controller, second numerical simulation is used to check if the design verify the requirements.

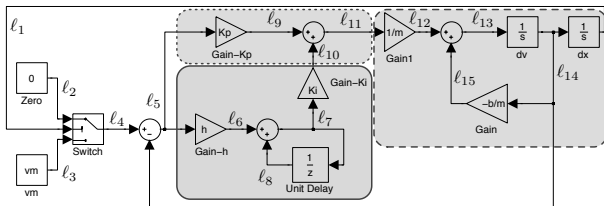


Figure 1. A cruise-control system in Simulink. We mark the plant (circled with dashed lines), the continuous controller (circled with dotted lines) and the discrete controller (circled with full lines).

However, numerical simulation alone cannot be trusted due to numerical approximations: even for this simple, linear example.

Of course, validation of low-level code implementation of embedded control software will always be mandatory in safety critical systems. Abstract interpretation-based static analysis [3, 8] in particular is more and more used. However, formal verification techniques applied at this level face two difficulties: the control algorithm is flooded with implementation details (mainly to increase efficiency) such as the use of fixed-point arithmetic or pointers, and there is no information on the physical environment which strongly influences the run of the control-command system. We thus believe that new methods to formally verify more complex properties such as the requirements on the vehicle controller have to be developed as the continuous environment must be taken into account to prove the requirements. As Simulink allows to model the environment and the discrete controller using the same formalism, we believe that it is the good level of description to apply formal verification on control embedded systems with their physical environment.

In that context, we believe that our approach fits the next challenges of formal verification defined in the articles [7, 18]. Our claim in this article is that, before defining formal verification methods on Simulink models, we must understand what Simulink does. In this article we define an operational semantics of Simulink models which will give us the fundamental basis to apply static analysis-based formal verification of control embedded software taking into account their physical environments.

3. Simulation engine overview

In this section, we briefly describe the main features of the numerical simulation of *dynamical systems* such as the one presented in Section 2. Indeed, the behavior of the Simulink solver relies on such methods and so does the semantics we define in Section 5.

3.1 Simulation steps

A classical mathematical model to represent dynamical systems is the *state-space representation*. In our case a state-space is described by a system of equations (see Equations (3)), t being the time variable, made of a *continuous-time state function* $f_x : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \rightarrow \mathbb{R}^{n_x}$, a *discrete-time state function* $f_d : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \rightarrow \mathbb{R}^{n_d}$ and an *output function* $g : \mathbb{R} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_d} \rightarrow \mathbb{R}^m$.

$$\dot{\mathbf{x}}(t) = f_x(t, \mathbf{x}(t), \mathbf{d}(t)) \quad \text{with} \quad \mathbf{x}(0) = \mathbf{x}_0 \quad (3a)$$

$$\bar{\mathbf{d}}(t) = f_d(t, \mathbf{x}(t), \mathbf{d}(t)) \quad \text{with} \quad \mathbf{d}(0) = \mathbf{d}_0 \quad (3b)$$

$$\mathbf{y} = g(t, \mathbf{x}(t), \mathbf{d}(t)) . \quad (3c)$$

The continuous state is \mathbf{x} , the discrete state is \mathbf{d} and the output is \mathbf{y} . For the sake of simplicity, we consider that the potential input \mathbf{u} of the system is encoded in the functions f_x , f_d and g . In this article, a bold symbol represents a vector of values or variables. We denote by n_x the number of continuous state variables, by n_d the number of discrete state variables and by m the number of outputs. The notation $\bar{\mathbf{d}}(t)$ stands for a difference operator: $\bar{\mathbf{d}}(t) = (d_1(t + s_1), \dots, d_{n_d}(t + s_{n_d}))$, where each discrete variable d_i evolves at rate s_i . Note that we adapt the classical definition of state-space, which is only continuous-time or only discrete-time, to represent the fact that Simulink models are hybrid systems, i.e., a mix of discrete-time and continuous-time systems.

The solution of the state-space equation is a flow function $\phi_x(t)$ which is the solution of the differential equation (Eq. (3a)) and a step function $\phi_d(t)$ which is the solution of the finite difference equation (Eq. (3b)). A close form of these functions is not computable in general, so the goal of the numerical simulation is to compute approximated solutions $\tilde{\phi}_x(t)$ and $\tilde{\phi}_d(t)$ based on a temporal discretization which may be dynamically chosen during the

simulation. As the function ϕ_d is a step function with discontinuities at specific time instants (given by the rates s_i), the temporal discretization will have to be chosen so that it contains all these instants. The functions $\tilde{\phi}_x$ and $\tilde{\phi}_d$ are defined as a sequence of approximations of the solution of the state-space equations on small intervals $[t_n, t_n + h_n]$. Each approximation is computed using a numerical solver that behaves well on small time intervals. So in Simulink, the simulation of dynamical systems (i.e., the solver) follows a simple simulation loop where t is the time and h is the integration step-size which defines the mesh of time:

```

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 values $\mathbf{x}_0, \mathbf{d}_0$ and h_0 are the initial values of the state variables and of the step-size respectively. The simulation loop behaves as follows. The output function of the system is first evaluated and the next value \mathbf{d}' of \mathbf{d} is stored. Then the solution of continuous state function f_x is computed with a numerical integration method on a small interval defined by the step-size h_n . Note that the result of this stage is a new value \mathbf{x}_{n+1} which will be used as the initial value of the next iteration. We present in Section 3.2 the algorithms for computing \mathbf{x}_{n+1} . The third step, find_zero_crossing, is the detection of particular events appearing between t_n and $t_n + h_n$, as for example the safety mechanism in Section 2. More details on zero-crossing methods are given in Section 3.3. Finally, the time t_n and the integration step-size h_n are updated. Note that the step-size may vary during the simulation (for more details see Section 3.2).

In order to have an approximation close to the mathematical model, the time instants chosen by the simulation loop should be such that they verify the following properties. Property 1 states that the numerical simulation will eventually end, that is the time t_n keeps increasing at each iteration until it reaches t_{end} .

Property 1 (Monotone simulation time). *Let \mathbf{x}_0 be the initial state. Let $\{t_k : k \in [0, n]\}$ be the set of $n + 1$ time instants taken during the simulation such that $t_0 = 0$ and $t_n = t_{\text{end}}$. Then we have:*

$$\forall i \in [0, n - 1], t_i < t_{i+1} .$$

Property 2 guarantees a good interaction between the discrete-time part and the continuous-time part of a system. Actually, to compute the value of \mathbf{x}_{n+1} at time $t_n + h_n$, the numerical integration methods require that f_x is (at least) continuous. In particular, this means that the discrete variables must be constant between t_n and $t_n + h_n$. Property 2 ensures this.

Property 2 (Consistent sampling time). *Let \mathbf{x}_0 be the initial state. Let $\{t_k : k \in [0, n]\}$ be the set of $n + 1$ time instants taken during the simulation. Let also $S = \{k \cdot s_i : k \in \mathbb{Z}, i \in [0, n_d]\}$ be the set of all discrete instants where the variables d_i are updated according to the rate s_i . Then we have:*

$$\forall t \in S, t_0 \leq t \leq t_n \Rightarrow t \in \{t_k : k \in [0, n]\} .$$

3.2 Numerical integration

We now describe how the continuous-time part (Eq. (3a)) of a dynamical system is updated during the simulation loop. To compute an approximation \mathbf{x}_{n+1} of \mathbf{x} at time $t_n + h_n$ from an approximation \mathbf{x}_n at time t_n , the solver must solve an *initial value problem*:

$$\dot{\mathbf{x}}(t) = f_x(t, \mathbf{x}(t), \mathbf{d}_n) \quad \text{with} \quad \mathbf{x}(t_n) = \mathbf{x}_n . \quad (4)$$

Let us note $f(t, \mathbf{x}) = f_x(t, \mathbf{x}, \mathbf{d}_n)$. An important known result [11] is that Equation (4) has a unique solution if $f : \mathbb{R} \times \mathbb{R}^{n_x} \rightarrow$

\mathbb{R}^{n_x} is continuous in the first variable and f satisfies a Lipschitz condition in the second variable. By extension, all functions piecewise Lipschitz continuous f have also a unique solution.

Simulink can use many of numerical integration methods, see [16] for a complete list of them to solve Equation (4). A numerical integration method computes the approximation \mathbf{x}_{n+1} using \mathbf{x}_n and approximations of the derivative of $\mathbf{x}(t)$ on the interval $[t_n, t_n + h_n]$. For example, the solver named ode23 in Simulink, based on the Bogacki-Shampine method [17], is a variable step-size method defined at Equations (5).

$$k_1 = f(t_n, \mathbf{x}_n) \quad (5a)$$

$$k_2 = f(t_n + (1/2)h_n, \mathbf{x}_n + (1/2)h_n k_1) \quad (5b)$$

$$k_3 = f(t_n + (3/4)h_n, \mathbf{x}_n + (3/4)h_n k_2) \quad (5c)$$

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h_n ((2/9)k_1 + (1/3)k_2 + (4/9)k_3) \quad (5d)$$

$$k_4 = f(t_n + h_n, \mathbf{x}_{n+1}) \quad (5e)$$

$$z_{n+1} = \mathbf{x}_n + h_n ((7/24)k_1 + (1/4)k_2 + (1/3)k_3 + (1/8)k_4) \quad (5f)$$

Usually, to compute the solution of an ordinary differential equation \mathbf{x}_{n+1} over the interval $[t_n, t_n + h_n]$ and from the initial value \mathbf{x}_n , we need to compute a few intermediate approximations in this interval (e.g., the values of k_2 and k_3). The solution \mathbf{x}_{n+1} is then obtained by a weighted mean of the intermediate results, see Equation (5d). Note that this method assumes a Lipschitz continuity condition on the interval $[t_n, t_n + h_n]$. We can know better understand Property 2 stating that no difference operator \bar{d} is updated in the interval $I = [t_n, t_n + h_n]$. This property guarantees that the function f is continuous over the interval I and the solution of the differential equation exists on this interval.

The main feature of variable step-size methods, like ode23, is that the step-size can be adapted during the simulation process (that is why we have h_n in Equations (5)) in order to keep the approximated solution $\tilde{\phi}(t_n) = \mathbf{x}_n$ close to the mathematical solution $\phi(t_n)$, that is $|\phi(t_n) - \tilde{\phi}(t_n)| \leq \delta$ for a given small value δ . To do so, the error is estimated as $\text{err} = |\mathbf{x}_{n+1} - z_{n+1}|$, that is as the distance between two approximation points with two different methods. This estimated error is used to *validate* the integration step from t_n to $t_n + h_n$ and compute the next step-size h_{n+1} . The integration step is validated [17] if:

$$\text{err} \leq \max(\text{atol}, \text{rtol} \times \max(|\mathbf{x}_{n+1}|, |\mathbf{x}_n|)) . \quad (6)$$

The values atol and rtol are the absolute and relative tolerances given by the user to keep a precise approximation. Note that these two values are problem dependent, that is the good choice of atol and rtol depends of the function f . Once the integration step is validated, we can adapt the value of the step-size, a simple method [17, p. 47] is given by:

$$h_{n+1} = h_n \times (\text{rtol}/\text{err})^{1/(q+1)} . \quad (7)$$

The value q is defined as $q = \min(p, \hat{p})$, where p and \hat{p} are the order of the method used to compute \mathbf{x}_{n+1} and z_{n+1} , e.g., $q = 2$ for ode23. The modification of the step-size is useful to reduce the number of steps in the simulation loop when the function f of Equation (4) has a smooth dynamic, but also to increase the precision of $\tilde{\phi}(t)$ when it varies a lot on one interval $[t_n, t_n + h_n]$. Note that an efficient method to adapt the step-size requires many other parameters such as h_{\min} and h_{\max} , the minimal and maximal values of h_n . We refer to [11, p. 167] for a complete description on such methods.

In this article, we only consider Simulink's solvers based on Runge-Kutta methods *i*) fixed step-size: ode1 (Euler's method), ode2 (Heun's method), ode3, ode4 (Runge-Kutta's method), ode5 (Dormand-Prince's method). *ii*) variable step-size: ode23, ode45. All these methods can be described by a Butcher table (see Figure 2(a)). The d_i represent the time instants of the intermediate steps needed to compute the solution of $\dot{\mathbf{x}} = f(t, \mathbf{x})$ over the inter-

d_2	a_{21}				0			
d_3	a_{31}	a_{32}			$\frac{1}{2}$	$\frac{1}{2}$		
\vdots	\vdots		\ddots		$\frac{3}{4}$	0	$\frac{3}{4}$	
d_s	a_{s1}	a_{s2}	\cdots	$a_{s,s-1}$	1	$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$
	w_1	w_2	\cdots	w_s		$\frac{2}{9}$	$\frac{1}{3}$	$\frac{4}{9}$
	w'_1	w'_2	\cdots	w'_s		$\frac{7}{24}$	$\frac{1}{4}$	$\frac{1}{3}$
								$\frac{1}{8}$

(a) General form.

(b) ode23 table.

Figure 2. Butcher table.

val $[t_n, t_n + h_n]$. The matrix made of the elements a_{ij} represents the weights used to approximate the interval solution from the previous intermediate steps. The elements w_i, w'_i (only for variable step-size methods) represent the latest weights to approximate the solution at time $t_n + h_n$ with two methods of different orders. For example, the Butcher table associated to ode23 is given at Figure 2(b). In consequence the elements of the Butcher table give a unified description for all the numerical integration methods members of the Rung-Kutta family.

3.3 Zero-crossing detection

In some cases, the simulation should detect when particular events happen without knowing the exact time of occurrence. Such phenomena are taken into account in *zero-crossings* methods. More precisely, we can represent such events by looking at the sign of some functions z , named *zero-crossing functions*. For example, the safety mechanism in Section 2 is associated to the zero-crossing function $z(t) = x(t) - 7.1$ detecting when the vehicle has reached 7.1 meters of distance. Therefore after each integration step, we have to check if the sign of $z(t)$ changed, this is the *zero-crossing detection*. Once we detect a particular event, we must find the exact time instant \tilde{t} when this event happens, this step is the *zero-crossing localization*. Usually, we use a bisection method to find \tilde{t} up to a given precision. More details are given in Section 5.4.2. In consequence the time is varying during this step, it explains why this process is only used with variable step-size integration methods.

Note that a cascade of zero-crossing events may happen during a numerical simulation, so Simulink adds elements to detect when such case happens, for example when the number of consecutive zero-crossings is too important. For the other issues with zero-crossing detection, we refer to [21].

4. Formalism: Simulink language

In this section, we present the subset of Simulink for which we defined our operational semantics. We also present how Simulink models are translated into an equation language on which our operational semantics is defined.

4.1 Overview of Simulink

Simulink is a graphical language that allows the designer to easily write dynamical systems by replacing the state-space representation (as given by Equations (3)) by a block diagram where all operations are represented by *blocks* connected with *lines*. Each block may have several inputs and outputs, which are named *ports*. The data that is exchanged between the blocks via the lines are named *signals*; a signal is a function $\ell : \mathbb{R}_+ \rightarrow \mathbb{R}$ that associates at each time instant a value. Note that in Simulink, the signals may be function of time into \mathbb{R}^n for some $n \in \mathbb{N}$, for the sake of simplicity we only consider real valued signals here. In Table 1, we present a core subset of Simulink blocks for which we defined our operational semantics. We also give the equations that make explicit how the input signals are linked to the output signals. For example, the addition block has two input ports (named ℓ_1 and ℓ_2 for example), one

output port (ℓ_3) and the equation $\ell_3 = \ell_1 + \ell_2$ means that the output signal on ℓ_3 , say $\ell_3(t)$, is such that: $\forall t, \ell_3(t) = \ell_1(t) + \ell_2(t)$. In the rest of this article, we will not distinguish lines and signals.

Let us remark that for the *Integrator*, the input signal is not directly linked to the output signals. Instead, this block has an internal state named x such that the output of the block is equal to the internal state. However, to compute the value of this state, we must solve a differential equation: if the input of the block is ℓ_1 , then the internal state is given by the equation $\dot{x}(t) = \ell_1(t)$. Equivalently, for *Unit Delay* blocks, there is an internal state named d whose value is given by a first order difference equation: we write $\bar{d} = \ell_1$ for $d(t+s) = \ell_1(t)$, where s is the sampling time of the block, which must be given by the block parameters.

The *Switch* block is also parametrized by a predicate p_r , which can be either $\ell_2 > r, \ell_2 \geq r$ or $\ell_2 \neq 0$, where ℓ_2 is the second input signal of the block and r a given constant value.

In Simulink, blocks may have many parameters that modify their behaviors (for example the predicate for the *Switch* block or the value c for a *Constant* block). All these parameters are made explicit by the equations. Another important parameter that can be set for almost all blocks is the sampling period p . Actually, it can be specified that a block must output a new value only at times $k \times p$, for all $k \in \mathbb{N}$. This option will also be made explicit in the equations: for example, if a *Sum* block has a sampling time s , we will write the equation $\ell_3 = S_s \ell_1 + \ell_2$, with $S_s = \{k \times s : k \in \mathbb{N}\}$. The meaning of this equations in term of signals is then:

$$\forall t \in \mathbb{R}_+, \ell_3(t) = \begin{cases} \ell_1(t) + \ell_2(t) & \text{if } t \in S_s \\ \lim_{\tau \rightarrow t, \tau < t} \ell_3(\tau) & \text{otherwise} \end{cases}$$

So in this case, the signal $\ell_3(t)$ is a step function whose value changes at each time $k \times s, k \in \mathbb{N}$.

Using these equations, we can reconstruct, from a Simulink model, the state-space equations of the dynamical system represented by the model. The simulation engine of Simulink computes a numerical simulation of this state-space representation using the techniques presented in Section 3.

4.2 Simulink equations from a Simulink model

From now on, we thus consider a Simulink program to be a sequence of equations described by the BNF rules given in Equations (8). An expression e is made of constants $r \in \mathbb{R}$, variables ℓ, x and d coming from a finite set \mathcal{V} , arithmetic operations ($\diamond \in \{+, -, \times, \div\}$) as well as Boolean expressions ($\bowtie \in \{<, \leq, >, \geq, =, \neq\}$) and conditional expression *if*. Equations eq are either non-temporal equations $\ell := e$, sampled equations $\ell :=_S e$, differential equations of the form $\dot{x} := e$ or a difference equations denoted by $\bar{d} :=_S e$. We associate to a Simulink model M a system of flow equations Eq_M (we denote this system Eq when M is clear from the context).

$$e ::= r \mid \ell \mid x \mid d \mid e_1 \diamond e_2 \mid e_1 \bowtie e_2 \mid \text{if}(e_1, e_2, e_3) \quad (8a)$$

$$eq ::= \ell :=_S e \mid \ell := e \mid \dot{x} := e \mid \bar{d} :=_S e \quad (8b)$$



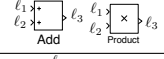



$$p ::= eq \mid eq; p \quad (8c)$$

Three comments: *i)* We do not have functions as Simulink always flatten its models. *ii)* Equations are easily and automatically generated by inspecting the content of the `mdl` file used to save Simulink models. *iii)* Our language is able to model a large subset of Simulink blocks as many of them are syntactic sugar e.g., *StateSpace* block is only a system of linear differential equations.

Remark 1. In Simulink, the *Integrator* block can be associated to many features which may change its behavior as resetting the integrator if a specified event occurs. In this article, we will only consider, for conciseness, the simplest version but our operational semantics is able to deal with all the other features.

Example 1. The Simulink program of the PI controller of Figure 1 is associated with the following equations:

Table 1. Subset of Simulink blocks

Library	Blocks	Representation	Equations	Description
Source	Constant		$\ell_1 = c$	Constant value
Sinks	Output		$\text{out1} = \ell_1$	System output
Arithmetic operations	Add/Mul		$\ell_3 = \ell_1 + \ell_2$ or $\ell_3 = \ell_1 \times \ell_2$	Addition/Multiplication
Signal routing	Switch		$\ell_4 = \text{if}(p_r(\ell_2), \ell_1, \ell_3)$	Conditional statement
Continuous-time	Integrator		$\{\ell_2 = x; \dot{x} = \ell_1; x(0) = \text{init}\}$	Continuous-time integration
Discrete-time	Unit Delay		$\{\ell_2 = d; \bar{d} = \ell_1; d(0) = \text{init}\}$	Discrete-time delay

$$\begin{aligned}
 \dot{x}_1 &:= \ell_{13}; \dot{x}_2 := \ell_{14}; \bar{d}_1 := \ell_7; \\
 \ell_{14} &:= x_1; \ell_1 := x_2; \ell_8 := d_1; \ell_2 := 0; \ell_3 := v_m; \\
 \ell_4 &:= \text{if}(\ell_1 < 7.1; \ell_3; \ell_2); \ell_5 := \ell_4 - \ell_{14}; \ell_6 :=_{S_{0.1}} h \times \ell_5; \\
 \ell_7 &:= \ell_6 + \ell_8; \ell_9 := K_p \times \ell_5; \ell_{10} := K_i \times \ell_7; \\
 \ell_{11} &:= \ell_9 + \ell_{10}; \ell_{12} := 1/m \times \ell_{11}; \ell_{13} := \ell_{12} + \ell_{15}; \\
 \ell_{15} &:= -b/m \times \ell_{14}
 \end{aligned}$$

Within the equations Eq, we distinguish four subsets that will be evaluated one after another during the simulation loop. They are:

- the major step equations $\text{Eq}(M)$ that contain all equations of the kind $\ell :=_S e$ or $\ell := e$ where e is an expression;
- $\text{Eq}(m)$ that contains all equations of the kind $\ell := e$, i.e., blocks without sampling;
- $\text{Eq}(d)$ that contains all equations of the kind $\bar{d} :=_S \ell$;
- $\text{Eq}(x)$ that contains all equations of the kind $\dot{x} := \ell$.

Note that $\text{Eq}(m)$ is included in $\text{Eq}(M)$ which contains, in addition to $\text{Eq}(m)$, all the sampled equations. In $\text{Eq}(M)$ and $\text{Eq}(m)$, the order in which equations appear is important. Actually, an equation links the output of a block with its input. So, if the output of block b_1 is the input of block b_2 , the equation eq_1 corresponding to b_1 must be evaluated before the equation eq_2 of b_2 . So the sequence of equations Eq is built using a dependency relation: blocks with internal state (Integrator and Unit Delay) are independent of the others, and a block b_1 is dependent of b_2 if an output of b_2 is an input of b_1 . Obviously there may be a loop in this relation (i.e., a block being dependent of itself) if there is a loop in the Simulink model without Integrator or Unit Delay blocks. In such cases, named *algebraic loop* in Simulink, the simulation engine must compute a fixpoint to compute the value of all signals within the loop. In this article, we assume that all loops in the Simulink model contain a Integrator or Unit Delay block.

Note also that the order in which equations $\dot{x} := \ell$ and $\bar{d} :=_S \ell$ appear is not important as the computation of the next value of the internal state x or d is done after the evaluation of g in the simulation loop (see Section 3), and this new value will be used at the next iteration of the simulation loop. So all internal states may be computed in parallel, we thus use the vector notations $\bar{d} := \ell$ and $\dot{x} := \ell$ for the equations in $\text{Eq}(d)$ and $\text{Eq}(x)$ respectively.

In the equations Eq, we made visible the sampling rate which is a parameter that can modify the behavior of a block. Another parameter must be taken into account, namely the zero-crossing functions. Actually, some blocks can define a zero-crossing event in order to detect a particular event. For example, the Saturated Integrator block tries to detect when its state enters or leaves an upper or a lower limit. Note that some blocks do not allowed zero-crossing detection while they are based on continuous-time inte-

gration, e.g., Transfer Fcn and StateSpace. Each block allowing zero-crossing defines one (or more) *local zero-crossing function* $z_i : \Sigma \rightarrow \mathbb{R}$, with Σ the set of states (see Section 4.3); for example the Saturated Integrator block b_i (whose state is x_i and with saturation values m and M), is associated with the functions $z_i^m(\sigma) = \sigma(x_i) - m$ and $z_i^M(\sigma) = M - \sigma(x_i)$. If the sign of one of these functions changes between states σ and σ' , a zero-crossing event is detected. As all zero-crossing events must be detected and handled simultaneously, we define the *global zero-crossing function* Eq_z as a function $\text{Eq}_z : \Sigma \rightarrow \mathbb{R}^d$, where d is the number of local zero-crossing functions in the Simulink model. If these functions are named z_1, \dots, z_d , then $\text{Eq}_z(\sigma) = (z_1(\sigma), \dots, z_d(\sigma))$.

Finally, we will denote by $\text{Eq}(\text{sampling})$ the set of all discrete sampling times that appear in the equations:

$$\text{Eq}(\text{sampling}) = \cup \{S \mid \ell :=_S e \in \text{Eq}(M) \vee \bar{d} :=_S e \in \text{Eq}(d)\} .$$

Example 2. If we consider our running example, then the set of equations Eq is made of: $\text{Eq}(M) = \{\ell_{14} := x_1; \ell_1 := x_2; \ell_8 := d_1; \dots\}$, $\text{Eq}(m)$ contains all equations of $\text{Eq}(M)$ except $\ell_6 :=_{S_{0.1}} h \times \ell_5$. Also, $\text{Eq}(d) = \{\bar{d}_1 = \ell_7\}$ and $\text{Eq}(x) = \{\dot{x}_1 = \ell_{13}; \dot{x}_2 = \ell_{14}\}$. The zero-crossing equation is $\text{Eq}_z(\sigma) = \ell_1 - 7.1$. $\text{Eq}(\text{sampling}) = \{0.1k \mid k \in \mathbb{N}\}$.

Our semantics is a transition system defined by a Simulink model and that depends on the solver parameters: we write the transitions as: $\text{Eq}, \pi \vdash \sigma \rightarrow \sigma'$, meaning that the state σ is modified by the equations Eq into a new state σ' . This modification depends on the solver parameters π , which are defined by the user before the simulation starts. We now define our notion of states and parameters, the derivation rules for the transitions are in Section 5.

4.3 Variables and states

The equations representing a Simulink program involve different kinds of variables: outputs of blocks ℓ_i , states of continuous blocks x_k and states of unit-delay blocks d_j . Two additional important variables will be necessary for the execution of a Simulink model: the time t and the step-size h . We denote by \mathcal{V} the set of variables of a Simulink model. For a model with n_b blocks, n_x continuous blocks (integrator for example) and n_d discrete blocks, we have: $\mathcal{V} = \{\ell_i, i \in [1, n_b]\} \cup \{x_i, i \in [1, n_x]\} \cup \{d_i, i \in [1, n_d]\} \cup \{t, h\}$.

Definition 1. The state σ of a model associates to each variable $v \in \mathcal{V}$ its value in \mathbb{R} . It is thus a mapping $\sigma : \mathcal{V} \rightarrow \mathbb{R}$ and Σ is the set of all states. For a variable $v \in \mathcal{V}$, let $\sigma(v)$ be the value of v in the state σ and let $\sigma[v \mapsto r]$ be the state σ' that is identical to σ except for the value of v :

$$\forall v' \in \mathcal{V}, \sigma'(v') = \begin{cases} \sigma(v') & \text{if } v' \neq v \\ r & \text{otherwise} \end{cases} .$$

As the internal state of `Integrator` and `Unit Delay` blocks are modified in parallel, we denote by $\mathbf{x} \in \mathcal{V}^{n_x}$ the vector containing the states of all continuous blocks of a Simulink model. Equivalently, $\mathbf{d} \in \mathcal{V}^{n_d}$ is the vector containing the states of all unit-delay blocks. In this article, a bold symbol \mathbf{x} represents a vector of values or variables, and x_i is the i^{th} component of \mathbf{x} . The arithmetic symbols $\diamond \in \{+, -, \times, \dots\}$ are used on vectors with:

$$\mathbf{z} = \mathbf{x} \diamond \mathbf{y} \Leftrightarrow \forall i, z_i = x_i \diamond y_i .$$

In order to modify or access to vectors in a state, we shall use the following shortcut notations:

$$\begin{aligned} \sigma(\mathbf{x}) &= (\sigma(x_1), \dots, \sigma(x_{n_x})), \quad \sigma(\mathbf{d}) = (\sigma(d_1), \dots, \sigma(d_{n_d})) \\ \sigma[\mathbf{x} \mapsto \mathbf{r}] &= \sigma[x_1 \mapsto r_1][x_2 \mapsto r_2] \dots [x_{n_x} \mapsto r_{n_x}] \\ \sigma[\mathbf{d} \mapsto \mathbf{r}] &= \sigma[d_1 \mapsto r_1][d_2 \mapsto r_2] \dots [d_{n_d} \mapsto r_{n_d}] \end{aligned}$$

4.4 Parameters

Before running the simulation, the user chooses various parameters, known as *configuration parameters* in Simulink, that control the precision of the solver and of the zero-crossing algorithm. In our formalism, all these parameters are given as a record π such that:

- $\pi(\epsilon_r)$ (resp. $\pi(\epsilon_a)$) is the relative (resp. absolute) tolerance. Note that $\pi(\epsilon_r) \in \mathbb{R}$ is a real number while $\pi(\epsilon_a)$ is a vector giving the absolute tolerance for each integrator block.
- $\pi(t_0)$ and $\pi(t_{\text{end}})$ are the start and end time.
- $\pi(h_0)$ (resp. $\pi(h_{\text{min}})$ and $\pi(h_{\text{max}})$) is the initial (resp. minimal and maximal) step-size.
- $\pi(\epsilon_z)$ is the tolerance used in the zero-crossing detection.

The record π also includes the Butcher table of the chosen solver (see Section 3.2): the i^{th} line of the table is recorded as $\pi(d_i) \in \mathbb{R}$ and $\pi(\mathbf{a}_i) \in \mathbb{R}^i$. The last line is given by $\pi(\mathbf{w}) \in \mathbb{R}^n$ and if the solver is a variable step-size method, it has one more line recorded as $\pi(\mathbf{w}') \in \mathbb{R}^n$.

Example 3. We consider again the Simulink model given at Figure 1 for which the parameters are (these are the default values given by Simulink for each parameter): $\pi(t_0) = 0$, $\pi(t_{\text{end}}) = 10$, $\pi(\epsilon_r) = 10^{-3}$ and $\pi(\epsilon_a)$ is the vector with all values equal to 10^{-6} ; $\pi(h_{\text{min}}) = t \cdot 16 \cdot 2^{-53}$ and $\pi(h_{\text{max}}) = (t_{\text{end}} - t_0)/50$; $\pi(\epsilon_z) = 10 \times 128 \times 2^{-53}$. We consider `ode23` solver presented in Section 3.2 so $\pi(d_1) = 1/2$, $\pi(\mathbf{a}_1) = 1/2$, etc.

5. Operational semantics

In this section, we define the operational semantics associated to the subset of Simulink language defined in Section 4. This semantics is a transition system that closely follows the Simulink simulation engine described in Section 3: each stage of the simulation loop is a transition and the main transition system is the composition of all these transitions. In order to keep notations manageable and our semantics understandable, we present it in a “top-down” way: we start with the high level transitions and decompose them into specialized transitions. This is done in Sections 5.1 to 5.4.

5.1 Run of a Simulink program

A state σ_0 is the initial state of a program `Eq` in a configuration π if $\sigma_0(\mathbf{x})$ and $\sigma_0(\mathbf{d})$ are the initial values as defined by the equations of `Eq` and $\sigma_0(t) = \pi(t_0)$ and $\sigma_0(h) = \pi(h_0)$. A state is final in a configuration π if $\sigma(t) = \pi(t_{\text{end}})$. A run of the system of equations `Eq` in a configuration π is a finite sequence of states $\sigma_0, \sigma_1, \dots, \sigma_n \in \Sigma$ such that σ_0 is an initial state, σ_n is a final state and $\forall i \in [0, n-1]$, $\text{Eq}, \pi \vdash \sigma_i \Rightarrow \sigma_{i+1}$, where the \Rightarrow transition is given in Figure 3. The transition \Rightarrow depends on various other transitions: the “major step transition” \xrightarrow{M} that evaluates the

$$\frac{\frac{\sigma(t) = \pi(t_{\text{end}})}{\text{Eq}, \pi \vdash \sigma \Rightarrow \sigma} \text{ SIMULATION-END}}{\frac{\sigma(t) < \pi(t_{\text{end}})}{\text{Eq}, \pi \vdash \sigma \xrightarrow{M} \sigma_1} \quad \frac{\sigma(t) < \pi(t_{\text{end}})}{\text{Eq}, \pi \vdash \sigma_1 \xrightarrow{u} \sigma_2} \quad \frac{\sigma(t) < \pi(t_{\text{end}})}{\text{Eq}, \pi \vdash \sigma_2 \xrightarrow{s} \sigma'}}{\text{Eq}, \pi \vdash \sigma \Rightarrow \sigma'} \text{ SOLVER}}$$

Figure 3. Operational semantics rules: global rules.

equations `Eq(M)`, the “update transition” \xrightarrow{u} that evaluates `Eq(d)` and the “solver transition” \xrightarrow{s} that evaluates `Eq(x)`.

5.2 Major and minor steps transition

We now define the \xrightarrow{o} transition that defines how a state $\sigma \in \Sigma$ is modified by a sequence of equations. In Figure 4 we give the rules that define this transition. We recall that $\bowtie \in \{<, \leq, >, \geq, =, \neq\}$, $\diamond \in \{+, -, \times, \div\}$. We see that it is very similar to the transitions of the operational semantics of an imperative programming language, as defined in [15]. The main difference comes from the affectation $\ell :=_S e$ whose effect depends on the simulation time: if the time t is one of the sampling times of the corresponding block (i.e., if $t \in S$), then the instruction is executed (rule `AFFS`), otherwise it has no effect on σ (rule `AFFNS`).

Based on the \xrightarrow{o} transition, we define the major steps output \xrightarrow{M} which evaluates all equations, and minor steps output \xrightarrow{m} which only evaluates the equations corresponding to continuous blocks. The transition \xrightarrow{M} (resp. \xrightarrow{m}) is just the application of \xrightarrow{o} on the set of major (resp. minor) step equations `Eq(M)` (resp. `Eq(m)`).

5.3 Update transitions

Once the output of all blocks at time t is computed, the internal state of unit-delay and continuous blocks can be computed. These internal states are the input of the other blocks at the next simulation step, i.e., at time $t+h$. The unit delay block is a memory block that outputs at time $t+h$ its input at time t . Thus, the update transition simply stores in the internal state the input value: this is stated by the rules `UPDATENS` and `UPDATES` of Figure 4. It is clear that the order in which these updates are made does not affect the execution, thus we can consider that they are all performed simultaneously. We use the vectorial notation $\sigma(\ell)$ and $\sigma(\mathbf{d})$ to model this. Let us remark that for this transition also, we must consider two cases: when the time belongs to the sampling time of the unit-delay (rule `UPDATES`) and when it is not the case (rule `UPDATENS`).

5.4 Solver transitions

Next, the internal state of all continuous blocks (integrator for example) must be computed: this means that the numerical solver computes a value $\mathbf{x}(t+h)$ for all the continuous states \mathbf{x} . Note that the step-size may be reduced by the solver in order to keep the integration error within the bounds given by π . Also, zero-crossing events will be detected and, if necessary, tightly approximated using a specific algorithm. Finally, once the step is validated, the next step-size is computed and the simulation time is increased. We decompose these three stages into: the integration transition \xrightarrow{i} , the zero-crossing transition $\xrightarrow{z_c}$ and the step-size transition \xrightarrow{h} that we detail in the next sections. The rule for the solver transition is:

$$\frac{\frac{\text{Eq}, \pi, \sigma \vdash \sigma \xrightarrow{i} \sigma_{so}}{\text{Eq}, \pi, \sigma \vdash \sigma_{so} \xrightarrow{z_c} \sigma_{zc}} \quad \frac{\text{Eq}, \pi \vdash \sigma_{zc} \xrightarrow{h} \sigma'}{\text{Eq}, \pi \vdash \sigma \xrightarrow{s} \sigma'}}{\text{Eq}, \pi \vdash \sigma \Rightarrow \sigma'} \text{ SOLVER}$$

5.4.1 Integrator transitions

We now describe the transitions for the integrator stage, i.e., how the continuous state $\mathbf{x}(t+h)$ is computed. Let us first remark that,

$$\begin{array}{c}
\frac{}{\langle r, \sigma \rangle \xrightarrow{o} r} \text{CTE} \quad \frac{}{\langle \ell, \sigma \rangle \xrightarrow{o} \sigma(\ell)} \text{VAR} \quad \frac{\langle e_1, \sigma \rangle \xrightarrow{o} r_1 \quad \langle e_2, \sigma \rangle \xrightarrow{o} r_2}{\langle e_1 \diamond e_2, \sigma \rangle \xrightarrow{o} r_1 \diamond r_2} \text{ARITH} \quad \frac{\langle e_1, \sigma \rangle \xrightarrow{o} r_1 \quad \langle e_2, \sigma \rangle \xrightarrow{o} r_2}{\langle e_1 \bowtie e_2, \sigma \rangle \xrightarrow{o} r_1 \bowtie r_2} \text{CMP} \\
\frac{\langle e_1, \sigma \rangle \xrightarrow{o} \text{true} \quad \langle e_2, \sigma \rangle \xrightarrow{o} r_2}{\langle \text{if } (e_1, e_2, e_3), \sigma \rangle \xrightarrow{o} r_2} \text{THEN} \quad \frac{\langle e_1, \sigma \rangle \xrightarrow{o} \text{false} \quad \langle e_3, \sigma \rangle \xrightarrow{o} r_3}{\langle \text{if } (e_1, e_2, e_3), \sigma \rangle \xrightarrow{o} r_3} \text{ELSE} \quad \frac{\langle eq_1, \sigma \rangle \xrightarrow{o} \sigma' \quad \langle eq_2, \sigma' \rangle \xrightarrow{o} \sigma''}{\langle eq_1; eq_2, \sigma \rangle \xrightarrow{o} \sigma''} \text{SEQ} \\
\frac{\sigma(t) \notin S}{\langle \ell :=_S e, \sigma \rangle \xrightarrow{o} \sigma} \text{AFFNS} \quad \frac{\sigma(t) \in S \quad \langle e, \sigma \rangle \xrightarrow{o} r}{\langle \ell :=_S e, \sigma \rangle \xrightarrow{o} \sigma[\ell \leftarrow r]} \text{AFFS} \quad \frac{\langle e, \sigma \rangle \xrightarrow{o} r}{\langle \ell := e, \sigma \rangle \xrightarrow{o} \sigma[\ell \leftarrow r]} \text{AFF} \\
\frac{\langle \text{Eq}(m), \sigma \rangle \xrightarrow{o} \sigma'}{\text{Eq}, \pi \vdash \sigma \xrightarrow{m} \sigma'} \text{MINOR} \quad \frac{\langle \text{Eq}(M), \sigma \rangle \xrightarrow{o} \sigma'}{\text{Eq}, \pi \vdash \sigma \xrightarrow{M} \sigma'} \text{MAJOR} \quad \frac{\text{Eq}(\mathbf{d}) = \bar{\mathbf{d}} :=_S \ell \quad \sigma(t) \notin S}{\text{Eq}, \pi \vdash \sigma \xrightarrow{u} \sigma} \text{UPDATE\text{NS}} \quad \frac{\text{Eq}(\mathbf{d}) = \bar{\mathbf{d}} :=_S \ell \quad \sigma(t) \in S}{\text{Eq}, \pi \vdash \sigma \xrightarrow{u} \sigma[\mathbf{d} \mapsto \sigma(\ell)]} \text{UPDATE\text{S}}
\end{array}$$

Figure 4. The output and update transitions.

$$\begin{array}{c}
\frac{\langle \text{sc}_1(\sigma_M(\mathbf{x}), \text{Eq}(\mathbf{x}), \pi), \sigma \rangle \xrightarrow{o} \sigma_1 \quad \langle \text{sc}_2(\sigma_M(\mathbf{x}), \text{Eq}(\mathbf{x}), \pi), \sigma_1 \rangle \xrightarrow{o} \sigma_2 \quad \text{check_err}(\sigma, \sigma_1, \sigma_2, \pi) = 1}{\text{Eq}, \pi, \sigma_M \vdash \sigma \xrightarrow{i} \sigma_M[\mathbf{x} \mapsto \sigma_1(\mathbf{x}), h \mapsto \sigma_1(h)]} \text{INTEGRATION-SUCCESS} \\
\frac{\langle \text{sc}_1(\sigma_M(\mathbf{x}), \text{Eq}(\mathbf{x}), \pi), \sigma \rangle \xrightarrow{o} \sigma_1 \quad \langle \text{sc}_2(\sigma_M(\mathbf{x}), \text{Eq}(\mathbf{x}), \pi), \sigma_1 \rangle \xrightarrow{o} \sigma_2 \quad \text{check_err}(\sigma, \sigma_1, \sigma_2, \pi) = 0 \quad \text{Eq}, \pi, \sigma_M \vdash \sigma_M[h \mapsto \max(\pi(h_{\min}), \frac{\sigma(h)}{2})]}{\text{Eq}, \pi, \sigma_M \vdash \sigma \xrightarrow{i} \sigma'} \text{INTEGRATION-FAIL}
\end{array}$$

Figure 5. The rules for the integrator transition.

as for the update transition, the order in which the internal state of each continuous block is computed is not relevant, all states can be computed simultaneously. We will thus use again the vector notation for this. The integration rules are given in Figure 5, we explain them in the sequel. Let us first recall that the equations for the integrator blocks are of the form $\dot{\mathbf{x}} := \ell$ (in the vector notation) and that $\text{Eq}(\mathbf{x})$ returns these equations.

To compute $\mathbf{x}(t+h)$, a numerical solver uses a composition of two kinds of computations. First, the derivatives at $t_i = t + d_i \cdot h$ are stored. In Simulink, this means evaluating the output of all blocks and storing the input of the integrator block. Here, d_i is the first coefficient of the i^{th} row of the Butcher table (see Section 3.2). Then, $x(t_i)$ is estimated using a linear extrapolation with the parameters of the Butcher table.

Once all extrapolations are done, the value at $t+h$ is defined as a linear extrapolation from $\mathbf{x}(t)$, using a linear combination of the derivatives at all t_i . The coefficients of this extrapolation are in the last row of the Butcher table. In the classical variable step-size scheme, the derivatives at points t_i are stored in variables named \mathbf{k}_i (\mathbf{k}_i is a vector of the size of $\mathbf{x}(t)$). We use the same notations and suppose that the concatenation of the vectors \mathbf{k}_i forms a matrix \mathbf{K} . We denote the matrix-vector multiplication by \times : if \mathbf{K} is a $n \times m$ matrix and \mathbf{v} a $m \times 1$ vector, then $\mathbf{K} \times \mathbf{v}$ is a $n \times 1$ vector.

In our semantics, evaluating the output of all blocks means applying the \xrightarrow{o} transition: the integrator transition is thus a sequence of numerical computations (for the extrapolation) and call to \xrightarrow{o} on the minor time step equations (for the derivatives). We will extend the state σ so that they also associate to each variable \mathbf{k}_i a value. We expand the equation $\dot{\mathbf{x}} := \ell$ in a solver-dependent code that will be evaluated by the \xrightarrow{o} transition. This code is of the kind:

$$\begin{array}{l}
\mathbf{k}_1 := \ell; \\
\mathbf{x} := \text{expl}(\mathbf{x}_0, \mathbf{k}_1, \pi(a_1), h); t := t + \pi(d_1)h; \text{Eq}(m); \mathbf{k}_2 := \ell; \\
\mathbf{x} := \text{expl}(\mathbf{x}_0, [\mathbf{k}_1 \mathbf{k}_2], \pi(a_2), h); t := t + \pi(d_2)h; \text{Eq}(m); \mathbf{k}_3 := \ell; \\
\vdots \\
\mathbf{x} := \text{expl}(\mathbf{x}_0, [\mathbf{k}_1 \dots \mathbf{k}_s], \pi(w), h)
\end{array}$$

We recall that $\text{Eq}(m)$ is the sequence of equations that must be executed during the minor time steps. Note that the first instruction is always $\mathbf{k}_1 := \ell$ as the first coefficient k_1 is always equal to

the derivative estimated at \mathbf{x} that was already computed by the \xrightarrow{m} transition. The expression $\text{expl}(\mathbf{x}_0, \mathbf{K}, \mathbf{a}, h)$, where \mathbf{x}_0 and \mathbf{a} are vectors and \mathbf{K} is a matrix, will be evaluated by the \xrightarrow{o} transition as:

$$\frac{\mathbf{r} = \mathbf{x}_0 + h \cdot (\mathbf{K} \times \mathbf{a})}{\langle \mathbf{x} := \text{expl}(\mathbf{x}_0, \mathbf{K}, \mathbf{a}, h), \sigma \rangle \xrightarrow{o} \sigma[\mathbf{x} \mapsto \mathbf{r}]}$$

For variable step-size methods, this code will be completed by another one that computes the second approximation used for error estimation. This second code is of the same kind as above. For a configuration π (i.e., a choice for the solver), we will denote $\text{sc}_1(\mathbf{x}_0, \dot{\mathbf{x}} := \ell, \pi)$ the code (sc stands for solver code) to compute the first approximation point (denoted \mathbf{x}_{n+1} in Section 3.2) and $\text{sc}_2(\mathbf{x}_0, \dot{\mathbf{x}} := \ell, \pi)$ the code to compute the second approximation point (denoted \mathbf{z}_{n+1} in Section 3.2). For fixed step solvers, this second code will be empty.

Example 4. Using *ode23* solver with our running example, $\text{sc}_1(\mathbf{x}_0, \dot{\mathbf{x}} := \ell, \pi)$ is interpreted as this sequence of equations (in vectorial notation, with t the transpose):

$$\begin{array}{l}
\mathbf{k}_1 := (\ell_{14} \quad \ell_{13}); \mathbf{x} := \mathbf{x}_0 + 0.5h (\ell_{14} \quad \ell_{13}); t := t + 0.5h; \\
\text{Eq}(m); \mathbf{k}_2 := (\ell_{14} \quad \ell_{13}); \mathbf{x} := \mathbf{x}_0 + 0.75h (\ell_{14} \quad \ell_{13}); \\
t := t + 0.75h; \text{Eq}(m); \mathbf{k}_3 := (\ell_{14} \quad \ell_{13}); \\
\mathbf{x} := \mathbf{x}_0 + h (k_1 \quad k_2 \quad k_3) \cdot (2/9 \quad 1/3 \quad 4/9)^t.
\end{array}$$

The computation of $\text{sc}_2(\mathbf{x}_0, \dot{\mathbf{x}} := \ell, \pi)$ is very similar.

We may now explain the two rules for the integrator transition. Both rules start by evaluating the code for the two stages of the solver (i.e., they compute \mathbf{x}_{n+1} and \mathbf{z}_{n+1}), thus getting two states σ_1 and σ_2 that contain two approximations for $\mathbf{x}(t+h)$. As explained in Section 3.2, the simulation step will be validated only if the distance between these two approximations is smaller than a (user-defined) bound. The function `check_err` returns true if it is the case, false otherwise. Following Equation (6), it is defined by:

$$\text{check_err}(\sigma, \sigma_1, \sigma_2, \pi) = \frac{\|\sigma_1(\mathbf{x}) - \sigma_2(\mathbf{x})\|_\infty}{\max(\max(|\sigma_1(\mathbf{x})|, |\sigma_2(\mathbf{x})|), \frac{\text{rtol}}{\text{rtol}})} \leq \text{rtol}.$$

If the `check_err` returns 1, then the step is validated and the new state is $\sigma_1(\mathbf{x})$ (rule INTEGRATOR-SUCCESS). Otherwise, the step is rejected and the integration starts over with a step-size reduced to $h/2$ (rule INTEGRATOR-FAIL).

Remark that, contrary to the semantics defined in [14], our semantics mask the backtracking of time inside the solver method call. This is indeed closer to what Simulink does as the time of the major steps are always in increasing order.

5.4.2 Zero-crossing transitions

To handle zero-crossing events, Simulink performs three tasks: zero-crossing detection, zero-crossing bracketing and finally passing through the zero-crossing. A good survey of how zero-crossing is detected and various algorithms to handle it can be read in [21].

The detection phase relies on the Eq_z function that we defined in Section 4.2. Assume that the state at time t (called major time state) was σ_M , and that the state after the integration transition is σ_{so} . Then, there is a zero-crossing event between σ_M and σ_{so} if $\text{Eq}_z(\sigma_M) \cdot \text{Eq}_z(\sigma_{so}) \leq 0$: this means (for example) that the output of the **Saturated Integrator** block at σ_M was between m and M and that it is above M at σ_{so} . We thus define the zero-crossing detection function $\text{dzc} : \Sigma \times \Sigma \times \text{Eq} \rightarrow \mathbb{B}$ by:

$$\begin{aligned} \text{dzc}(\sigma_1, \sigma_2, \text{Eq}) &= (\text{Eq}_z(\sigma_1) \cdot \text{Eq}_z(\sigma_2) \leq \mathbf{0}) \\ &\wedge (\text{Eq}_z(\sigma_1) \neq \mathbf{0} \vee \text{Eq}_z(\sigma_2) \neq \mathbf{0}) . \end{aligned}$$

The first condition means that the sign of Eq_z must change between σ_1 and σ_2 , the second states that at least one of these values must be different from 0. Remark that in this formula, $\mathbf{0}$ is the vector containing only zeros; the tests \leq and \neq are done component-wise. So, a zero-crossing event is detected on the whole program if a zero-crossing event is detected in one of its blocks.

To detect a zero-crossing event, we compute the output of each block using the \xrightarrow{m} transition on σ_{so} and call the dzc function. The rule ZC-F in Figure 6 deals with the case when no zero-crossing event is detected: the numerical integration is validated and so the simulation time is increased. The rule ZC-T deals with the case when a zero-crossing event is detected: the zero-crossing event is located (using the \xrightarrow{loc} transition), which yields two states, σ_L and σ_R , such that there is no zero-crossing between σ_M and σ_L and one zero-crossing between σ_L and σ_R . We explain the localization algorithm later. Then, the solver increases the simulation time up to $\sigma_R(t)$, i.e., just after the zero-crossing event. To do so, it first advances up to $\sigma_L(t)$ using a major output (\xrightarrow{m} and \xrightarrow{u} transitions in the ZC-T rule), then it sets \mathbf{x} to $\sigma_R(\mathbf{x})$ and t to $\sigma_R(t)$. The simulation is then ready to start a new step.

The localization algorithm is what is called the *search loop*: the solver brackets the zero-crossing event by states σ_L and σ_R and reduce this interval until the time precision $|\sigma_L(t) - \sigma_R(t)|$ is smaller than the parameter $\pi(\epsilon_z)$. This is represented by the rules LOC-F and LOC-T. To reduce the bracketing, the \xrightarrow{lr} transition works as follows (see rules LR1 and LR2 in Figure 6). The time \tilde{t} at which the zero-crossing occurs is estimated using interpolation (function computeTz , explained later) and the state of the system at \tilde{t} is estimated using an approximation method (interpolateX , explained later); this yields a new state σ , and, depending on the result of $\text{dzc}(\sigma_L, \sigma, \text{Eq})$, we set σ_L or σ_R to σ , and repeat.

To estimate the time \tilde{t} at which the zero-crossing event occurs, the solver does a linear interpolation between $\text{Eq}_z(\sigma_L)$ and $\text{Eq}_z(\sigma_R)$ and solves when this interpolation crosses zero. Thus, the function computeTz is defined in Equation (9) with $\mathbf{x}_L = \text{Eq}_z(\sigma_L(\mathbf{x}))$ and $\mathbf{x}_R = \text{Eq}_z(\sigma_R(\mathbf{x}))$.

$$\text{computeTz}(\sigma_L, \sigma_R) = \min_L \left(\sigma_L(t) - \mathbf{x}_L \cdot \frac{\sigma_R(t) - \sigma_L(t)}{\mathbf{x}_R - \mathbf{x}_L} \right) \quad (9)$$

The values \mathbf{x}_L and \mathbf{x}_R are vectors and the numerical operations (including the division) are performed component-wise, so the formula $(\sigma_L(t) - \mathbf{x}_L \cdot \frac{\sigma_R(t) - \sigma_L(t)}{\mathbf{x}_R - \mathbf{x}_L})$ computes a vector \mathbf{t} of times at which the linear interpolation crosses 0. The result of computeTz is the smallest component of \mathbf{t} that is greater than $\sigma_L(t)$: this is the meaning of the \min_L function, i.e., $\min_L(\mathbf{t}) =$

$\min\{t_i \mid t_i \geq \sigma_L(t)\}$, where $\mathbf{t} = (t_1, \dots, t_m)$. Remark that, as their is a zero-crossing event between σ_L and σ_R , we have: $\text{computeTz}(\sigma_L, \sigma_R) \in [\sigma_L(t), \sigma_R(t)]$.

To approximate the state \mathbf{x} at t , the solver uses a method called continuous extension [17, p. 54]. This method is specific to the numerical integration method and yields a very good polynomial approximation of the solution of the differential equation on the interval $[t, t+h]$. Moreover, this approximation is very fast to compute, thus well suited for the zero-crossing process. Such continuous approximation exists for most numerical methods used in Simulink [17]. For example, for `ode23`, the interpolation method is the cubic Hermite interpolation: see Equation (10) with $t_1 = \sigma_1(t)$, $t_2 = \sigma_2(t)$, $h = t_2 - t_1$ and $\tau = (t - t_1)/h$. Moreover, the term p_1 (resp. p_2) is the derivative of the state at time t_1 (resp. t_2). These values are easily computed in Simulink: it is the input of the integrator block in states σ_1 and σ_2 , respectively.

$$\begin{aligned} \text{interpolX}(\sigma_1, \sigma_2, t) &= (2\tau^3 - 3\tau^2 + 1)\sigma_1(\mathbf{x}) + (\tau^3 - 2\tau^2 + \tau)hp_1 \\ &\quad + (-2\tau^3 + 3\tau^2)\sigma_2(\mathbf{x}) + (\tau^3 - \tau^2)hp_2 \quad (10) \end{aligned}$$

In Equation (10), σ_1 is the state of the system at time t and σ_2 is the state at time $t+h$: we will thus use for σ_1 the state before the \xrightarrow{i} transition (σ_M in the rules of Figure 6) and for σ_2 the state after (σ_{so} in Figure 6). Note that we always use the same function (i.e., interpolX uses the same states for σ_1 and σ_2) for the whole search loop, as it is the best approximation we can have on $[t, t+h]$.

Example 5. We illustrate the impact of zero-crossing process on the simulation by considering once again our running example. Recall that the safety mechanism, described in Section 2, has to stop the vehicle as soon as it reaches 7.1 meters of distance. The zero-crossing function $z(t) = x(t) - 7.1$ where $x(t)$ is the position of the vehicle is associated to this mechanism. The interesting step is between time 1.5 and 1.6. The numerical integration starting at time 1.5 with $x = 6.769693$ and provides as a result $x = 7.507278$ at time 1.6. However, a zero-crossing is detected and the zero-crossing location process estimates the time $\tilde{t} = 1.54433$. As a result, a new major step is computed at time 1.54433 and the simulation restarts from this point

5.4.3 Step-size update transitions

Once the zero-crossing event has been handled, the simulation time at the next simulation step is known. If no zero-crossing event was detected, it is $\sigma_M(t) + \sigma_{so}(h)$ (i.e., the time computed by the solver); if there was a zero-crossing event, it is $\sigma_R(t)$. Now, the step-size of the next simulation step must be computed. There are two reasons for changing the step-size. First, the solver may decrease or increase the step-size in order to control the error or improve the performance. Second, the simulation time must be compatible with the sampling times chosen for all blocks in the system: if a block has a sampling rate of s seconds, the simulation must make a major step at each time instant $t_k = k \times s$ for $k \in \mathbb{N}$.

So, the solver first changes the step-size according to a formula like Equation (7) (see Section 3.2). Let us denote $\text{changeH}(h, \pi)$ the function that computes this new step-size h' . Then, the simulation engine checks if there is a sampling time between t_M and $t_M + h'$. If so, it reduces h' so that $t_M + h'$ is the first sampling time. This is done by the rule of Equation (11).

$$\frac{h' = \text{changeH}(\sigma(h), \pi)}{t' = \min\{\tau \in \text{Eq}(\text{sampling}) : \tau > \sigma(t)\}} \text{UPDATE-H} \quad (11)$$

$$\text{Eq}, \pi, \sigma_M \vdash \sigma \xrightarrow{h} \sigma[h \mapsto \min(h', t' - \sigma(t))]$$

Recall that $\text{Eq}(\text{sampling})$ is the set of all sampling times for the considered Simulink model, as defined in Section 4.4.

$$\begin{array}{c}
\frac{\text{Eq}, \pi \vdash \sigma_{so}[t \mapsto \sigma_M(t) + \sigma_{so}(h)] \xrightarrow{m} \sigma' \quad \text{dzc}(\sigma_M, \sigma', \text{Eq}) = \text{false}}{\text{Eq}, \pi, \sigma_M \vdash \sigma_{so} \xrightarrow{zc} \sigma_{so}[t \mapsto \sigma_M(t) + \sigma_{so}(h)]} \text{ZC-F} \\
\frac{\text{Eq}, \pi \vdash \sigma_{so}[t \mapsto \sigma_M(t) + \sigma_{so}(h)] \xrightarrow{m} \sigma' \quad \text{dzc}(\sigma_M, \sigma', \text{Eq}) = \text{true} \quad \text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_M, \sigma' \xrightarrow{loc} \sigma_L, \sigma_R \quad \text{Eq}, \pi \vdash \sigma_L \xrightarrow{m} \sigma_1 \quad \text{Eq}, \pi \vdash \sigma_1 \xrightarrow{u} \sigma'}{\text{Eq}, \pi, \sigma_M \vdash \sigma_{so} \xrightarrow{zc} \sigma'[\mathbf{x} \mapsto \sigma_R(\mathbf{x}), t \mapsto \sigma_R(t)]} \text{ZC-T} \\
\frac{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{lr} \sigma'_L, \sigma'_R \quad |\sigma'_L(t) - \sigma'_R(t)| \leq \pi(\epsilon_z)}{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{loc} \sigma'_L, \sigma'_R} \text{LOC-F} \\
\frac{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{lr} \sigma'_L, \sigma'_R \quad |\sigma'_L(t) - \sigma'_R(t)| > \pi(\epsilon_z) \quad \text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma'_L, \sigma'_R \xrightarrow{loc} \sigma''_L, \sigma''_R}{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{loc} \sigma''_L, \sigma''_R} \text{LOC-T} \\
\frac{\tilde{t} = \text{computeTz}(\sigma_L, \sigma_R) \quad \mathbf{r} = \text{interpolX}(\sigma_M, \sigma_{so}, \tilde{t}) \quad \text{Eq}, \pi \vdash \sigma_L[\mathbf{x} \mapsto \mathbf{r}, t \mapsto \tilde{t}] \xrightarrow{m} \sigma}{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{a} \sigma} \text{ZC-APPROX} \\
\frac{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{a} \sigma \quad \text{dzc}(\sigma_L, \sigma, \text{Eq}) = \text{true}}{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{lr} \sigma_L, \sigma} \text{LR1} \quad \frac{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{a} \sigma \quad \text{dzc}(\sigma_L, \sigma, \text{Eq}) = \text{false}}{\text{Eq}, \pi, \sigma_M, \sigma_{so} \vdash \sigma_L, \sigma_R \xrightarrow{lr} \sigma, \sigma_R} \text{LR2}
\end{array}$$

Figure 6. Operational semantics rules: zero-crossing detection and localization.

Example 6. A simpler version of `changeH` for `ode23` is:

$$h' = \begin{cases} h/temp & \text{if } err \leq rtol \text{ and } temp > 0.2 \\ 5.0h & \text{if } err \leq rtol \text{ and } temp \leq 0.2 \\ \max(h_{\min}, 0.5h) & \text{otherwise (decreasing step-size)} \end{cases}$$

with $err = h(\|\mathbf{x}_{n+1} - \mathbf{x}_n\|_\infty / \max(\max(|\mathbf{x}_{n+1}|, |\mathbf{x}_n|), atol/rtol))$, the inner max being the maximum value of all elements of both vectors, and $temp = 1.25(err/rtol)^{1/3}$. Note that to update h we have to keep in memory the integration error err , see Section 5.4.1.

5.5 Remarks about the semantics

All the rules we have presented so far define our operational semantics for Simulink. It handles variable step-size solvers, zero-crossing detection and models that combine discrete and continuous blocks. As far as we know, this semantics is the first to gather and formalize all the information on the Simulink solver that is found in the documentation of Simulink, in the text books on numerical integration [17] and some pieces of source code available in the Simulink distribution. However, to keep the presentation readable, we have voluntarily omitted some details that we now present.

First, fixed step-size solvers do not allow zero-crossing events, which is not the case in the \xrightarrow{zc} transition. To handle this, we only need to make the `dzc` function solver dependent: it returns 0 for fixed step-size solvers. Also, during the computation of the zero-crossing time (function `computeTz`, Equation (9)), two special cases must be handled. First, if $0 \in \text{Eq}_z(\sigma_L)$, we always have `computeTz`(σ_L, σ_R) = $\sigma_L(t)$. Simulink has a solution for this case: it sets $\sigma_R(t)$ to $\sigma_L(t) + \epsilon$, where ϵ is the machine precision, and does a first-order interpolation to have the state $\sigma_R(\mathbf{x})$. Second, if their is a component of σ_L and σ_R such that $\mathbf{x}_L(i) = \mathbf{x}_R(i)$, we have a division by zero in Equation (9). In this case, we can easily check that no zero-crossing event exists for this component, so we can omit it in the computation of `computeTz`.

Finally, when the step-size is modified, we must check that h remains between $\pi(h_{\max})$ (the maximal step-size) and $\pi(h_{\min})$ (the minimal step-size). So the rule of Equation (11) should be modified to limit the new value of h .

6. Correctness of the semantics

As the source code for Simulink is not fully available, it is difficult to formally prove that the semantics we propose exactly describes what Simulink computes. However, we are confident that it is exact for two reasons: it conforms to the properties mentioned in

Section 3.1 and, on various examples, it conforms to the results of Simulink simulations. This is detailed in the rest of this section.

Properties verified by our semantics We give two theorems that prove that our semantics conforms to Properties 1 and 2. Both theorems are easily proved using our derivation rules, we only list as proof sketches the rules that make the theorems true.

Theorem 1 (Conform to Property 1). *Let σ_0 be the initial state. Let $\{\sigma_k : k \in [1, n]\}$ be the sequence of states such that $\sigma_0 \Rightarrow \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n$. For each $k \in [0, n]$, let $t_i = \sigma_i(t)$. Then we have: $\forall i \in [0, n-1], t_i < t_{i+1}$.*

Proof sketch. The only rule in which time could go “backward” is the integrator transition \xrightarrow{i} , during the various stages of multi-stage solvers. However, as the transition $\sigma \Rightarrow \sigma'$ only occurs once a step is validated, we know that $\sigma'(t) \geq \sigma(t)$. \square

Theorem 2 (Conform to Property 2). *Let σ_0 be the initial state. Let $\{\sigma_k : k \in [1, n]\}$ be the sequence of states such that $\sigma_0 \Rightarrow \sigma_1 \Rightarrow \dots \Rightarrow \sigma_n$. For each $k \in [0, n]$, let $t_i = \sigma_i(t)$. Let also $S = \text{Eq}(\text{sampling})$ be the set of all sampling times for the blocks of the Simulink program. Then we have:*

$$\forall t \in S \cap [t_0, t_n], t \in \{t_i : i \in [0, n]\}.$$

Proof sketch. Consequence of the rule `UPDATE-H` (Eq. (11)). \square

Numerical correctness of the semantics To establish the correctness of our semantics, we developed a prototype simulator that uses the equation representation of a Simulink program and applies the rules of our semantics. We tested it on three systems that are representative of the different features of Simulink. First, we used a spring-mass system³ which is a purely continuous system without zero-crossing detection nor sampling rates. Then we used our running example which mixes continuous and discrete behaviors as well as zero-crossing events. Finally, we tested our simulator on the bouncing ball system³ which is an example for which zero-crossing algorithms are intensively used. This example also uses so called zero-crossing signals, which are special actions taken only when a zero-crossing event is detected. For conciseness we did not present such signals in our semantics but it is able to handle them.

For all these examples, our simulation gives exactly the same results as Simulink: the time instants chosen by the simulator and the

³<http://matlabtutor.com>

values of the output variables are the same. We are thus confident that our semantics is correct with respect to Simulink simulation.

7. Related work and conclusion

Related work A few works tried to define a formal semantics of Matlab/Simulink considering all its features. Caspi et al. [4] use the synchronous language Lustre to define the semantics of a discrete-time subsets of Simulink. Agrawal and al. [1] and Tiwari [19] based their semantics on hybrid automata. The latter quickly describes the effect of numerical integration without precisely describing the Simulink solver. Nevertheless none of these works define formally the semantics of Simulink solver: instead of defining the result of Simulink simulation, they formally define what Simulink *should* compute, i.e., the solution of the state-space equations encoded by the Simulink model. Our approach is different as we gave a semantics of Simulink that formally defines the result of the numerical simulation. The work by Denckla et al. [9] defines a semantics to a block diagram language very similar to Simulink. Their semantics can mix both discrete and continuous time blocks, but it does not consider zero-crossing nor variable step solvers that are probably the main features of Simulink simulation engine.

Conclusion We presented an operational semantics of Simulink's models. We hence emphasized all aspects of the solver, especially the major steps and the minor steps. More precisely, we described the numerical methods used in the solver such as numerical integration methods and zero-crossing process. Our semantics has two main contributions. First, it mixes both continuous- and discrete-time blocks while previous works only focused on one of it. Secondly, our semantics is solver-dependent, so that we can compute the result of a simulation using different solver or solver parameters. As it covers all aspects of the Simulink solver (e.g., variable step-size methods and zero-crossing localization), this work improves previous attempts to define a formal semantics of Simulink [4, 9].

In this article, we used a core subset of Simulink blocks, including both discrete and continuous aspects of the language. We want to stress out that this subset is large enough to capture more advanced features like enabled or triggered subsystems. Actually, a subsystem with an “enabling” signal can be replaced by all its blocks, each of it preceded by a switch block indicating if it must be executed. Using this transformation, we were able to simulate systems with triggered and enabled subsystems in our simulator and obtained the same results as Simulink. Remark that this work revealed an unreported bug¹ in the Simulink simulation engine when dealing with enabled subsystems with integrator blocks and fixed-step solvers. We cannot describe this bug in details due to space limitation, but its effect is that the output of the subsystem when it becomes disabled is a minor step, which, according to Simulink documentation, should never happen. So, the results of the simulation are very different depending on the chosen solver.

We believe that this semantics can be used for various purposes. First, we want to apply formal methods, in particular abstract interpretation based techniques, on Simulink models to compute (an over-approximation of) the distance between the simulation (as defined by our semantics) and a perfect, mathematical execution. We want to adapt techniques from Goubault et al. [10] for the static analysis of numerical programs where the distance between the program execution using floating-point numbers and the mathematical semantics using real numbers is computed as the sum of simple errors associated with each operation. This could be adapted to determine which part of the simulation loop causes the biggest error and thus help to increase the trust we can have in Simulink outputs.

Another interesting follow-up of this work is to define a set-based simulation, i.e., the simulation of a model where some parameters (or input values) are unknown but within some set. This could be used to determine the behavior of a system with uncertain parameters and thus add some non-determinism in the modelling of a physical system. The main challenge in this direction is the handling of zero-crossing events for which techniques coming from hybrid systems analysis could be used [13].

References

- [1] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *ENCS*, 109:43–56, 2004.
- [2] R. Alur, A. Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *EMSOFT*, pages 89–98. ACM, 2008.
- [3] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. In *UML and Formal Methods*. IEEE, 2010.
- [4] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. *ACM Transaction on Embedded Computing Systems*, 4(4):779–818, 2003.
- [5] A. Chapoutot and M. Martel. Abstract Simulation: a static analysis of Simulink models. In *ICESS*, pages 83–92. IEEE Press, 2009.
- [6] C. Chen, J. Dong, and J. Sun. A formal framework for modeling and validating simulink diagrams. *Formal Aspects of Computing*, 2009.
- [7] P. Cousot. Integrating physical systems in the static analysis of embedded control software. In *APLAS*, volume 3780 of *LNCS*, pages 135–138. Springer, 2005.
- [8] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
- [9] B. Denckla and P. Mosterman. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In *IEEE Conference on Decision and Control*, 2005.
- [10] E. Goubault, M. Martel, and S. Putot. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification*, volume 2991 of *LNCS*, pages 306–313. Springer, 2003.
- [11] E. Hairer, S. Norsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, 2nd edition, 1993.
- [12] A. Kanade, R. Alur, F. Ivancic, S. Ramesh, S. Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow models. In *CAV*, volume 5643 of *LNCS*, 2009.
- [13] C. Le Guernic and A. Girard. Zonotope-hyperplane intersection for hybrid systems reachability analysis. In *HSCC'08*, volume 4981 of *LNCS*, pages 215–228. Springer, 2008.
- [14] E. A. Lee and H. Zheng. Operational semantics of hybrid systems. In *HSCC*, number 3414 in *LNCS*. Springer, 2005.
- [15] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
- [16] L. Shampine and M. Reichelt. The MATLAB ODE Suite. *Journal on Sci. Comput.*, 18(1):1–22, 1997.
- [17] L. Shampine, I. Gladwell, and S. Thompson. *Solving ODEs with MATLAB*. Cambridge Univ. Press, 2003.
- [18] J. Sifakis. A vision for computer science – the system perspective. *Central European Journal of Computer Science*, 1(1):108–116, 2011.
- [19] A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI Intl., 2002.
- [20] A. Tiwari, N. Shankar, and J. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, 2003.
- [21] F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *17th IFAC World Congress*, pages 7967–7972, 2008.

¹ Similar to <http://www.mathworks.fr/support/bugreports/361167>.