# Internship Report — MPRI M2
# Hybrid System Models with Transparent Assertions

Henri Saudubray, supervised by Marc Pouzet, Inria PARKAS

**General Context** — Hybrid systems modelers such as Simulink[1] are essential tools in the development of embedded systems which evolve in physical environments. They allow for precise descriptions of hybrid systems through both continuous-time behaviour defined with Ordinary Differential Equations (ODEs) and discrete-time reactive behaviour similar to what is found in the synchronous languages such as Lustre [PHP87]. The Zélus language [BP13] aims to reconcile synchronous languages with hybrid systems, by taking a synchronous language kernel and extending it with continuous-time constructs similar to what is found in tools like Simulink. Continuous-time behaviour is computed through the use of an off-the-shelf ODE solver such as Sundials CVODE [BIP16; Hin+05].

**Research Problem** — The simulation of hybrid system models, as done in Simulink and Zélus, uses a single ODE solver instance to simulate the entire model at once. This raises a difficult problem: subsystems which seemingly should not interfere with each other end up affecting each other's results. This is due to the chosen integration method. An adaptive solver like Sundials CVODE will vary its step length throughout the integration process, and the addition of new, unrelated ODEs in the system can influence these step lengths, affecting the results obtained for pre-existing ODEs. This is particularly problematic in the case of run-time executable assertions [CR06; HLR92], which are typically expected to be transparent: they should not affect the final result of the computation.

We therefore aim to define a new execution model for hybrid system models, which allows for clear separation between a program and its assertions, in such a way that the results obtained by executing a model with and without its assertions are the same.

**Proposed Contributions** — To solve this, we propose a new runtime for the Zélus language that simulates assertions with their own solvers in order to maintain the separation between assertions and the model they operate on. We first present a low-level denotational semantics for hybrid models, similar to the S-functions of Simulink or the FMI/FMU standard of Modelica, and use this interpretation to give an operational semantics of hybrid model simulations as programs in the synchronous subset of Zélus, using OCaml as our meta-language. The runtime is then *lifted* into Zélus, allowing direct manipulation of hybrid simulations in Zélus proper. The addition of assertions is then a simple modification of the simulation algorithm to handle models with an associated assertion.

**Arguments Supporting Their Validity** — The interpretation of a simulation as a discrete program and their lifting into the source language allows for direct and composable manipulation of systems in isolation. This allows the developer to separate critical parts of the model in order to monitor their execution in isolation. This is already sufficient to represent observers [HLR92] as done in synchronous languages such as Lustre. The added convenience of compiling assertions as distinct models allows the programmer to focus on modelisation without concerning themselves with the required separation of assertions from their model.

The Zélus compiler is currently being reimplemented, and the code associated with this report is in the process of being included in this implementation as the simulation engine. As such, this new runtime has been tested on a variety of examples from the gallery of examples of Zélus; these are available in

[1] https://www.mathworks.com/products/simulink
[2] https://www.codcberg.org/17maiga/hsim

the main repository[2]. A compilation pass from assertions as a syntactic construct to the representation of assertions as a sub-model is currently being implemented.

**Summary and Future Work** — During this internship, we have implemented a new runtime allowing for distinct simulation of models and access to the continuous results without the introduction of discrete events in the model. This has been used to implement assertions in the spirit of Lustre observers. At the time of writing, a compilation pass from assertions in the source language to their internal representation as a separate hybrid model is currently being implemented.

Multiple questions remain. The completion of the compilation pass is an immediate concern; one could then consider a generalization of this compilation pass to *nested* assertions (that is, assertions within assertions), with a recursive definition of a model with assertions. The simulation algorithm for such a model has already been implemented inside the runtime.

The lifting of simulations into the language raises several questions. As discrete Zélus allows for the manipulation of past values in the spirit of Lustre, with operators such as `pre`, we must be able to store values produced by the simulation in memory. Unfortunately for us, this is not always feasible, as explained in Section 2.3.3; the solver's internal manipulation of the state may render previously computed values invalid[3]. One could envision a static analysis to forbid such manipulations, with types being annotated depending on their ability to be copied and stored in memory, similar to Standard ML's *eqtypes* or Haskell's typeclasses.

**Notes and Aknowledgements** — For convenience, we use notation inspired by OCaml's records throughout this report. This is translated into products and projections on these products as expected. The notation $v\#e$ denotes the access to the record member $e$ on the value $v$.
The following work has been implemented in OCaml, and samples of the code are used for illustration purposes. Due to space concerns, OCaml type definitions are ommitted from the main body of the report, and are instead given in Appendix C. They are a direct translation into OCaml of the definitions given in the report. All of the associated code can additionally be found at https://codeberg.org/17maiga/hsim.

I wish to thank my supervisor, Marc Pouzet, for his priceless advice and insight throughout this internship. I am grateful to Timothy Bourke for his kind help with understanding the inner workings of ODE solvers and zero-crossing detection methods, as well as for his advice on this report. Finally, I thank Anne Bouillard, Grégoire Bussonne, Charles de Haro, Paul Jeanmaire, Jean-Baptiste Jeannin, Balthazar Patiachvili, and Loïc Sylvestre for their warm welcome to the PARKAS team and the fascinating discussions throughout my internship.

[3]This is not a problem exclusive to Zélus; Lustre itself suffers from the same issue with externally defined datatypes.

# 1 Introduction

Hybrid system modelers such as Simulink or Modelica have become ubiquitous in the development of embedded systems interacting with physical environments. Their ability to describe both continuous-time behaviour defined using Ordinary Differential Equations, discrete-time behaviours similar to the approach of the synchronous languages such as Lustre [PHP87], and the interactions between the two lends itself perfectly to the modelisation of the interactions between a program and its environment.

Modelers such as Simulink or Modelica are distinct from the classical description of hybrid systems as hybrid automata [Hen00] in their focus on concrete simulation of hybrid models as executable programs. Zélus follows the same approach, by extending a synchronous language kernel *à la* Lustre with continuous-time constructs and compiling models down to a low-level, statically scheduled representation of models as a set of transition functions over an inner state. This internship continues in this direction by providing a precise, executable semantics of the simulation of such a representation using OCaml.

The execution of hybrid models relies on a numerical ODE solver, which computes approximations to the model's behaviour in continuous time. A single solver is used to approximate the behaviour of the entire model; this choice of implementation unfortunately raises unforeseen difficulties. Indeed, the parallel simulation of independent blocks causes interferences between the two, changing the simulation results. This is not a consequence of numerical error, but of the internal simulation engine implementation. The ODE solver approximates the entire model at once, and as such seemingly independent parts of the model end up affecting each other through their impact on the solver's behaviour.

This is particularly unfortunate in the case of run-time assertions. These are typically expected to have no impact on the execution; we call them "transparent", in the sense that their execution does not change the results of the program. Continuous-time assertions may introduce their own ODEs as part of their computation, and as such, impact the simulation of the rest of the model.

To avoid this interference, we propose a new runtime for the simulation of a hybrid model with assertions, where assertions are simulated using their own ODE solver, thus preventing their interference with the model they observe. We first present a low-level denotational semantics of hybrid system models as a collection of functions operating on an inner state, and consider the solving machinery of hybrid system modelers through this interpretation. We then combine a hybrid model with this solving machinery in order to obtain a synchronous operational semantics of the simulation. We briefly discuss some implementation details to motivate some of the choices made. Finally, we use the interpretation of a simulation as a synchronous program to implement the simulation of continuous-time assertions independently from their parent model.

# 2 Hybrid System Model Simulation

The simulation of a hybrid system model is a function from signals to signals. Signals are functions from time to values, modelling the evolution of a value in time. The exact meaning of time depends on the nature of the model. Three possible situations may occur: discrete-time models, akin to those found in synchronous languages like Lustre [PHP87]; continuous-time models with no discontinuities; and hybrid models, which involve both discrete and continuous behaviours.

## 2.1 Discrete-Time Models

Zélus starts from a synchronous language kernel *à la* Lustre, and extends it with continuous-time constructs [BP13]. This synchronous language kernel allows for the description of variables evolving in time through streams of values. The notion of time is logical, and is represented as a series of discrete instants; time is then a value in $\mathbb{N}$, and streams are functions from $\mathbb{N}$ to their respective codomains:

```
(** Run a model on a list of inputs. *)
let dsim (DNode model) input =
  let rec run s = function
    | []      -> []
    | i :: is -> let (o, s) = model.step s i in (o :: run s is) in
  run model.s0 input
```

Listing 1: Discrete simulation in OCaml

$$Stream(V) = \mathbb{N} \to V$$

Given a stream $s : Stream(V)$, we denote $s_n$ the $n$-th value $s(n)$ of the stream.

Computation occurs in successive steps performed at each instant, and may depend on values computed at previous instants. This interpretation of time allows a program written in the synchronous kernel to be considered independently of its physical implementation. Nothing in this representation tells us anything about how much physical time passes between successive instants.

The programs expressed in this kernel, called *discrete nodes*, are functions on streams. At each time instant, given inputs $I$, they produce outputs $O$. Producing these outputs may also depend on previously computed values through operators like `pre(e)`, which returns the value of its sub-expression `e` at the previous instant, and `e1 -> e2`, which returns its left-hand side `e1` at the first instant and its right-hand side `e2` afterwards. This requires nodes to store some previously computed values. To encode this, we define a low-level representation of nodes as Mealy machines, with a state and step function. Nodes therefore operate on an inner state of type $S$: previously computed values must be stored inside this state in order to refer to them afterwards. The behaviour of a node is represented by a step function $step : S \to I \to O \times S$ and an initial state $s_0 : S$, used at the first instant. Given a set of inputs and the current state, the $step$ function produces a set of outputs and a new, updated state. This function is then called at each instant, taking as input the current value of the input signal, and the state produced by the previous instant.

Since programs may wish to reset the state of a node (for instance, when writing automata; further motivation will be given in the following sections), nodes also define a reset function $reset : S \to R \to S$. Since nodes may be parameterized by a value, this reset function takes in an additional reset parameter $R$ and the previous state, and returns an updated state. A discrete model with input $I$ and output $O$ is then a triple of an initial state, and a step and reset function:

$$DNode(I, O, R, S) \stackrel{\text{def}}{=} \{s_0 : S; step : S \to I \to O \times S; reset : S \to R \to S\}$$

Its definition in OCaml is found in Listing 10.

The simulation of such a model then defines two streams: the inner state $s$ and the output $o$:

$$dsim(m)(i_n) = o_n \text{ where } (o_n, s_{n+1}) = m\#\text{step } (i_n, s_n) \text{ and } s_0 = m\#s_0$$

A possible implementation of this simulation in OCaml, where streams are represented by lists of values, is given in Listing 1.

As an example, consider the program in Listing 2, written in the discrete subset of Zélus (it could have been written in Lustre in a similar way). This program computes the evolution of the Van der Pol oscillator in time. The Van der Pol oscillator is defined by the two differential equations

$$\frac{dx}{dt} = y \qquad \frac{dy}{dt} = \mu(1 - x^2)y - x$$

with $\mu$ a scalar parameter. The `f_integr` and `b_integr` nodes implement forward and backward Euler integrators. In more detail, the `f_integr` node take as input a stream `x0` representing the initial value

```
let h = 0.01 (* Integration time step. *)
let mu = 5.0 (* Dampening strength.    *)

(* Forward Euler integrator. *)
let node f_integr(x0: float, x': float) = (x: float) where
  rec x = x0 -> pre(x +. x' *. h)

(* Backward Euler integrator. *)
let node b_integr(x0: float, x': float) = (x: float) where
  rec x = x0 -> pre(x) +. x' *. h

(* Van der Pol oscillator. *)
let node vanderpol_discrete() = (x: float, y: float) where
  rec x = b_integr(1.0, y)
  and y = f_integr(1.0, (mu *. (1.0 -. (x *. x)) *. y) -. x)
```

Listing 2: Van der Pol oscillator in discrete Zélus

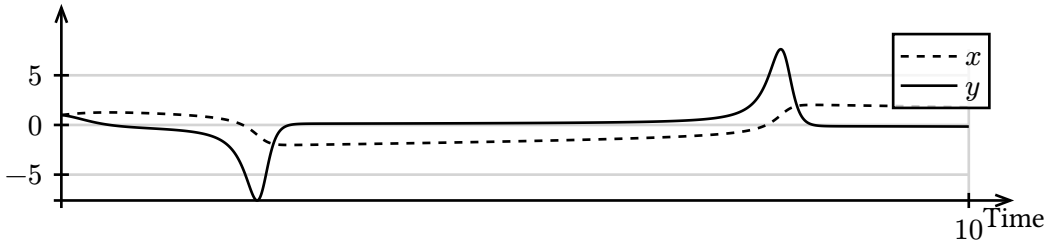


Figure 1: Simulation of Listing 2 with $h = 0.001$

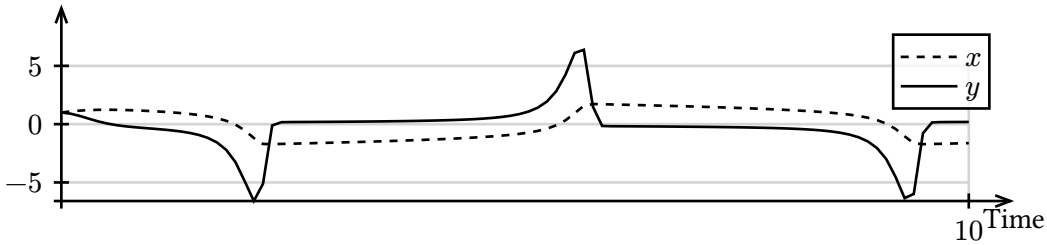


Figure 2: Simulation of Listing 2 with $h = 0.1$

of the signal to be integrated, and a stream `x'` representing the derivative of this same signal, sampled at a predefined integration step `h`. It then defines a new stream `x`, approximating the integral of `x'`, as follows:

$$\mathtt{x}_0 = \mathtt{x0}_0 \qquad \mathtt{x}_{n+1} = \mathtt{x}_n + \mathtt{x'}_n \cdot \mathtt{h}$$

The `f_integr` and `b_integr` nodes are used to compute an approximation of the solution to a restricted form of an initial value problem: given a function $x'(t)$ computing the derivative of a variable $x$ with respect to time (that is, $\frac{dx}{dt}(t) = x'(t)$), and an initial value $x_0$ for this variable, its solution is a function of time $x(t)$ whose derivative is $x'$, and whose value at $t = 0$ is $x(0) = x_0$.

The `vanderpol_discrete` node then uses these integrators to approximate the behaviour of the Van der Pol oscillator. Given an initial position at $x = 1$ and $y = 1$, we can formulate the oscillator through an initial value problem. We can then use two integrators to approximate solutions to $x$ and $y$. The output of `vanderpol_discrete` at instant $n$ is then the pair of the coordinates $x$ and $y$ at time $n \times h$.

## 2.2 Continuous-Time Models

While the model in Listing 2 is simple to understand, it is somewhat rigid: the integration method is fixed, as well as the time step. The simulation results strongly depend on these parameters, as seen in

```
let mu = 5.0
let hybrid vanderpol_continuous() = (x, y) where
  rec der x = y init 1.0
  and der y = (mu *. (1.0 -. (x *. x)) *. y) -. x init 1.0
```

Listing 3: Van der Pol oscillator in continuous Zélus

Figure 1 and Figure 2. Even worse, given a time step of $h = 1$, we quickly encounter `NaN` values. This is due to the shape of the Van der Pol oscillator curve; it alternates between steeper and softer phases, and the integration step must be precise enough to avoid divergence during the steep phases (if the integration step is too big, the high values for the derivatives cause the results to reach the maximum representable floating-point number, after which we obtain `NaN`), with the unfortunate consequence that the simulation in softer phases will be unnecessarily slow. The programmer therefore has to think not only about the model being described, but also about the integration scheme, its impact on performance and the interaction between the model and integrator.

We can do better. Rather than remain in the discrete world, Zélus allows us to express a signal as a function of *continuous time*. Time is no longer logical and represented by a series of discrete instants, but rather physical and continuous. A model is now a function of signals on physical time. Given an input signal of type $I$, it defines a continuously evolving inner state $s$ of type $S$, and an output signal of type $O$. This is represented through an initial state $s_0 : S$ and two functions. The derivative function $der : I \to S \to S'$ computes the derivative $S'$ of the inner state $S$ at a given time using the value of the input signal $i$ and the inner state at that time ($\frac{ds}{dt}(t) = der(i(t), s(t))$); it must be continuous[4]. The output function $out : I \to S \to O$ computes the output $o$ of the model at a given time given the value of the input signal and the inner state at that time ($o(t) = out(i(t), s(t))$). A continuous model is then a tuple of an initial state and of these two functions:

$$CNode(I, O, S, S') \stackrel{\text{def}}{=} \{s_0 : S; der : I \to S \to S'; out : I \to S \to O\}$$

For instance, the model of Listing 2 can be expressed in continuous time as seen in Listing 3. Here, `x` and `y` are expressed directly as initial value problems. The notation `der x = e init e0` expresses that the derivative of `x` with respect to time is `e`, and that the value of `x` at time `t = 0` is `e0`.

A major difference between the discrete and continuous models is that the description of the continuous model is kept separate from the ODE solving machinery. Nothing in Listing 3 expresses any constraints for how the two initial value problems of `x` and `y` are solved – we leave this detail to the language implementation. This allows for greater flexibility in the simulation process, because independence from the solver means we can choose our approximation method independently from the model being simulated.

## 2.3 Numerical ODE Solvers

The simulation of a continuous model solves the initial value problem posed by the initial state $s_0$ and the derivative function $der$, and uses this solution in order to compute the output signal with $out$. This is done using a numerical solver which approximates the solution, such as Sundials CVODE — the `integr` node from Listing 2 is another example of a numerical solver (albeit not a very good one). In general, numerical ODE solvers can be considered through a simple interface: given an initial value problem for a signal $y : Time \to Y$, in the form of a maximum time $stop$, a derivative function $f : [0, stop] \to Y \to Y'$ such that for all $t \in [0, stop]$, $\frac{dy}{dt}(t) = f(t, y(t))$, and an initial value $y_0 : Y$ such that $y(0) = y_0$, a numerical ODE solver provides a function

$$csolve(f)(y_0) : (h : Time) \to (h' : Time) \times (dky : [0, h'] \to Y)$$

[4]This restriction is enforced in Zélus by a typing pass: see [BP13] for details.

This function, given a requested horizon $h : Time$ (this represents the date up to which we wish to know the approximation of the solution), returns a new horizon $h' \leq h$ and an approximation $dky : [0, h'] \to Y$ of the solution to the initial value problem, that is, $dky(t) \approx y(t)$ for all $t \in [0, h']$. This function is called a *dense solution*.[5]

### 2.3.1 Sequential Interpretation

Of particular interest is the fact that we use numerical ODE solvers to compute approximations *sequentially*. Since the solver does not necessarily return an approximation up to the requested horizon, we may need to perform multiple calls to $csolve$ in order to obtain the approximation up to the requested horizon. Furthermore, solvers can be classified in two broad categories: single-step, stateless solvers such as the Runge-Kutta methods, and multi-step, stateful solvers such as SUNDIALS CVODE; the main difference being that the latter *remember* some information about the previous calls to $csolve$ and use this information to improve the approximation. These two characteristics suggest an interpretation of an ODE solver as a particular kind of discrete model. Its internal state is the memory of its previous calls (in the case of a stateless solver, this is empty), and its $step$ function is simply the call to $csolve$. The only missing element is the initialization of the solver with an initial value problem, which can be done as part of the $reset$ function.

More formally, a single call to the $csolve$ function provides us with an approximation of the solution up to the returned horizon $h'$, which may be less than the requested date $h$. To obtain an approximation of the solution at a later date, we must perform another call, this time with initial state $dky(h')$, which is the best approximation of the value of $y(h')$. This new call will provide us with a new horizon $h'' \geq h'$, and a new approximation $dky' : [h', h''] \to Y$. This is then repeated as often as needed to build a larger approximation of the solution.

This sequential process allows a synchronous interpretation of an ODE solver as a discrete node. Rather than producing a single function of continuous time, an ODE solver is a synchronous node that takes in a stream of requested horizons and produces a stream of dense functions and associated horizons.

$$Dense(A) \stackrel{\text{def}}{=} \{h : Time; u : [0, h] \to A\}$$

The ODE solver, given a stream of requested horizons, produces a stream of dense solutions, and operates on an internal state $S$, whose definition depends on the solver being used. Its reset parameter is an initial value problem for a function $y : Time \to Y$, with an initial value $y_0 : Y$ such that $y(0) = y_0$, a maximum horizon $stop : Time$ and a function $f : [0, stop] \to Y \to Y'$ computing the derivative of $y$ (that is, $\frac{dy}{dt}(t) = f(t, y(t))$ for all $t \in [0, stop]$):

$$IVP(Y, Y') \stackrel{\text{def}}{=} \{y_0 : Y; stop : Time; f : [0, stop] \to Y \to Y'\}$$

When simulating a continuous-time model $m$, this initial value problem is obtained using the model's initial state and $der$ function, composed with the current input $i$ (i.e. $f(t, s) = der(i(t), s)$). An ODE solver can thus be considered as a particular kind of discrete node:

$$CSolver(Y, Y', S) \stackrel{\text{def}}{=} DNode(Time, Dense(Y), IVP(Y, Y'), S)$$

A continuous-time signal of type $V$ is now represented as a stream of interval-defined functions, that is, a function from $\mathbb{N}$ to $Dense(V)$. Successive values in the stream are interpreted as successive intervals on the time domain. Given a stream of dense functions $v : \mathbb{N} \to Dense(V)$, the corresponding signal $w : Time \to V$ is defined as

[5] The notation *dky* and the name *dense solution* are taken from the SUNDIALS CVODE interface.

$$w(t) = \begin{cases} v_0 \# u(t) & \text{if } t \in [0, e_0] \\ v_n \# u(t - e_{n-1}) & \text{if } t \in (e_{n-1}, e_n] \text{ for some } n > 0 \\ \text{undefined} & \text{otherwise} \end{cases} \qquad e_n = \sum_{i=0}^{n} v_i \# h$$

where $e_n$ is the stream of instants at which the solver stops. We assume dense functions to be continuous on their domain. However, nothing prevents discontinuities from occurring at the joining points of the stream, that is, for the stream $s$ above, we might have that $s_n \# u(s_n \# h) \neq s_{n+1} \# u(0)$. The ODE solver does not itself introduce discontinuities; the only discontinuities in the system are those introduced by the input signal.

### 2.3.2 Interferences

It is important to note that the solver approximates solutions to the *entire* initial value problem at once. That is, if the initial value problem is composed of two or more unrelated ODEs (in the sense that they operate on distinct sets of variables), the solver does not consider these ODEs separately; rather, it computes approximations to the entire system at once. This can lead to some unexpected behaviour. Some solvers, such as Sundials CVODE, adapt their step length according to the system being approximated. If given a particularly "steep" curve (say, a sine wave with a high frequency), the step length is shortened to mitigate errors; instead, if given a "gentler" curve, the step length is lengthened to increase efficiency. Of course, the approximation obtained depends on the length of the integration steps the solver performs; integrating the same curve with different step lengths yields different results.

The addition of a new, unrelated ODE to a pre-existing system can then alter the results obtained for this system. If the newly added ODE is "steep", the solver reduces its step length to mitigate error, and computes an approximation for the entire system using this new step length. This differs from the steps the solver would have taken had the new ODE not been included; and so, the results obtained for the rest of the system are different. This is particularly important: the simultaneous integration of two unrelated systems yields different results from their integration in isolation[6].

Of course, one could consider a different approximation method, where each ODE is integrated independently with its own ODE solver, rather than all together as a whole system. This is called distributed simulation, and while it solves the issue of interference between unrelated systems, it raises other difficulties and performance concerns, and is more difficult to implement. Zélus chooses instead to live with the consequences of using a single solver for the entire system.

### 2.3.3 Solver Steps and Simulation Steps

The simulation of a continuous-time system with an ODE solver is now considered as a synchronous node. Rather than continuous-time signals, it operates on streams of interval-defined functions. At each step, it takes the value provided by its input signal, initializes the ODE solver with an appropriate initial state and derivative function, and performs a step of the solver to obtain an approximation of the solution to the initial value problem. It then uses this approximation and the model's output function to build an output value.

Since the ODE solver does not necessarily reach the requested horizon in a single step, the simulation may need to execute several steps of the underlying solver for each dense function provided as input. That is, for an input value defined on the time interval $[0, h]$, the ODE solver produces a list of approximations such that their "concatenation" represents the solution over the full interval $[0, h]$, with each item in the list being the result of a step of the solver. Since the ODE solver does not introduce discontinuities, it is safe to consider this concatenation as a single continuous function.

[6] An example of this interference can be found at https://www.codeberg.org/17maiga/hsim, in the `exm/zelus/parallel` folder.

Unfortunately, some ODE solvers (such as Sundials CVODE) work in such a way that stepping the solver multiple times per step of the simulation is infeasible. Solvers operate on an internal state (in Sundials CVODE, this is called a session), and the approximation returned by a step of the solver depends on this internal state. Some solvers rewrite this internal state in-place during a step, invalidating previously produced approximations. We cannot then step the solver multiple times, as all but the last approximation produced will be unusable. But stepping the solver once per simulation step would mean that the solver would have to store its input values until the solver is ready to work on them, which we cannot do either: the input stream might be the output of another simulation, in which case all but the latest input value would be unusable as well.

To solve this conundrum, we wrap the dense functions in our stream with an option type, representing the "readiness" of the simulation to accept a new input value.

$$Option(V) \stackrel{\text{def}}{=} V \cup \{None\} \qquad Signal(V) \stackrel{\text{def}}{=} Option(Dense(V))$$

Rather than stepping the solver multiple times per step of the simulation, we step the solver once, and return the output up to the horizon reached by the solver. If the solver has not reached the input's horizon, we perform another step, giving *None* as input to the simulation, and do so until the solver reaches the input's horizon, after which the simulation simply returns *None*. Once this occurs, it is safe to provide a new dense function as input and begin integration again.

The simulation can then take as input as many *None* values as necessary for the solver to "have time" to reach the horizon of the input. The input stream must then contain as many successive *None* as needed after a dense function for the solver to reach the horizon requested by this dense function. The simulation assumes the input signal takes this form; if a new input value is provided to the simulation before it is done integrating the previous one, no guarantee is made on the correctness of the results.

The simulation of a continuous-time model with a solver is then a special case of a discrete node, with a complex internal state $S$:

$$\begin{aligned} csim : CNode(I, O, S_M, S'_M) \to CSolver(S_M, S'_M, S_S) \to \\ DNode(Signal(I), Signal(O), Unit, Signal(I) \times S_M \times S_S) \end{aligned}$$

A step of the simulation can take three forms, depending on its input and on the state of the simulation. If the input is a new dense function, we assume we are done integrating the previous input. We reset the solver to take into account the new input value in the initial value problem (the model's derivative function uses the input, and so the derivative function given to the solver must change). If the input is *None* and we are not done integrating, we call the solver again and use the approximation returned to build a dense value for the output stream. If the input is *None* and we are done integrating, we do nothing: the simulation is waiting for the next input value. A possible implementation in OCaml is given in Listing 4.

## 2.4 Hybrid Models

Continuous-time models allow for precise descriptions of physical systems and continuous behaviours. However, they lack the ability to describe discrete *events*. For instance, consider the model of a bouncing ball. We can describe its behaviour in the air with two ODEs for the ball's position $y$ (the distance from the ground) and speed $y'$:

$$\frac{dy}{dt}(t) = y'(t) \qquad \frac{dy'}{dt}(t) = -g$$

where $g$ is the gravitational constant ($g \approx 9.81$). Coupled with an initial position and speed, this gives us our initial value problem, which can be approximated as seen above. However, nothing here

```
(** Simulation of a continuous model, as a discrete node. *)
let csim (CNode model) (DNode solver) =
  let s0 = (None, model.s0, solver.s0) in
  let step (current_input, mstate, sstate) new_input =
    match (new_input, current_input) with
    | (Some input, None) ->
        let ivp_f t m = model.fder (input.u t) m in
        let ivp = { y0=mstate; f=ivp_f; h=input.h } in
        None, (Some i, mstate, solver.reset sstate ivp)
    | (None, Some input) ->
        let ({h; u=dky}, sstate) = solver.step sstate input.h in
        let u t = model.fout (input.u t) (dky t) in
        let current_input = if h >= input.h then None else current_input in
        Some {h; u}, (current_input, dky h, sstate)
    | (None, None) -> None, (None, ms, ss)
    | (Some _, Some _) -> assert false in
  let reset (_, ms, ss) () = (None, model.y0, solver.y0) in
  DNode { s0; step; reset }
```

Listing 4: Continuous simulation in OCaml

describes the ball's bouncing behaviour as it touches the ground: it will fall until the end of time. We would ideally like to identify the instant at which the ball touches the ground, stop the simulation at this instant, and perform some changes to the model state to represent the impact of the bounce (say, negate the speed and scale it by a constant), before resuming the simulation with the updated state.

The question of discrete events comes up whenever we wish to include discrete behaviour in a continuous model, such as when modelling the interaction of a discrete program with its physical environment. A controller for a water heater, for instance: the controller turns on or off the heating (discrete change) based on the water's temperature (continuous change). Since discrete models do not include any notion of time, and simply work on the sequence of values they are given, nothing tells us when, in continuous time, we should perform discrete steps. There are many possible choices. We could, for instance, pick a step length $p$ and say that the discrete step is performed periodically at every $p$. In practice, hybrid system modelers like Simulink and Zélus use *zero-crossings*. They monitor a certain value during simulation, and perform a discrete step whenever this value changes from strictly negative to positive or null.

More formally, a zero-crossing on a function $z : [0, h] \to \mathbb{R}$ occurs at time $t \in [0, h]$ if any of the following conditions are met:

$$\begin{aligned} &(z(t-\varepsilon) < 0) \wedge (z(t) > 0) \\ \vee\ &(z(t-\varepsilon) < 0) \wedge (z(t) = 0) \wedge (z(t+\varepsilon) \geq 0) \\ \vee\ &(z(t-\varepsilon) = 0) \wedge (z(t) = 0) \wedge (z(t+\varepsilon) > 0) \end{aligned}$$

with $\varepsilon \in \mathit{Time}$ a strictly positive, solver-dependent constant representing the maximum precision of the zero-crossing detection mechanism[7] (see Section 2.5).

An important point is that discrete events should take *no time* to execute. The physical time of the model does not change during discrete steps. This is similar to the approach of the synchronous languages, where the execution of a step is considered to be instantaneous. Additionally, multiple discrete steps may occur directly after one another. The time basis should reflect this: if we use $\mathit{Time} = \mathbb{R}_+$, successive discrete steps would occur at the same time, and we have no way to distinguish the

[7] For instance, if the zero-crossing mechanism represented time with floating-point numbers, a sensible choice for $\varepsilon$ could be `epsilon_float`.

order of execution, or even represent as a function whose codomain is a single value. In the superdense semantics of [LZ05], the time basis is the set $\mathbb{R}_+ \times \mathbb{N}$, ordered lexicographically ($(t_1, n_1) < (t_2, n_2)$ iff $t_1 < t_2$ or $t_1 = t_2 \wedge n_1 < n_2$)[8]. At each physical instant $t : \mathbb{R}_+$, any number $n$ of discrete steps may occur in successive logical instants $(t, 0), (t, 1), ..., (t, n)$. In our stream representation of signals, discrete instants are instead represented by dense functions with horizon $h = 0$ (that is, defined on the interval $[0, 0]$). The order of execution of successive discrete steps is simply the order given by the stream[9].

A hybrid model describes such systems whose behaviour goes through both continuous and discrete phases. Its state $S$ contains both discrete and continuous parts: the continuous part $Y$ is defined by ODEs, and evolves during continuous phases, while the discrete part is only modified during discrete steps, and must be constant during continuous phases[10]. The model defines functions $cget : S \to Y$ and $cset : S \to Y \to S$ to get and set the continuous state $Y$ from the whole state $S$. To handle zero-crossings, a model with input signal $I$ defines a zero-crossing function $zero : S \to I \to Y \to Z_o$, where $Z_o$ is a vector of values to be monitored for zero-crossings. The inner state $S$ also maintains a vector of Boolean flags $Z_i$, representing the events corresponding to the values in $Z_o$ (a flag is set to true if its corresponding event has occured), and a function $zset : S \to Z_i \to S$ to update the state when an event has been detected.

For implementation reasons, a hybrid model also defines two additional functions $jump : S \to \mathbb{B}$ and $horizon : S \to Time$. The $horizon$ function allows a model to provide a horizon after which no further integration must occur. This is used to indicate whether or not the simulation, after a discrete step, must perform another discrete step (the horizon is 0 in this case); and for the `period` construct, which represents a recurring discrete event. The $jump$ function is used to indicate whether or not a discrete step has introduced a discontinuity in the model's state: if so, the solver must be reset to take this change into account. These two functions exist mainly for implementation purposes: see Section 2.6 for more details.

Finally, a hybrid model defines all functions required by discrete and continuous models:

$$\begin{aligned} HNode(I, O, R, S, Y, Y', Z_i, Z_o) \stackrel{\text{def}}{=} \{ & s_0 : S; \\ & cget : S \to Y; cset : S \to Y \to S; zset : S \to Z_i \to S; \\ & step : S \to I \to Z_i \to O \times S; der : S \to I \to Y \to Y'; \\ & out : S \to I \to Y \to O; zero : S \to I \to Y \to Z_o; \\ & reset : S \to R \to S; horizon : S \to Time; jump : S \to \mathbb{B}\} \end{aligned}$$

[8] This is not the only possible choice; for instance, in [Ben+12], the semantics of hybrid systems is expressed using non-standard analysis, and the time basis is the set of hyperreals ${}^*\mathbb{R}$.

[9] The interpretation of a stream of dense functions $x : \mathbb{N} \to Dense(V)$ as a function of superdense time $w : (\mathbb{R}_+ \times \mathbb{N}) \to V$ is then defined through the following recursive functions:

$$f(t, n, x, i) = \begin{cases} x_i\#u(t) & \text{if } t < x_i\#h \\ g(x_i\#u(0), n-1, x, i+1) & \text{if } t = x_i\#h \\ f(t - x_i\#h, n, x, i+1) & \text{otherwise} \end{cases}$$

$$g(v, n, x, i) = \begin{cases} v & \text{if } n = 0 \\ x_i\#u(0) & \text{if } x_i\#h \neq 0 \\ g(x_i\#u(0), n-1, x, i+1) & \text{otherwise} \end{cases}$$

$$w(t, n) = f(t, n, x, 0)$$

The stream representation is quite similar to the hybrid sequences of [Kay+11, sec. 3.4].

[10] This restriction is enforced by typing: see [BP13] for more details.

```
let hybrid ball(y0, y'0) = y where
  rec der y  = y'    init y0
  and der y' = -9.81 init y'0 reset z -> -0.8 *. last y'
  and     z  = up(-. y)
```

Listing 5: The bouncing ball in Zélus

Zélus provides several ways to specify zero-crossing events, of which the `up(e)` construct is the most common. It monitors its subexpression `e` for zero-crossings, and triggers an event whenever a zero-crossing occurs on `e`. Constructs like `present` and `reset` allow models to execute discrete behaviour when an event is triggered. These constructs are compiled down to an internal representation quite similar to *HNode* (see [Bou+15] for more details). Continuous-time models are a special case of hybrid ones, where the discrete step function does not do anything, and the zero-crossing function does not monitor any signals; and Zélus compiles them down to the same internal representation.

Going back to our bouncing ball, we monitor the expression $-y$ for zero-crossings. Whenever this expression becomes positive, the ball touches the ground. When this zero-crossing event is triggered, we represent the effect of the bounce by negating the speed, so that the ball starts moving up again, and decreasing it by a small factor, to represent the loss of inertia from the collision. A possible implementation in Zélus is given in Listing 5. Whenever the zero-crossing event `z` occurs, `y'` is negated and scaled down; the notation `last y'` represents the left-limit of `y'`.

## 2.5 Zero-crossing Detection

The monitoring of the zero-crossing expressions $Z_o$ requires a mechanism to detect zero-crossings, termed a zero-crossing solver. This solver, given a function $g : Time \to Y \to Z_o$ computing a vector of values to be monitored for zero-crossing, an initial value $y_0 : Y$, and a dense function $y : Dense(Y)$ defined up to a horizon $h : Time$, attempts to locate the first occurrence of a zero-crossing event on the interval $[0, h]$. Multiple methods exist; one of the oldest and most used methods is the Illinois method [Sny53], used by default in Simulink and reimplemented in Zélus. In general, the zero-crossing solver can be summarized as providing a function

$$zsolve : (Time \to Y \to Z_o) \to Dense(Y) \to Time \times Option(Z_i)$$

taking as input a zero-crossing function and a dense solution $v$, and returning a pair of a horizon $h \in [0, v\#h]$ and an optional vector of Boolean flags $z$, such that if the zero-crossing solver detects one or more zero-crossing events, $h$ is the earliest instant at which a zero-crossing occurs, and $z$ is not null; otherwise, $h = v\#h$, and $z$ is null.

Similarly to the ODE solver, a zero-crossing solver can be seen as a synchronous node. Since the zero-crossing function does not change during continuous behaviour (it takes as argument the continuous part of the state, and the discrete part is considered constant during integration), it may be used as a reset parameter; the input is a stream of dense solutions, and the output is a stream of pairs of reached horizons and optional zero-crossings.

$$ZSolver(Y, Z_i, Z_o, S) \stackrel{\text{def}}{=} DNode(Dense(Y), Time \times Option(Z_i), Time \to Y \to Z_o, S)$$

### 2.5.1 Combining with an ODE solver

A zero-crossing solver may be combined with an ODE solver to obtain the full solver mechanism used by the simulation of a hybrid system. This full solver both performs approximation of the solution to the initial value problem of the model, as well as zero-crossing detection using this approximation. That is, given an ODE solver $cs : CSolver(Y, Y', S_C)$ and a zero-crossing solver $zs : ZSolver(Y, Z_i, Z_o, S_Z)$, their composition takes the form of a new synchronous node $s : Solver(Y, Y', Z_i, Z_o, S_C \times S_Z)$, where

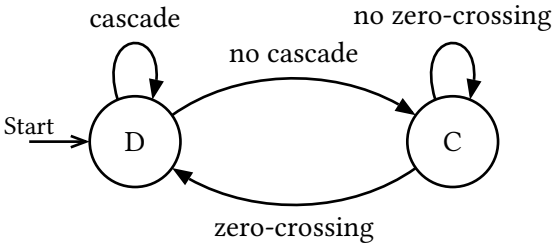


Figure 3: Overview of the simulation loop; $D$ and $C$ represent discrete and continuous step modes

$$Solver(Y, Y', Z_i, Z_o, S) \stackrel{\text{def}}{=} DNode(Time, Dense(Y) \times Option(Z_i), IVP(Y, Y') \times ZCP(Y, Z_o), S)$$

A step of the full solver $s$ takes in a horizon $h : Time$. It then performs a step of the underlying ODE solver $cs$ with input $h$, obtaining a dense function $v$ defined up to a horizon $h' \leq h$ approximating the solution of the initial value problem, and uses this dense function as input to the zero-crossing solver $zs$, which returns a new horizon $h'' \leq h'$ and an optional zero-crossing event $z$. The final horizon $h''$ is then used as the horizon of $v$ (since $v$ is defined on $[0, h']$ and $h'' \leq h'$, then $v$ is defined on $[0, h'']$). Finally, it returns the dense function $v$ with horizon $h''$ and the optional zero-crossing event $z$.

$$s\#step\ (h) \stackrel{\text{def}}{=} (v', z) \text{ where } v = cs\#step\ (h),\ \ (h', z) = zs\#step\ (v) \text{ and } v' = \{h = h', u = v\#u\}$$

The full solver mechanism is then paired with a hybrid model to construct a node representing the simulation of the model.

## 2.6 The Simulation Algorithm

The simulation of a full hybrid model $m : HNode(I, O, R, S_M, Y, Y', Z_i, Z_o)$ with a solver $s : Solver(Y, Y', Z_i, Z_o, S_S)$ can also be seen as a synchronous node, operating on streams of dense functions. Simulation steps take two forms: discrete steps perform state changes and side effects, and continuous steps approximate the solution to the initial value problem of the model and monitors for zero-crossing events. The simulation alternates between these two modes as needed, switching from continuous to discrete steps if a zero-crossing event occurs, and from discrete to continuous steps if no additional discrete steps are necessary. A high-level overview of the simulation's behaviour is given in Figure 3.

The simulation's internal state stores five things: the internal states $s_m : S_M$ and $s_s : S_S$ of the model and solver, respectively; the current simulation $mode$ (either idle, discrete or continuous); a Boolean flag $r$ indicating whether we should reset the solver before the next continous step (see Section 2.6.3); the current input $i : Option(Dense(I))$; and the current simulation time with respect to the input's domain $now \in [0, i\#h]$, used in discrete steps to obtain the correct input. We use the same trick as with continuous-time models to solve the problem of the ODE solver taking several steps to integrate a single input value: we ask that the input stream contains enough successive $None$ values for the ODE solver to finish integrating the current input value (as explained in Section 2.3).

$$State(S_M, S_S, I) = \{s_m : S_M; s_s : S_S; mode : Mode; r : \mathbb{B}; i : Option(Dense(I)); now : [0, i\#h]\}$$

The simulation of a hybrid system is then a discrete node on streams:

$$\begin{aligned} hsim : HNode(I, O, R, S_M, Y, Y', Z_i, Z_o) \to Solver(Y, Y', Z_i, Z_o, S_S) \to \\ DNode(Signal(I), Signal(O), R, State(S_M, S_S, I)) \end{aligned}$$

Its step function's behaviour varies depending on the simulation mode; the following sections describe these behaviours in more details.

### 2.6.1 Discrete steps

A discrete step occurs whenever a zero-crossing event is triggered or a new value is obtained by the simulation. Zero-crossing events may be triggered in two different ways: either by detection using the zero-crossing solver during a previous continuous step, or resulting from the action of a previous discrete step as indicated by *horizon*. New input values require a discrete step to be performed in order to reset the underlying solver. Discrete steps may modify the entire model state, and perform side effects. The simulation's physical time does not advance during a discrete step.

A discrete step of the simulation simply calls the model's *step* function with the appropriate inputs, constructs a dense function defined on $[0, 0]$ using the output (as described in Section 2.4, this represents a discrete step), and updates the simulation state as needed for the next step. Four possible situations arise. If the model's indicated horizon (obtained with the *horizon* function defined by the model) requires us to perform another discrete step, we keep the simulation in discrete mode. If no other discrete step must be performed, but we are done integrating the current input (the current time $now$ is greater than or equal to the current input's horizon $i\#h$), we cannot proceed further, and must wait for additional input; and so we switch to idle mode. If no other discrete step must be performed and we have not reached the input's horizon, we switch to continuous mode. In this case, we may have to reset the solver. This occurs if the current discrete step has caused a discontinuity (as indicated by the model's *jump* function), or if the state's reset flag $r$ is set, in which case we build out a new initial value problem and zero-crossing problem using the current model state and reset the solver.

A possible implementation of the discrete step in OCaml is given in Listing 6 as the `dstep` function.

### 2.6.2 Continuous steps

Continuous steps advance time, approximate the solution to the model's initial value problem using the ODE solver, and monitor the model's zero-crossing function for zero-crossing events using the zero-crossing solver. They operate on a restricted part of the model's state; only the continuous part of the state may be modified, and the discrete part is constant. Furthermore, no side-effects may occur during continuous steps. These restrictions are enforced by a typing pass during the compilation process [Bou+15].

A continuous step performs a call to the solver to obtain both an approximation of the solution to the model's initial value problem and an optional zero-crossing event. It builds a dense function representing the output on the approximation's domain using the model's *out* function. Then, it updates the simulation state as needed for the next step. Once again, four possible situations arise. If a zero-crossing event has occured, we must perform a discrete step, and so we switch to discrete mode and update the model's state to take into account the zero-crossing event with the *zset* function. If no zero-crossing event has occurred, but we have reached the end of the current input (the horizon reached by the solver is greater than or equal to the current input's horizon), we must perform a discrete step as well, and so we switch to discrete mode. If no zero-crossing event has occured and we have not reached the current input's horizon, but we have reached the model's desired stopping point (as indicated by the model's *horizon* function), we must again perform a discrete step, and so we switch to discrete mode. Otherwise, we can continue integrating; we keep the simulation mode as continuous.

A possible implementation of the continuous step in OCaml is given in Listing 6 as the `cstep` function.

### 2.6.3 Complete definition

The full step function then performs the correct kind of step depending on the simulation mode; if the mode is Idle, it simply returns *None* and the unmodified state. When a new dense function is provided as input, it updates the current input and time, sets the reset flag, and switches to discrete mode. Once

again, it expects the input stream to contain as many successive $None$ values as needed for the solver to integrate the entire input, as explained in Section 2.3.

The simulation's reset function simply resets the model using its $reset$ function, sets the simulation mode to Idle, and sets the reset flag $r$ in the simulation state, so that the next discrete step resets the solver before integration.

Finally, its initial state is simply the initial states of the model and solver, the mode set to idle, an empty current input ($None$) and a current time set at 0.

Its implementation in OCaml is given in Listing 6.

## 2.7 Implementation Details

While the above algorithm works, it suffers from a few flaws which limit its efficiency. Indeed, resetting the solver at every new input value is counterproductive. ODE solvers with adaptive step lengths (such as Sundials CVODE) begin integration by performing very small steps in time, and increase their step length later, as they obtain more information on the function they are currently integrating. Resetting the solver slows down the progress of the simulation, as such ODE solvers will perform shorter steps than if they had not been reset.

If two successive input values can be considered to be continuous (that is, the second one only extends the first, with no discontinuity at the joining point), there is no particular reason why we should reset the solver. This occurs for instance between successive continuous steps. The ODE solver does not itself introduce discontinuities, and so the simulation should not reset its solver if taking as input two successive continuous steps of an ODE solver. As a first solution, we can equip our dense values with an additional bit of information $c$ representing whether the next value in the stream is simply an extension of themselves:

$$Dense(V) \stackrel{\text{def}}{=} \{h : Time; u : [0, h] \to V; c : \mathbb{B}\}$$

When building the output value, the simulation knows how this output value behaves compared to the next one: if we just performed a continuous step and the next step is also continuous, we know that the next output value will simply be an extension of the current one, and we can include this information in the current output value. In all other cases, it is safe to consider that successive values are discontinuous. When a simulation receives an input value, it can then reset the solver only if necessary.

Another issue comes from the impossibility of stepping the solver more than once per step of the simulation, as seen in Section 2.3. Since adaptive solver begin with small integration steps, output values will be defined on small intervals. If we compose simulations together, the resulting output will be defined on smaller and smaller intervals, even though this is not always necessary, as the ODE solvers do not introduce discontinuities between their steps.

We can impose another restriction on our ODE solvers to mitigate this issue. If the solver provides a function to copy its internal state, allowing us to preserve the validity of the previous approximations, we can safely step the solver multiple times per step of the simulation and concatenate the results. A discrete node with state copies operating on a state $S$ defines an additional function $copy : S \to S$ returning a copy of the inner state, which may be used for the rest of the simulation. Previously computed approximations then depend on the original copy of the state, which remains untouched by later steps.

$$DNodeC(I, O, R, S) \stackrel{\text{def}}{=} \{s_0 : S; step : S \to I \to O \times S; reset : S \to R \to S; copy : S \to S\}$$

```
(** Discrete simulation step. *)
let dstep (HNode model) (DNode solver) state =
  let i = Option.get state.i in
  let (o, sm) = model.step state.sm (i.u state.now) in
  let state =
    if model.horizon sm <= 0 then { state with sm }
    else if state.now >= i.h then { state with mode=Idle; i=None; sm }
    else if model.jump sm || state.r then (* Reset solver. *)
      let ivp = { h=i.h; y0=model.cget sm; f=fun t y -> model.fder sm (i.u t) y } in
      let zcp = { y0=model.cget sm; f=fun t y -> model.fzer sm (i.u t) y } in
      let ss = solver.reset (ivp, zcp) state.ss in
      { state with mode=Continuous; sm; ss; r=false }
    else { state with mode=Continuous; sm } in
  (Some { h=0.0; u=fun _ -> o }, state)

(** Continuous simulation step. *)
let cstep (HNode model) (DNode solver) state =
  let i = Option.get state.i in
  let stop = min (model.horizon state.sm) i.h in
  let (({ h=now; u=dky }, z), ss) = solver.step state.ss stop in
  let sm = model.cset state.sm (dky now) in
  let out = { h=now; u=fun t -> model.fout sm (i.u t) (dky t) } in
  let state = match z with
    | Some z ->
        let sm = model.zset sm z in
        { state with mode=Discrete; sm; ss; now }
    | None ->
        if model.horizon sm <= 0.0 || now >= i.h
        then { state with mode=Discrete; sm; ss; now }
        else { state with mode=Continuous; sm; ss; now } in
  (Some out, state)

(** Complete simulation algorithm. *)
let hsim (HNode model) (DNode solver) =
  let s0 = { mode=Idle; i=None; now=0.0; sm=model.s0; ss=solver.s0; r=true } in
  let step state i = match (i, state.mode) with
    | Some _, Idle ->
        let state =  { state with mode=Discrete; i; now=0.0; r=true } in
        dstep (HNode model) (DNode solver) state
    | None, Discrete -> dstep (HNode model) (DNode solver) state
    | None, Continuous -> cstep (HNode model) (DNode solver) state
    | None, Idle -> (None, state)
    | Some _, _ -> assert false in
  let reset state r =
    { state with mode=Idle; sm=model.reset r state.sm; r=true } in
  DNode { s0; step; reset }
```

Listing 6: Simulation of a hybrid model in OCaml

Given a solver with state copies, the simulation can then perform multiple steps of the solver, performing a state copy in between each step and concatenating the approximations returned by the solver. This concatenation $join : Dense(V) \times Dense(V) \to Dense(V)$ is defined as expected:

$$join(l, r) = \{h = l\#h + r\#h; u = \lambda t.\ \text{if } t \leq l\#h \text{ then } l\#u(t) \text{ else } r\#u(t + l\#h)\}$$

This does not free us from the option type in our signals, however; the simulation may still produce more than one dense function as output per dense function as input (for instance, if a zero-crossing event occurs).

### 2.8 Lifting the Runtime

An interesting consequence of interpreting simulations as discrete nodes is that we can reasonably consider manipulating them directly inside the language. This takes the form of a module in Zélus' standard library, providing several primitives and utility functions to create and manipulate simulations and signals. For instance, the function

```
val solve : ('i -C-> 'o) -S-> ('i signal -D-> 'o signal)
```

takes as input a continuous-time model (this is indicated by the `-C->` arrow) from `'i` to `'o` and producing a discrete-time node (indicated by the `-D->` arrow) from `'i signal` to `'o signal`[11]. It represents the simulation *with a dedicated instance* of an ODE solver of its argument; that is, the ODE solver used to simulate the argument of `solve` is separate from the one used for the rest of the program. We can now choose which parts of our program we wish to simulate in isolation from the rest. The primitives `compose` and `synchr`, whose signatures are

```
val compose :
  ('a signal -D-> 'b signal) -S->
  ('b signal -D-> 'c signal) -S->
  ('a signal -D-> 'c signal)
```

```
val synchr :
  ('a signal -D-> 'b signal) -S->
  ('a signal -D-> 'c signal) -S->
  ('a signal -D-> ('b * 'c) signal)
```

allow for the composition and synchronization of independent simulations. Indeed, standard composition of simulations does not work: the output of the first would not take into account that the second simulation requires *None* values as input until it is done integrating its current input. We need to ensure that this requirement is met. This is simple: to simulate $f \circ g$, step $g$ and then $f$ when $f$ is done integrating (i.e. when $f$ returns *None*), otherwise only step $f$. Furthermore, two simulations will not necessarily advance at the same speed, and a synchronization primitive is required to simulate two systems in parallel. Its behaviour is simple: step only the simulation that has not progressed as far as the other one, and return the solution only on the interval on which both are defined.

These two combinators have been implemented in OCaml, but it is interesting to note that both of these could just as well be implemented in discrete Zélus, given the right tools to manipulate dense functions (in particular, getting their horizon and splitting them into two smaller, successive dense functions). One could even imagine that simulations themselves are implemented in Zélus; they are only discrete nodes implementing a particular automaton, and discrete Zélus provides all of the necessary constructs to implement automata. Given an interface allowing us to instanciate and call a solver, this seems entirely feasible.

## 3 Hybrid Observers and Assertions

Expressing assertions on the state of a program is an established technique both for formal verification and defensive, runtime checks [CR06; Hoa69]. We focus on runtime executable assertions; these are executed along with the code to check a Boolean property, and interrupt the execution if the property is not met. In OCaml, for instance, the `assert` instruction checks that a certain expression evaluates to `true` at runtime, and raises an exception otherwise. This is an example of a defensive use of runtime assertions as a way to avoid unwanted behaviour: the programmer chooses to suspend execution rather than proceed with a state which they consider invalid.

[11] Due to some restrictions in the higher-order facilities of Zélus, the argument to `solve` must be statically known (that is, we must be able to allocate the necessary memory before starting the execution of the program); this is represented by the `-S->` arrow.

```
let node f (*...*) =
  let v = (*...*) in
  assert (
    let p = f_integr(0.0, v) in
    p >= 0.0
  ); v
```

```
let node assertion_f(v) =
  let p = f_integr(0.0, v) in
  p >= 0.0
let node f (*...*) =
  let v = (*...*) in
  let ok = assertion_f(v) in
  (v, ok)
```

Listing 7: A discrete assertion and its idealized translation as an observer.

```
let hybrid assertion_f(v) =
  let der p = v init 0.0 in
  up(-. p)
let hybrid f (*...*) =
  let v = (*...*) in
  let ok = assertion_f(v) in
  (v, ok)
```

Listing 8: A naïve implementation of a continuous observer.

An expected feature of run-time assertions is that they should not affect the rest of the computation (except stopping execution when they are not fulfilled). We call them *transparent*, in the sense that, running the program with or without assertions, if no error is raised, should produce the same result. Of course, this property requires the expression evaluated in the assertion to not cause any visible side-effects: in OCaml, if the assertion modifies mutable data or performs I/O operations, executing the assertion is not transparent. Still, if the subexpression respects certain criteria, we can safely assume that the presence of the assertion will not affect the final result.

In synchronous languages like Lustre, the equivalent of the run-time assertions of OCaml is an *observer*: a node whose sole purpose is to monitor its input stream to check a certain property. Listing 7 gives an example of an assertion in discrete Zélus, and an idealized translation of this assertion into a separate observer node. The `assertion_f` node monitors its input stream, and returns a Boolean property (here, that a certain value, obtained by integrating another stream with the `f_integr` node of Listing 2, is always positive), with no effect on the rest of the computation in `f`. The Boolean stream is returned along with the output value of `f`, and the caller of `f` will propagate this information up to the main node, where it can be monitored by the user.

In general, a discrete assertion on a model $M : DNode(I, O, R, S)$ can be seen as another node whose input stream is the state $S$ of the parent model, and whose output stream is a Boolean value; it defines its own internal state, with its own local variables, subnodes, and so on and so forth. The parent model then calls the assertion with its internal state during each step.

In continuous-time, one could imagine a similar way of encoding such behaviour. Listing 8 presents a possible implementation of the behaviour of Listing 7 in continuous time. Rather than using inequalities, which are considered discontinuous and are not allowed in continuous code, we use a zero-crossing event to monitor the signal, and perform the appropriate side effect if the event occurs. The integration is once again handled by the simulation, as described in Section 2.6.

Unfortunately, this implementation does not meet our criteria for assertions. Indeed, adding ODEs to a model changes the approximation returned by the ODE solver, as explained in Section 2.3, even if the new ODEs are entirely unrelated to the existing ones. The implementation in Listing 8 does not separate the body of the assertion from its parent model, and the simulation runs both the model and its assertion at the same time. Therefore, the assertion may impact the results of its parent model, and is not transparent. We wish instead to simulate the assertion independently from its parent model, with its own ODE solver.

## 3.1 Models With Assertions

Since an assertion can be considered as a separate model operating on the inner state of its parent model, we can represent a model with assertions as a pair of the parent model $m$, operating on its inner state $S_M$, and a list of assertion models. All assertions operate on the same input datatype $S_M$ (the state of the parent model) and return Boolean output signals $\mathbb{B}$.

$$\begin{aligned} ANode(I, O, R, S_M, Y, Y', Z_i, Z_o) \stackrel{\text{def}}{=} \big\{ & m : HNode(I, O, R, S_M, Y, Y', Z_i, Z_o); \\ & a : List\Big(ANode\Big(S_M, \mathbb{B}, R_A, S_A, Y_A, Y'_A, {Z_i}_A, {Z_o}_A\Big)\Big)\big\} \end{aligned}$$

Note that the output signal is Boolean even in continuous time. This is normally impossible in continuous Zélus, except if the signal is constant during continuous phases; a Boolean signal changing during continuous time could lead to discontinuities, and so operations like conditionals or Boolean operators are rejected by a typing pass [BP13]. There are two possible interpretations of this Boolean output: either assertions benefit from relaxed typing rules for their output, and are allowed a limited subset of discrete behaviours in continuous time; or assertions in continuous time are defined in terms of zero-crossing events, and as such the output of the assertion will be constant during continuous phases (in fact, the output will be constantly true, as the simulation would not have entered a continuous step otherwise). Both interpretations lead to the same updated simulation algorithm, as we will see in Section 3.2. The *ANode* datatype is recursive: indeed, nothing prevents assertions from containing their own assertions, and so on and so forth.

While this representation allows for a lot of expressivity, in most cases, a model with a single assertion suffices. Multiple assertions may be combined as a single one by simply taking the conjunction of their outputs, and nested assertions (assertions within assertions) can be checked as part of the simulation of their parents. We can then define a simpler datatype

$$\begin{aligned} ASNode(I, O, R, S_M, Y, Y', Z_i, Z_o) \stackrel{\text{def}}{=} \{ & m : HNode(I, O, R, S_M, Y, Y', Z_i, Z_o); \\ & a : DNode(Signal(S_M), \mathbb{B}, R, S_A)\} \end{aligned}$$

where the assertion is a single, discrete node operating on dense functions of the model state and returning Boolean values. During compilation, assertions are compiled down to hybrid models, and turned into discrete simulations using a variant of the `solve` function of Section 2.8. This is the current target model of the Zélus compiler; however, simulations of both *ANode* and *ASNode* are implemented in the runtime.

## 3.2 Updated Simulation

The simulation of a system with assertions requires little adjustments from the original simulation algorithm. The main difficulty resides in the fact that we need to construct a dense function of the entire parent model's state. For discrete steps, this is simple: we build a constant function on the model's state defined on the interval $[0, 0]$. For continuous steps, the ODE solver provides us with a dense function of the continuous part of the state. Since the discrete part of the state is constant during integration, we can use the approximation returned by the solver, combined with the model's *cset* function, to build a dense function of the entire state. We need to be careful, however: if the *cset* function rewrites the model's state in-place (this is the case in the code produced by the Zélus compiler), we must update the state back to its value at the horizon reached by the ODE solver before starting the next simulation step.

The simulation state then stores a copy of the model's continuous state at the reached horizon after every continuous step. Before every step, we update the model's state to ensure its correctness. Continuous steps now produce an additional dense function of the state, which is used as input to the

```
let acstep (HNode model) (DNode solver) state =
  let i = (*...*) in let stop = (*...*) in
  let (({ h=now; u=dky }, z), ss) = solver.step state.ss stop in
  let out = { h=now; u=fun t -> model.fout state.sm (i.u t) (dky t) } in
  let dst = { h=now; u=fun t -> model.cset state.sm (dky t) } in
  let state = (*...*) in
  (Some out, state, Some dst)

let asim (ASNode { m=HNode model; a=DNode assertion }) (DNode solver) =
  let s0 = { (*...*); y=None; sa=a.state } in
  let step state i = match (i, state.mode) with
    | (*...*)
    | None, Discrete ->
        let state = { state with sm=model.cset state.sm state.y } in
        let o, state = dstep (HNode model) (DNode solver) state in
        let y = model.cget state.sm in
        let st = { h = 0.0; u = fun _ -> state.sm } in
        let b, sa = assertion.step state.sa (Some st) in
        assert b; (o, { state with sa; y })
    | None, Continuous ->
        let state = { state with sm=model.cset state.sm state.y } in
        let o, state, st = acstep (HNode model) (DNode solver) state in
        let y = model.cget state.sm in
        let b, sa = assertion.step state.sa st in
        assert b; (o, { state with sa; y })
    | (*...*) in
  let reset state r =
    let sm = model.reset r state.sm and sa = assertion.reset r state.sa in
    { state with mode=Idle; sm; sa; r=true } in
  DNode { s0; step; reset }
```

Listing 9: Simulation of a model with assertions in OCaml

assertion. A step of the simulation then performs a step of the parent model, as seen in Section 2.6, and uses the additional output to step the simulation of the assertion as often as needed for it to reach the model's reached time, checking the assertion's output at each step. The updated OCaml implementation is given in Listing 9.

# 4 Related Work

Simulink provides *observer blocks* to monitor signals during execution. These allow for both logging and monitoring of properties without interference with the model, and may be simulated with their own solver or together with the main model. They do not, however, allow for nested assertions; an observer block may not observe another observer block.

# 5 Conclusion and Future Work

Due to space constraints, we do not repeat the conclusion here, and instead refer the reader to Page 2 for a summary and discussion of future work.

# Appendix A — Bibliography


[Ben+12] Benveniste, Albert ; Bourke, Timothy ; Caillaud, Benoît ; Pouzet, Marc: Non-standard semantics of hybrid systems modelers. In: *Journal of Computer and System Sciences* vol. 78 (2012), Nr. 3, pp. 877–910. — In Commemoration of Amir Pnueli

[BIP16] Bourke, Timothy ; Inoue, Jun ; Pouzet, Marc: Sundials/ML: interfacing with numerical solvers.. In: *ACM Workshop on ML*, 2016

[Bou+15] Bourke, Timothy ; Colaço, Jean-Louis ; Pagano, Bruno ; Pasteur, Cédric ; Pouzet, Marc: A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages.. In: Franke, B. (ed.): *Compiler Construction, Lecture Notes in Computer Science*. vol. 9031. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015 — ISBN 978-3-662-46662-9, pp. 69–88

[BP13] Bourke, Timothy ; Pouzet, Marc: Zélus: a synchronous language with ODEs.. In: *Proceedings of the 16th international conference on Hybrid systems: computation and control* . Philadelphia Pennsylvania USA : ACM, 2013 — ISBN 978-1-4503-1567-8, pp. 113–118

[CR06] Clarke, Lori A. ; Rosenblum, David S.: A historical perspective on runtime assertion checking in software development. In: *ACM SIGSOFT Software Engineering Notes* vol. 31 (2006), Nr. 3, pp. 25–37

[Hen00] Henzinger, Thomas A.: The Theory of Hybrid Automata.. In: Inan, M. K. ; Kurshan, R. P. (eds.): *Verification of Digital and Hybrid Systems*. Berlin, Heidelberg : Springer, 2000 — ISBN 978-3-642-59615-5, pp. 265–292

[Hin+05] Hindmarsh, Alan C ; Brown, Peter N ; Grant, Keith E ; Lee, Steven L ; Serban, Radu ; Shumaker, Dan E ; Woodward, Carol S: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. In: *ACM Transactions on Mathematical Software (TOMS)* vol. 31, ACM New York, NY, USA (2005), Nr. 3, pp. 363–396

[HLR92] Halbwachs, N. ; Lagnier, F. ; Ratel, C.: Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. In: *IEEE Transactions on Software Engineering* vol. 18 (1992), Nr. 9, pp. 785–793

[Hoa69] Hoare, C. A. R.: An axiomatic basis for computer programming. In: *Commun. ACM* vol. 12 (1969), Nr. 10, pp. 576–580

[Kay+11] Kaynar, Dilsun K. ; Lynch, Nancy ; Segala, Roberto ; Vaandrager, Frits: *The Theory of Timed I/O Automata, Synthesis Lectures on Distributed Computing Theory*. Cham : Springer International Publishing, 2011 — ISBN 978-3-031-00875-7

[LZ05] Lee, Edward A. ; Zheng, Haiyang: Operational Semantics of Hybrid Systems.. In: Morari, M. ; Thiele, L. (eds.): *Hybrid Systems: Computation and Control, Lecture Notes in Computer Science*. vol. 3414 : Springer Berlin Heidelberg, 2005 — ISBN 978-3-540-25108-8, pp. 25–53

[PHP87] Pilaud, Daniel ; Halbwachs, N ; Plaice, J.A.: LUSTRE: A declarative language for programming synchronous systems.. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*. vol. 178, 1987, p. 188

[Sny53] Snyder, J.N.: Inverse interpolation, a real root of f (x)= 0. In: *University of Illinois Digital Computer Laboratory, ILLIAC I Library Routine H1-71* vol. 4 (1953)

# Appendix B — Table of Contents

# Appendix C — Additional Code

## OCaml Type Definitions

This appendix contains all OCaml type definitions used in code examples throughout the report. They are direct translations into OCaml of the mathematical definitions given in the body of the report.

```
(** Discrete-time model. *)
type ('i, 'o, 'r, 's) dnode = DNode of {
  s0 : 's;                    (* Initial state.  *)
  step : 's -> 'i -> 'o * 's; (* Step function.  *)
  reset : 's -> 'r -> 's;     (* Reset function. *)
}
```

Listing 10: Discrete-time model in OCaml

```
(** Continuous-time model. *)
type ('i, 'o, 's, 'sd) cnode = CNode of {
  s0 : 's;                (* Initial state.       *)
  fder : 'i -> 's -> 'sd; (* Derivative function. *)
  fout : 'i -> 's -> 'o;  (* Output function.     *)
}
```

Listing 11: Continuous-time model in OCaml

```
(** Hybrid model. *)
type ('i, 'o, 'r, 's, 'y, 'yd, 'zi, 'zo) hnode = HNode of {
  s0 : 's;                       (* Initial state. *)
  cget : 's -> 'y;               (* Get the continuous part of the state. *)
  cset : 's -> 'y -> 's;         (* Set the continuous part of the state. *)
  zset : 's -> 'zi -> 's;        (* Set the zero-crossing information.    *)
  horizon : 's -> time;          (* Get the current horizon.              *)
  jump : 's -> bool;             (* Get discontinuity information.        *)
  reset : 's -> 'r -> 's;        (* Reset function.                       *)
  step : 's -> 'i -> 'o * 's;    (* Discrete step function.               *)
  fder : 's -> 'i -> 'y -> 'yd; (* Derivative function.                   *)
  fzer : 's -> 'i -> 'y -> 'zo; (* Zero-crossing function.               *)
  fout : 's -> 'i -> 'y -> 'o;  (* Output function.                      *)
}
```

Listing 12: Hybrid model in OCaml

```
(** Dense function. *)
type 'a dense = {
  h : time;       (* Horizon. *)
  u : time -> 'a; (* Function on [0, h]. *)
}
```

Listing 13: Dense function in OCaml

```
(** Initial value problem. *)
type ('y, 'yd) ivp = {
  y0 : 'y;               (* Initial position.    *)
  stop : time;           (* Stop time.           *)
  f : time -> 'y -> 'yd; (* Derivative function. *)
}
```

Listing 14: Initial value problem in OCaml

```
(** ODE solver. *)
type ('y, 'yd, 's) csolver =
  (time, 'y dense, ('y, 'yd) ivp, 's) dnode
```

Listing 15: ODE solver as a discrete-time model in OCaml

```
(** Zero-crossing problem. *)
type ('y, 'zo) zcp = {
  y0 : 'y;                (* Initial position. *)
  f : time -> 'y -> 'zo; (* Zero-crossing function. *)
}
```

Listing 16: Zero-crossing problem in OCaml

```
(** Zero-crossing solver. *)
type ('y, 'zi, 'zo, 's) zsolver =
  ('y dense, time * 'zi option, ('y, 'zo) zcp, 's) dnode
```

Listing 17: Zero-crossing solver as a discrete-time model in OCaml

```
(** Full solver. *)
type ('y, 'yd, 'zi, 'zo, 's) solver =
  (time, 'y dense * 'zi option, ('y, 'yd) ivp * ('y, 'zo) zcp, 's) dnode
```

Listing 18: Complete solver mechanism in OCaml

```
(** Hybrid simulation mode. *)
type mode = Discrete | Continuous | Idle

(** Hybrid simulation state. *)
type ('i, 'sm, 'ss) state = {
  sm : 'sm;            (* Model state.     *)
  ss : 'ss;            (* Solver state.    *)
  mode : mode;         (* Simulation mode. *)
  r : bool;            (* Reset flag.      *)
  i : 'i dense option; (* Current input.   *)
  now : time;          (* Current time.    *)
}
```

Listing 19: Hybrid simulation state in OCaml

```
(** Hybrid model with a single assertion. *)
type ('i, 'o, 'r, 'sm, 'sa, 'y, 'yd, 'zi, 'zo) asnode = ASNode of {
  m : ('i, 'o, 'r, 'sm, 'y, 'yd, 'zi, 'zo) hnode; (* Model.     *)
  a : ('sm signal, bool, 'sa) dnode;              (* Assertion. *)
}
```

Listing 20: Hybrid model with assertion in OCaml