Building a Hybrid Systems Modeler on Synchronous Languages Principles

Albert Benveniste*, Timothy Bourke†§, Benoît Caillaud*, Jean-Louis Colaço‡, Cédric Pasteur‡ and Marc Pouzet¶§†
*Inria, Campus de Beaulieu, 263 Avenue Général Leclerc, 35042 Rennes. Email: Firstname.Name@inria.fr
† Inria, 2 Rue Simone IFF, 75012 Paris. Email: Timothy.Bourke@inria.fr
‡ ANSYS/Esterel-Technologies, 9 Rue Michel Labrousse, 31100 Toulouse. Email: Firstname.Name@ansys.com
§ École normale supérieure, PSL University, 45 rue d'Ulm, 75230 Paris. Email: Marc.Pouzet@ens.fr
¶ Université Pierre et Marie Curie, Sorbonne Université, 4 Place Jussieu, 75005 Paris

***Abstract*—Hybrid systems modeling languages that mix discrete and continuous time signals and systems are widely used to develop Cyber-Physical systems where control software interacts with physical devices. Compilers play a central role, statically checking source models, generating intermediate representations for testing and verification, and producing sequential code for simulation and execution on target platforms.**

This paper presents a novel approach to the design and implementation of a hybrid systems language, built on synchronous language principles and their proven compilation techniques. The result is a hybrid systems modeling language in which synchronous programming constructs can be mixed with Ordinary Differential Equations (ODEs) and zero-crossing events, and a runtime that delegates their approximation to an off-the-shelf numerical solver.

We propose an ideal semantics based on non standard analysis, which defines the execution of a hybrid model as an infinite sequence of infinitesimally small time steps. It is used to specify and prove correct three essential compilation steps: (1) a type system that guarantees that a continuous-time signal is never used where a discrete-time one is expected and conversely; (2) a type system that ensures the absence of combinatorial loops; (3) the generation of statically scheduled code for efficient execution.

Our approach has been evaluated in two implementations: the academic language Zélus, which extends a language reminiscent of Lustre with ODEs and zero-crossing events, and the industrial prototype Scade Hybrid, a conservative extension of Scade 6.

I. Introduction

Hybrid systems modeling tools [1] like Simulink[1] are widely used in the development of embedded systems. They have evolved progressively from interfaces to numeric solvers aimed solely at simulation to fully fledged languages for programming executable models of dynamical systems that comprise control software and models of its physical environment [2]. Models are not only simulated; they are tested, debugged, and verified, from the early stages of the design, and right along the development chain, until the automatic generation of embedded target code. In state-of-the-art methodologies, compilers statically check source models, produce intermediate representations used by testing and verification tools, and generate sequential code for either efficient simulation or execution on target platforms. The various compilation steps can be tricky to design and implement. How can we ensure that they are semantics preserving, that there is no need to program and verify things twice [3]?

[1]https://www.mathworks.com/products/simulink.html

In this paper, we focus on the design, semantics, and implementation of such languages, following the viewpoint that "hybrid modeling languages are programming languages that happen to have a hybrid systems semantics" [4]. We focus on modeling with Ordinary Differential Equations (ODEs) and do not consider Differential Algebraic Equations (DAEs).

A Programming Language view of Hybrid Systems Modeling

Hybrid systems modeling languages present several new difficulties compared to general purpose programming languages.

The first difficulty is that correctness cannot be defined as the perfect match of an implementation with a specification when part of a model represents a physical device or system rather than a software component. Approximations are generally unavoidable. They result from a lack of exact knowledge of the physical system or from the need to consider simplified models. The science of physics and control engineering treats the important question about the fidelity of a model with respect to the real world [5]. We address solely here programming language issues.

While approximations are unavoidable, they should not arise from artifacts of the modeling tool. It is essential that the simulations of a given model be reproducible both within a single tool and across different tools. Modeling should therefore rely on a *modeling language* equipped with a *mathematical semantics,* i.e., an unambiguous and preferably simple definition of what a given model means, and when its semantics is defined. It must be possible to check model executions against their semantics.

The second difficulty is the gap between an ideal mathematical semantics and the numerical behavior of a model. This gap is due to approximation schemes for differential equations and state events and cannot be avoided in general. One solution is to incorporate details of the simulation engine, and even particular solvers, into the semantics of models [6]. The integration scheme and simulation machinery can be hardwired into the hybrid model to give a purely discrete model [7], possibly through a synchronous encoding [8]. It is also possible to generate a synchronous program, replacing differential equations by difference equations and state events by edge detection. But these approaches burden the semantics with low-level details and make it more complex. It becomes difficult to distinguish the properties of the model itself, which should be independent of a particular solver or integration scheme. Furthermore, solvers require certain invariants in order to give reliable results. For example, the function to be integrated must be continuous and the function for state event detection must be continuously differentiable—both properties that are easily broken (e.g., by importing a function containing a conditional, modulo operator, or side effect). A concrete semantics complements but does not replace an ideal one, that is independent of a given solver. An ideal semantics is enough for proving that compilation steps are semantics preserving and for statically ensuring certain invariants, e.g., that functions to be integrated are continuous and free of side effects.

One final difficulty of hybrid systems modeling languages is the multiplicity of language constructs. These languages offer the expressiveness, and thus the complexity of a general purpose language to which it adds parallelism and the mix of discrete and continuous-time signals together with a numerical solver for their simulation: models mix stream equations, ODEs, zero-crossing events, hierarchical automata, and also side effects, loops, different forms of modular composition, for instance subsystem blocks, and calls to external functions produced by other tools. Moreover, all of these features can be composed in parallel.

While this expressiveness is unquestionably useful for writing real applications, the undisciplined composition of language constructs leads to fragile and 'unsafe' models [9] that are difficult to understand modularly and debug because the behavior of a block can change dramatically when other unconnected blocks are added in parallel. Problems can arise, for instance, in the intricate interactions between discrete and continuous-time: when a continuous-time signal is used where a discrete-time signal is expected and vice versa [10]; when a block explicitly refers to the internal time steps of the solver—the so-called *major steps* of Simulink solvers; when a call is made during integration to a function that unexpectedly performs a side effect or is discontinuous; or when two parallel blocks write to the same shared variable. Some of these situations are detected statically by tools and trigger errors or warnings, but not all of them. The static detection is not imposed as a type discipline: some unsafe models pass the static check while some safe ones do not. The bottom line is that the complexity of actual hybrid systems modeling languages makes the definition of a comprehensive formal static and dynamic semantics difficult to achieve.

Far from being abstract philosophical concerns, these difficulties have practical consequences. System design teams in industry do not like being confronted with unsafe models. They often adopt restrictive programming disciplines by which some operators or certain of their combinations are prohibited [11], [12], [13]. Ensuring that such programming disciplines guarantee the absence of problems is not easy. More importantly, this approach may be more restrictive than a mathematically sound acceptance or rejection discipline.

For the above reasons, we chose the following approach:

1) To identify a minimal language kernel of orthogonal programming constructs that is expressive enough to write realistic hybrid models;
2) to define a detailed static and dynamic semantics of the language and its compilation steps.

A Synchronous Approach to Hybrid Systems Modeling

Synchronous languages [14] partly address the preceding questions but focus solely on discrete-time dynamical systems. The synchronous abstraction is to consider that computations and communication occur instantaneously. This allows the mathematical definition of an ideal and simple semantics: signals are modeled as *stream*, that is, infinite sequences of values, that advance synchronously, and systems are modeled as functions over streams. Fidelity with respect to real-time constraints is checked a posteriori by verifying that the worst case execution time of the generated code is less than the period of execution or the minimum inter-arrival time of triggering inputs. The expressiveness of synchronous languages has been purposefully tuned to ensure important safety properties. Their static semantics — the set of verifications performed by the compiler to characterize correct models — and their dynamic semantics — what is defined by the model — have been mathematically specified. In particular, compilers statically reject programs that are not proven to be deterministic and free of deadlocks, and generate sequential code that executes in bounded time and space. Such features contribute to the fact that SCADE [15],[2] an industrial synchronous language implementation, is now the reference for critical software development in civil avionics.

Nevertheless, synchronous languages do not adequately model or efficiently simulate systems that mix discrete- and continuous-time signals. This causes a break in the development chain, with one language for initial hybrid modeling and another for modeling control software, with the risk of mismatches and task duplications.

To address some of the above issues, we experimented with a new approach to the design, semantics, and implementation of a hybrid systems modeling language that reuses and extends the principles and compilation techniques developed for synchronous languages. As a basic language, we took a synchronous and purely functional language, in the style of LUSTRE, and conservatively extended it with ODEs and zero-crossing events [16], and later adding hierarchical automata [17]. It allows modeling discrete- and continuous-time systems, and expressing their complex interactions. We introduced an ideal semantics based on nonstandard analysis [10] that models executions as infinite sequences of infinitesimally short reactions. Programs must respect a strict typing discipline that forbids using a discrete-time signal where a continuous-time one is expected and vice versa, and also explicitly referring to the major steps of the solver; every side-effect and state change must be aligned with a zero-crossing [18]. Algebraic loops, which may lead to systems with zero or many solutions, are statically rejected [18]. Finally, programs are compiled into statically scheduled code

[2]http://www.esterel-technologies.com/products/scade-suite/

that is paired with an off-the-shelf numerical ODE solver [19], namely SUNDIALS CVODE [20]. These results form the foundation of ZÉLUS [21][3] and the industrial prototype SCADE HYBRID [19] based on SCADE 6 [15]. In the latter, it was possible to reuse all the existing infrastructure like static checking, intermediate languages, and optimisations, with little modification. The extensions for hybrid features require only 5% additional lines of code. Moreover, the proposed language extension is conservative in that classical synchronous features are compiled as before—the same synchronous code is used for both simulation and execution on target platforms.

The resulting language can be used at the specification and implementation stages, both to write a high level hybrid model of the whole system or to focus on a software component only whose embedded code will be automatically generated by the compiler. It can be used as a classical synchronous language, to write only discrete-time programs, without even knowing its hybrid features, and symetrically, as a simple interface language to write ODEs, without even knowing its synchronous features. More interestingly, software components can be programmed in a synchronous language and tested with a programmed model of the physical environment.It is also possible to start from an entirely continuous-time model, and to progressively replace some continuous-time systems by discrete-time ones or vice versa, within a single language and programming model. This combination allows for using continuous-time models (or discrete-time ones) where best suited, to write higher-level executable specifications, and to detect design bugs earlier.

The resulting language is not the composition of two distinct languages and compilers nor does it aggregate several models of computations, as Ptolemy [22] does, for instance. The programming constructs are those of a synchronous dataflow language, namely function definition and application, hierarchical automata, and data-flow equations. These constructs are extended to support continuous-time systems. Static verifications like typing, causality analysis, and initialization analysis have been adapted and extended to apply to the whole language. Finally, the generation of executable code is defined as a small extension to an existing synchronous language compiler.

Organization of the Paper

Section II presents fundamental issues in the design of languages that mix discrete and continuous time with a focus on semantic models. Section III treats an extended example: a Newton's cradle with two colliding pendulums. Section IV treats causality and scheduling. Section V develops the distinction between discrete- and continuous-time signals and systems through a static type discipline. Section VI describes the run-time system, which delegates the simulation of continuous dynamics to an off-the-shelf numerical ODE solver. Section VII describes the architecture of the ZÉLUS language and its compiler and reports on the experimental extension of SCADE 6 called SCADE HYBRID. We conclude with a discussion in Section VIII.

[3]http://zelus.di.ens.fr

This paper follows a tutorial style. Technical details are found in the papers [10], [16], [17], [18], [19], [21], [23].

II. DESIGNING CPS MODELING LANGUAGES

In this section we review and discuss some difficulties that arise as consequences of making hybrid systems modeling tools overly flexible and permissive. The main difficulties relate to the coexistence of continuous and discrete time in models. We develop our discussion using a minimal core of a hybrid systems modeling language, which includes data-flow equations over infinite sequences or *streams*, ODEs, a switch construct, and the possibility to hierarchically compose such elements in parallel to build libraries of predefined subsystems. Models of this core language are written in the concrete syntax of ZÉLUS (so the reader can experiment on their own) but they can easily be written in other languages, like, e.g., PTOLEMY or Simulink. We illustrate issues using several small examples, but the goal is to accurately identify problems in larger, more complicated models.

Details related to the material of this section are found in [16], [18].

A. Mixing Discrete and Continuous Time

Let us begin with purely discrete-time or purely continuous-time systems. Following the tagged-signal model [24], a signal x is a function from a time domain $\mathbb{T}$—a totally ordered set—to a set of values. A deterministic system is a function from a set of input signals to a set of output signals. In a block diagram language, such functions can be defined by writing equations on signals: an equation $(x = e)$, where x is a variable name and e is an expression, is an invariant $(x = e)(t)$ that must hold at every time $t \in \mathbb{T}$, that is, $\forall t \in \mathbb{T}, x(t) = e(t)$, where $x(t)$ is the value of signal x at time t and $e(t)$ is the value of expression e at time t. In expressions, external n-ary operations (e.g., arithmetic or boolean operations that apply pointwise), that is, $(op(e_1, \ldots, e_n))(t) = op(e_1(t), \ldots, e_n(t))$. When E_1 and E_2 are two homogeneous equations, i.e., when they are defined on the same time domain $\mathbb{T}$, the parallel composition E_1 and E_2 means

$$\forall t \in \mathbb{T}, \quad (E_1 \texttt{ and } E_2)(t) = E_1(t) \wedge E_2(t). \qquad (1)$$

The above definitions hold for any time domain $\mathbb{T}$, like, for instance, the integers $\mathbb{N} = \{0, 1, 2, \ldots\}$ or the non-negative reals $\mathbb{R}_{\geq 0}$. We illustrate it with two examples.

Discrete-time: Consider a linear filter defined as the composition of two equations over signals:

```
let node filter(x) = y where
  rec y = 0.2 *. x +. s
  and s = 0.8 *. (0.0 fby y)
```

The keyword node declares that `filter` is a discrete-time function. This node takes an input stream `x` and produces an output stream `y`—by *stream*, we mean a discrete time signal defined over time index $\mathbb{N}$. The output is defined by a recurrence equation; the keyword `rec`, for 'recursive', means that the `y` on the left and right sides of the equations refers to the same stream. The expression $0.0\,\texttt{fby}\,y$ denotes the unit delay applied to the stream y. If $x = (x_n)_{n\in\mathbb{N}}$ and $y = (y_n)_{n\in\mathbb{N}}$ are two streams:

$$\forall n \in \mathbb{N}_{>0},\ (x\,\texttt{fby}\,y)_n = y_{n-1} \text{ and } (x\,\texttt{fby}\,y)_0 = x_0. \qquad (2)$$

The $+.$ and $*.$ stand for addition and multiplication of floating-point numbers. They apply pointwise to all stream elements. The meaning of the two equations that define `y` and `s` is simply:

$$\begin{array}{rcl} \forall n \in \mathbb{N},\ \ y_n & = & (0.2 * x + s)_n = 0.2 * x_n + s_n \\ s_n & = & (0.8 * (0.0\,\texttt{fby}\,y))_n \\ & = & 0.8 * (0.0\,\texttt{fby}\,y)_n \\ & = & 0.8 * y_{n-1} \text{ if } n > 0 \\ & = & 0.8 * 0.0 = 0.0 \text{ if } n = 0 \end{array}$$

Up to syntactic details, the linear filter can be written the very same way in LUSTRE [25], [26] or SCADE 6 [15].[4]

Continuous-time: The second example models a ball falling with initial height `y0` and velocity `v0` as two continuous-time equations composed in parallel.

```
let hybrid falling(y0, v0) = y where
  rec der v = -9.81 init v0
  and der y = v init y0
```

This declaration defines the function `falling` with two inputs `y0` and `v0`, and an output `y`. The keyword `hybrid` indicates that the function relates continuous-time signals. This function is readily expressed in any hybrid systems modeling language, e.g., Simulink [5] or PTOLEMY [27].[6] Its semantics, taking $\mathbb{R}_{\geq 0}$ as the time domain, is:

$$\forall t \in \mathbb{R}_{\geq 0}, v(t) = \texttt{v0}(0) + \int_0^t -9.81\,dt = -9.81\,t$$

$$\forall t \in \mathbb{R}_{\geq 0}, y(t) = \texttt{y0}(0) + \int_0^t v(t)\,dt = \texttt{y0}(0) - 9.81\int_0^t t\,dt$$

Even though a numerical solver would compute a discrete approximation, i.e., two sequences $(y_n)_{n\leq\max}$ and $(v_n)_{n\leq\max}$ on increasing instants $t_n \in \mathbb{R}$, with $t_n < t_{n+1}$ up to some horizon $t_{\max}$, its exact solution is defined by an ideal solver semantics [28]. Besides the problem of relating the ideal semantics to a numerical approximation, these two homogeneous models, whether in discrete- or continuous-time, do not pose any particular difficulties.

Mixing discrete- and continuous-time: In contrast, consider now the two following hybrid models that blend discrete- and continuous-time domains:

```
der time = 1.0 init 0.0
and  cpt = 0.0 fby (cpt +. time)                (3)
```

and;

```
      cpt = 0.0 fby (cpt +. 1.0)
and der y = cpt init 0.0                        (4)
```

The definition (1) for parallel composition and the application of n-ary operators no longer apply: in the first example, `time` is a continuous-time signal (its domain is $\mathbb{R}_{\geq 0}$) but `cpt` is expected to be a stream (its domain is $\mathbb{N}$) because the unit delay `fby` expects two streams. In the second, `cpt` is a stream but `y` is a continuous-time signal and `cpt` should also be continuous-time. Nothing in the model indicates how to relate the (logical) discrete-time domain of `cpt` with the (metric) continuous-time domain of `time` and `y`. It is *wrongly typed*: it combines signals defined on incomparable time domains.

There is, however, a situation where the composition of a stream and a continuous-time signal makes perfect sense. Consider a continuous-time signal `z` that is true on a sequence of increasing instants $t_n \in \mathbb{R}$, $n \in \mathbb{N}$ with $0 \leq t_0$ and $t_n < t_{n+1}$ and false everywhere else. Call such a `z` a discrete *clock*.[7]

> Call *zero-crossing* a clock that is true at t_i if and only if some continuous signal x crosses zero from below: $x(t_i - h) < 0$ for all h such that $-\varepsilon < h < 0$ and $x(t_i + h) > 0$ for all h such that $0 < h < \varepsilon$, where ε is a small enough positive number. (5)

A signal is deemed *discrete* if its changes are activated on a discrete clock, defined as follows [16, §2]:

> A clock is termed *discrete* if it has been declared so, or if it is a sub-sampling of a discrete clock, or if it is a zero-crossing. Otherwise, it is termed *continuous*. (6)

A periodic timer, e.g., that ticks every 0.1 seconds, is an example of a discrete clock. It can be defined, albeit sub-optimally, by a sawtooth signal with slope one, initialized to -0.1, and reset every time it crosses zero. We are now ready to write a corrected version of model (3) where the signal `cpt` is piecewise constant and changes only when `z` is true.

```
let hybrid continuous_with_discrete(z) = cpt
  where
   rec der time = 1.0 init 0.0
   and cpt =
         present z -> 1.0 fby (cpt +. time)
         init 0.0
```

The function `continuous_with_discrete` takes as input a discrete clock `z` and returns the signal `cpt`. The parallel composition of the two equations is perfectly valid: the stream $1.0\,\texttt{fby}\,(\texttt{cpt} + .\texttt{time})$ is aligned with the sequence of instants where `z` is true. The construct `present`...`init` returns a piecewise constant signal, initialized with 0.0 which changes at every occurrence of `z`.

$$\begin{array}{rcll} \texttt{cpt}(t) & = & 0 & 0 \leq t < t_0 \\ \texttt{cpt}(t_0 + h) & = & 1 & 0 \leq h < t_1 - t_0 \end{array}$$

and, for $0 \leq h < t_{n+1} - t_n$,

$$\begin{array}{rcl} \texttt{cpt}(t_n + h) & = & (\texttt{cpt} + \texttt{time})(t_{n-1}) \\ & = & \texttt{cpt}(t_{n-1}) + \texttt{time}(t_{n-1}) \end{array}$$

The above function illustrates the typical situation of a software controller activated periodically on a discrete clock `z` and composed in parallel with a continuous-time model of a plant, as in the following.

```
let hybrid model(z) = o where
```

[4]The `fby` is a primitive operator of SCADE. In LUSTRE, it can be replaced by the initialization operator $->$ and uninitialized delay `pre` with the property that $x\,\texttt{fby}\,y =_{\text{def}} x \texttt{->} \texttt{pre}\,y$.

[5]$\texttt{der}\,y = e\,\texttt{init}\,v_0$ stands for $y = \frac{1}{s}(e)$ inititialized to v_0 in Simulink.

[6]http://ptolemy.eecs.berkeley.edu/ptolemyII/

[7]Models may be Zeno, i.e., t_n may not tend to infinity with n. This problem cannot, in general, be detected at compile time and it leads to problems at run-time, e.g., simulations that run very slowly or that go beyond the Zeno point due to floating-point errors. This is a known and unavoidable problem.

```
rec der x = plant(x, u) init x0
and y = f(x, u)
and o = present z -> command(y) init u0
```

where `command` is a synchronous stream function whose logical steps are aligned on `z`.

A discrete clock like `z` can be used to specify different mode changes, e.g., to reset the value of an integrator. For example, a sawtooth signal with slope 1.0 and reset to 0.0 every time `z` is true, is written:

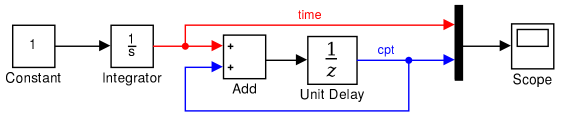
```
der time = 1.0 init 0.0 reset z -> 0.0
```

The signal `z` can be defined with a zero-crossing function. In ZÉLUS, the expression `up(x)` defines a discrete clock that is true when the signal `x` crosses zero from a strictly negative value to a strictly positive one.

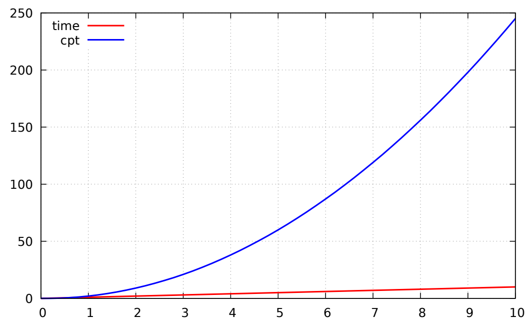
Yet, using $\mathbb{R}$ as a time domain means that we cannot describe a zero-crossing which would be the instantaneous consequence of a previous one (also called a cascade where events appear in sequence but in zero time). We will see later that the previous example can still be justified in this more general setting.

Up to syntactic details, the two preceding models can be written almost as is, in any hybrid systems modeling language, e.g., Simulink[8] or PTOLEMY.

Unsafe mixing of discrete- and continuous time: What if an ill-defined combination of discrete- and continuous-time signals is written as in (3) and (4)? It is tempting to perform implicit conversions between streams and continuous-time signals. But taking $\texttt{cpt}(t) = \texttt{cpt}_{\lfloor t \rfloor}$, where $\lfloor t \rfloor$ is the integer fraction of t, would not make any more sense than $\texttt{cpt}(t) = \texttt{cpt}_{i(t)}$, with $i : \mathbb{R} \mapsto \mathbb{N}$ being any monotone surjective function. The following example, in Simulink this time, illustrates the consequences of such a liberal approach.



(a) Basic model



(b) Simulation of basic model

Figure 1. Composition of models that mix discrete- and continuous-time blocks in Simulink (R2016b).

We consider example (3) again, reprogramming it as in Figure 1a, where the `fby` is replaced by Simulink's delay operator which "holds and delays its input by one iteration".[9] This operator is applied here to a continuous-time signal, though nothing explicitly specifies its discrete-time steps, that is, when "iterations" occur. The constant 1 is connected to the input port of an integrator that is fed directly into a feedback loop involving a unit delay with inherited sampling time. Simulation results are shown in Figure 1b. While the value of time is adequately approximated, it is not at all clear what the output signal cpt should be. In Simulink, cpt is an interpolation of the cumulative "time" computed at so called *major steps* which depend on when the solver decides to stop.[10] For this model, the compiler infers the discrete steps of the unit delay to be the major steps of the simulation algorithm. The model's meaning thus depends critically on the mechanics of the simulation engine. Consequently, if solver parameters, for instance, the minimum step size or the error tolerance, are changed, the value computed for cpt may change significantly. Furthermore, the solver chooses major step sizes for the entire model and not just for individual fragments, which works contrary to the ideal of a component-based approach. In particular, adding an unrelated component in parallel changes the behavior of the original component significantly.

In the above example, we could have equivalently used the "memory block" $\texttt{mem}(x)$[11] instead of the unit delay. If x is a signal, $\texttt{mem}(x)$ is a piecewise constant signal which holds the value of x from the previous major step. If those steps are taken at increasing instants $t_i \in \mathbb{R}$, $\texttt{mem}(x)(t_0) = x_0$ where $t_0 = 0$ and x_0 an initial value and $\texttt{mem}(x)(t_i + h) = x(t_{i-1})$ for $i > 0$, $0 \leq h < t_{i+1} - t_i$. Formally, a memory block takes a discrete clock `z` and an input value `x` such that:

```
let hybrid memory(z, x) =
  present z -> (0.0 fby x) init 0.0
```

Thus, if `major_step` is a discrete clock that denotes the instants where major steps occur, $\texttt{mem}(x)$ is an abbreviation for $\texttt{memory}(\texttt{major\_step}, \texttt{x})$, exhibiting the fact that its value does not depend solely on `z`. Moreover, `major_step` is not just the disjunction of all explicit zero-crossings in a model; it is also true at some intermediate instants where the numerical solver stops for other reasons. With variable step numerical solvers, it is difficult to predict the actual value of `major_step`. Thus, having a block in a model that explicitly refers to the major step make the whole model sensitive to low-level solver choices. Several other operators in the Simulink standard library explicitly refer to major steps, including the memory,[12]

[8]For Simulink, replace `present/init` by a sub-system block triggered on `z` and `fby` by the unit delay of the discrete-time library. The ODE with reset can be implemented directly using the integrator block of Simulink.

[9]See https://www.mathworks.com/help/simulink/slref/unitdelay.html.

[10]A Simulink simulation is structured as a pair of nested loops [29, p. 1-10]. The outer loop calculates the values of a model's signals and states at a single instant of simulation time. Its successive iterations generate the successive instants of a simulation. They are termed *major steps*. The inner loop both integrates continuous states for use in the next iteration of the outer loop and locates zero-crossings. Its successive iterations are termed *minor steps*. The values they calculate are provisional; the solver may require multiple iterations to determine an appropriate approximation.

[11]See https://www.mathworks.com/help/simulink/slref/memory.html.

[12]See https://www.mathworks.com/help/simulink/slref/memory.html.

derivative,[13] time/transport delay,[14] rate limiter,[15] and backlash blocks.[16]

There are understandable reasons to allow operators that implicitly refer to major steps, as Simulink does. Programmers can explicitly exploit the low-level cycles of the simulation algorithm and features like function-call subsystems; even to the point of implementing solvers within a model [30]. This approach can also reduce the number of zero-crossing events and thus give faster simulations. Blocks that explicitly rely on the major steps are definitely useful. However, their use is constrained by rules and guidelines. For instance, the documentation for the memory block advises to "avoid using the Memory block when both these conditions are true: Your model uses the variable-step solver `ode15s` or `ode113`. The input to the block changes during simulation".

Conclusions: The preceding analysis leads first to a question. Is it possible to replace some of "unsafe" blocks described above with safer constructs that have a similar—if not necessarily identical—meaning? And second, to the requirement for a compile-time analysis to reject unsafe combinations of discrete- and continuous-time equations and, in turn, the possibility to guarantee certain properties of accepted programs. This may seem easy for small models, but it is far from trivial for real-size system models. It requires a careful consideration of the mathematical semantics of hybrid models, which we develop in the following sections.

B. Background: Ideal Solver Semantics and Superdense Time

The semantic model of a programming language is supposed to be faithfully implemented by interpreters and compilers, but numerical approximations are unavoidable for simulating models with non-trivial continuous-time dynamics. Must we give up on precision altogether? Must we incorporate the details of numerical solvers into the language semantics as in [31]? An alternative approach, the *Ideal Solver Semantics*, was advocated by Liu and Lee [28], and later further developed by Lee [32]. We briefly present it next.

ODE solvers such as SUNDIALS CVODE [20] compute approximate solutions to the problem of finding a solution to an ODE until some specified stopping event less than a given $t_{max} \in \mathbb{R}$. Formally, for $t_0, t_1 \in \mathbb{R}, t_0 < t_1 \leq t_{\max}$, a signal $x(t)$ is *defined over* $[t_0, t_1]$ *and beyond* if there exists $s_1 > t_1$ such that $x(t)$ is defined over $[t_0, s_1)$. For I a non-empty open interval of $\mathbb{R}$ and $h : I \mapsto \mathbb{R}$, *h possesses a zero-crossing at* $t \in I$ if there exists an open interval $J \subseteq I$ containing t and such that, inside J, $h(s) < 0$ before t and $h(s) > 0$ after t. ODE solvers address the following problem:

Problem 1. *Given $t_0 \in \mathbb{R}$, $f : \mathbb{R}^m \times \mathbb{R} \mapsto \mathbb{R}^m$, $x_0 \in \mathbb{R}^m$, and $g_i : \mathbb{R}^m \times \mathbb{R} \mapsto \mathbb{R}$ for $i = 1, \ldots, k$, find an instant $t_1 > t_0$ and an $\mathbb{R}^m$-valued signal $x(t)$ defined over $[t_0, t_1]$ such that:*

1) x satisfies the ODE $x'(t) = f(x(t), t)$ with initial condition $x(t_0) = x_0$; and

2) t_1 is the smallest instant at which at least one of the functions $t \mapsto g_i(x(t), t)$ possesses a zero-crossing.

If found, the instant t_1 is called the event of zero-crossing.

Variable step solvers adapt their time steps to varying stiffness of f and the detection of the zero-crossings of g. For the solver to work properly, it is therefore advisable that both f and g be smooth enough, at least f continuous and g continuously differentiable.

So far Problem 1 specifies exit conditions from continuous modes. In addition, we must describe how the system state x is reset after time t_1 for the next phase of the dynamics. It has been identified by previous authors [33], [28], [32], [4] that the transition from one continuous mode to the next one may require several steps of a discrete time automaton. A simple example illustrating this need is the Newton's Cradle with three balls or more. The transfer of inertia from the first to the last ball occurs via a cascade of successive transfers, from each ball to the next one.

With reference to the above discussion, the *Ideal Solver Semantics* of a model consists of:

1) an oracle returning an *exact* solution to Problem 1, completed by
2) the specification, for each event, of the discrete time automaton specifying the reset conditions for the state, together with its exit conditions.

Resetting may take several successive computation steps, for which extra instants are required. To unambiguously represent this, Lee and Zheng [4] use *superdense time* [33]:

$$\mathbb{T}^{\mathsf{sd}} =_{\text{def}} \mathbb{R}_{\geq 0} \times \mathbb{N}$$

with the lexicographic order defined by $(s, m) < (t, n)$ if $s < t$, or $s = t$ and $m < n$. A signal is a function $\mathbb{T}^{\mathsf{sd}} \to D$, where D is the value domain. At each real time t, a signal takes a totally ordered sequence of values indexed by $\mathbb{N}$.

This time base is used to represent signals whose dynamics alternate between smooth trajectories of positive duration and sequences of events where the signal takes a finite but arbitrary number of successive values in zero time, the last of which gives an initial value for the next continuous phase. The representation of a signal in superdense time is defined [32]:[17]

$x : \mathbb{T}^{\mathsf{sd}} \to D$ is a partial function with domain $\{(t, n) \mid 0 \leq n \leq N_x(t)\}$, where $N_x : \mathbb{R}_{\geq 0} \to \mathbb{N}$ is the *timeline* of signal x. (7)

For all $t \in \mathbb{R}_{\geq 0}$, $x(t, n)$ is defined for all $0 \leq n \leq N_x(t)$ and undefined otherwise. If $N_x(t)$ is the constant zero, then (7) defines a signal indexed by $\mathbb{R}_{\geq 0} \times \{0\}$, which is isomorphic to $\mathbb{R}_{\geq 0}$. This defines an embedding of ordinary continuous time signals in superdense time signals.

We say that *x has an event at* $t \in \mathbb{R}_{\geq 0}$ if $N_x(t) > 0$, and that x is *chattering free* if the set of t such that x has an event at t is a finite or diverging sequence [32]. We are mainly interested in chattering-free systems here, as they capture the

[13] See https://www.mathworks.com/help/simulink/slref/derivative.html
[14] See https://www.mathworks.com/help/simulink/slref/transportdelay.html
[15] See https://www.mathworks.com/help/simulink/slref/ratelimiter.html
[16] See https://www.mathworks.com/help/simulink/slref/backlash.html

[17] [32] defines signals as total functions $x : \mathbb{T}^{\mathsf{sd}} \to D \cup \{\epsilon\}$ whose value is frozen beyond the timeline and possibly absent at some instant. Here, we only consider non-absent signals for which the two definitions are equivalent.

situations where successive modes, in which the continuous-time dynamics are smooth, are traversed in a non-Zeno way. This is illustrated in Figure 2. In particular, an event occurs

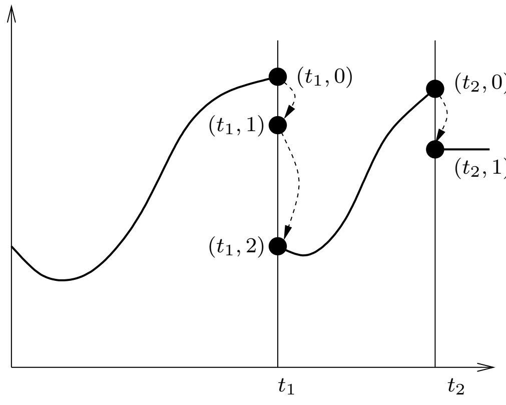


Figure 2. A chattering-free signal $x(t, n)$ in superdense time. The vertical axis is the domain $\mathbb{R}$ of values for x and the horizontal axis is the set of instants $\mathbb{R}_{\geq 0}$. The trajectory is the solid curve augmented with black bullets at events t_1 and t_2. The ordering of discrete instants at the two events is indicated by dashed arrows. The self-loops indicate "freezing" when the height is reached.

at t_2 where x has a jump typical of the resetting of an ODE. At this event, the bullet $x(t_2, 0)$ represents the left-limit of x at t_2, whereas $x(t_2, 1)$ represents the actual value of x at t_2, after which x remains constant. An event also occurs at time t_1, followed by two others.

C. *Synchronous Models in Nonstandard Time*

The ideal solver semantics is adequate to describe the execution of a hybrid system model. In our development, we had a different objective, however, expecting from the semantics:

- to support the study of the parallel composition of programs, and to be modular — meaning that the semantics of complex programs is derived by simple constructions from the semantics of the primitives of the language;
- to provide a mathematical justification of all the compilation steps, from the model to the sequential code paired with the numerical solver, and to make sure those steps can be defined modularly;
- to define a set of static constraints that ensure certain safety invariants for programs that are accepted by the compiler.

The semantic developments of synchronous languages had the same objectives. Hence, not surprisingly, our answer borrows ideas from synchronous languages. It relies on the notion of nonstandard semantics, a technique that essentially treats continuous time as if it were discrete. Let us first recall the essence of synchronous models.

1) Synchronous Models: In this model, input, output and local signals are sequences of values or *streams* that advance synchronously; systems are synchronous stream functions defined by a set of mutually recursive equations over streams. It is very clear here that time is logical [3]: there is no hypothesis on the physical duration between two successive values nor a way to express it *internally* by composing stream functions. an equation $x = a$, where a is an expression, means that $x_n = a_n$, for all $n \in \mathbb{N}$. The parallel composition of two equations is synchronous, that is, $x = a$ and $y = b$, where a and b are expressions, means $(x_n = a_n) \wedge (y_n = b_n)$ for all $n \in \mathbb{N}$. Operations are implicitly lifted to stream. E.g., if $x = (x_n)_{n\in\mathbb{N}}$ and $y = (y_n)_{n\in\mathbb{N}}$, then $x + y = (x_n + y_n)_{n\in\mathbb{N}}$ and a constant like 1 stands for an infinite constant stream.

As an example, consider the forward Euler integration function for the ODE $y' = u$ with $y(0) = i_0$ parameterized by a step size h:

```
let node forward_euler(h)(i, u) = y where
  rec y = i fby (y +. (h *. u))                 (8)
```

In this example, h is a static parameter, which can only be instantiated by an expression whose value is computable at compile time (otherwise the program does not type check). The function forward_euler(h) takes two input streams, i and u, and returns an output stream y such that:

$$\begin{array}{lcll} \forall n \in \mathbb{N}, \quad y_n & = & (i\,\texttt{fby}\,(y + (h * u)))_n & \\ & = & i_0 & \text{if } n = 0 \\ & = & y_{n-1} + (h * u_{n-1}) & \text{otherwise} \end{array}$$

The following function is the forward Euler with reset.

```
let node forward_euler_r(h)(i0, i, r, u)
  = (sy, y)
  where rec sy = i0 fby (y +. (h *. u))
        and y = if r then i else sy             (9)
```

It returns a pair of streams $(\texttt{sy}, \texttt{y})$, where sy is y except at instants when r is true, where it takes the value of i.

For a given stream function, the compiler generates a statically scheduled, loop-free function, which computes a step of the system—we call it the *step* function. Given the current inputs i_n, u_n and a current internal state, this step function returns the current output y_n and a new state (in practice, the state is modified in place). This compilation is possible when the function have no instantaneous feedback, i.e., the value of y_n does not depend on itself. In contrast, writing the following function leads to a compile-time error:

```
let node forward_euler(h)(i, u) = y where
  rec y = y +. (h *. u)
      ^^^^^^^^^^^^^^^^^
Causality error: this expression depends
instantaneously on itself.
```

Here, $y(n)$ depends instantaneously on itself. Rejecting instantaneous feedback loops is a sufficient condition to ensure that fix-point equations have a unique solution. Moreover, it ensures that statically scheduled code can be generated. For every function definition, the *causality analysis* of ZÉLUS computes a type signature that expresses the dependencies between inputs and outputs. This signature is then used everywhere the function is called. If the causality analysis fails, the compilation stops.

2) Synchronous Models in Continuous-Time: Let us attempt to directly extend our previous reasoning to a model defining continuous-time signals, e.g., made of ODEs and equations. For example, consider the model of the temperature in a tank that is heated or not.

```
let hybrid heater(i, heat) = tp where
```

```
  rec der tp = v init i
  and v = if heat then k1 *. (k2 -. temp)
          else k3 *. (k4 -. temp)
```

The function `heater` takes two inputs `i`, `heat` and returns one output `tp`. Suppose that `k1`, `k2`, `k3` and `k4` are constants. The body defines the signals tp and v such that:

$$\begin{array}{rcll} tp' & = & k_1(k_2 - tp)+ & \text{if } heat \text{ is true} \\ tp' & = & k_3(k_4 - tp) & \text{if } heat \text{ is false} \\ t(0) & = & i_0 & \end{array} \qquad (10)$$

Forgetting the derivative definition gives a causality error:

```
let hybrid heater(i, heat) = tp where
  rec tp = v
  and v = if heat then k1 *. (k2 -. temp)
          else k3 *. (k4 -. temp)

Causality error: here is an example of a cycle
v < tp, tp < v
```

There is an intuitive justification for this: because the defined set of mutually recursive equations is cyclic, it is not possible to compute a statically scheduled function that returns the current value of the derivative of tp as a function of tp. For the first version, the compiler produces a statically scheduled function which computes the current derivative, from the current value of `i`, `heat` and `tp`. When given to a numerical solver [34], this solver returns a sequence of approximations $tp(t_n)$ for increasing values of time $t_n \in \mathbb{R}^+$ and $n \in \mathbb{N}$. Thus, for continuous-time signals, an integrator breaks causality loops just as unit delays do for discrete-time signals. Yet, this intuitive justification is not satisfactory as it builds on details about how an ODE solver functions, and it says nothing bout the more general situation when discrete and continuous-time computations depend on each other. In particular, are we sure that the signal `heat` is constant during integration? Solver-agnostic reasoning would be preferable.

We would like to justify that the above function is causal and compute a causality signature prior to code generation in the same way we do when writting synchronous stream functions. Could we not simply reuse the straightforward justification we had for data-flow equations in discrete-time? It turns out that nonstandard analysis makes this possible.

3) Nonstandard Analysis: Nonstandard analysis [35], [36] extends the set $\mathbb{R}$ of real numbers into a superset ${}^\star\mathbb{R}$ of *hyperreals* (also called *nonstandard reals*) with an infinite set of infinitely large numbers and infinitely small numbers. The key properties of hyperreals that we need are the following [36]:

- There exist *infinitesimals*, which are hyperreals that are smaller in absolute value than any real number: An infinitesimal $\partial \in {}^\star\mathbb{R}$ is such that $|\partial| < a$ for any positive $a \in \mathbb{R}$. For x and y, two hyperreals, write $x \approx y$ if $x - y$ is an infinitesimal.
- All relations, operators, and propositional formulas valid over $\mathbb{R}$ are also valid over ${}^\star\mathbb{R}$. For example, ${}^\star\mathbb{R}$ is a totally ordered set. The arithmetic operations $+$, $\times$, etc. can be lifted to ${}^\star\mathbb{R}$; and so on. A hyperreal x is *finite* if there is a standard finite positive real number a such that $|x| < a$.
- Every finite hyperreal $x \in {}^\star\mathbb{R}$ possesses a unique standard real number $st(x) \in \mathbb{R}$ such that $st(x) \approx x$; we call $st(x)$ the *standard part* of x.
- Let $t \mapsto x(t), t \in \mathbb{R}$ be an $\mathbb{R}$-valued (standard) signal. Then:[18]

  x is continuous at instant $t \in \mathbb{R}$ if and only if, for any infinitesimal $\partial \in {}^\star\mathbb{R}$, $x(t+\partial) \approx x(t)$; (11)

  x is differentiable at instant $t \in \mathbb{R}$ if and only if there exists $a \in \mathbb{R}$ such that, for any infinitesimal $\partial \in {}^\star\mathbb{R}$, $\frac{x(t+\partial)-x(t)}{\partial} \approx a$. In this case, $a = x'(t)$. (12)

We can consider the following set of instants:

$$\mathbb{T} \subseteq {}^\star\mathbb{R} \qquad \mathbb{T} = 0, \partial, 2\partial, 3\partial, \dots = \{n\partial \mid n \in {}^\star\mathbb{N}\} \qquad (13)$$

where ${}^\star\mathbb{N}$ denotes the set of *hyperintegers*, consisting of all integers augmented with additional infinite numbers called *nonstandard*. The important point here is that, on one hand, any finite real time $t \in \mathbb{R}$ has an element of $\mathbb{T}$ that is infinitesimally close to it (informally, $\mathbb{T}$ covers $\mathbb{R}$), and, on the other, $\mathbb{T}$ is discrete in that every instant $n\partial$ has a predecessor $(n-1)\partial$ and a successor $(n+1)\partial$.

4) Nonstandard Semantics: Using $\mathbb{T}$ as a time base amounts to indexing signals with the discrete index $n \in {}^\star\mathbb{N}$ defined in (13). Following Suenaga, Sekine, and Hasuo [37], we call the signals indexed by $\mathbb{T}$ *hyperstreams*—the mention of “stream” emphasizes their discrete-time nature. Whenever needed for clarity, we will indicate hyperstreams with the star-prefix notation “${}^\star x$”.

Consider, for example, the forward Euler scheme, implemented by the function `forward_euler` defined in ZÉLUS in (8). Interpret the stream equation for `y` with time base $\mathbb{T}$ defined in (13) and an infinitesimal time step $h = \partial$, as the hyperstream satisfying the recurrence equation:

$${}^\star y_{n+1} = {}^\star y_n + \partial \times {}^\star u_n \quad , \quad {}^\star y_0 = {}^\star i_0 \qquad (14)$$

Since ∂ is infinitesimal, it turns out by (12) that,

for any $t \in \mathbb{R}$ and any $n\partial \in \mathbb{T}$ such that $n\partial \approx t$, $\frac{1}{\partial}({}^\star y_{n+1} - {}^\star y_n) \approx y'(t)$ holds, (15)

which means that (14) approximates the solution of the ODE:

$$y' = u \quad y(0) = i(0) \qquad (16)$$

up to an infinitesimal error. Since such an error is smaller than any positive error, it can be ignored and the stream equation (14) can be seen as a perfect semantics for the ODE. By (11) and (12), the above analysis holds regardless of the particular choice for ∂ provided that it is infinitesimal. All of this legitimates that we call (14) the *nonstandard semantics* of the ODE (16):

```
let hybrid int(i, u) = y where
  rec der y = u init i
```

As another example, consider the integration with possible reset, written:

[18]In (11) and (12), we abuse the notation slightly by invoking the value of signal x at nonstandard instants; see [36] for a formalization.

```
let hybrid int_with_reset(r, i, u) = y where
  rec der y = u init i reset r -> i
```

Its nonstandard semantics is:

$$\begin{aligned} {}^\star y_0 &= i_0 \\ {}^\star y_{n+1} &= \begin{cases} i_{n+1} & \text{if } {}^\star r_{n+1} \text{ occurs} \\ {}^\star y_n + \partial \times {}^\star u_n & \text{otherwise} \end{cases} \end{aligned} \tag{17}$$

We can express this nonstandard semantics using (9), that is, $\mathtt{forward\_euler_r(h)(i,i,r,u)}$ with $\mathtt{h} = \partial$, the infinitesimal basic time step.

To summarize, for a hybrid function, we have replayed the simple semantics for expressions, equations and parallel composition that we gave for a stream function in Section II-C1, replacing $\mathbb{N}$ by ${}^\star\mathbb{N}$ and considering that every step is of infinitesimal length ∂.

The different constructions proposed for the extension ${}^\star\mathbb{R}$ amount to invoking the axiom of choice [36], [10]. Hence, not surprisingly, nonstandard analysis is not effective. No computer exists that can compute with hyperreals. So what do we gain by considering nonstandard semantics? As we have turned differential equations into difference equations, some symbolic reasoning and transformations or static analyses like the detection of instantaneous loops can be easily justified and extended to the whole language. This discrete-time interpretation of a hybrid model is also helpful to prove important properties of hybrid system models, see Section V-B and the properties (1) and (2) therein.

5) From Nonstandard Semantics to Superdense Time Semantics: Once the required program analyses and transformations have been performed, it remains to establish a bridge between:

- the nonstandard semantics of the program that supported our analyses and compilation steps, and
- a standard semantics that formally specifies what should be executed in practice; we chose the Ideal Solver Semantics for this.

This bridge is formalized as the *standardization* of the nonstandard semantics. Its justification is provided in [10], [23], we describe the intuition below.

Consider first the case of the ODE with reset (17). Standardizing the tuple of signals $({}^\star y, {}^\star u, {}^\star r)$ consists in finding a standard domain and a tuple (y, u, r) over it such that ${}^\star y \approx y, {}^\star u \approx u, {}^\star r \approx r$.

- Outside the occurrences of the event ${}^\star r$, ${}^\star x$ satisfies the forward-Euler scheme which we know standardizes to the ODE $x' = u$—so much for continuous-time dynamical systems of timebase $\mathbb{R}_{\geq 0}$.
- How should we standardize the dynamics (17) at an occurrence of event ${}^\star r$? Simply by mapping them to the two successive values ${}^\star y_n, {}^\star y_{n+1}$ occurring in sequence within an infinitesimal amount of time, which itself standardizes as zero time. We thus eventually recover the handling of events performed by the superdense time approach [4], [32].

The standardization of the nonstandard semantics of (17) is illustrated in Figure 3. If several events occur consecutively, the standardization of of a nonstandard signal is a signal in

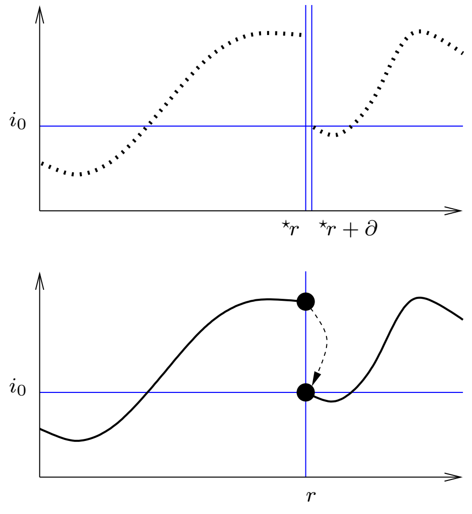


Figure 3. Nonstandard semantics (top) of (17) and the superdense time semantics (bottom) resulting from the standardization of the former. The nonstandard semantics is dashed to indicate that it is discrete time.

superdense time,and it is defined as follows [18], [23]. For $t \in \mathbb{R}$ and $T \subseteq \mathbb{T}$, define the *halo* of t as

$$T_t =_{\text{def}} \{t' \in \mathbb{T} \mid t' \approx t\}$$

i.e., the set of nonstandard instants belonging to $\mathbb{T}$ and infinitely close to t. For ${}^\star x$ a nonstandard signal, its *standardization* x is defined as follows:

1) For $t \in \mathbb{R}_{\geq 0}$, define the standard signal $st({}^\star x)$ by $st({}^\star x)(t) =_{\text{def}} \{st({}^\star x(t')) \mid t' \in T_t\}$. That is, for every real standard time t, we take the set of all standard parts of ${}^\star x(t')$ when t' ranges over the halo of t.
2) The standardization of ${}^\star x$ is the superdense time signal x defined as follows:
   - If $st({}^\star x)(t) = \{v\}$ is a singleton; then $N_x(t) = 0$ and $x(t, 0) = v$;
   - If $st({}^\star x)(t)$ is not a singleton, let $T'_t \subseteq T_t$ be the set of nonstandard instants $t' \in T_t$ at which ${}^\star x$ has a non infinitesimal change, that is, ${}^\star x(t') \not\approx {}^\star x(t' - \partial)$. In the considered case, T'_t is not empty and two cases occur:
     - Either $T'_t = \{t'_1, \ldots, t'_m\}$ is finite; we then define the timeline of x as $N_x(t) = m$ and set
     $$x(t,n) = \begin{cases} st({}^\star x(t'_1 - \partial)) & \text{for } n = 0 \\ st({}^\star x(t'_n)) & \text{for } n = 1, \ldots, N_x(t) \\ \text{undefined} & \text{for } n > N_x(t) \end{cases}$$
     - Or $T'_t = \{t'_1, \ldots\}$ is infinite, which corresponds to a Zeno behavior; then $N_x(t)$ is undefined and so is $x(t, n), n \in \mathbb{N}$ and $x(t', n)$ for any $t' > t$.

In [10], [18], [23], any chattering free hybrid system has a nonstandard semantics that can be standardized to a model defined everywhere in superdense time, i.e., the last case does not occur. This final model corresponds to the definition given by the ideal solver semantics.

The important point about standardization is that it is performed as a final step on the global program. In particular, neither the causality analysis nor the discrete/continuous typing that we develop in Section V rely on superdense semantics for their soundness.

The two semantic domains — nonstandard time and superdense time — complement each other: the former is useful to design and justify modular compile time checking and symbolic transformations, while the latter justifies the final simulation loop in which the resolution of ODEs is delegated to a solver.

III. The Newton's Cradle example

Our subsequent developments are illustrated by a running example, sketched in Figures 4 and 5.

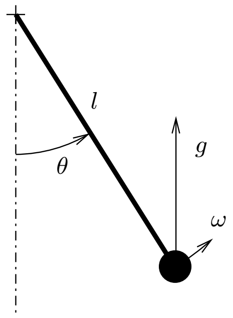


Figure 4. Pendulum in polar coordinates.

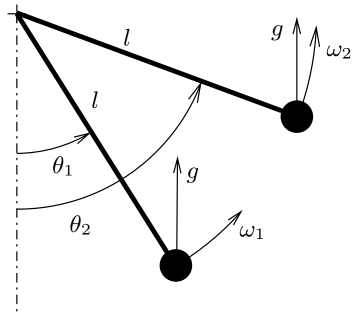


Figure 5. Newton's Cradle with two colliding pendulums.

Consider a pendulum of length l as shown in Figure 4. It is defined in polar coordinates, where θ is the angle of the pendulum relative to the vertical axis, ω is the angular velocity, and g is an acceleration parallel to the vertical axis. The dynamics of the pendulum are defined by the following system of ODEs:

$$\left\{ \begin{array}{lcl} \theta' & = & \omega \\ \omega' & = & a \sin\theta \quad \text{where } a = g/l \end{array} \right. \tag{18}$$

We consider next a Newton's Cradle consisting of two pendulums of equal mass, depicted in Figure 5. Outside collision events, each pendulum is governed by an instance of model (18) and we index the two instances with the subscripts 1 and 2. When the pendulums collide, it is assumed that the interaction between the two balls is elastic, meaning that the energy of the system is invariant. A collision occurs whenever the pendulums are in contact ($\theta_1 \geq \theta_2$) and their relative velocity is negative ($\omega_1 > \omega_2$). Since the balls have equal masses, a collision results in an exchange of angular velocities:

$$\left\{ \begin{array}{lcl} \omega_1^+ & = & \omega_2^- \\ \omega_2^+ & = & \omega_1^- \end{array} \right. \tag{19}$$

This is a physically meaningful illustrative example of small scale. It does not blend plant and control but it combines continuous dynamics and mode changes with event handling, the central difficulty. It does not exhibit the complexity issues of large CPS systems but it is sufficient to illustrate in detail some semantic issues. Prior to presenting our approach, we first review some difficulties in modeling this system using the reference tool Simulink. This will illustrate that, despite its small size, this example already exhibits some interesting and representative difficulties.

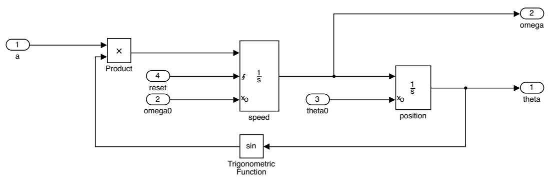


Figure 6. Simulink model of the pendulum

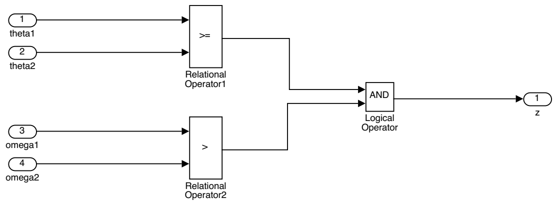


Figure 7. Simulink model of the collision detector

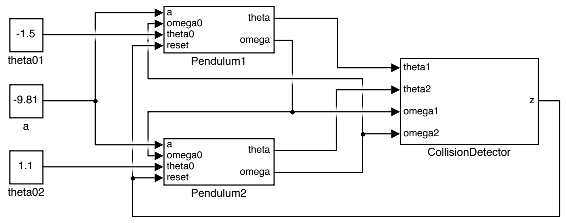


Figure 8. Simulink model of the Newton's Cradle with two pendulums

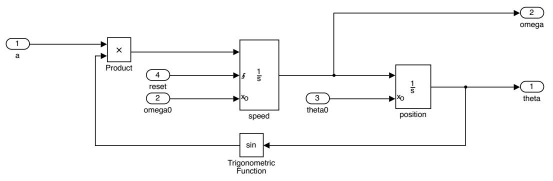


Figure 9. Simulink model of the pendulum with state port

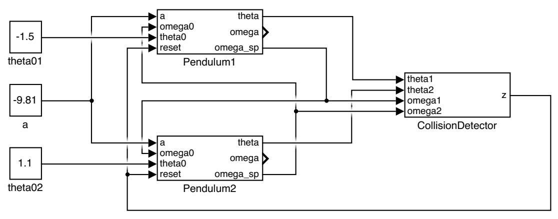


Figure 10. Simulink model of the Newton's Cradle with state ports

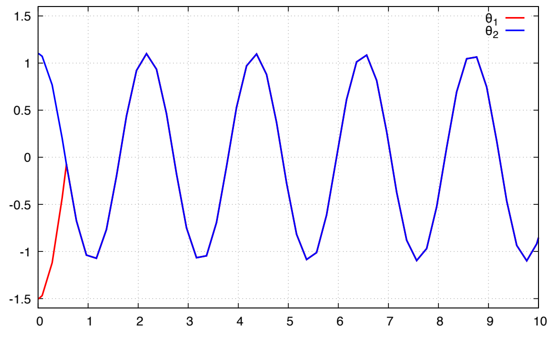


Figure 11. Incorrect behavior of the model with state ports. Trajectory of variable θ_1 (resp. θ_2) is shown in red (resp. blue).

A. Difficulties and Pitfalls

The Simulink model of the single pendulum of Figure 4 and model (18) is given in Figure 6. Modeling the Newton's Cradle seems to be an easy task in Simulink by interconnecting two instances of the pendulum model with the collision detector corresponding to formulas (19) and specified in Simulink as shown in Figure 7. The whole model is shown in Figure 8. However, the simulation of this Simulink model fails at the instant of the first collision, with a "*Block diagram 'pendulums' contains* 1 *algebraic loop(s)*" error message. The reason is that at the instant of reinitialization of the ω_1 and ω_2 integrators, the reset values of the integrators depend on one another. The error message even contains a hint of how this issue can be solved : "*Use integrator state port to avoid algebraic loops*". Indeed, the help pages give the following definition of the state port : "*The output of the state port is the same as the output of the block's standard output port except for the following case. If the block is reset in the current time step, the output of the state port is the value that would have appeared at the block's standard output if the block had not been reset.*". In other words the value of the state port is the left-limit of the integrator's output.

The corrected model, using state ports, is shown Figures 9 and 10. This model simulates without reporting an error. However, the trajectory of the system is incorrect, as shown in Figure 11. This is the consequence of a miscompilation, explained in detail in [18]. In a nutshell, the reinitialization of the integrators are scheduled sequentially, without temporarily storing the state of the first integrator to be reinitialized so that it can be used later to reinitialize the second one. The consequence is that the new velocity of one of the pendulums is not set to be the left limit of the velocity of the other pendulum, but rather, its new velocity. This explains why the pendulums have identical trajectories after the first collision.

This problem can be avoided in Simulink by using two memory blocks $\mathtt{mem}(x)$ that introduce a small delay in the angular velocity variables ω_1 and ω_2; see Figure 12. When a collision occurs, the memory blocks break the algebraic loop and the values of the output ports of the memory blocks are numerically close (but not exactly equal) to the left-limits of the angular velocities. Figure 13 shows the values of ω_1 and $\omega_{1b} = \mathtt{mem}(\omega_1)$ at every major step. z is true at the instant of the contact (line with $\mathtt{z} = \mathtt{1}$ in Figure 13). Note that right after this instant (line below), Simulink takes an extra step that is very close in time to the previous one.[19] It is at this instant that the speeds ω_1 and ω_2 are exchanged, not at the instant where the contact is detected. This extra delay makes the simulation imprecise from a numerical point of view and possibly unpredictable. Yet, for this simple example, the

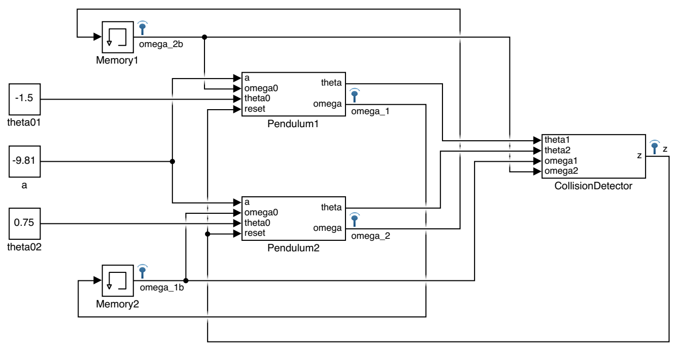


Figure 12. Simulink model of the Newton's Cradle with memory blocks

t	z	omega_1	omega_1b
0.080212582126707	0	0.7842526	0.1569379
0.306505169555583	0	2.9055798	0.7842526
0.560709471482921	0	4.2594777	2.9055798
0.560709471482928	1	-2.2754531	4.2594777
0.560709471482935	0	-2.2754531	-2.2754531
0.839389940219895	0	-1.2708350	-2.2754531

Figure 13. Data logged from the Newton's Cradle with memory blocks

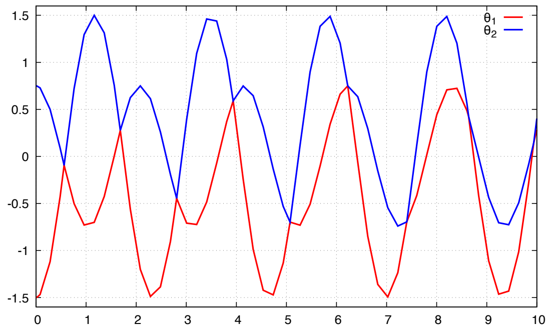


Figure 14. Behavior of the model with memory blocks. Red (resp. blue) pendulum 1 (resp. 2).

[19]This extra step can serve to detect that a zero-crossing signal goes from zero to a strictly positive (or negative) value.

simulation results, given in Figure 14, match the expected behavior. The need to use the memory block to break an algebraic loop means that the written model diverges from the intended mathematical model; it becomes polluted by implementation details.

Two lessons can be drawn from this example. First, the problem with the incorrect reinitialization of the integrators reveals that the scheduling of the successive actions in response to mode change events requires some care. We believe it is best to establish a solid mathematical basis for the compilation steps that schedule reinitialization actions at mode change events. Second, we should remember the discussion of Section II-A where we gave strong arguments for not providing solver-related primitives that rely on major steps. The trouble we had using the memory block in the Newton's Cradle supports this position.

B. *The Newton's Cradle in* ZÉLUS

The $\mathtt{mem}(y)$ operator is unsafe in the general case as its definition does not depend solely on y but on the whole system in which it is used. There is nonetheless a situation where the memory block is useful and still safe: the model only refers to the previous integration step during a discrete step. This situation is very common: it occurs, for example, in a system with two continuous modes M_1 and M_2 producing a signal x, where each mode starts with a value computed previously by the solver and changes between states are aligned on a discrete clock. $\mathtt{mem}(x)$ is used to pass values between the two modes. This actually occurs in the Newton's Cradle example.

Instead of using the operator $\mathtt{mem}(x)$, we propose to replace it by a more constrained one that we call $\mathtt{last}(x)$. At instants when x is left continuous, it stands for the *left limit* of x. The compiler will nonetheless restrict its use through compile-time checks to avoid the pitfalls of the memory block. It will statically reject situations where the value of $\mathtt{last}(x)$ may change during integration.

```
let g = -9.81
let l = 1.0

let hybrid pend(a, omega0, theta0, r) =
                              (last omega, theta)
 where
  rec der theta = omega init theta0
  and der omega = a *. (sin theta)
                  init 0.0 reset r -> omega0

let hybrid newtons_cradle() = (theta1, theta2)
 where
  rec (omega1, theta1) =
          pend(g *. l, omega2, -1.5, contact)
  and (omega2, theta2) =
          pend(g *. l, omega1, 1.1, contact)
  and contact =
         (up(-. theta1 +. theta2))
             on (omega1 -. omega2 < 0.0)
```

Figure 15. The Newton's Cradle model written in ZÉLUS.

Figure 15 shows the Newton's Cradle model written in ZÉLUS. In this model, g is the acceleration and l is the length of the pendulum. The function pend has four arguments: a, omega0, theta0, and r. It returns a pair (last omega, theta), where last omega is the left limit of omega. The function contains two ODEs and omega is reset to the current value of omega0 whenever the r event occurs. Finally, the main function newtons_cradle takes no arguments and returns the pair (theta1, theta2) of signals. The body is made of three equations, one for each pendulum and one that defines the signal contact as a state event that is true when the zero-crossing $\mathtt{up}(-.\,\mathtt{theta1}\ +.\,\mathtt{theta2})$ occurs, that is, when $\mathtt{theta2} >= \mathtt{theta1}$ becomes true and $(\mathtt{omega1}\ -.\,\mathtt{omega2} < 0.0)$ is true.

The memory block is replaced by the operator $\mathtt{last}(.)$. Note that its value is only used at a discrete clock — precisely when $\mathtt{up}(-.\,\mathtt{theta1}\ +.\,\mathtt{theta2})$ is true — and this is checked statically. More generally, ZÉLUS will forbid writing blocks that explicitly refer to the major steps of the solver. Examples are the memory block, the unit delay, the time and transport delay, the rate limiter and backlash that must all be computed on a discrete clock.

In the next section, we develop the nonstandard semantics of the ZÉLUS program of Figure 15 for the Newton's Cradle.

C. *Nonstandard Semantics of the Newton's Cradle*

Throughout this section, whenever convenient, we use greek symbols ω, θ to refer to the variables with extended names omega, theta.

We define ⋆pend to be the nonstandard semantics of the function pend defined in Figure 15, which, given a, ω^0 (we write ω^0 to avoid a double subscript), and θ_0 returns the hyperstreams $({}^\star\mathtt{last}(\omega), {}^\star\theta)$ such that:[20]

$$\begin{array}{lcl} {}^\star\theta_{n+1} & = & {}^\star\theta_n + \partial\, {}^\star\omega_n \\ {}^\star\theta_0 & = & \theta_0 \\ {}^\star\omega_{n+1} & = & \mathtt{if}\ {}^\star r_{n+1}\ \mathtt{then}\ {}^\star\omega^0_{n+1}\ \mathtt{else}\ {}^\star\omega_n + \partial\,(a\,sin({}^\star\theta_n)) \\ {}^\star\omega_0 & = & 0 \end{array} \tag{20}$$

and the semantics of $\mathtt{last}(\mathtt{omega})$ is simply:

$$\begin{array}{lcl} {}^\star\mathtt{last}(\omega)_{n+1} & = & {}^\star\omega_n \\ {}^\star\mathtt{last}(\omega)_0 & = & 0 \end{array} \tag{21}$$

Hence $\mathtt{last}(\mathtt{omega})$ breaks an instantaneous dependence: its output does not depend instantaneously on omega. The nonstandard semantics ⋆newtons_cradle is a function with no inputs that returns ${}^\star\theta_1, {}^\star\theta_2$—the following are equations on streams, and ${}^\star x = {}^\star y$ means that, for all $n \in {}^\star\mathbb{N}, {}^\star x_n = {}^\star y_n$.

$$\begin{array}{l} {}^\star(\omega_1, \theta_1) = {}^\star\mathtt{pend}(g * l, {}^\star\omega_2, -1.5, {}^\star c) \\ {}^\star(\omega_2, \theta_2) = {}^\star\mathtt{pend}(g * l, {}^\star\omega_1, 1.1, {}^\star c) \\ {}^\star c = ({}^\star\mathtt{up}(-{}^\star\theta_1 + {}^\star\theta_2))\ {}^\star\mathtt{on}\ ({}^\star\omega_1 - {}^\star\omega_2 < 0.0) \end{array} \tag{22}$$

The operations ⋆up and ⋆on are defined as follows. The boolean signal ${}^\star\mathtt{up}(x)$ is true when x crosses zero, from a negative to a positive value, in a finite number of infinitesimal steps:

$$\begin{array}{lcll} {}^\star\mathtt{up}(x)_0 & = & \mathit{false} & \\ {}^\star\mathtt{up}(x)_{n+1} & = & \exists m \in \mathbb{N} & (x_{n-m} < 0) \\ & & \wedge & (x_{n-m+1} = 0) \wedge \ldots \\ & & \wedge & (x_n = 0) \wedge (x_{n+1} > 0) \end{array} \tag{23}$$

[20]To simplify the notation, we consider a and the initial conditions constant.

If c is a discrete clock and d a boolean, c on d defines a sub-clock of c: it is true when c occurs, that is, c is true and the boolean condition d is true:

$$(c \,{}^\star\text{on}\; d)_n \quad = \quad {}^\star c_n \wedge {}^\star d_n \tag{24}$$

The nonstandard semantics of the Newton's Cradle is given by (20)–(24).

The nonstandard interpretation `pend` and `newtons_cradle` corresponds to the definition of the discrete-time functions `disc_pend` and `disc_newtons_cradle` parameterized by `h`, and defining up by an edge front detection.

```
let node disc_pend(h)(a, omega0, theta0, r) =
                                  (last_omega, theta)
  where
    rec theta = forward_euler(h)(theta0, omega)
    and (last_omega, omega) =
          forward_euler_r(h)(0.0, r, omega0,
                                   a *. (sin theta))

let node disc_newtons_cradle(h) = (theta1, theta2)
 where
  rec (omega1, theta1) =
          disc_pend(h)(g *. l, omega2, -1.5, contact)
  and (omega2, theta2) =
          disc_pend(h)(g *. l, omega1, 1.1, contact)
  and contact = (up(-. theta1 +. theta2))
                    on (omega1 -. omega2 < 0.0)
```
(25)

Of course, `disc_newtons_cradle`(∂) is not effective since computers can calculate with infinitesimals. On the other hand, the model `disc_newtons_cradle(h)` with `h` small (and standard) is not the nonstandard semantics. Also, it is not the way `newtons_cradle` should be implemented and this is not the way it is implemented by ZÉLUS. It would yield poor simulations, particularly at the zero-crossing events causing mode changes. The nonstandard semantics is not meant to be an operational semantics for execution, but rather a mathematical semantics supporting symbolic analyses and compilation steps. We illustrate this in the next section with the causality analysis and scheduling of ZÉLUS models.

IV. Causality Analysis and Scheduling

Causality or *algebraic* loops [38] (2-34) appear in models when the current value of a signal depend instantaneously on itself. They are not bad in essense but they pose problems of well-definedness (existence, unicity) and compilation. They prevent the simulation from statically ensuring the existence of fix-points, and compilers from generating statically scheduled code. For ZÉLUS, we decided to statically reject those loops. For that, we designed a type system that expresses causality relations between signals and check for the absence of instantaneous cycles [23].

The nonstandard semantics allow to directly adapt the causality analysis and scheduling compilation steps from data-flow synchronous languages like LUSTRE. It interprets a hybrid program as a discrete-time one where time progresses by infinitesimal steps. The nonstandard semantics of an ODE is its forward Euler scheme, albeit with an infinitesimal time step. We introduce a special operator `last`, whose nonstandard semantics is the unit-delay in the $\mathbb{T}$-time base—the intent is to capture the left limit of a signal, whenever it exists. Hence, causality can be defined in a uniform manner and is interpreted here in its weakest form as a dependence relation: all the computations involved in a reaction must be acyclic.

The nonstandard semantics of a hybrid model provides a simple but rigorous criterion for rejecting or accepting ZÉLUS models, based on causality arguments. As an example, consider the following two definitions of a discrete-time integrator followed by the continuous time integrator with reset:

```
let node backward_euler(h, x0, xprime) = x
 where
   rec x = x0 -> pre(x) +. h *. xprime

let node forward_euler(h, x0, xprime) = x
 where
   rec x = x0 -> pre(x +. h *. xprime)

let hybrid int(x0, z, xprime) = x
 where
   rec der x = xprime init x0 reset z -> x0
```

The compiler computes two causality type signatures that express the dependence relation between inputs and outputs.

```
val euler_backward : {}. 'a * 'a * 'a -> 'a
val euler_forward :
              {'a < 'b}. 'b * 'a * 'b -> 'a
val int : {'a < 'b}. 'a * 'a * 'b -> 'a
```

For that, every expression is associated to a time tag and the relation between those tag must be a partial order. The first signature means that for any time tag `'a`, all inputs and output can be computed all together. The second one express that for any time tags `'a` and `'b`, if input `h` and `xprime` have tag `'b` and `x0` have tag `'a`, the result is also on tag `'a`. The relation between tags $'\mathtt{a} <' \mathtt{b}$ means that with all computations done on tag `'a` are scheduled before those with tag `'b`. The causality type signature indeed express that the output `x` does not depend on `h` nor `xprime` but only on `x0`. The last type signature expresses that the output `x` does not depend on `xprime`.

The ZÉLUS models `pend` and `newtons_cradle` of Figure 15 are both accepted since all their dependency cycles are broken by a unit delay in the nonstandard semantics.

In contrast, consider the ZÉLUS programs below, `cyclic_pend` and `cyclic_newtons_cradle`(), which are minor variations of `pend` and `cyclic_newtons_cradle`(). The causality analysis finds a cyclic dependency involving `omega1` and `omega2` and `cyclic_newtons_cradle`() is thus rejected.
The causality analysis of hybrid system models is thus directly inherited from the synchronous languages, thanks to the help of the nonstandard semantics.

Remark 1. For its soundness in the nonstandard semantics, the causality analysis uses the fact that hyperstreams progress by discrete steps. Mathematically speaking, this relies on the fix-point theory for recursive equations over hyperstreams, see [39], [40], [10], [23]. According to this theory, the desired fix-point is obtained as the limit of an iteration producing overlapping partial executions of increasing length, and the causality criteria guarantee that the length of these partial executions actually grows until the entire time line is covered. This technique applies to time lines more general than discrete

```
let hybrid cyclic_pend
      (a, omega0, theta0, res) = (omega, theta)
 where
  rec der theta = omega init theta0
  and der omega = a *. (sin theta)
                  init 0.0 reset res -> omega0

let hybrid cyclic_newtons_cradle() = (theta1, theta2)
 where
  rec (omega1, theta1) =
    cyclic_pend(g *. l, omega2, -1.5, contact)
  and (omega2, theta2) =
    cyclic_pend(g *. l, omega1, 1.1, contact)
  and contact =
    (up(-. theta1 +. theta2))
                   on (omega1 -. omega2 < 0.0)
```

Figure 16. The model `cyclic_newtons_cradle()`.

time in $\mathbb{N}$ [39], [40]. Unfortunately, it does not apply to continuous time lines such as $\mathbb{R}_{\geq 0}$ or $\mathbb{T}^{\mathrm{sd}}$, see [40, pages 54–55] and the discussion about ODEs therein. The problem is that, for an instant $t \in \mathbb{R}$, there is no *next* instant. Our nonstandard semantics, therefore, offers new features not provided by the Ideal Solver Semantics.

V. Discrete- vs. Continuous-time

The examples discussed in Section II-A conflate the discrete time expected by operators like the unit delay, and the discretization time steps chosen by the simulation engine. In our approach, we carefully distinguish the two time bases with the objectives of (1) excluding the operational details of numeric solvers from the semantic model; (2) facilitating compositional reasoning on models; and (3) reducing the effect of solver parameters and choices on simulation results obtained for discrete signals. Our approach consists in developing a static type system that classes expressions as being discrete or continuous time. Details are given in [16], [18], [23].

A. Analysis of Examples

Our first design choice in defining Zélus is to forbid, within Zélus programs, any access to operational details of numeric solvers. No statement in Zélus uses the major step of the solver—in contrast to the `mem()` operator of Simulink. In particular, the model of Figure 12 with its two memory blocks cannot be written in Zélus. The Zélus programmer is expressly prevented from writing such models.

Our second concern is to prevent the undisciplined mixing of discrete and continuous time bases. Distinguishing the two time bases requires inferring, from the program text, which computations are performed in continuous-time and which are performed in discrete-time. The time base of discrete-time computations must be defined unambiguously, based on local and visible program features and not on global and hidden properties of the execution engine. We achieve this in the Zélus prototype by:

1) reusing, in Zélus, the constructs of Lustre (more precisely, of its descendent Lucid Synchrone [41]) for the description of discrete-time dynamics; and
2) introducing a type system to statically separate the continuous parts of a program, which must be exercised by the numerical solver, from the discrete parts, which must not act during integration.

Recall that a signal is deemed discrete if its changes are activated on a discrete clock, see (6). In practice, this means that all discrete changes in a Zélus program are synchronized with a zero-crossing event. The Simulink example from Figure 1a is expressed in Zélus as follows:

```
let hybrid wrong() = cpt where
  rec der time = 1.0 init 0.0
  and  cpt = 0.0 fby (cpt +. time)
```

In this example, the ODE defining `time` is composed in parallel with a stream equation. The compiler rejects it with the following error message:

```
  >  der time = 1.0 init 0.0
     and cpt = 0.0 fby (cpt +. time)
  >            ^^^^^^^^^^^^^^^^^^^^^
  Type error: this is a discrete expression
  and is expected to be continuous.
```

If the stream equation is activated on the discrete clock, the program is accepted with the infered type printed below:

```
let hybrid continuous_with_discrete(z) = cpt
  where
   rec der time = 1.0 init 0.0
   and cpt = present z -> 1.0 fby (cpt +. time)
             init 0.0

val continuous_with_discrete : zero -C-> float
```

In the following section, we provide a sketch of the static typing we use to systematically accept or reject programs at compile time, with explanations in case of rejection. Then, we briefly explain how this technique extends to higher level constructs such as mode machines.

B. Static Typing: Simple Programs

Checking whether a signal is discrete-time (meaning that its instants of change are supported by a discrete-time clock) is undecidable. We thus take a more pragmatic point of view: by convention, a signal is typed *discrete* if it is activated on a zero-crossing event, and otherwise it is typed *continuous*.

The intuition behind the type system is to give a type of the form $t_1 \xrightarrow{k} t_2$ to a function f where k is a kind with three possible values. If $k = \mathtt{C}$, f can only be used in a block activated on a continuous clock. If $k = \mathtt{D}$, f must be activated by a discrete clock. If $k = \mathtt{A}$, then f can be used in expressions of any kind, that is, f is a combinatorial function. Kinds can be compared such that for all k, $k \subseteq k$ and $\mathtt{A} \subseteq k$. The type language is:

$$\begin{array}{lcl} \sigma & ::= & \forall \beta_1, \ldots, \beta_n . t \xrightarrow{k} t \\ t & ::= & t \times t \mid \beta \mid bt \\ k & ::= & \mathtt{D} \mid \mathtt{C} \mid \mathtt{A} \\ bt & ::= & \mathtt{float} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{zero} \end{array}$$

where σ defines types schemes and $\beta_1, \ldots, \beta_n$ are type variables. A type t can be a pair $(t \times t)$, a type variable (β) or a base type (bt), and `zero` stands for the type of a zero-crossing condition whose only value constructor is `up(.)`.

Typing must keep track of both the types of defined nodes (signal functions) and local signals. A global environment G assigns type schemes (σ) to global identifiers and an environment H assigns types to variables.

$$\begin{array}{lcl} G & ::= & [f_1 : \sigma_1; \ldots ; f_n : \sigma_n] \\ H & ::= & [\,] \mid H, x : t \end{array}$$

If H_1 and H_2 are two environments, $H_1 + H_2$ is the union of the two, provided their domains are disjoint.

Typing is defined by asserting judgments like these two:

$$\text{(TYP-EXP)}\quad G, H \vdash_k e : t \qquad\qquad \text{(TYP-ENV)}\quad G, H \vdash_k E : H'$$

The predicate (TYP-EXP) states that under the global environment G and the local environment of signals H, expression e has type t and kind k. The predicate (TYP-ENV) states that equation E produces the type environment H' and has kind k.

Programs are typed under an initial global environment G_0 containing, in particular, the type signature for imported primitives. As an example, we give the signature of addition, equality, and the conditional. They are all of kind A since they can be executed on either a discrete clock or a continuous clock. The unit delay has kind D. The zero-crossing function up(e) must be activated on a continuous clock, and hence its kind is C.

$$\begin{array}{lcl} (+) & : & \texttt{int} \times \texttt{int} \xrightarrow{\texttt{A}} \texttt{int} \\ (=) & : & \forall \beta.\beta \times \beta \xrightarrow{\texttt{A}} \texttt{bool} \\ \texttt{if} & : & \forall \beta.\texttt{bool} \times \beta \times \beta \xrightarrow{\texttt{A}} \beta \\ \texttt{pre}\,. & : & \forall \beta.\beta \xrightarrow{\texttt{D}} \beta \\ .\,\texttt{fby}\,. & : & \forall \beta.\beta \times \beta \xrightarrow{\texttt{D}} \beta \\ \texttt{up}(.) & : & \texttt{float} \xrightarrow{\texttt{C}} \texttt{zero} \end{array}$$

The integer addition operator maps a pair of integers to an integer and can be used in any context. Other arithmetic and logical operators have the same kind and similar types, only varying in whether the arguments and results are of type int, float, or bool. The synchronous primitives for uninitialized delay (pre $\cdot$), initialized delay ($\cdot$ fby $\cdot$), and initialization ($\cdot$ -> $\cdot$) are of kind D since they contain internal discrete state variables. Their type signatures are *polymorphic* [42], [43]: each involves the quantified type variable β since they can be applied to values of any primitive type. The operator for zero-crossing detection (up($\cdot$)) has kind C since it can only be used on a continuous clock. We introduce the type zero to represent zero-crossing events.

The less-than-or-equal-to operator (<=), like the other relational operators, takes two arguments of the same primitive type and returns a boolean value. Perhaps surprisingly, such operators have kind D even though they have no internal state. This ensures that boolean signals never change during integration, and thus that the active branch of if and other branching constructs never change during integration.

The complete set of typing rules is presented elsewhere [16, §3.2] and we only describe a few typical rules here. As a first example, consider the rule for ODEs with no reset:

$$\text{(ODE)}\quad \frac{G, H \vdash_{\texttt{C}} e_1 : \texttt{float} \qquad G, H \vdash_{\texttt{D}} e_2 : \texttt{float}}{G, H \vdash_{\texttt{C}} \texttt{der}\, x = e_1\, \texttt{init}\, e_2 : [x : \texttt{float}]}$$

The typing assumptions are given to the left of the '$\vdash$' symbol. They comprise the global environment G described above, and a local environment H mapping variables to their types. The '$\vdash$' is annotated with the context in which a rule applies. The predicate over equations below the line is conditional on the two predicates over expressions above the line. All together, the rule says that an equation $\texttt{der}\, x = e_1\, \texttt{init}\, e_2$ is well typed when defined on a continuous clock if (1) the derivative expression e_1 yields a floating-point value on a continuous clock, (2) the initialization expression e_2 yields a floating-point value on a discrete clock. In ZÉLUS, the general form of an ODE with reset handler is written:

$$\texttt{der}\, x = e\, \texttt{init}\, ei\, \texttt{reset}\, z_0 \texttt{ -> } e_0 \mid \ldots z_n \texttt{ -> } e_n$$

The handler is a (possibly empty) list of pairs z_i -> e_i where z_i is an event and e_i an expression which gives the initial value of x when z_i is true. E.g.:

```
der x = 1.0 init 0.0
        reset z1 -> 1.0 | z2 -> 2.0
```

The signal x has derivative 1.0. It is reset to value 1.0 when z1 is true and to value 2.0 when z2 is true. The typing rule is extended to account for the reset handler:

$$\text{(ODE-RESET)}\quad \frac{\begin{array}{cc} G, H \vdash_{\texttt{C}} e : \texttt{float} & G, H \vdash_{\texttt{D}} e : \texttt{float} \\ \forall i \in I. G, H \vdash_{\texttt{C}} z_i : \texttt{zero} & G, H \vdash_{\texttt{D}} e_i : \texttt{float} \end{array}}{G, H \vdash_{\texttt{C}} \texttt{der}\, x = e\, \texttt{init}\, ei\, \texttt{reset}\, z_i \texttt{ -> } e_i \mid_{i \in I} : [x : \texttt{float}]}$$

Compared to the previous definition, the event expressions z_i must be of type zero which is the type for events. The expressions e_i must be of type float and are evaluated only when the event z_i occurs, thus at a discrete-time instant, hence the kind D. The use of a discrete context for initialization expressions is coherent with the principles laid out above, since the language ensures that all variable initializations and reinitializations occur at precise instants that are aligned with zero-crossings. In any case, variables cannot be directly assigned during numerical integration. The typing rules for equations successively construct a new local environment that is later required to be consistent with the assumed one (H). The der construct can only be used on a continuous clock since no rules are given for combinatorial or discrete clocks.

As a second example, consider the following rule for the parallel composition of sets of equations:

$$\text{(AND)}\quad \frac{G, H \vdash_k E_1 : H_1 \qquad G, H \vdash_k E_2 : H_2}{G, H \vdash_k E_1\, \texttt{and}\, E_2 : H_1 + H_2}$$

This rule applies for any context k but requires that the sets of equations being composed are well typed in the same context. In fact, it is this rule that is violated in the first ZÉLUS reimplementation of Figure 1a. The resulting environment, $H_1 + H_2$, is the (disjoint) composition of H_1 and H_2.

Applying type systems as sketched in this section gives a precise and unambiguous specification of the set of hybrid models that are valid in a given language (subject to other analyses). The rules permit formal reasoning and guide the implementation of the compiler typing pass.

Kinds also play an important role during compilation, as will be seen in Section VI. An expression has kind `D` when it must be compiled with an internal discrete state and activated on a *discrete* clock, `C` when it must be compiled for activation by a solver on a *continuous* clock, and `A` when it is combinatorial and can be activated on any clock.

Finally, models that have passed the type checking and are causally correct verify the following safety properties:

Theorem 1 ([18], [23]).

1) *Discrete state variables do not change between successive events;*
2) *If, in addition, no primitive statement hides a discontinuity, then all variables have continuous trajectories between two successive events;*
3) *Under this additional assumption, signal expressions labeled discrete and continuous are updated at disjoint instants, respectively.*

The Newton's Cradle example of Figure 15 is shown to be correctly typed in the sense of this section.

C. Static Typing: Hierarchical Automata

It turns out that the very same type system just naturally supports hierarchical automata describing nested multiple modes [17]. Consider for example, a function `sum` that computes the stream of natural numbers and `bounce` which models a bouncing ball.

```
let g = 9.81

let node sum() = o where rec o = (0 fby o) + 1

let hybrid bounce(yi, yi') = (y, y', z) where
  rec der y' = -. g init yi'
                reset z -> -0.8 *. last y'
  and der y = y' init yi
  and z = up(-. y)

let hybrid count_bounces_and_stop
       (epsilon, yi, yi') = (y, s) where
  rec automaton
      | Bounce ->
         (* z and y' are local to the state *)
         local z, y' in
         do (y, y', z) = bounce(yi, yi')
         and s = present z -> sum() init 0
         until z on (y' < epsilon) then Stop
      | Stop ->
         (* s is unchanged, i.e., *)
         (* implicitly, s = last s *)
         do y = 0.0 done
      end
```

Figure 17. The program `count_bounces_and_stop`.

The function `count_bounces_and_stop` of Figure 17 is defined by a two state automaton: in the initial state `Bounce`, it counts the number of bounces until `z` on ($\mathtt{y}' < \mathtt{epsilon}$), that is, when there is both a contact and the speed of the ball is less than `epsilon`. When this event occurs, the next state is `Stop` in which the position `y` remains constant at 0.0. The absence of an explicit definition for `s` in this state means that its value remains constant. This program is valid because the discrete-time function `sum` in equation `present z` $->$ `sum() init 0` is computed at zero-crossing instants and defines a (piecewise constant) hyperstream. Removing the `present` construct would result in a typing error based on the rule: when declaring a function of kind k, for example, `node` or `hybrid`, all computations must be of the same kind.

The typing rules for automata [17] follow the same pattern described above, but their formalization is technically more involved since automata themselves are syntactically more complicated than basic equations. It turns out that six rules suffice to treat both discrete and hybrid automata. These rules precisely encode the sort of informally stated guidelines typically given in the documentation of hybrid modelers. The Stateflow User's Manual, for instance, includes a section entitled "Design Considerations for Continuous-Time Modeling in Stateflow Charts" [44, 19-22], the goal of which is to ensure that a model does not depend critically on "side effects" of the numerical simulation algorithm. Informal rules like "Compute derivatives only in during actions" and "Update local data only in transition, entry, and exit actions" are readily and precisely encoded in terms of syntax-based `A`/`D`/`C` rules.

VI. The Simulation Loop

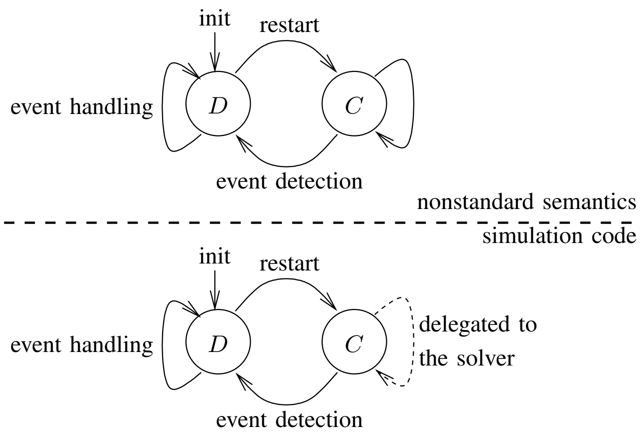


Figure 18. From the nonstandard semantics (top) to the actual simulation code (bottom).

Now, for a given ZÉLUS model, assume we have at hand (1) its nonstandard semantics and associated causality analysis and (2) its `D`/`C` (discrete/continuous) static typing. Assume also that the model is correct, both causally and for continuous/discrete typing. Since the model is causally correct, we can statically schedule all the computation actions involved in the step function of this model. By statement 3 of Theorem 1, expressions typed as `D` and `C` must be evaluated at disjoint instants. We can thus collect them, respectively, in the `D` and `C` states of a two-state automaton. This automaton is depicted at the top of Fig. 18. It is discrete time and the time step is infinitesimal.

We know that the actual system satisfies the two properties 1 and 2 of Theorem 1. This justifies the architecture of the actual simulation code shown at the bottom of Fig. 18. The structure is the same as for the semantics except for two changes which correspond to performing standardization:

- The nonstandard discrete-time self-loop in mode C at the top of Fig. 18 is replaced by a dashed transition depicting the delegation of this self-loop to a numerical solver to solve the ODEs and detect zero-crossings. The latter operates until the next event is detected.
- The nonstandard discrete-time self-loop in mode D at the top of Fig. 18 is mapped to the same self-loop with identical scheduling, albeit acting in the $\mathbb{N}$-component of superdense time.The scheduling of the reset or restart actions at instants of mode change comes directly from the scheduling of the nonstandard semantics.

By doing so, the invocation of a solver implementing some numerical approximation scheme occurs only at the last step of compilation. Besides the very final code generation phase, compilation steps are unpolluted by approximations. Details regarding this section are found in [19].

A. Principles

We now describe the practical architecture of the simulation loop of ZÉLUS. The first choice to make in implementing a hybrid system is how to solve ODEs. Creating an efficient and numerically accurate numerical solver is a daunting and specialist task. Reusing an existing solver is more practical, with two possible choices: either (a) generate a *Functional Mock-Up Unit* (FMU) using the standardized *Functional Mock-Up Interface* (FMI) and rely on an existing simulation infrastructure [45], [46]; or (b) use an off-the-shelf numerical solver like CVODE [20] and program the main simulation loop. The latter corresponds to the *co-simulation* variant (CS) of FMI [47], where each FMU embeds its own solver.

The simulation loop of a hybrid system is the same no matter which option is chosen. It can be defined formally as a *synchronous function* that defines four streams $t(n)$, $lx(n)$, $y(n)$, and $z(n)$, where: $n \in \mathbb{N}$; $t(n) \in \mathbb{R}$ is the increasing sequence of instants at which the solver stops (the major steps of Simulink); $lx(n)$ is the value at time $t(n)$ of the *continuous state variables*, that is, of all variables defined by their derivatives in the original model; $y(n)$ is the value at time $t(n)$ of the *discrete state*; and $z(n)$ indicates any *zero-crossings* at instant $t(n)$ on signals monitored by the solver, that is, any signals that become equal to or pass through zero.

The synchronous function has two modes: the discrete mode D contains all computations that may change the discrete state or that have side effects. The continuous mode C is where ODEs are solved. The two modes alternate according to the execution scheme summarized in Figure 18.

The Continuous Mode C: In this mode, the solver computes an approximation of the solution of the ODEs and monitors a set of expressions for zero-crossings. Code generation is independent of the actual solver implementation. We abstract it by introducing a function $solve(f)(g)$ parameterized by f and g where:

- $x'(\tau) = f(y(n), \tau, x(\tau))$ defines the derivatives of continuous state variables x at instant $\tau \in \mathbb{R}$;
- $upz(\tau) = g(y(n), \tau, x(\tau))$ defines the current values of a set of zero-crossing signals upz, indexed by $i \in \{1, \dots, k\}$.

The continuous mode C computes

$$(lx, z, t, s)(n+1) = solve(f)(g)(s, y, lx, t, step)(n)$$

where:

$s(n)$	is the internal state of the solver at instant $t(n) \in \mathbb{R}$. Calling $solve(f)(g)$ updates the state to $s(n+1)$;
x	is an approximation of a solution of the ODE, $x'(\tau) = f(y(n), \tau, x(\tau)) \quad , \quad t(n) \leq \tau < t(n+1)$ It is parameterized by the current discrete state $y(n)$ and initialized at instant $t(n)$ with the value of $lx(n)$, that is, $x(t(n)) = lx(n)$;
$lx(n{+}1)$	is the value of x at $t(n+1)$, that is: $lx(n+1) = x(t(n+1))$ lx is a discrete-time signal updated at the instants $t(n)$ whereas x is a continuous-time signal;
$t(n+1)$	is bounded by the horizon $t(n) + step(n)$ that the solver has been asked to reach, that is: $t(n) \leq t(n+1) \leq t(n) + step(n)$
$z(n+1)$	signals any zero-crossings detected at time $t(n+1)$. An event occurs with a transition to the discrete mode D when horizon $t(n)+step(n)$ is reached, or when at least one of the zero-crossing signals $upz(i)$, for $i \in \{1, \dots, k\}$ crosses zero,[21] which is indicated by a true value for the corresponding boolean output $z(n+1)(i)$: $event = z(n+1)(0) \vee \dots \vee z(n+1)(k) \vee (t(n+1) = t(n) + step(n))$

If the solver raises an error, e.g., division by zero or failure to converge, we consider that the simulation fails.

The Discrete Mode D: All discrete changes occur in this mode. It is entered when an event is raised during integration. During a discrete phase, the function $next$ defines y, lx, $step$, $encore$, z, and t:

$$\begin{aligned}(y, lx, step, encore)(n+1) &= next(y, lx, z, t)(n)\\ z(n+1) &= false\\ t(n+1) &= t(n)\end{aligned}$$

[21]The function $solve(f)(g)$ abstracts from the actual implementation of zero-crossing detection. To account for a possible zero-crossing at the horizon $t(n) + step(n)$, the solver may integrate over a strictly larger interval $[t(n), t(n) + step(n) + margin]$, where $margin$ is a solver parameter.

$$z(n+1)(i) = \left\{\begin{array}{l}(\forall \tau \in [t(n), t(n+1)[\ .\ upz(\tau)(i) < 0)\\ \wedge\ \exists m \leq margin\ .\\ \quad (\forall \tau \in [t(n+1), t(n+1)+m]\ .\ upz(\tau)(i) \geq 0)\end{array}\right.$$

This definition assumes that the solver also stops whenever a zero-crossing expression passes through zero from positive to negative.

where

$y(n+1)$	is the new discrete state; outside of mode D, $y(n+1) = y(n)$;
$lx(n+1)$	is the new continuous state, which may be changed directly in the discrete mode;
$step(n+1)$	is the new step size;
$encore(n+1)$	is true if an additional discrete step must be performed. Function $next$ can decide to trigger instantaneously another discrete event causing an *event cascade* [10];
$t(n)$	the simulation time, is frozen in discrete phases.

The initial values for $y(0)$, $lx(0)$ and $s(0)$ are given by an initialization function *init*. Finally, $solve(f)(g)$ may decide to reset its internal state if the continuous state changes. If $init\_solve(lx(n), s(n))$ initializes the solver state, we have:

$$\begin{aligned} reinit &= (lx(n+1) \neq lx(n)) \\ s(n+1) &= if\ \ reinit\ \ then\ \ init\_solve(lx(n+1), s(n)) \\ &\quad\ \ else\ \ s(n) \end{aligned}$$

Writing $solve(f)(g)$ abstracts from the actual choice of integration method and zero-crossing detection algorithm. A more detailed description of $solve(f)(g)$ would be possible — e.g., an automaton with two states: one that integrates, and one that detects zero-crossings — but with no influence on the code generation problem which must be independent of such simulation details.

Given a program written in a high-level language, we must produce the functions $init$, f, g, and $next$. In practice, they are implemented in an imperative language like C. Code generation for hybrid models has thus much in common with code generation for synchronous languages.

B. *Implementation*

The operation of the continuous mode is fairly standard. In our implementation, we used the SUNDIALS CVODE solver [20], [48], which imposes particular but typical constraints on the runtime and generated code. We also implemented the Bogacki-Shampine and Dormand-Prince explicit Runge-Kutta schemes [49] using Butcher tableau, and they act as “drop in” replacements for the four main solver functions (for creation, reinitialization, stepping, and interpolation).

The transition into the continuous mode must reinitialize the solver if the continuous state values were changed directly, or if the dynamics that govern them were modified by a change to the discrete state. Otherwise, the solver may reuse the previous interpolant or calculated state value to continue. Such reinitializations are best avoided, however, since they reduce the efficiency of the solver and since they are not needed if an event only triggered an update of a visualization or log file.

VII. THE COMPILATION SUITE

In this section we present a brief description of our two experimental developments based on the principles presented in this article: ZÉLUS and its industrial sister SCADE Hybrid. Technical details can be found elsewhere [19].

A. ZÉLUS

The compiler architecture for hybrid programs is based on those of existing compilers for data-flow synchronous languages like SCADE 6 and Lucid Synchrone, as described for instance in [50]. After initial checks, it consists in successive rewritings of the source program into intermediate languages, and ending with sequential code in a target language, typically C. The different passes are shown in Figure 19:

1) Parsing transforms code in the source language into an abstract syntax tree;
2) typing checks programs according to the system of [16]. In the language extended with ODEs, this system distinguishes continuous and discrete blocks to ensure the correct separation of continuous and discrete behaviors;
3) causality analysis verifies the absence of causality loops [18];
4) control structures like hierarchical state machines are rewritten into data-flow equations, following the method presented in [51]. A small modification accounts for the fact that transitions are executed in a discrete context whereas the bodies of states are continuous;
5) traditional optimizations, like dead-code removal and common sub-expression elimination, are performed, after which the generation of sequential code begins;
6) scheduling orders equations based on data dependencies;
7) programs are translated into an intermediate sequential object language (named SOL in the case of SCADE) [50]. This language has been extended to to deal with the new constructs for ODEs and zero-crossings;
8) slicing specializes the sequential function generated for each node into three functions: f, which defines the derivatives, g, which defines the zero-crossing signals, and $next$, which defines the function that implements discontinuous changes.
9) dead-code removal eliminates useless code from each function. For instance, derivatives need not be computed by the $next$ function and zero-crossing values are always false during integration;
10) finally, the sequential code is translated to imperative code. For ZÉLUS, the target language is OCaml. For SCADE HYBRID, the target language is C.

The compiler passes in gray in Figure 19 are those which must be modified in, or added to (dashed borders), a traditional synchronous language compiler. For SCADE HYBRID, the modifications were relatively minor w.r.t SCADE — around 10% of each pass—and do not require major changes to the existing architecture. Together with the new passes, they amount to 5% of the total code size of the compiler. ZÉLUS, on the contrary, was written from scratch but has the same basic structure.

B. SCADE 6

From SCADE 6 *to* SCADE HYBRID*:* SCADE Suite is an integrated design environment for critical applications including model-based design, simulation, verification and qualifiable/certified code generation. SCADE Suite has been used for more than twenty years to design critical software, such

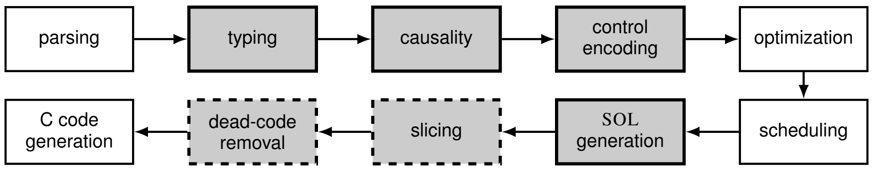


Figure 19. Compiler architecture (modified passes are gray; new ones are also dashed)

as flight control and engine control systems, automatic pilots, power and fuel management systems, rail interlocking systems and signaling, emergency braking systems, overspeed protection, nuclear power plant controls, ADAS in cars, and many other aerospace, railway, energy, automotive, and industrial applications.

Applications are implemented using the formally defined SCADE 6 language which is a synchronous data-flow language combining dataflow constructs as in LUSTRE with control-flow structures like hierarchical state machines. The SCADE Suite KCG code generator generates C or Ada code from a SCADE model. The code generation and the sanity checks are qualified/certified for various safety standards (DO-178C/DO-330 at TQL-1, IEC 61508 at SIL 3, EN 50128 at SIL 3/4, and ISO 26262 software up to ASIL D). In a qualification context, SCADE models replace detailed textual requirements. KCG certification allows eliminating verification activities at code level and unit testing: the generated code is guaranteed to implement the behavior specified by the input model.

SCADE HYBRID is an extension of SCADE 6 supporting hybrid systems combining continuous- and discrete-time dynamics. It is based on the ideas presented in this paper. The KCG Hybrid code generator is based on KCG, and reuses as much as possible of the existing (discrete) SCADE 6 language and compiler architecture The goal was to validate the approach — designing SCADE HYBRID as a minimal extension of SCADE— on an industrial compiler and at a bigger scale (around 50kloc of OCaml), and with additional constraints, including full traceability from the input model to the generated code.

The conclusions were positive: in KCG Hybrid, the fraction of the source code that has been modified or added with reference to KCG, is approximately 5%. The architecture of the code generator remains the same and the static checks were readily extended to the hybrid context.

Description: KCG Hybrid generates C code with bounded memory. In particular, the discrete parts of a model are compiled as in KCG. This guarantees that the code used for simulation is the same as that executed on target platforms.

The code generated by KCG Hybrid can be paired with a runtime to target one of the following backends:

- a model-exchange "Functional Mockup Unit" (FMU) that respects the *FMI for Model Exchange 1.0* [45] *or 2.0* [46] *standards*. Such FMUs describe a mix of ODEs and discrete events. They are simulated, with or without other components, by an external numerical solver provided by a host. The execution model of FMI [45, Section 2.9] resembles the scheme described in the previous section. A small generic runtime implements the FMI API by calling KCG generated code.
- a co-simulation FMU that respects the *FMI for Co-Simulation 1.0* [47] *or 2.0* [46] *standard*. Such FMU embeds its own solver, which is CVODE in our case.
- a standalone executable.

The code generated by KCG is independent of the backend. It is paired with a target-specific runtime. The traceability information generated by KCG Hybrid is used to generate model-specific "glue" code and XML descriptions for FMUs.

It is also possible to import FMUs (version 1.0 or 2.0) into a SCADE Hybrid model:

- a model-exchange FMU is considered as an imported `hybrid` operator. Note that importing a model-exchange FMU generated from another SCADE Hybrid model is equivalent to calling that operator directly.
- a co-simulation FMU is considered as an imported `node`.

This allows SCADE Hybrid to be used as a tool to orchestrate FMUs, where the behavior of each FMU and their combination is well defined.

Scade Hybrid in a certified context: Although the qualification of KCG allows to reduce activities on the generated code, it is still necessary to perform verification and validation activities at the specification level.To perform these activities, one needs to be able to provide a test environment which is powerful enough to describe the environment but also deterministic to be able to replay the test sessions with the same results. This is very important in the certification domain as the applicant must be able to present evidence of all activities performed during the development process.

Being able to mix continuous and discrete parts with deterministic semantics and reproducible simulations is a marked improvement on current practice. By extending SCADE with continuous capabilities it is possible to perform tests and simulations that combine continuous and discrete parts, with a well-defined semantics for their interactions. This guarantees the correctness of the simulation and expected behaviors can be assessed in advance. The goal of SCADE Hybrid is to increase the level of confidence in such mixed models.

Furthermore, the increasing need for on-line monitoring and failure detection and isolation calls for including, in the monitoring software, parts of the physical system model. That model is used to generate measurement predictions, under the hypothesis that the system behaves as the model predicts. These predictions can be compared to actual measurements for failure detection and isolation. Having a hybrid system modeling tool with a well defined mathematical semantics and integrated with a certified embedded code generator is a first step towards embedded code involving physical models.

VIII. Discussion, Perspectives, and Conclusion

In this concluding section we first discuss issues, limitations, extensions, and perspectives, directly related to this work. In a second part, we widen the perspective by discussing other issues not addressed here, but important for modeling languages and tools for Cyber-Physical System (CPS).

A. *Our work: modeling is programming*

In this work, we consider hybrid systems from a programming language perspective, thus following the approach advocated by Lee and Zheng [4].

We aimed to make ZÉLUS as simple as LUSTRE and a conservative extension of it. We reused and adapted synchronous language principles and techniques as much as possible and followed the same philosophy of rejecting unsafe models at compile time. In a sense, ZÉLUS is equipped with the weakest modeling discipline guaranteeing that accepted models are safe.

In order to achieve this, our modeling discipline had to be formally sound. This required the development of an ideal modular semantics as a reference for the design of static analyses and compilation steps, but orthogonal to numerical aspects related to ODE discretization schemes. The semantics had to support both the continuous-time dynamics within different modes, and the reset or restart actions performed at events of mode change. We found nonstandard analysis to be instrumental for defining a mathematical and modular semantics of hybrid models. Its aim is not to serve as an operational semantics providing reference executions but rather as a support for the analyses that identify unsafe models and statically reject them, for proving certain invariants for accepted models, and for providing adequate execution schemes for the reset or restart actions performed at events of mode change.

All of this is a reuse of techniques developed for synchronous languages and adapted to the nonstandard semantics. Not everything can be inherited, however. There are two specific issues, namely the clean separation between discrete- and continuous-time dynamics, and the run-time architecture in which the execution of continuous modes is delegated to an off-the-shelf ODE solver. The background from synchronous languages still provided good guidance for how to proceed with these novel issues.

For ZÉLUS, our heritage is still limited to LUSTRE and some programming constructs and compilation techniques borrowed from LUCID SYNCHRONE [52]. More elaborated semantic studies were developed for the synchronous languages ESTEREL and SIGNAL, with an emphasis on so called *constructive semantics* [53]. Because ZÉLUS extends LUSTRE, no sophisticated causality analysis as in ESTEREL [54], [55] appeared necessary to write complicated models. Neither we really needed the SIGNAL "clock calculus" [56]. Regarding the discrete/continuous typing system, types are attached to functions and expressions. We could have considered instead a type system of finer granularity for signals. Our design choice favored simplicity. So far, our type system seems accurate enough for not rejecting too many practically meaningful programs. Several other analyses — either static, dynamic, or a combination of both — would be useful. One is the analysis that detects situations where cascades of events are bounded; another is to ensure the absence of critical races between events.

Other tools and languages, like PTOLEMY [27], have adopted a different approach where checks on models are performed at run time. It is possible to dynamically check that a model has no instantaneous loops or that a signal expected to be continuous does no jumps during integration (up to some threshold). We have taken the opposite standpoint by favoring the detection of unsafe models at compile time. The consequence is that we do reject good models because the type systems we developed are not expressive enough. An experimental study, for example writting a comprehensive library of discrete/continuous/hybrid blocks, will help deciding whether the type systems are overly constraining or not. A preliminary experiment is reported in [57]. Finally, the discovery of numerical difficulties related to stiffness remains run time—and rules out the need for overly restrictive programming disciplines in industrial contexts. Performing rich analyses at compile time, while constraining the users, may detect errors in models early; it also allows form removing run-time checks and to statically schedule the computation of the step function and the reset actions, which leads to more efficient code.

B. *A wider perspective*

Besides the observations directly related to the presented material, there are other issues that are important for CPS modeling tools in general.

Using nonstandard analysis to prove properties: Besides providing the semantic foundations for compilation, the nonstandard semantics also supports *symbolic manipulations* and possibly the formal proofs of properties on models. This aspect has not yet been considered.

Surviving too many events: In ZÉLUS, and all other existing modeling languages for hybrid systems, runs alternate continuous-time phases and events of mode change, possibly in (finite) cascades. There are, however, situations in which events become too frequent. A typical example is the bouncing ball in which jumps occur more and more frequently, thus exhibiting a Zeno behavior—the bouncing ball is, for this reason, considered a difficult benchmark for hybrid systems modeling tools. Many other examples exist. An ideal diode with no leakage is modeled by the equations $i \geq 0$, $v \geq 0$, and $iv = 0$. In a context where the status of the diode oscillates between passing or not passing, events become too frequent for the solver to stop at each of them. The same issue arises in multibody mechanics with contact, where the modeling of contact events is in itself an issue. For all these cases, zero-crossing events are not appropriate. So-called *nonsmooth systems solvers* [58] are to be preferred, as they use *time-stepping* discretization schemes that do not stop at events. Nonsmooth systems solvers are particularly effective for multibody mechanics or electronic circuits with ideal semiconductors [59], where they prove to behave much better than classical schemes. The class of discontinuities they handle are the *complementarity conditions*, of which the above perfect diode model is the simplest instance. Today, nonsmooth system solvers are available through dedicated tools like Siconos,[22] and cannot easily be called from existing modeling languages. Offering such features is certainly a useful objective. As these special solvers are good in particular situations but not everywhere, they cannot replace mainstream solvers but rather complement them. The best way of achieving such combinations remains an issue.

Dealing with large sparse models: Huge CPS models such as smart grids or smart buildings are typically sparse. This means that they are composed of a great number of local subsystems interacting only locally.[23] Sparsity is a *structural* property, related to the topology of the system block-diagram. Being symbolic in nature, sparsity should be preferably handled at compile time. Existing hybrid systems modeling tools do not take advantage of the model's sparsity. A striking illustration is that discretization time steps are adapted in time but *uniform in space*. For example, opening a window in a single room in the thermodynamical model of a large building will impact the simulation in the whole building, whereas the physical effect is clearly local. It is thus tempting to investigate alternative discretization methods such as *Quantized State Solvers* (QSS), in which variables are quantized in their value domain, and time progresses locally for each variable until the current value quantum must be exited [60], [61], [62], [63]. QSS solvers implement discrete event approximations of continuous-time systems. Alternative approaches exist such as *multi-rate simulation* [63] in which multiple variable time steps are managed locally in space. More generally, advances made in the High Performance Computing community regarding solvers for very large sparse ODE systems [62] remain to be applied in the context of CPS.

From ODEs to DAEs: In this work we restricted ourselves to ODE-based modeling of hybrid systems, commonly specified by means of dataflow block-diagrams. Block diagrams, however, suffer from limits in expressivity as is illustrated in Figure 20. This figure shows two models of the same

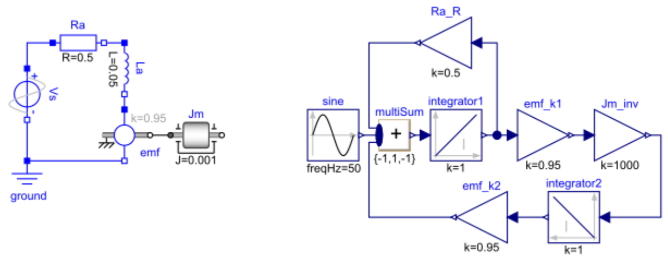


Figure 20. DAE (left) vs. ODE (right) based modeling.

system comprising a simple model of an electrical motor and of the rotational inertia of the motor. On the left-hand side a Modelica component diagram of the system is shown that connects physical components through non-directed interactions resulting from the first principles of physics. On the right-hand side, the same model is shown as a block diagram in which input/output oriented blocks are connected with a directed wiring manually specified by the designer. Adding one more physical components is straightforward in the first diagram, whereas it may necessitate a complete redesign of the second one. Physical component diagrams are mathematically described by DAEs of the form $f(x', x, u) = 0$, where u and x are vectors of variables, and f is a vector of functions. Note that there is no notion of input versus output. Multi-mode DAE-based models constitute the basis for component-oriented languages for physical modeling like the open standards Modelica[24] and VHDL-AMS[25] or proprietary languages like Simscape.[26] DAE-based modeling is a considerable progress but it raises a number of difficulties, related to both the mathematics of DAEs and multi-mode DAEs, the algorithms for compilation of such models, and the underlying languages, particularly for the definition and handling of events. An extensive literature can be found at https://www.modelica.org/publications.

Nonstandard semantics can be used to give a precise semantics for multi-mode DAEs and particularly their mode changes [64]. The extension to DAE-based modeling of the causality analysis performed in ZÉLUS, however, is drastically different and definitely more involved—it is known as *structural analysis* in the context of DAE-based modeling languages. Structural analysis involves the notion of *differentiation index*, irrelevant in our context. The reader interested in the reuse of the ideas from synchronous languages in this subject is referred to [65], [64].

REFERENCES

[1] L. Carloni, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli, "Languages and tools for hybrid systems design," *Foundations and Trends in Electronic Design Automation*, vol. 1, no. 1/2, 2006.

[2] K. J. Åström, "Modeling and Simulation: from Physics to Software," March 19 2014, invited talk at Collège de France. [Online]. Available: http://www.college-de-france.fr/site/gerard-berry/seminar-2014-03-19-17h00.htm

[3] G. Berry, "Real Time programming: Special purpose or general purpose languages," in *IFIP Congress*. Elsevier Science Publishers, 1989, pp. 11–17.

[4] E. A. Lee and H. Zheng, "Operational semantics of hybrid systems," in *Hybrid Systems: Computation and Control (HSCC)*, vol. 3414. Zurich, Switzerland: LNCS, March, 9-11 2005.

[5] L. Ljung, *System Identification — theory for the user*, 2nd ed. Upper Saddle River, NJ: PTR Prentice Hall, 1999.

[6] O. Bouissou and A. Chapoutot, "An operational semantics for simulink's simulation engine," in *LCTES*, 2012, pp. 129–138.

[7] P. J. Mosterman, J. Zander, G. Hamon, and B. Denckla, "Towards Computational Hybrid System Semantics for Time-Based Block Diagrams," in *IFAC Proceedings volume, Volume 42, Issue 17*, 2009, pp. 376–385. [Online]. Available: http://dx.doi.org/10.3182/20090916-3-ES-3003.00065

[8] E. A. Lee and H. Zheng, "Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems," in *EMSOFT*, Salzburg, Austria, September 30-October 3 2007.

[9] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis, "Translating Discrete-Time Simulink to Lustre," *ACM Transactions on Embedded Computing Systems*, 2005, special Issue on Embedded Software.

[10] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "Nonstandard semantics of hybrid systems modelers," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 877–910, 2012.

[22] http://siconos.gforge.inria.fr/4.1.0/html/index.html

[23] We do not consider here the issue of dynamic instantiation and removal of subsystems, which is the core difficulty of Systems of Systems.

[24] https://www.modelica.org/documents/ModelicaSpec33.pdf

[25] https://standards.ieee.org/findstds/standard/1076.1-2007.html

[26] https://mathworks.com/products/simscape/

[11] F. M. Company, “Structured analysis and design using Matlab/Simulink/Stateflow: Modelling style guidelines, version 2.4.2,” 1999.

[12] MathWorks Automotive Advisory Board (MAAB), “Controller style guidelines for production intent using MATLAB, Simulink and Stateflow,” Apr. 2001.

[13] D. Buck and A. Rau, “On modelling guidelines: Flowchart patterns for STATEFLOW,” *Gesellschaft für Informatik, FG 2.1.1: Softwaretechnik Trends*, vol. 21, no. 2, 2001.

[14] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, “The synchronous languages 12 years later,” *Proceedings of the IEEE*, vol. 91, no. 1, Jan. 2003.

[15] J.-L. Colaco, B. Pagano, and M. Pouzet, “Scade 6: A Formal Language for Embedded Critical Software Development,” in *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

[16] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, “Divide and recycle: types and compilation for a hybrid synchronous language,” in *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Chicago, USA, Apr. 2011, pp. 61–70.

[17] ——, “A hybrid synchronous language with hierarchical automata: static typing and translation to synchronous code,” in *EMSOFT*, S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds. ACM, 2011, pp. 137–148.

[18] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet, “A type-based analysis of causality loops in hybrid systems modelers,” in *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC’14, Berlin, Germany, April 15-17, 2014*, M. Fränzle and J. Lygeros, Eds. ACM, 2014, pp. 71–82. [Online]. Available: http://doi.acm.org/10.1145/2562059.2562125

[19] T. Bourke, J. Colaço, B. Pagano, C. Pasteur, and M. Pouzet, “A synchronous-based code generator for explicit hybrid systems languages,” in *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, B. Franke, Ed., vol. 9031. Springer, 2015, pp. 69–88. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46663-6_4

[20] A. Hindmarsh, P. Brown, K. Grant, S. Lee, R. Serban, D. Shumaker, and C. Woodward, “SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers,” *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 363–396, Sep. 2005.

[21] T. Bourke and M. Pouzet, “Zélus: A synchronous language with ODEs,” in *Hybrid Systems: Computation and Control (HSCC)*. Philadelphia, USA: ACM, Apr. 2013, pp. 113–118.

[22] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, *Heterogeneous Concurrent Modeling and Design in Java*, Memorandum UCB/ERL M04/27, EECS, University of California, Berkeley, CA USA 94720, July 2004.

[23] A. Benveniste, T. Bourke, B. Caillaud, B. Pagano, and M. Pouzet, “A Type-based Analysis of Causality Loops in Hybrid Systems Modelers,” *Nonlinear Analysis: Hybrid Systems*, vol. 26, pp. 168–189, Nov. 2017. [Online]. Available: https://hal.inria.fr/hal-01549183

[24] E. A. Lee and A. Sangiovanni-Vincentelli, “A framework for comparing models of computation,” *IEEE Transactions on CAD*, vol. 17, no. 12, December 1998.

[25] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous dataflow programming language LUSTRE,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[26] N. Halbwachs and P. Raymond, “A tutorial of Lustre,” 2002, http://www-verimag.imag.fr/Publications-Synchrones.html.

[27] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[28] J. Liu and E. A. Lee, “On the causality of mixed-signal and hybrid models,” in *Hybrid Systems: Computation and Control, 6th International Workshop, HSCC 2003 Prague, Czech Republic, April 3-5, 2003, Proceedings*, ser. Lecture Notes in Computer Science, O. Maler and A. Pnueli, Eds., vol. 2623. Springer, 2003, pp. 328–342. [Online]. Available: https://doi.org/10.1007/3-540-36580-X_25

[29] *Simulink: Developing S-functions*, The MathWorks, Inc., Natick, MA, USA, Mar. 2017.

[30] P. J. Mosterman, J. Zander, G. Hamon, and B. Denckla, “A computational model of time for stiff hybrid systems applied to control synthesis,” *Control Engineering Practice*, vol. 20, no. 1, pp. 2–13, Jan. 2012.

[31] P. Mosterman, J. Zander, G. Hamon, and B. Denckla, “Towards computational hybrid system semantics for time-based block diagrams,” in *3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS’09)*, Spain, Sep. 2009, pp. 376–385, keynote paper.

[32] E. A. Lee, “Constructive models of discrete and continuous physical phenomena,” *IEEE Access*, vol. 2, pp. 797–821, 2014. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2014.2345759

[33] O. Maler, Z. Manna, and A. Pnueli, “From timed to hybrid systems,” in *REX Workshop*, ser. Lecture Notes in Computer Science, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, Eds., vol. 600. Springer, 1991, pp. 447–484.

[34] G. Dahlquist and Å. Björck, *Numerical Methods in Scientific Computing: Volume 1*. SIAM, 2008.

[35] A. Robinson, *Nonstandard Analysis*. Princeton Landmarks in Mathematics, 1996, ISBN 0-691-04490-2.

[36] T. Lindstrøm, “An invitation to nonstandard analysis,” in *Nonstandard Analysis and its Applications*, N. Cutland, Ed. Cambridge Univ. Press, 1988, pp. 1–105.

[37] K. Suenaga, H. Sekine, and I. Hasuo, “Hyperstream processing systems: Nonstandard modeling of continuous-time signals,” in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’13. New York, NY, USA: ACM, 2013, pp. 417–430.

[38] *Simulink 7—User’s Guide*, 7th ed., The Mathworks, Natick, MA, U.S.A., Mar. 2010.

[39] P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis, “Actors without directors: A kahnian view of heterogeneous systems,” in *Hybrid Systems: Computation and Control, 12th International Conference, HSCC 2009, San Francisco, CA, USA, April 13-15, 2009. Proceedings*, ser. Lecture Notes in Computer Science, R. Majumdar and P. Tabuada, Eds., vol. 5469. Springer, 2009, pp. 46–60. [Online]. Available: https://doi.org/10.1007/978-3-642-00602-9_4

[40] E. Matsikoudis and E. A. Lee, “The fixed-point theory of strictly causal functions,” *Theor. Comput. Sci.*, vol. 574, pp. 39–77, 2015. [Online]. Available: https://doi.org/10.1016/j.tcs.2015.01.036

[41] J.-L. Colaço and M. Pouzet, “Type-based Initialization Analysis of a Synchronous Data-flow Language,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 3, pp. 245–255, August 2004. [Online]. Available: sttt04.pdf

[42] R. Hindley, “The principal type-scheme of an object in combinatory logic,” *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, Dec. 1969.

[43] R. Milner, “A theory of type polymorphism in programming,” *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, Dec. 1978.

[44] *Stateflow User’s Guide (R2017b)*, The Mathworks, Natick, MA, USA, Mar. 2017.

[45] MODELISAR, *Functional Mock-up Interface for Model Exchange v1.0*, 2010.

[46] ——, *Functional Mock-up Interface for Model Exchange and Co-Simulation v2.0*, 2010.

[47] ——, *Functional Mock-up Interface for Co-Simulation v1.0*, 2010.

[48] T. Bourke, J. Inoue, and M. Pouzet, “Sundials/ML: interfacing with numerical solvers,” in *ACM Workshop on ML*. Nara, Japan: ACM, Sep. 2016.

[49] L. Shampine, I. Gladwell, and T. S., *Solving ODEs with Matlab*. Cambridge University Press, 2003.

[50] D. Biernacki, J.-L. Colaco, G. Hamon, and M. Pouzet, “Clock-directed Modular Code Generation of Synchronous Data-flow Languages,” in *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008. [Online]. Available: lctes08a.pdf

[51] J.-L. Colaço, B. Pagano, and M. Pouzet, “A Conservative Extension of Synchronous Data-flow with State Machines,” in *ACM International Conference on Embedded Software (EMSOFT’05)*, Jersey city, New Jersey, USA, September 2005. [Online]. Available: emsoft05b.pdf

[52] M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI, April 2006.

[53] G. Berry, “The constructive semantics of pure Esterel,” 1999, draft book.

[54] T. R. Shiple and G. Berry, “Constructive analysis of cyclic circuits,” in *Proceedings of the International Design and Test Conference ITDC 96*, Paris, France, 1996.

[55] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer, 2007. [Online]. Available: https://doi.org/10.1007/978-0-387-70628-3

[56] T. Amagbegnon, L. Besnard, and P. L. Guernic., “Implementation of the data-flow synchronous language Signal,” in *Programming Languages Design and Implementation (PLDI)*. ACM, 1995, pp. 163–173.

[57] T. Bourke, F. Carcenac, J.-L. Colaço, B. Pagano, C. Pasteur, and M. Pouzet, "A Synchronous Look at the Simulink Standard Library," in *ACM International Conference on Embedded Software (EMSOFT)*, Seoul, October 15-20 2017.

[58] Acary Vincent and Brogliato Bernard, *Numerical Methods for Nonsmooth Dynamical Systems. Applications in Mechanics and Electronics*, ser. Lecture Notes in Applied and Computational Mechanics. Springer-Verlag, 2008, vol. 35.

[59] V. Acary, O. Bonnefon, and B. Brogliato, *Nonsmooth Modeling and Simulation for Switched Circuits*, ser. Lecture Notes in Electrical Engineering. Springer Verlag, Oct. 2010, vol. 69. [Online]. Available: https://hal.inria.fr/inria-00522358

[60] F. Cellier and E. Kofman, *Continuous System Simulation*. Springer-Verlag, 2006.

[61] J. Fernandez and E. Kofman, "A Stand-Alone Quantized State System Solver for Continuous System Simulation." *Simulation: Transactions of the Society for Modeling and Simulation International*, vol. 90, no. 7, pp. 782–799, 2014. [Online]. Available: files/qss_solver_v4.pdf

[62] F. Casella, "Simulation of large-scale models in modelica: State of the art and future perspectives," in *Proc. of the 11th Int. Modelica Conference*, H. Elmqvist and P. Fritzson, Eds. Versailles, France: Modelica Association, Sep. 2015.

[63] F. Bergero, A. Ranade, and F. Casella, "QSS and Multi-rate Simulation of Object-oriented Models," in *Proceedings of the 7th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, ser. EOOLT '16. New York, NY, USA: ACM, 2016, pp. 69–77. [Online]. Available: http://doi.acm.org/10.1145/2904081.2904091

[64] A. Benveniste, B. Caillaud, H. Elmqvist, K. Ghorbal, M. Otter, and M. Pouzet, "Structural analysis of multi-mode DAE systems," in *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, HSCC 2017, Pittsburgh, PA, USA, April 18-20, 2017*, G. Frehse and S. Mitra, Eds. ACM, 2017, pp. 253–263. [Online]. Available: http://doi.acm.org/10.1145/3049797.3049806

[65] ——, "Structural Analysis of Multi-Mode DAE Systems," Inria, Research Report RR-8933, Feb. 2017. [Online]. Available: https://hal.inria.fr/hal-01343967