A hybrid synchronous language with hierarchical automata: Static typing and translation to synchronous code

Albert Benveniste, Timothy Bourke, Benoit Caillaud, Marc Pouzet

To cite this version:

Albert Benveniste, Timothy Bourke, Benoit Caillaud, Marc Pouzet. A hybrid synchronous language with hierarchical automata: Static typing and translation to synchronous code. EMSOFT 2011 - Embedded Software, Oct 2011, Taipei, Taiwan. 10.1145/2038642.2038664 . hal-00654113

HAL Id: hal-00654113
https://inria.hal.science/hal-00654113

Submitted on 19 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Hybrid Synchronous Language with Hierarchical Automata*

Static Typing and Translation to Synchronous Code

Albert Benveniste
INRIA Rennes
albert.benveniste@inria.fr

Timothy Bourke†
INRIA Paris-Rocquencourt
timothy.bourke@inria.fr

Benoît Caillaud
INRIA Rennes
benoit.caillaud@inria.fr

Marc Pouzet
DI, École normale supérieure
marc.pouzet@ens.fr

ABSTRACT

Hybrid modeling tools like SIMULINK have evolved from simulation platforms into development platforms on which testing, verification and code generation are also performed. It is critical to ensure that the results of simulation, compilation and verification are consistent. Synchronous languages have addressed these issues but only for discrete systems.

Reprising earlier work, we present a hybrid modeler built from a synchronous language and an off-the-shelf numerical solver. The main novelty is a language with hierarchical automata that can be arbitrarily mixed with data-flow and ordinary differential equations (ODEs). A type system statically ensures that discrete state changes are aligned with zero-crossing events and that the function passed to the numerical solver has no side-effects during integration. Well-typed programs are compiled by source-to-source translation into synchronous code which is then translated into sequential code using an existing synchronous language compiler.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.3.2 [**Language classifications**]: Data-flow languages

General Terms

Algorithms, Languages, Theory

Keywords

Real-time systems; Hybrid systems; Synchronous languages; Block diagrams; Compilation; Semantics; Type systems

† Located at DI, École normale supérieure.

* This work was supported by the SYNCHRONICS large scale initiative of INRIA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0714-7/11/10 ...$10.00.

1. INTRODUCTION

Embedded systems are usually envisioned as discrete components that interact with a physical environment. This explains much of the success of hybrid modeling tools such as SIMULINK,[1] which have progressively evolved from simulation platforms to development platforms where a single source code is used for simulation, testing, formal verification and the generation of embedded code. It is becoming critical to place such tools on a firm semantical basis to be able to prove that the results of simulation, compilation and verification are mutually consistent: simulation results must be reproducible; generated code must be faithful to the original model and efficient; verification results on the model should also be valid for generated code.

Synchronous languages [2] already addressed these issues for a class of discrete real-time systems for which time- and resource-bounded code can be generated. They abstract completely from the physical environment, treating it solely as a source of discrete sampled inputs and a recipient for discrete outputs. They concern themselves with the details of how discrete instants are produced but largely ignore the gaps in between. The physical environment, on the other hand, continues to evolve during such gaps. It is usually modeled with ordinary differential equations (ODEs) or with differential algebraic equations (DAEs) that interact closely with discrete controllers and may exhibit discontinuities at discrete instants. While a synchronous description of the whole is possible, replacing differential by difference equations, obtaining both efficiency and precision in simulations really necessitates the use of a numerical variable-step solver. This calls for a clear separation of continuous parts, which must be compiled specifically for activation by the solver, from discrete parts, which can be left unchanged.

Building on earlier work [1], we propose a language for hybrid systems by extending an existing synchronous language with ODEs and an off-the-shelf numerical solver (SUNDIALS CVODE [8] is used in our prototype compiler). The synchronous language is used both for programming and as a target for code generation. The extension is *conservative* with respect to the synchronous subset: any synchronous function is compiled/optimized/executed exactly as if it were written in a regular synchronous language. There are two primary motivations for this approach. First,

[1] `http://www.mathworks.com/products/simulink`

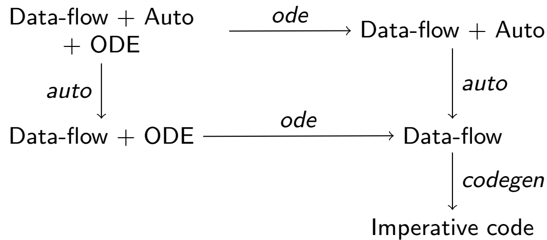

Figure 1: Possible compilation approaches

synchronous languages are already a well-understood solution for modeling and compiling discrete components: their extension with hybrid features provides an integrated solution for modeling and simulating systems together with their environments. This is particularly relevant for systems that are mostly discrete, but that incorporate some physical elements. Second, when physical systems are simulated on a computer, it is not possible to observe continuous variables at all values of simulated time t. On the contrary, t must be increased through a series of discrete increments. A synchronous language can thus act as a compilation target for hybrid models. Models are first translated into the synchronous subset of the language which is then compiled and linked with a numerical solver for the approximation of continuous trajectories. Note that the exact instants to sample are not necessarily known in advance nor periodic, but rather arise from the combined dynamics of continuous and discrete elements; that is, from the to and fro between a numerical solver and a discrete program. By translating the hybrid part into synchronous code and leaving the synchronous part unchanged, we can reuse most of the existing code generation and optimization techniques developed for synchronous languages [2] and also reduce the risk of differences between simulated and compiled code.

Contribution and organization of the paper.

The main novelty of this paper is a language which includes sophisticated control structures, namely hierarchical automata. These control structures can be composed in parallel or hierarchically, together with data-flow equations and ODEs with possible reset. Importantly, the control structures are valid in both discrete and continuous contexts. For example, the automata can be used to alternate between discrete control laws as in Mode Automata [12] or between sets of differential equations as in Hybrid Automata [7].

A type system is required to distinguish, at compile time, *discrete* computations from *continuous* ones. It ensures that discrete state changes are aligned with zero crossing events and determines which parts of a program are transformed for execution by a solver and which remain unchanged. Functions generated from well-typed programs are provably free of side effects when executed by a solver during the approximation of integral values. Maintaining this property is not easy when the bodies of control structures contain differential equations; there are several subtle typing issues. For example, transitions from one state to another in hybrid automata must only occur on zero-crossing events. One of the contributions of this paper is to clarify these issues.

We present a source-to-source transformation, for well-typed programs, that translates the full language into a synchronous subset amenable to code generation by a synchronous compiler. The ultimate goal is to translate programs written in a hybrid data-flow language with automata into imperative code, i.e., to go from the top-left of Figure 1 to the bottom-right. Given that synchronous programs with control structures (Data-flow+Auto) can already be translated to data-flow synchronous code (Data-flow) [12, 5], and that (Data-flow + ODE) can be translated to data-flow synchronous code (Data-flow) [1], there are two possible approaches. The first is to define a translation from (Data-flow+Auto+ODE) to (Data-flow+ODE) that removes control structures but not hybrid definitions. The second is to define a translation from (Data-flow+Auto+ODE) to (Data-flow+Auto) that removes hybrid constructs but not control structures. The latter translation, that effectively translates Hybrid Automata into Mode Automata, is considered in this paper. It is modular and maintains the structure of programming constructs.

This paper continues in Section 2 by recalling both the need for types when mixing discrete synchronous code with ODEs, and also the essentials of translating hybrid programs into synchronous code. It sketches the typing and compilation issues specific to control structures. In Section 3, the syntax of a basic synchronous language with control structures is presented, and the semantics of hierarchical automata are recalled. The extension of the language with ODEs is detailed in Section 4, and its type system is presented. The source-to-source translation is presented in Section 5. Related work is discussed in Section 6.

2. OVERVIEW

Following Lee and Zheng [10], we consider hybrid systems from a programming language perspective. We treat, in particular, the design, typing and compilation of a language which mixes data-flow equations and first-order ODEs. We do not consider verification.

Hybrid systems, by their very nature, involve the interwoven interaction of continuous dynamics and discrete events. Considered in isolation, a system's continuous dynamics can be expressed as an initial value problem $x' = f(t, x)\,\mathsf{init}\,x_0$, where $t \in \mathbb{R}^{\geq 0}$ represents physical time, x is a vector of continuous state variables, x' is a vector of instantaneous derivatives and x_0 is an initial state vector. Given f, a numerical solver (like SUNDIALS [8]) approximates the value of x for increasing instants $t_1, t_2, \ldots$ of t. The gap between instants, the 'step size', may vary. And as, internally, a solver may jump back and forth when going from t_i to t_{i+1} to achieve acceptable accuracy, f must be free of side effects, that is, combinatorial, at each (integration) step.

The first extension to this scheme to account for hybrid systems is to control the interruption of continuous trajectories. There are essentially two techniques: time horizons and vectors of *zero-crossing* expressions. Time horizons are useful for efficiently treating models that include periodically-executed discrete tasks. This is an important class of systems but not the only one: real applications also exhibit modes with discrete changes, in both the controller and the environment, that are not necessarily periodic or known in advance (e.g., an emergency braking system, or a clutch lock-up model, or a fuel control system[2]). Such systems are more adequately addressed by the zero-crossing mecha-

[2]See the SIMULINK/STATEFLOW automotive examples.

nism. Given a zero-crossing function $z = g(t, x)$, a solver can check, during integration, for changes of sign in elements of z that may indicate discontinuities in continuous state variables, or other interesting events. Periodic timers can, in fact, be encoded as a particular form of zero-crossing. For example, this fragment defines a periodic signal:

```
der p = 1.0 init -2.0 reset -2.0 every z  and  z = up(p)
```

The value of `p` increases with slope 1 from an initial value of -2, and is reset to -2 every time `p` crosses zero.[3] The variable `z` is true at instants $(2\,n)_{n\in\mathbb{N}^{>0}}$. Three types of action are possible in response to such discrete (zero-crossing) events: (1) reset elements of the continuous state vector x (as in the example above); (2) change the values of discrete state variables, which are implicit input parameters for f and g; (3) trigger side-effects, e.g., update a display. There may be a sequence of such actions, a *cascade*, before the solver is invoked again to approximate continuous behaviors. Thus, the overall execution of a hybrid system alternates between *continuous* phases where integration is performed by a solver and *discrete* phases where side-effects may occur.

In [1], an elementary, LUSTRE-like language extended with ODEs is introduced, along with a typing and compilation scheme for producing code linked with a black-box numerical solver. A type system is used to statically separate the continuous part, which is exercised by the numerical solver, from the discrete part, which is programmed in a synchronous language like LUSTRE or LUCID SYNCHRONE, and which must not act during integration. A signal is deemed discrete if activated on a discrete clock, defined [1, §2]:

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

A simple type system guarantees that when a program incorporates an ODE, all discrete changes, the aforementioned actions 1, 2 and 3, are aligned with zero-crossing events. The need for a type system can be understood from this example:

```
der time = 1.0 init 0.0  and  y = sum(time)
```

In this example, an ODE defining `time`[4] occurs in parallel with a synchronous function `sum`, whose definition is immaterial, but to be specific, could be assumed as:[5]

```
let node sum(x) = cpt where
  rec cpt = x -> (pre cpt +. x)
```

The example program is rejected because the computation of `time` is defined on a continuous clock while the computation of `y` is expected to run on a discrete clock.

Following [1], we reject such programs statically. The intuition is to assign a *kind* $k \in \{\mathtt{A}, \mathtt{D}, \mathtt{C}\}$ to every expression and equation. 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. In the example, `sum` is of kind `D`, whereas `der y = 1.0 init 0.0` is of kind `C`. The function `sum` cannot be activated directly on the stream `time` because the two have different kinds.

The original program should instead be written:

```
 der time = 1.0 init 0.0
and
 y = sum(time) every up(ez) init 0.0
```

where $\mathtt{e_z}$ is an expression to be monitored for zero-crossing events. The semantics of y are now $\mathtt{y}(0) = 0$, i.e., `y` is initialized to 0, and, if $t_{\mathtt{z}_1}, t_{\mathtt{z}_2}, ..., t_{\mathtt{z}_i}, \ldots$ are the instants of zero-crossing of $\mathtt{e_z}$, $\mathtt{y}(t_{\mathtt{z}_i}) = \Sigma_{j\leq i}(\mathtt{time}(t_{\mathtt{z}_j}))$, i.e., `y` is only computed at instants where $\mathtt{e_z}$ crosses zero, and otherwise it keeps its previous value, i.e., $\mathtt{y}(t) = \mathtt{y}(t_{\mathtt{z}_i})$ for $t_{\mathtt{z}_i} \leq t < t_{\mathtt{z}_{i+1}}$.

Well-typed programs mixing ODEs and synchronous code activated on zero-crossings can be translated into purely synchronous code. Such a source-to-source translation ensures traceability between the source code and its operational form and allows the reuse of the existing optimization techniques in synchronous compilers. The principle is to translate a hybrid function activated on a continuous clock—one that contains ODEs and zero-crossings—into a regular synchronous function. The function is augmented with extra inputs for passing continuous state values and zero-crossing notifications, and also with extra outputs for returning instantaneous derivative values, updated continuous state values, and zero-crossing expression values. As an example, consider a bouncing ball with initial position y_0 and speed y'_0:

```
let hybrid bouncing(y0,y'0) = y where
  rec der y  = y' init y0
  and der y' = -9.81 init y'0
               reset -0.9 *. last y' every up(-. y)
```

Whenever the ball hits the ground (`y` changes from positive to negative), its speed is reset to `-0.9 *. last y'`, where `last y'` is the left limit of signal `y'` when `y'` is left-continuous (when y crosses zero), as is the case here, and the previous value of `y'` otherwise. The translation gives:

```
let node bouncing((y0,y'0), ((ly,ly'), z)) =
                         (y, ((y,y'), (dy,dy'), upz))
  where
    rec dy  = y' and y = y0 -> ly and upz = -. y
    and dy' = -9.81 and last_y' = y'0 -> ly'
    and y' = if z then -0.9 *. last_y' else last_y'
```

The `der` definition of `y` is replaced with two new definitions: `dy` defines the instantaneous derivative of `y`; and `y` is equal to `y0` at the very first instant, and thereafter to `ly`, the value estimated by the solver.[6] The `der` definition of `y'` is also replaced: initially `last_y'` is equal to `y'0`, and thereafter it is equal to the input `ly'`; `y'` takes the same values unless the zero-crossing input `z` is true, in which case it becomes `-0.9 *.  last_y'`. In general, an extra '`last_y`' definition is needed to ensure correct initialization, in the presence of the `last` operator. The variable `upz` equals `-.  y`. In this system, `ly` and `ly'` are extra inputs to be computed by the solver whereas `dy` and `dy'` are extra outputs to be returned to the solver.[7]

---

[3] This can be written in SIMULINK as an integral with state port (here written st) and rising zero-crossing detection: $p, st = \frac{1}{s}(-2.0, \uparrow(st), 1)$.

[4] $\forall t \in \mathbb{R}^{\geq 0}, \mathtt{time}(t) = 0 + \int_0^t 1\,dt = t$

[5] $\forall n \in \mathbb{N}, \mathtt{sum}(\mathtt{x})(n) = \Sigma_{i=0}^{n}(x_i)$

[6] The LUSTRE initialization operator, $\cdot \mathtt{->} \cdot$ has the semantics: $(x \mathtt{->} y)_0 = x_0$ and for all $n \in \mathbb{N}^{>0}$, $(x \mathtt{->} y)_n = y_n$.

[7] To avoid excessive copying, the extra inputs and outputs can instead be stored in arrays with in-place modification.

This translation is fine for a language without control structures. But control structures, such as hierarchical automata, are essential for designing complex systems with different operating modes. Such automata may appear at many levels of a system, in both discrete and continuous contexts. The problem, then, is not whether to extend the basic language, but rather how to do so.

We explain our approach with an example automaton:

```
let hybrid two(y0,y'0,z1,z2) = y where
  rec init y = y0
  and automaton
      | Gravity ->
          local y' in
          do der y' = -9.81 init y'0 and der y = y'
          until z1 then Rising done
      | Rising ->
          do der y = 1.0
          until z2 then Gravity done
      end
```

The initial value of `y` is given by the equation `init y = y0`. The dynamics of `y` depend on the active mode of the automaton. In `Gravity`, the initial mode, $y'(t) = -9.81$. In `Rising`, $y'(t) = 1$. The notation `until z1 then Rising` defines a weak preemption with reset: the body of state `Gravity` is executed before the condition `z1` is tested. If it is true, `Rising` becomes the active mode in the next reaction and the equations it contains will be reinitialized.

This program can be transformed into a discrete one.

```
let node two((y0,y'0,z1,z2), (ly,ly')) =
                              (y, ((y,y'), (dy,dy'))) where
    rec y = y0 -> ly
    and automaton
        | Gravity ->
            do dy' = -9.81 and dy = y'
            and y' = y'0 -> ly'
            until z1 then Rising done
        | Rising ->
            do dy = 1.0 and y' = ly' and dy' = 0.0
            until z2 then Gravity done
        end
```

As before, two inputs, `ly` and `ly'`, are added, as are two pairs of outputs: state variables `y` and `y'` and instantaneous derivatives `dy` and `dy'`. While `y'` was initially declared local, it cannot be returned as an output unless its scope is extended, and it and its derivative must be given values in the other mode: we set them, respectively, to the estimate from the solver (`ly'`) and to 0.0; inactive states are effectively suspended (the same convention as SIMULINK). The result can be compiled by existing tools using standard techniques to transform automata into data-flow definitions [5].

In the previous example, the discrete events `z1` and `z2` are inputs. In general, discrete events may be computed locally and values may be communicated between states. Consider:

```
let hybrid bouncing(y0, y'0, start) = y where
 rec init y = y0
 and automaton
     | Await ->
        do der y = 0.0 until start then Bounce(y'0) done
     | Bounce(v) ->
        local c, y' in
        do der y' = -9.81 init v and der y = y'
           and c = up(-. y)
        until c on (y' < eps) then Await
            | c then Bounce(-0.9 *. y')
        done
     end
```

The program has two modes. It waits in the initial one until the input `start` becomes true. It then enters the `Bounce` mode, which contains a variable `c` defined to be true when a zero-crossing occurs on the expression `-.  y`. When that event occurs, two transitions are possible. They are prioritized from top to bottom: if condition `y' < eps` is true (written `c on (y' < eps)`), the automaton returns to the `Await` mode where the ball does not move; otherwise, `Bounce` is re-entered with a new value for the parameter `v`, which is used in the concomitant reset of `y'` to model a bounce.

In the translation, the `der` definitions are treated as before. For the zero-crossing expression, a new input `z` replaces `up(-.  y)`, and a new output `upz` is defined to be `-.  y`.

```
let node bouncing((y0, y'0, start), ((ly, ly'), z)) =
                    (y, ((y, y'), (dy, dy'), upz)) where
  rec y = y0 -> ly
  and automaton
      | Await ->
         do dy = 0.0 and dy' = 0.0
            and upz = 0.0 and y' = ly'
         until start then Bounce(y'0) done
      | Bounce(v) ->
         local c in
         do  dy' = -9.81 and dy = y' and c = z
             and upz = -. y and y' = v -> ly'
         until c & (y' < eps) then Await
             | c then Bounce(-0.9 *. y')
         done
      end
```

Transitions only occur on discrete events, viz. `start` and `up(-.y)`; it is impossible for them to fire during integral approximation, as `z` and `start` will be false. The value of `upz` is only relevant if `Bounce` is active; it is set to 0 in `Await`.

Note that a discrete equation cannot be executed in a mode of a continuous automaton, even if, unlike the previous example with `sum`, it does not interact with the other continuous variables. Consider, for example:

```
let hybrid wrong(z) = (y, c) where
  rec automaton
      | First -> do der y = 2.0 init 1.0
                 and c = 0 -> pre c + 1
                 until z then First done
```

In this automaton, an ODE `y` occurs in parallel with a discrete counter `c`. If $[0, t_z] \subseteq \mathbb{R}^{\geq 0}$ such that $\mathtt{z}(t_z) = \mathit{true}$ and for all $t \in [0, t_z)$, $\mathtt{z}(t) = \mathit{false}$, initially, the semantics of `y` is:

$$\forall t \in [0, t_z), y(t) = 1 + \int_0^t 2\, dt = 1 + 2\, t$$

The counter itself is defined on a discrete time basis. Then, what is the semantics of the parallel composition of the computation of y with that of c? Taking an initial part $[0..l] \subseteq \mathbb{N}$ for some $l \in \mathbb{N}$, it could be tempting to state that:

$$\forall n \in [1..l], c_n = c_{n-1} + 1 \text{ and } c_0 = 0$$

But then, how should $[0, t_z)$ and $[0..l]$ be related and what is the value of l? The two equations do not share the same timescale (y runs on a continuous clock whereas c should be activated on a discrete one) and the meaning of their parallel composition is thus unclear. Moreover, once compiled, it would be necessary to continuously increment c during integration. This is an error which must be statically detected.[8]

[8]Rewritten in STATEFLOW, the compiler complains: *In continuous time Stateflow charts, outputs cannot be read if they*

Perhaps surprisingly, this new case is subsumed by the previously proposed system of kinds, which rejects `wrong`. The system of kinds clarifies some of the restrictions imposed by the MathWorks Stateflow compiler.

The source-to-source translation of automata is only valid if the generated transition and zero-crossing functions are side-effect free. This is guaranteed by extending the analysis that assigns kinds. Basically, a kind is assigned to a whole automaton, and the equations in the bodies of every state must have that kind. It is possible, though, in automata of kind `C`, to put discrete computations on transitions since they can only be activated on a discrete clock. This simple type discipline ensures that, once compiled, the transition function is free of side-effects during integration. In the next two sections, we formalize the static typing of programs with hierarchical automata and the source-to-source translations.

3. A SYNCHRONOUS LANGUAGE WITH AUTOMATA

3.1 Syntax

We first consider a synchronous language that mixes data-flow equations and hierarchical automata. Hybrid features are added in Section 4.

$$
\begin{array}{lcl}
d & ::= & \texttt{let}\ k\ f(p) = e \mid d; d \\
k & ::= & \texttt{D} \mid \texttt{A} \\
e & ::= & x \mid v \mid e\, \texttt{fby}\, e \mid \texttt{last}(x) \mid f(e) \mid (e, e) \mid \texttt{let}\ E\ \texttt{in}\ e \\
p & ::= & (p, p) \mid x \\
E & ::= & x = e \mid E\ \texttt{and}\ E \\
 & & \mid \texttt{automaton}\ (S(p) \longrightarrow u\ \texttt{unless}\ s^*)^+ \\
u & ::= & \texttt{local}\ x\ \texttt{in}\ u \mid \texttt{do}\ E\ \texttt{until}\ s^* \\
s & ::= & e\ \texttt{then}\ S(e) \mid e\ \texttt{continue}\ S(e)
\end{array}
$$

A program is a sequence of global declarations (d) of functions over signals ($\texttt{let}\ k\ f(p) = e$), where k is the kind of the function: $k = \texttt{A}$ declares that f is combinatorial; $k = \texttt{D}$ declares that f may contain delays or automata.[9] Expressions are composed of variables (x), immediate values (v), initialized delays ($e\, \texttt{fby}\, e$), last values of signals ($\texttt{last}(x)$), applications of functions to arguments ($f(e)$), pairs (e, e) and local declarations ($\texttt{let}\ E\ \texttt{in}\ e$), which ultimately take the value of an e that may contain variables defined in a set of equations E. Patterns, p, are variables (x) or pairs of patterns (p, p). Other tuples are shorthand for nested pairs.

Equations are denoted E and include variables x of value e ($x = e$), sets of equations in parallel ($E\ \texttt{and}\ E$), local declarations of variables x in equations ($\texttt{local}\ x\ \texttt{in}\ E$), and automata ($\texttt{automaton}\ (S(p) \longrightarrow u\ \texttt{unless}\ s^*)^+$).

An automaton is a list of handlers ($S(p) \longrightarrow u\ \texttt{unless}\ s^*$): $S(p)$ is a parameterized state name, u is the state body and s^* is a sequence of strong transitions. The body u may contain variables local to the state ($\texttt{local}\ x\ \texttt{in}\ u$) but it is ultimately a set of equations E paired with a sequence of weak transitions ($\texttt{do}\ E\ \texttt{until}\ s^*$).[10] There are two types of transitions: either the target state is reset on entry ($e\ \texttt{then}\ S(e)$) or it is resumed ($e\ \texttt{continue}\ S(e)$), i.e. entered with history. $S(e)$ gives the the target state name (S) and a value (e) to pattern match against that state's parameters. The shorthand S, i.e., a state with no parameters, stands for $S(x)$ in equations, with x a fresh variable, and $S(())$ in transitions.

are ever written to in during actions or transition conditions. Therefore, the use of 'c' (#26) is illegal in the following context: State First #19 in Chart 'Chart' (#18): during: `c := c + 1;` (Simulink version 7.7.0.471).

[9] In the concrete syntax, `A` is omitted and `D` is written `node`.

[10] The keywords `unless` and `until` are omitted in the concrete syntax for empty transition sequences.

3.2 Examples and Intuitive Semantics

The distinction between combinatorial (kind `A`) and stateful functions (kind `D`) is necessary as they have different implementations. Combinatorial functions are typically imported from the host language (e.g., integer or boolean operators) and have no internal state. Such functions can be composed to form new combinatorial functions which are translated directly into the host language. Their type signature is of the form $t_1 \xrightarrow{\texttt{A}} t_2$ with t_1 and t_2 as input and output types respectively. A stateful function, in comparison, may contain stateful operators (e.g., `fby`, `pre`, `->`), automata or other stateful functions. Their implementation in the host language needs an extra argument for passing a mutable internal state record—all instances of such a function share the same compiled code but each is allocated a distinct state value. Their type signature is of the form $t_1 \xrightarrow{\texttt{D}} t_2$. The kind of a function is validated by static typing.

The semantics of data-flow equations is the one of Lustre and not recalled here. Automata deserve more attention. We adopt the automata of Lucid Synchrone [14], whose detailed semantics is given in [4]. The two transition kinds (strong and weak) and the two ways of entering a target state (history and reset) give four possible types of transitions, which is typical in the majority of languages or tools for embedded system design that have hierarchical automata, like, for example, the hierarchical automata of Scade 6, Ptolemy II [9], Statecharts [6], and Stateflow. An automaton's behavior in a reaction depends on its immediate prior state S_{prior}; initially the first in the sequence (which must take `()` as argument). It is computed in four steps:

1. For strong preemptions, we consider the $(s_i)_{i \in I}$ of the handler $\texttt{unless}\,(s_i)_{i \in I}$ of S_{prior}. Each s_i has either the form $e_i\ \texttt{then}\ S_i(se_i)$, or the form $e_i\ \texttt{continue}\ S_i(se_i)$. All $(e_i)_{i \in I}$ are computed to give a list of booleans.

2. If none of the $(e_i)_{i \in I}$ are true, the current state remains active for the reaction: $S_{active} = S_{prior}$. Otherwise, the branch corresponding to the earliest true expression e_k gives the active state: $S_{active} = S_k(v)$, where v results from evaluating se_k.

3. The body u of the active state S_{active} is reset if the last transition, be it strong or weak, had the form $\cdot\ \texttt{then}\ \cdot$. Execution passes through all $u = \texttt{local}\ x\ \texttt{in}\ u'$ into u', and when $u = \texttt{do}\ E\ \texttt{until}\ (s_j)_{j \in J}$, E is executed. Afterward, all weak transitions $(s_j)_{j \in J}$ are computed to find the state in which to begin the next reaction. Each s_j has one of the forms $e_j\ \texttt{then}\ se_j$, or $e_i\ \texttt{continue}\ se_i$. All $(e_i)_{i \in J}$ are computed.

4. If none of the $(e_i)_{i \in J}$ are true, the next state, i.e. S_{prior} at the next reaction, will be S_{active}. Otherwise, the branch corresponding to the earliest true expression $e_{k'}$ gives the next state $S_{k'}(v)$, where v results from immediately evaluating $se_{k'}$.

4. HYBRID SYNCHRONOUS LANGUAGE

This basic language is extended with three constructs: ordinary differential equations, tests for defining zero-crossing events and conditionals on those events. We present a minimal extension which neglects ODE resets and synchronous code activations on zero-crossings. They are, however, considered elsewhere [1] and implemented in our prototype.

4.1 Extended Syntax of the Language

In this section, we consider an extended language:

$$\begin{aligned} k &::= \mathtt{D} \mid \mathtt{A} \mid \mathtt{C} \\ e &::= \ldots \mid \mathtt{up}(e) \mid e\,\mathtt{on}\,e \\ E &::= \ldots \mid \mathtt{init}\,x = e \mid \mathtt{der}\,x = e \end{aligned}$$

A new element $\mathtt{C}$ is added to the set of kinds (k), for functions whose bodies must be activated on a continuous clock.

There are two new expression types. $\mathtt{up}(e)$ detects a rising zero-crossing on e, i.e., the instant when e changes from negative to non-negative and $e_1\,\mathtt{on}\,e_2$ is present if e_1, a zero-crossing, is present and e_2, a boolean expression, is true.

The equations are extended with declarations of an initial value e_0 for x ($\mathtt{init}\,x = e_0$), and an instantaneous derivative ($\mathtt{der}\,x = e$). The two are used in combination to define an ODE $x' = e$ with initial value e_0, for which we also allow the syntactic shortcut: $\mathtt{der}\,x = e\ \mathtt{init}\ e_0$.

4.2 Static Typing

As discussed in Section 2, we must distinguish at compile time what is *discrete* from what is *continuous*. We adopt the convention that a signal is typed *discrete* if it is declared so or is activated on a zero-crossing event using the result of an $\mathtt{up}(\cdot)$ or $\cdot\,\mathtt{on}\,\cdot$ construct. Otherwise, it is typed *continuous*.

Instead of associating this information with signals, we take a simpler approach and associate it with functions: a function f is given a type signature $t_1 \xrightarrow{k} t_2$ where t_1 is the type of its input, t_2 is the type of its output and k is a kind. If $k = \mathtt{C}$, f may only be used in a continuous context. If $k = \mathtt{D}$, f may only be used in a discrete context. If $k = \mathtt{A}$, then f is a combinatorial function that can be used in expressions of any kind. There is a subkinding relation where, for all k, $k \subseteq k$ and $\mathtt{A} \subseteq k$. The type language is:

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

where σ defines types schemes and $\beta_1, ..., \beta_n$ are type variables. A type t is either a pair ($t \times t$), a type variable (β) or a base type (bt), and $\mathtt{zero}$ is the type of zero-crossing conditions, with $\mathtt{up}(\cdot)$ and $\cdot\ \mathtt{on}\ \cdot$ as the only constructors.

As is typical, the typing rules are defined with respect to typing environments. We write G for the environment that accumulates over the sequence of global declarations in a program, mapping each to a type scheme (σ). And we write H for a local environment mapping each variable x to its type t and status: $\mathtt{init}(x) : t$ if the initial value is defined but not the derivative, $\mathtt{der}(x) : t$ if the derivative is defined but not the initial value, $\mathtt{last}(x) : t$ if the $\mathtt{last}(\cdot)$ operator may be applied, and $x : t$ otherwise.

$$G ::= [f_1 : \sigma_1; \ldots; f_n : \sigma_n]$$

$$H ::= [\,] \mid H, x : t \mid H, \mathtt{last}(x) : t \mid H, \mathtt{init}(x) \mid H, \mathtt{der}(x)$$

Operations on Environments.

If H_1 and H_2 are two environments, $H_1 + H_2$ is their union, provided their domains are disjoint; H_1, H_2 is their concatenation; $H_1 * H_2$ is a new environment such that

$$\begin{aligned} &(H_1 + [x : t]) * (H_2 + [x : t]) = (H_1 * H_2) + [x : t] \\ &(H_1 + [\mathtt{last}(x) : t]) * (H_2 + [\mathtt{last}(x) : t]) = \\ &\qquad\qquad (H_1 * H_2) + [\mathtt{last}(x) : t] \\ &(H_1 + [\mathtt{init}(x) : t]) * (H_2 + [\mathtt{der}(x) : t]) = \\ &\qquad\qquad (H_1 * H_2) + [\mathtt{last}(x) : t] \end{aligned}$$

(where $+$ and $*$ are associative and commutative). The initial environment G_0 gives the type signatures of imported operators (of kind $\mathtt{A}$), synchronous primitives (of kind $\mathtt{D}$), and zero-crossing constructors:

$$\begin{aligned} (+) &: \mathtt{int} \times \mathtt{int} \xrightarrow{\mathtt{A}} \mathtt{int} \\ (=) &: \forall \beta.\beta \times \beta \xrightarrow{\mathtt{A}} \mathtt{bool} \\ \mathtt{pre}(\cdot) &: \forall \beta.\beta \xrightarrow{\mathtt{D}} \beta \\ \cdot\,\mathtt{fby}\,\cdot &: \forall \beta.\beta \times \beta \xrightarrow{\mathtt{D}} \beta \\ \cdot\,\mathtt{->}\,\cdot &: \forall \beta.\beta \times \beta \xrightarrow{\mathtt{D}} \beta \\ \mathtt{up}(\cdot) &: \mathtt{float} \xrightarrow{\mathtt{C}} \mathtt{zero} \\ \cdot\,\mathtt{on}\,\cdot &: \mathtt{zero} \times \mathtt{bool} \xrightarrow{\mathtt{A}} \mathtt{zero} \end{aligned}$$

REMARK 1 ($\mathtt{up}(\cdot)$: REAL OR BOOLEAN?). *The function* $\mathtt{up}(x)$ *takes an* x *of type* $\mathtt{float}$. *Alternatively,* x *could be a* $\mathtt{bool}$*; with the mapping* $\{\mathtt{tt} \mapsto 1.0, \mathtt{ff} \mapsto -1.0\}$ *for the solver. The methods are equally expressive, but the modified secant method in* SUNDIALS *converges faster in the first.*

The type of transition guards.

Transition guards in discrete automata have type $\mathtt{bool}$. They are evaluated when a reaction occurs and the source state of their transition is active. Guards in continuous automata are the same conceptually, but in practice discrete events are communicated as zero-crossings. Thus, we write $k(\mathtt{bool})$ for the type of guards in context k, which is defined $\mathtt{C}(\mathtt{bool}) = \mathtt{zero}$, $\mathtt{D}(\mathtt{bool}) = \mathtt{bool}$ and $\mathtt{A}(\mathtt{bool}) = \mathtt{bool}$.

Generalization and Instantiation.

The types of program declarations are generalized to type schemes (σ) by quantifying over all free variables, which is valid since they are defined globally.

$$\begin{aligned} gen(t_1 \xrightarrow{k} t_2) \quad &= \quad \forall \beta_1, ..., \beta_n.t_1 \xrightarrow{k} t_2 \\ &\quad\quad \text{if } \{\beta_1, ..., \beta_n\} = FV(t_1 \to t_2) \end{aligned}$$

The variables in a type scheme σ can be instantiated, and the kind k of a function can be replaced with any kind k' where $k \subseteq k'$. Let $Inst(\sigma)$ be the set of all such instantiations.

$$\frac{k \subseteq k'}{(t \xrightarrow{k'} t')[t_1/\beta_1, ..., t_n/\beta_n] \in Inst(\forall \beta_1, ..., \beta_n.t \xrightarrow{k} t')}$$

Typing is defined by the predicates (TYP-ENV) stating that the equation E produces the type environment H' and has kind k, (TYP-EXP) stating that, in the global environment G and the local environment H, an expression e has type t and kind k and (TYP-PAT) which defines the type and environment produced by a pattern p, for which kinds are unnecessary.

(TYP-ENV)	(TYP-EXP)	(TYP-PAT)
$G, H \vdash_k E : H'$	$G, H \vdash_k e : t$	$\vdash_{pat} p : t, H$

(INIT)
$$\frac{G, H \vdash_{\mathtt{C}} e : t}{G, H \vdash_{\mathtt{C}} \mathtt{init}\, x = e : [\mathtt{init}(x) : t]}$$

(DER)
$$\frac{G, H \vdash_{\mathtt{C}} e : \mathtt{float}}{G, H \vdash_{\mathtt{C}} \mathtt{der}\, x = e : [\mathtt{der}(x) : \mathtt{float}]}$$

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

(EQ)
$$\frac{G, H \vdash_k e : t}{G, H \vdash_k x = e : [x : t]}$$

(APP)
$$\frac{t \xrightarrow{k} t' \in \mathit{Inst}(G(f)) \qquad G, H \vdash_k e : t}{G, H \vdash_k f(e) : t'}$$

(LETIN)
$$\frac{G,\; H, H_0 \vdash_k E : H_0 \qquad G,\; H, H_0 \vdash_k e : t}{G, H \vdash_k \mathtt{let}\; E \;\mathtt{in}\; e : t}$$

(CONST)
$$G, H \vdash_k 4 : \mathtt{int}$$

(LAST)
$$G, H + [\mathtt{last}(x) : t] \vdash_k \mathtt{last}(x) : t$$

(VAR)
$$G, H + [x : t] \vdash_k x : t$$

(VAR-LAST)
$$G, H + [\mathtt{last}(x) : t] \vdash_k x : t$$

(PAIR)
$$\frac{G, H \vdash_k e_1 : t_1 \qquad H \vdash_k e_2 : t_2}{G, H \vdash_k (e_1, e_2) : t_1 \times t_2}$$

(PAT-PAIR)
$$\frac{\vdash_{pat} p_1 : t_1, H_1 \qquad \vdash_{pat} p_2 : t_2, H_2}{\vdash_{pat} p_1, p_2 : t_1 \times t_2, H_1 + H_2}$$

(PAT-VAR)
$$\vdash_{pat} x : t, [x : t]$$

(DEF-NODE)
$$\frac{\vdash_{pat} p : t_1, H_p \qquad G, H_p \vdash_k e : t_2}{G \vdash \mathtt{let}\; k\; f(p) = e : [f : gen_G(t_1 \xrightarrow{k} t_2)]}$$

(DEF-SEQ)
$$\frac{G \vdash d_1 : G_1 \qquad G, G_1 \vdash d_2 : G_2}{G \vdash d_1; d_2 : G_1 + G_2}$$

Figure 2: Typing rules for data-flow equations

(AUTO)
$$\frac{\forall i \in I, j \in J_i, \quad \vdash sp_i : [S_i : t_i], H_i \qquad G, H + H_i \vdash_k u_i : H'_i, [S_l : t_l]_{l \in L_i} \qquad G, H + H_i \vdash_k^{\mathtt{A}} s_j : [S_j : t_j]}{G, H \vdash_k \mathtt{automaton}\; (sp_i \longrightarrow u_i \;\mathtt{unless}\; (s_j)_{j \in J_i})_{i \in I} : \Pi_{i \in I} H'_i}$$

(LOCAL-IN)
$$\frac{G, H + [x : t] \vdash_k u : H'}{G, H \vdash_k \mathtt{local}\; x \;\mathtt{in}\; u : H'}$$

(DO-UNTIL)
$$\frac{G, H \vdash_k E : H' \qquad \forall j \in J, \quad G, H' \vdash_k^k s_j : [S_j : t_j]}{G, H \vdash_k \mathtt{do}\; E \;\mathtt{until}\; (s_j)_{j \in J} : H', [S_j : t_j]_{j \in J}}$$

(PARAM-STATE)
$$\frac{\vdash p : t_p, H_p}{\vdash S(p) : [S : t_p], H_p}$$

(THEN)
$$\frac{G, H \vdash_{k_1} e : k_2(\mathtt{bool}) \qquad G, H \vdash_{\mathtt{D}} se : t}{G, H \vdash_{k_2}^{k_1} e \;\mathtt{then}\; S(se) : [S : t]}$$

(CONTINUE)
$$\frac{G, H \vdash_{k_1} e : k_2(\mathtt{bool}) \qquad G, H \vdash_{\mathtt{D}} se : t}{G, H \vdash_{k_2}^{k_1} e \;\mathtt{continue}\; S(se) : [S : t]}$$

Figure 3: Typing rules for automata (see also comments in Section 4.2)

Three other predicates are used for typing automata: (UNTIL) defines the environment defined by the body and computes the list of target states together with the types of their parameters; (THEN-CONTINUE) defines the target state which can be reached by a handler e `then` se or e `continue` se, k_1 is a restriction on the kind of the guard expression, and k_2 gives a restriction on its type; and (S-PARAM) returns the type and environment associated with a state pattern.

(UNTIL)
$$G, H \vdash_k u : H', [S_l : t_l]_{l \in L}$$

(THEN-CONTINUE)
$$G, H \vdash_{k_2}^{k_1} s : [S_l : t_l]_{l \in L}$$

(S-PARAM)
$$\vdash S(p) : [S : t_p], H_p$$

The typing rules are presented in Figures 2 and 3. Figure 2 is treats data-flow primitives.

Rule (INIT). An initialization of a variable x is well-typed if the defining expression e is well-typed. It takes the type t and kind k of e. The status of x is marked $\mathtt{init}(x)$.

Rule (DER). An expression e defining the derivative of a variable x must be of type `float` and kind `C`, and there must be an accompanying `init`. The status of x is marked $\mathtt{der}(x)$.

Rule (AND). Parallel equations must be well-typed with the same kind.

Rule (EQ). The kind k and type t of a variable x defined by an equation are those of the expression e, provided it is well-typed.

Rule (APP). A function application is well-typed if the kind k and argument type t of the function f can be instantiated to match those of a well-typed argument e. The result has (instantiated) type t' and the same kind.

Rule (LETIN). A local definition is well-typed if the equations E and resulting expression e are well-typed in an extended environment. Both must have the same kind k.

Rule (CONST). Set of rules illustrated by an integer constant.

Rules (LAST), (VAR) and (VAR-LAST). Occurrences of $\mathtt{last}(\cdot)$ and variables must be respect the enclosing environment.

Rule (PAIR). The rule for pairing is similar to the one for the parallel composition of equations.

Rules (PAT-PAIR) and (PAT-VAR). These rules build an initial environment for patterns.

Rule (DEF-NODE). The type of a node is generalized from the types of the input pattern t_1 and defining expression t_2. The kind k of the expression e must match the declared kind.

Rule (DEF-SEQ). Function definitions are typed sequentially.

Figure 3 defines the typing of control structures:

Rule (AUTO). In the typing rule for automata, the kind $k \in \{\mathtt{D}, \mathtt{C}\}$. The state pattern of each handler sp_i is typed, associating the state label S_i with the type of its parameter, and producing an environment H_i comprising the variables bound in the pattern and their types. The body and weak preemption conditions u_i are typed in a combination of the global environment H and that of the parameter variables H_i to yield an environment H'_i, comprising the names defined by the body, and, for each condition, the destination state label S_l and the type of the associated expression t_l. The strong preemption conditions s_j are typed similarly. The (AUTO) rule is conditional on additional constraints, stating that target states of strong and weak preemptions belong to the set of states defined by the automaton:

$$[S_l : t_l]_{l \in L_i} \subseteq [S_i : t_i]_{i \in I} \qquad [S_j : t_j]_{j \in J_i} \subseteq [S_i : t_i]_{i \in I}$$

Rule (LOCAL-IN). An x defined in u can be made local to u.

Rules (DO-UNTIL) and (PARAM-STATE). These rules apply respectively to state bodies E (and their weak preemption conditions s_j) and to state patterns.

Rules (THEN) and (CONTINUE). Guard expressions e must be of type $k_2(\mathtt{bool})$, which is `bool` for discrete automata and `zero` for continuous ones. This constraint guarantees that transitions are only taken in response to discrete events. The context of the guard expression is given by k_1. For strong transitions, it will be `A` (combinatorial), and for weak transitions it will either be `C` or `D` depending on the overall kind of an automaton. The expression yielding the argument for the destination state se is always of kind `D`, which allows synchronous functions to be executed in continuous automata when a transition fires.

5. TRANSLATION TO SUBSET

Well-typed programs can be translated to the synchronous subset of the language. The translation removes ODEs and zero-crossings and leaves discrete computations unchanged. It yields a sequential transition function that can be processed by a synchronous compiler.

The translation replaces zero-crossing operators and ODE definitions. Each zero-crossing operator $\mathtt{up}(e)$ is replaced by adding a new input z_i of type `bool`, which replaces the operator occurence, and a new output u_i of type $\mathtt{float} \times \mathtt{bool}$ with a defining equation. For each ODE $\mathtt{der}\, x = e$ with initialization $\mathtt{init}\, x = e_0$ we add a new input variable lx, two new output variables x and dx, and a local variable $last_x$. Two new equations are added to define dx and $last_x$, and the original equation defining x is modified.

Extra inputs and outputs are collected into vectors during the translation. For the inputs, we write zv to denote a vector of zero-crossing variables $[z_1, \ldots, z_n]$, and lxv for a vector of continuous state variables $[lx_1, \ldots, lx_k]$. For the outputs, we write upv for vectors of zero-crossing expression variables $[up_1, \ldots, up_n]$, xv for vectors of continuous state variables $[x_1, \ldots, x_k]$ and dxv for the vectors of instantaneous derivatives $[dx_1, \ldots, dx_k]$. The concatenation of two vectors is written $[z_1, \ldots, z_n] \mathbin{@} [z'_1, \ldots, z'_m]$, it yields a new vector $[z_1, \ldots, z_n, z'_1, \ldots, z'_m]$.

The translation is defined by $TrDef(d)$, $TrEq(E)$, $Tr(e)$ and $TrAuto((sp_i \longrightarrow \ldots)_{i \in I})$. We describe the first three, that treat data-flow primitives, before discussing the last.

The $TrDef(d)$ function addresses function declarations, it relies on an auxilliary mapping $KindOf(f)$ that gives the kind $k \in \{\mathtt{A}, \mathtt{D}, \mathtt{C}\}$ for the definition of a function f. Discrete functions are not modified by the translation. Hybrid functions are altered by the addition of inputs and outputs which are passed through internal occurrences of zero-crossing operators $\mathtt{up}(e)$, ODEs $\mathtt{der}\, x = e$ **and** $\mathtt{init}\, x = e_0$, and nested hybrid function applications $f(e)$.

The $TrEq(E)$ function treats equations. It returns a tuple $\langle zv, upv, lxv, xv, dxv, E' \rangle$, where the vectors contain the required inputs and outputs and E' is the translated equation (multiple equations are combined with **and**).

The $Tr(e)$ function defines the translation of an expression. It returns a tuple $\langle e', zv, upv, lxv, xv, dxv, E \rangle$, with the usual vectors of inputs and outputs, and where e' is the translated expression and E are the auxilliary definitions.

These three functions are defined in Figure 4. To simplify the presentation, we assume non-conflicting names for all variables, and furthermore that for every variable x there are corresponding (and unique) variables: lx, $last_x$ and dx.

1. Variables y and constants v are not changed. They return empty vectors for all variables and equations.
2. Applications of $\mathtt{last}(y)$ are replaced with the corresponding initialized last value $last_y$ whose eventual definition is guaranteed by the type system.
3. The operator $\mathtt{up}(e)$ is replaced by a fresh variable z that is added as a zero-crossing input. Another fresh variable u is defined by a new equation $u = (e, \mathtt{tt})$ and added to the list of zero-crossing outputs, the second element of the pair is a status flag used in the translation of automata.[11]
4. $e_1 \,\mathtt{on}\, e_2$ is translated into the boolean operation &.
5. Local definitions `let` E `in` e are flattened. This is sound, given appropriate renaming, in the basic language. But the compiler contains a more intricate treatment to avoid problems across automaton states, and to preserve the order of side-effects.
6. Applications of combinatorial (`A`) and discrete (`D`) functions are not changed.
7. For applications of hybrid (`C`) functions, we must account for the additional inputs and outputs added by the transformation. We thus introduce fresh input variables z and lx, adding them to the appropriate vectors, a local variable r to hold the result of the function call, and three fresh output variables up, x and dx. The local and output variables are defined by a new equation where the function is invoked, and the original expression is replaced by the variable holding the result. Note that the structure of nested hybrid function applications is reflected in the tree structure of the types of the new inputs and outputs.

[11] Signals [4] are used instead of explicit status values in the full language where the added equation becomes $\mathtt{emit}\ u = e$.

$\mathit{Tr}(y)$	$=$	$\langle y, [\,], [\,], [\,], [\,], [\,], [\,]\rangle$
$\mathit{Tr}(v)$	$=$	$\langle v, [\,], [\,], [\,], [\,], [\,], [\,]\rangle$
$\mathit{Tr}(\texttt{last}(y))$	$=$	$\langle \mathit{last}_y, [\,], [\,], [\,], [\,], [\,], [\,]\rangle$ where last_y is the initialized last value corresponding to y
$\mathit{Tr}(\texttt{up}(e))$	$=$	$\mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\langle z, z.zv, u.upv, lxv, xv, dxv, u = (e', \texttt{tt})\ \texttt{and}\ E\rangle$ where z and u are fresh variables.
$\mathit{Tr}(e_1\ \texttt{on}\ e_2)$	$=$	$\mathit{let}\ \langle e'_1, zv_1, upv_1, lxv_1, xv_1, dxv_1, E_1\rangle = \mathit{Tr}(e_1)\ \mathit{in}$ $\mathit{let}\ \langle e'_2, zv_2, upv_2, lxv_2, xv_2, dxv_2, E_2\rangle = \mathit{Tr}(e_2)\ \mathit{in}$ $\langle e'_1 \,\&\, e'_2, zv_1 @ zv_2, upv_1 @ upv_2, lxv_1 @ lxv_2, xv_1 @ xv_2, dxv_1 @ dxv_2, E_1\ \texttt{and}\ E_2\rangle$
$\mathit{Tr}(\texttt{let}\ E\ \texttt{in}\ e)$	$=$	$\mathit{let}\ \langle zv_1, upv_1, lxv_1, xv_1, dxv_1, E_1\rangle = \mathit{TrEq}(E)\ \mathit{in}$ $\mathit{let}\ \langle e', zv_2, upv_2, lxv_2, xv_2, dxv_2, E_2\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\langle e', zv_1 @ zv_2, upv_1 @ upv_2, lxv_1 @ lxv_2, xv_1 @ xv_2, dxv_1 @ dxv_2, E_1\ \texttt{and}\ E_2\rangle$
$\mathit{Tr}(f(e))$	$=$	$\mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\langle f(e'), zv, upv, lxv, xv, dxv, E\rangle$ if $\mathit{KindOf}(f) \in \{\texttt{A}, \texttt{D}\}$
$\mathit{Tr}(f(e))$	$=$	$\mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\langle r, z.zv, up.upv, lx.lxv, x.xv, dx.dxv, (r, up, x, dx) = f(z, lx, e')\ \texttt{and}\ E\rangle$ if $\mathit{KindOf}(f) = \texttt{C}$ and where $r, z, up, lx, x,$ and dx are fresh variables.
$\mathit{TrEq}(x = e)$	$=$	$\mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\langle zv, upv, lxv, xv, dxv, x = e'\ \texttt{and}\ E\rangle$
$\mathit{TrEq}(\texttt{init}\ x = e_0)$	$=$	$\mathit{let}\ \langle e'_0, zv_0, upv_0, lxv_0, xv_0, dxv_0, E_0\rangle = \mathit{Tr}(e_0)\ \mathit{in}$ $\langle zv_0, upv_0, lxv_0, xv_0, dxv_0, (\mathit{last}_x = e'_0\ \texttt{->}\ lx)\ \texttt{and}\ E_0\rangle$ where lx and last_x correspond with x.
$\mathit{TrEq}(\texttt{der}\ x = e)$	$=$	$\mathit{let}\ \langle e', zv, upv, lx.lxv, xv, dxv, E\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\langle zv, upv, lx.lxv, x.xv, dx.dxv, (x = \mathit{last}_x)\ \texttt{and}\ (dx = e')\ \texttt{and}\ E\rangle$ where lx, last_x and dx correspond with x.
$\mathit{TrEq}(\texttt{automaton}\ (sp_i \longrightarrow \ldots)_{i\in I})$	$=$	$\mathit{let}\ \langle (sp'_i \longrightarrow \ldots)_{i\in I}, zv, upv, lxv, xv, dxv\rangle = \mathit{TrAuto}((sp_i \longrightarrow \ldots)_{i\in I})\ \mathit{in}$ $\langle zv, upv, lxv, xv, dxv, \texttt{automaton}\ (sp'_i \longrightarrow \ldots)_{i\in I}\rangle$
$\mathit{TrEq}(E_1\ \texttt{and}\ E_2)$	$=$	$\mathit{let}\ \langle zv_1, upv_1, lxv_1, xv_1, dxv_1, E'_1\rangle = \mathit{TrEq}(E_1)\ \mathit{in}$ $\mathit{let}\ \langle zv_2, upv_2, lxv_2, xv_2, dxv_2, E'_2\rangle = \mathit{TrEq}(E_2)\ \mathit{in}$ $\langle zv_1 @ zv_2, upv_1 @ upv_2, lxv_1 @ lxv_2, xv_1 @ xv_2, dxv_1 @ dxv_2, E'_1\ \texttt{and}\ E'_2\rangle$
$\mathit{TrDef}(\texttt{let}\ k\ f(y) = e)$	$=$	$\mathit{let}\ \langle e', zv, upv, lxv, xv, dxv, E\rangle = \mathit{Tr}(e)\ \mathit{in}$ $\texttt{let D}\ f(zv, lxv, y) = \texttt{let}\ E\ \texttt{in}\ (e', upv, xv, dxv)$ if $k = \texttt{C}$
$\mathit{TrDef}(\texttt{let}\ k\ f(y) = e)$	$=$	$\texttt{let}\ k\ f(y) = e$ if $k \in \{\texttt{A}, \texttt{D}\}$

Figure 4: Translation of Data-flow Equations into Synchronous Code

8. For basic equations $x = e$, the expression is translated.
9. An initialization of a continuous state variable x is replaced by a new definition for the corresponding last_x variable, which takes an initial value in the initial reaction or the first reaction after a reset and otherwise the value given as input by the solver.
10. A derivative equation $\texttt{der}\ x = e$ is replaced by two new equations; one defining the value of the output x—which, in the absence of resets is just the initialized last variable last_x—and the other defining an output for the instantaneous derivative dx. The state variable x must be an output so that the corresponding continuous state can be reset in discrete reactions. A variable lx for the last state value from the solver is added to the input vector.
11. The translation of automata are discussed below.
12. Parallel (sets of) equations are translated separately and the results combined.
13. Continuous function definitions are translated into discrete functions where the zv and lxv vectors are turned into patterns over inputs, and the upv, xv and dxv vectors are returned as outputs.
14. Discrete and combinatorial functions are not changed.

Translation of Automata.

Automata are translated by a function

$$\mathit{TrAuto}((sp_i \longrightarrow u_i\ \texttt{unless}\ (s_j)_{j\in J_i})_{i\in I})$$

that returns a tuple

$$\langle (sp_i \longrightarrow u'_i\ \texttt{unless}\ (s'_j)_{j\in J_i})_{i\in I}, zv, lxv, upv, xv, dxv\rangle.$$

The translation is only performed for automata in continuous contexts; discrete automata are not translated.

Some auxiliary definitions are used in the translation of automata. We write $x \in v$ to mean that x is an element in

$$
\begin{array}{l}
\mathit{TrUntil}(\texttt{local}\ x\ \texttt{in}\ u) = \\
\quad \mathit{let}\ \langle u', zv', upv', lxv', xv', dxv' \rangle = \mathit{TrUntil}(u)\ \mathit{in} \\
\quad \mathit{if}\ x \in xv' \\
\quad \mathit{then}\ \langle \texttt{local}\ last_x\ \texttt{in}\ u', zv', upv', lxv', xv', dxv' \rangle \\
\quad \mathit{else}\ \langle \texttt{local}\ x\ \texttt{in}\ u', zv', upv', lxv', xv', dxv' \rangle \\
\mathit{TrUntil}(\texttt{do}\ E\ \texttt{until}\ (s_j)_{j \in J}) = \\
\quad \mathit{let}\ \langle E', zv', upv', lxv', xv', dxv' \rangle = \mathit{TrEq}(E)\ \mathit{in} \\
\quad \langle \texttt{do}\ E'\ \texttt{until}\ (s_j)_{j \in J}, zv', upv', lxv', xv', dxv' \rangle \\
\mathit{TrAuto}((sp_i \longrightarrow u_i\ \texttt{unless}\ (s_j)_{j \in J_i})_{i \in I}) = \\
\quad \mathit{let}\ (\langle u'_i, zv'_i, upv'_i, lxv'_i, xv'_i, dxv'_i \rangle = \mathit{TrUntil}(u_i))_{i \in I}\ \mathit{in} \\
\quad \mathit{let}\ zv\ = \oplus_{i \in I} zv_i \quad \mathit{in} \\
\quad \mathit{let}\ upv = \oplus_{i \in I} upv_i\ \mathit{in} \\
\quad \mathit{let}\ lxv = \oplus_{i \in I} lxv_i \quad \mathit{in} \\
\quad \mathit{let}\ xv\ = \oplus_{i \in I} xv_i \quad \mathit{in} \\
\quad \mathit{let}\ dxv = \oplus_{i \in I} dxv_i\ \mathit{in} \\
\quad \mathit{let}\ (u''_i = add(u'_i)(\{u = (0.0, \texttt{ff}) \mid u \in upv \setminus upv_i\}))_{i \in I} \\
\quad \mathit{in} \\
\quad \mathit{let}\ (u'''_i = add(u''_i)(\{dx = 0.0 \mid dx \in dxv \setminus dxv_i\}))_{i \in I}\ \mathit{in} \\
\quad \mathit{let}\ (u''''_i = add(u'''_i)(\{x = lx \mid x \in xv \setminus xv_i\}))_{i \in I} \qquad \mathit{in} \\
\quad \langle (sp_i \longrightarrow u''''_i\ \texttt{unless}\ (s_j)_{j \in J_i})_{i \in I}, zv, upv, lxv, xv, dxv \rangle
\end{array}
$$

Figure 5: Translation of Automata

the vector v,[12] and $x \in v_1 \setminus v_2$ to mean that $x \in v_1 \wedge x \notin v_2$. The union of vectors l_1 and l_2 is written $l_1 \oplus l_2$, and defined:

$$
\begin{array}{lcl}
[\,] \oplus l & = & l \oplus [\,] = l \\
(x.xs) \oplus ys & = & x.(xs \oplus ys)\ \text{if}\ x \notin ys \\
(x.xs) \oplus ys & = & xs \oplus ys\ \text{otherwise}
\end{array}
$$

This operator is lifted to sets of vectors:

$$lx = \oplus_{i \in I}(lx_i) = lx_{i_1} \oplus ... \oplus lx_{i_n}\ \text{if}\ I = \{i_1, ..., i_n\}$$

For a body u and a set of equations E, $add(u)(E)$ builds a new body from the given one by adding every $e \in E$.

The translation of automata is presented in Figure 5. $\mathit{TrAuto}(\cdot)$, translates the bodies and weak preemptions of all states $i \in I$, via the auxiliary function $\mathit{TrUntil}(\cdot)$. The vectors zv, upv, lxv, xv and dxv for an automaton are formed from the unions of the corresponding vectors from each state body. All of the variables in the output vectors—upv, xv and dxv—have to be defined in every state body. For every variable up that is not already defined in a state, we add the equation $up = (0.0, \texttt{ff})$; the reason is explained below. For an undefined derivative variable, we add the equation $dxv = 0.0$. And for an undefined state variable, we add the equation $x = lx$, where lx is the corresponding (uninitialized) state input. The strong transition expressions $(s_j)_{j \in J}$ are left unchanged since the guards are combinatorial and the actions are discrete.

$\mathit{TrUntil}(\cdot)$ transforms the equations in body E but leaves the transition expressions $(s_j)_{j \in J}$ unchanged—we make the usual assumption that each weak transition guard is replaced with a variable defined in the state body. Local variable declarations are left in place, unless they apply to a continuous state variable, since these variables must be completed in the other states and available as outputs. But if x is revealed, the corresponding $last_x$ variable must be hidden since otherwise it must also be defined in the other states.

[12]Here, to simplify the presentation, we only consider unstructured vectors.

Changes to the execution model.

In the work presented in this paper, we try to extend our earlier approach [1] as conservatively as possible. Nevertheless, two adjustments to the original execution model are required when automata are added.

A hybrid program is executed by alternating between a continuous phase where the program is exercised by a numeric solver and a discrete phase where the effects of zero-crossings are calculated. Discrete phases succeed one another for as long as new zero-crossings are detected. An extra discrete reaction is required whenever a weak transition is fired to ensure that initialization of the successor state occurs in a discrete reaction and not inside the numeric solver. In the current implementation an extra discrete reaction is always executed after no more zero-crossings have been detected (and it may itself generate new zero-crossings).

The value of a zero-crossing output up becomes $(0.0, \texttt{ff})$ when the parent state of the zero-crossing is inactive. The 0.0 value ensures that a zero-crossing is not generated immediately after returning to the active state. Setting the status flag to `ff` tells the discrete solver not to generate a (spurious) zero-crossing even if the previous value of up was negative. Note that zero-crossing detection is not reset by a self-looping `then` transition.

6. RELATED WORK AND DISCUSSION

Related Work.

There are other hybrid data-flow languages that include automata [3]. The two which have most influenced our approach are HYVISUAL/PTOLEMY II and Simulink/Stateflow.

Lee and Zheng [10] have already considered hybrid systems modelers from a programming language perspective, and there are many similarities between their approach and ours. In particular, their implementation of HYVISUAL[13] on top of PTOLEMY II[14] allows arbitrary hierarchical nestings of automata, continuous data-flow networks, and discrete data-flow networks. Models are executed by cycling between discrete and continuous phases. A discrete phase is executed as long as the discrete state keeps changing. Rather than define a type system to separate discrete from continuous computations, there is a 'simple consistency check' to forbid direct connections between discrete and continuous ports.

In later work [11], Lee and Zheng show that the synchronous model is powerful enough to encode continuous-time. We take a similar approach but use the synchronous model both as a semantic base and as a compilation target, permitting us to exploit existing compilation and analysis techniques. Moreover, source-to-source translations have good traceability which is important in some domains. Nevertheless, our approach is more limited in some respects. We do not, for instance, consider multiple numerical solvers.

The SIMULINK/STATEFLOW system provides interacting discrete and continuous data-flow networks and automata. Rather than a formal type system, a number of restrictions for continuous-time modeling are proposed [13, 16-26] and enforced by the Stateflow compiler. These rules aim to eliminate side-effects during numerical integration ('minor time steps'): assignments and function calls may only occur on transitions and when entering or exiting a state, and the

[13]`http://ptolemy.eecs.berkeley.edu/hyvisual/`
[14]`http://ptolemy.berkeley.edu/ptolemyII/`

derivatives of continuous variables may only be set, using a `*_dot` notation, in *during* actions

The behavior of zero-crossings in STATEFLOW is also interesting. Transition guards in continuous automata are arbitrary boolean expressions—using input events makes a chart 'triggered' rather than continuous. When zero-crossings are enabled for a chart, the simulation engine tries to accurately detect the instants of transition, not by monitoring the details of sub-expressions, but instead by changing the value of a zero-crossing expression from negative to positive upon state entry.[15] Charts are executed just before $(t - \epsilon)$ the calculated time of a zero-crossing (t), and again just afterward $(t + \epsilon)$. In this way, cascades of zero-crossings are also possible, but some time always passes between any two.

While SIMULINK/STATEFLOW is obviously much richer and more polished than our proposal, we believe that our approach is not without value. For instance, rather than effectively having two simulation engines [3, p. 28], we seek a consistent and simple integration of continuous automata and data-flow networks. Furthermore, our goal of better understanding the fundamentals of such tools mandates that we seek a small set of primitives, rather than attempt to provide a large number of features.

Boolean guards.

The encoding for boolean guards mentioned in Remark 1 is enough to detect enabling in a continuous phase, but not to have transitions with enabled guards taken continually. This alternative can be implemented using intricate encodings of zero-crossings, but it would also be possible to either change the definition of $\mathtt{up}(x)$ to $(x \geq 0 \,\texttt{->}\, z)$, that is true after a reset if x is not negative and then equal to the input from the solver, or to add primitives from which the same behaviour can be constructed, i.e. to add an $\mathtt{init}\colon \mathtt{bool} \overset{\mathtt{C}}{\rightarrow} \mathtt{zero}$, that only tests its argument when reset, and an $\mathtt{orz}\colon \mathtt{zero} \times \mathtt{zero} \overset{\mathtt{C}}{\rightarrow} \mathtt{zero}$ for combining events.

Compiling automata first.

This paper presents one of the paths shown in Figure 1, but we have also experimented with translating automata before ODEs. Our type system is not fine enough, however, to maintain the information implicit to an automaton that guarantees sound interaction between discrete and continuous behaviors. In simple terms: the intermediate results of compilation are not well-typed. This is a subject of our continuing research.

Sharing solver resources.

Another subject of our continuing research is to find a natural way of sharing zero-crossings and continuous states across mutually-exclusive automaton modes. This turns out to be quite delicate in our current execution and semantic models, and it is an inadequacy that we would like to correct.

7. CONCLUSION

This paper introduced a language for programming explicit hybrid systems with synchronous concurrency, ordinary differential equations, and hierarchical automata. A synchronous language is used at the programming level and as a target for code generation via a source-to-source translation. Unsound combinations of continuous and discrete elements are rejected by a type system. Our approach elucidates some aspects of designing hybrid modelers.

[15] This is mentioned in the documentation; and the insensitivity to guard dynamics is also evident in debugger traces.

8. REFERENCES

[1] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *ACM SIGPLAN/SIGBED Languages, Compilers, Tools & Theory for Embedded Systems (LCTES)*, Chicago, USA, April 2011.

[2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[3] L.P. Carloni, R. Passerone, A. Pinto, and A.L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations & Trends in Electronic Design Automation*, vol. 1, 2006.

[4] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing signals and modes in synchronous data-flow systems. In *ACM Int. Conf. on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006.

[5] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *ACM Int. Conf. on Embedded Software (EMSOFT'05)*, Jersey City, New Jersey, USA, September 2005.

[6] D. Harel. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3:231–275, 1987.

[7] T. Henzinger. The theory of hybrid automata. *NATO ASI Series F: Computer & Systems Sciences*, 170:265–292, 2000. Springer-Verlag.

[8] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. SUNDIALS: Suite of nonlinear and differential / algebraic equation solvers. *ACM Trans. Mathematical Software*, 31(3):363–396, September 2005.

[9] E.A. Lee. Finite state machines and modal models in Ptolemy II. Technical Report UCB/EECS-2009-151, EECS Uni. Calfornia at Berkeley, November 2009.

[10] E.A. Lee and H. Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume 3414 of *LNCS*, Zurich, Switzerland, March 2005. Springer-Verlag.

[11] E.A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, Salzburg, Austria, September/October 2007.

[12] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[13] The MathWorks, Natick, MA, USA. *Stateflow 7 User's Guide*, 2011.

[14] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006. (`www.di.ens.fr/~pouzet/lucid-synchrone`).