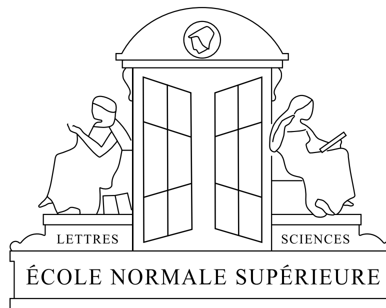
Research Internship Report

with the Parkas Inria/ENS team in the Computer Science Department
of the École normale supérieure, from March to August 2012.



Supervised by Marc Pouzet (marc.pouzet@ens.fr).

Mixing Continuous and Discrete Time in a Synchronous Language

Pablo Rauzy
pablo.rauzy@ens.fr

Computer Science Department — École normale supérieure

July 19, 2012

Abstract

A hybrid system is a system that exhibits both discrete and continous behaviors (e.g., a programmed controller and the evolution of its environment). ZÉLUS is a hybrid synchronous programming language, allowing the programmation and modelization of hybrid systems. The modelization of hybrid systems works by alternating between continuous phases and discrete steps. During continuous phases the continuous state of the system evolves in time with respect to differential equations. At discrete steps the discrete state of the system may change instantaneously.

The main result of this report is a static analysis of ZÉLUS code which detects if there can be an infinite number of discrete steps between two continuous phases. The presence of a finite number of such steps is a necessary condition to ensure time progress during a simulation of the system.

The question of the automatic translation of hybrid automata, a formalism to analyze/modelize hybrid systems, into ZÉLUS code, is also raised. Small results in this direction are shown and then a roadmap for pursuing this translation is proposed.

Contents

1 Hybrid synchronous languages at Parkas **3**
1.1 The Parkas team . . . 3
1.2 Synchronous languages . . . 3
1.3 Hybrid systems . . . 4

2 ZÉLUS: an hybrid synchronous language **5**
2.1 Presentation of the language . . . 5
2.2 `MiniZélus` . . . 6
2.3 Example of `MiniZélus` program . . . 7

3 Static analysis of cascades of discrete zero-crossings **7**
3.1 Discrete/continuous control-flow graph . . . 8
3.2 Building the discrete/continuous control-flow graphs . . . 8
3.3 Examples of discrete/continuous control-flow graphs . . . 9
3.4 Adding support for automata . . . 12
3.5 Results . . . 13

4 Working with hybrid automata in ZÉLUS **14**
4.1 What is a hybrid automaton . . . 14
4.2 An example of hybrid automaton . . . 15
4.3 Expressing boolean expressions using zero-crossings . . . 16
4.4 From hybrid automata to ZÉLUS code: propositions . . . 18

5 Conclusion **19**

Thanks

I would like to thank everyone in the Parkas team for welcoming me as an intern in their group and for the interesting conversations we had, whether it was about work or not. Among them I would especially like to thank Marc Pouzet, my supervisor, without whom this internship wouldn't have been possible.
I would also like to give special thanks to the other interns and the PhD students of the team with whom I had great fun and who provided moral support when I needed it.
And last but not least, I would like to thank Isabelle Delais and Joëlle Isnard for their helpful administrative support.

1 Hybrid synchronous languages at Parkas

1.1 The Parkas team

Parkas: Parallélisme des Réseaux de Kahn Synchrones.
The Parkas team addresses the design, semantics and compilation of languages for the implementation of provably safe and efficient computing systems. It is driven by the ideal of a unique source code used both to program and simulate a wide variety of systems, including (1) embedded real-time controllers (e.g., fly-by-wire, engine controller); (2) computationally intensive, numerical and non-numerical applications; (3) simulations of a large number of embedded systems in close interaction (in factories, electrical or sensor networks, train tracking, etc.). All these applications drive the design and implementation of formally defined languages, where the generated code is guaranteed to be "correct by construction" with respect to the single source specification. In addition, all simulations can be replayed, including the environment, thanks to determinism. All safety-critical parts of the generated code are validated by annotations, in the form of automated formal proofs. All performance critical parts are likewise annotated. When appropriate, their combined distributed performance is proven to meet real-time constraints against all tested environments.

Our research team is characterized by its attachment to the simplicity and complementarity of the functional, data-flow model of Kahn process networks with the theory and practice of synchronous languages.

We believe that the current application domain of synchronous languages could be widened significantly, provided that we answer two kinds of questions. First of all, the ability to program and simulate a discrete controller together with its environment, made of (a huge number of) other processes added/removed dynamically and possibly evolving in continuous time. Second, the ability to generate efficient parallel code from a synchronous specification, targeting modern architectures including shared-memory multicore processors, non-uniform manycore architectures, tiled processor arrays, and heterogeneous systems with hardware accelerators (GPU, DSP, FPGA). Answering this second kind of question involves reasoning about architecture constraints while preserving modular composition, and supporting relaxed synchronous models with jittering, buffered communication.

To work towards our goal, we leverage formal principles and practical experience in language design, semantics, type theory, concurrency models, synchronous circuits, code generation, compiler optimization, polyhedral compilation algorithms, and parallel hardware.

(Description taken from `http://www.di.ens.fr/ParkasOverview.html`.)

1.2 Synchronous languages

Synchronous programming languages such as Esterel[16], Lustre[3], and Lucid Synchrone[4] support the synchronous programming paradigm. The principle of synchronous programming is to make the same abstraction for programming languages as the synchronous abstraction in digital circuits. Synchronous circuits are indeed designed at a high-level of abstraction where the timing characteristics of the electronic transistors are abstracted away. Each gate of the circuit (and, or, ...) is therefore assumed to compute its result instantaneously, each wire is assumed to transmit its signal instantaneously. A synchronous circuit is clocked and at each tick of its clock, it computes instantaneously its output values and the new values of its memory cells from the its input values and the current values of its memory cells.

The synchronous abstraction makes reasoning about time in a program a lot easier, thanks to the notion of logical clock ticks: a synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be instantaneous and simultaneous. The synchronous abstraction eliminates the non-determinism resulting from the interleaving of concurrent behaviors, which allows thinking about parallelism in a clean and simple context. These properties

also allows deterministic semantics [1, 2], thereby making synchronous programs more amenable to formal analysis, verification and certified code generation, and usable as formal specification.

1.3 Hybrid systems

A *discrete system* is a system with a finite number of states. Typical examples are circuits, digital chips, and synchronous programs.

A *continuous system* is a system with a continuous behaviour and a real-valued state space. Physical systems with quantities like time, temperature, speed, acceleration etc. are continuous systems. Their evolution over time can be modelized by continuous functions or differential equations.

A **hybrid system** [18] is a system that exhibits both continous and discrete behaviors. Such a system has a continous state and a discrete state. The continous state evolves in time during *continuous phases* whereas the discrete state changes instantly at *discrete steps* between the continuous phases.

Among hybrid systems, some are intrinsically hybrid: a glass of water being filled up which suddenly overflows undergoes an instantaneous change in physical behavior, the same is true for a bouncing ball at each impact. Others systems have discrete and continuous components: a programmed controller which induces discrete changes and its envrironment which evolves continuously, an elevator or an automatic transmission system are examples of hybrid systems.

Example. As an example we will model a water tank. The tank leaks at a constant rate v, and water is added to the tank at a constant rate w if the tap is turned on, or not at all if the tap is turned off. We want the level of water x (*continuous environment*) to never be higher than $r1'$ and we want the tank never to be empty (x must be greater than $r2'$). For safety, we decide that the tap is turned on before the tank is actually empty but not to soon either, when x is below $r2$. For the same reason we decide that the tap is turned off before reaching the maximum authorized level but not to soon either, when x is over $r1$. The tap is controlled by a programmed computer (*discrete controller*) which receives data from sensors placed in the tank at depths corresponding to $r1$, $r1'$, $r2$, and $r2'$. Figure 1 illustrates this example.

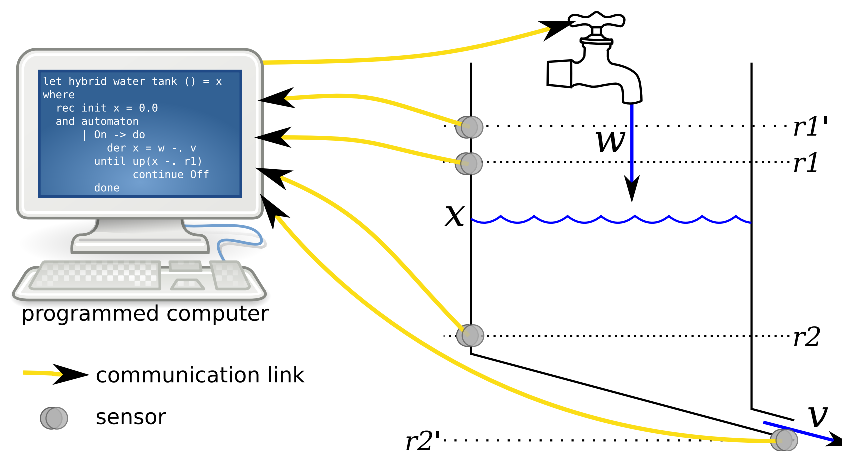


Figure 1: Illustration of the water tank hybrid system

2 ZÉLUS: an hybrid synchronous language

Before presenting our work on a static analysis in Section 3 ("Static analysis of cascades of discrete zero-crossings", p. 7), we start with a presentation of the ZÉLUS programming language. After that, the part of the language that we are working with, which we will call `MiniZélus`, is presented in more detail. Finally, an example of `MiniZélus` program is given.

2.1 Presentation of the language

Hybrid modelers such as Simulink have become corner stones of embedded systems development. They allow both discrete controllers and their continuous environments to be expressed in a single language. Despite the availability of such tools, there remain a number of issues related to the lack of reproducibility of simulations and to the separation of the continuous part, which has to be exercised by a numerical solver, from the discrete part, which must be guaranteed not to evolve during continuous phases.

Synchronous programming languages such as Lustre [3] and then SCADE[1] have proven their abilities to address the problems raised by critical embedded systems. The idea of ZÉLUS is to reuse what was learned from synchronous programming languages for the discrete part and to add ordinary differential equations (ODEs) to be able to modelize the continuous part of hybrid systems.

The kernel of ZÉLUS is a minimal Lustre-like synchronous language, based on the first-order part of Lucid Synchone [4], without clocks, in which data-flow equations can be mixed with ODEs with possible reset. The integration is carried out by Sundials[2], an external[3] numerical solver.

The ZÉLUS language also has hierarchical automata [6] that can be arbitrarily mixed with data-flow equations and ODEs.

The type system [7] of the language can statically distinguish discrete computations from continuous ones to ensure that signals are used in their proper domains. It also 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.

A semantics [5] based on non-standard analysis which gives a synchronous interpretation to the whole language, clarifies the discrete/continuous interaction and the treatment of zero-crossings, and also allows the correctness of the type system to be established.

The ZÉLUS language has been developed in the Parkas team by Marc Pouzet and Timothy Bourke.

Similar efforts have been made at Berkeley by Edwards A. Lee and Haiyang Zheng working on Ptolemy [9, 10].

Zero-crossing. A zero-crossing occurs when the graph of a function crosses zero from negative to positive. It is an event that can happen while the numerical solver is integrating functions (e.g., while the environement evolves). Using ZÉLUS's `up` construct, one can ask the numerical solver to watch for particular zero-crossings. When a watched zero-crossing occurs, a discrete step is triggered. Changes to the discrete state of the program can happen (e.g., the controller's program can act) at these discrete steps.
There are several possible definitions of such an event. For a given signal x, a zero-crossing could be triggered either when $x < 0$ and then $x \geq 0$, or when $x \leq 0$ and then $x > 0$, or when $x < 0$ and then $x > 0$. In ZÉLUS, zero-crossings are observed by Sundials which uses the first definition.

[1] `http://www.esterel-technologies.com/products/scade-suite/`
[2] `https://computation.llnl.gov/casc/sundials/main.html`
[3] using ML-Sundials: `http://www.di.ens.fr/ParkasSoftware#MlSundials`

2.2 MiniZélus

The work done during this internship is only concerned with hybrid ZÉLUS code, which can contain the up operator to watch for zero-crossings. This means we do not treat some parts of the language, such as the pre operator which can only be used in entirely discrete nodes ("nodes" is the Lustre, Lucid Synchrone, and now ZÉLUS name for what would be a function in other programming paradigms).

For clarity and simplicity reasons we only present the subset of the ZÉLUS language that we are working with. The BNF of this subset, which we will call MiniZélus, is presented in Code 1.

Code 1 MiniZélus' BNF

```
x ::= Variables (x, y, u, v, ...)
v ::= Values (42, 13.37, ...)
o ::= Operators (+, -, *, ...)
q ::= State name ("Foo", "Q51", ...)

e ::= x
    | v
    | <e> o <e>
    | "last" x

z ::= "up" <e>

h ::= <z> "->" <e> [ "|" <h> ]*

s ::= q
    | "do" x = <e> [ "and" x = <e> ]* "in" q

t ::= <z> "then" s [ "|" <t> ]*
    | <z> "continue" q [ "|" <t> ]*

u ::= "do" <E> "until" <t>

E ::= "init" x "=" <e>
    | "der" x "=" <e> [ "init" <e> ] [ "reset" <h> ]
    | x = "present" <h>
    | x = <e>
    | "automaton" ( "|" q "->" <u> [ "unless" <t> ] "done" )+
    | <E> "and" <E>
```

The meaning of the expressions presented in Code 1 are explained the in following table:

last x	Is the value of the left limit x.
up(e)	Watches for zero-crossings of the expression e continuously in time. It emits the value zero when a zero-crossing of e occurs.
init x = e	Gives to x the initial value of the expression e at the initial instant.

`x = present z_1 -> e_1 | ... | z_n -> e_n`	Resets the value of `x` to the value of the expression `e_i` when the zero-crossing `z_i` occurs.
`der x = e init e_0 reset z_1 -> e_1 | ... | z_n -> e_n`	Gives to `x` the initial value of the expression `e_0` and says that `x` evolves in time with respect to the differential equation $\dot{x} = e$. When the zero-crossing `z_i` occurs, the value of `x` is reset to the value of the expression `e_i` at the zero-crossing instant.
`x = e`	Gives to `x` the same value as `e` continuously in time.
`| Q -> do ... done`	Creates an automaton mode, named `Q`.
`unless t`	Is a guard which prevent entering the mode if a particular condition is verified, in that case it switches to another mode of the automaton (in which the unless guard is skipped).
`until t`	Creates a switch to other modes of the automaton.
`z continue Q`	Switches to mode `Q` when condition `z` (expressed as a zero-crossing) occurs, the value of the continuous variables are conserved.
`z then Q`	Switches to mode `Q` when condition `z` (expressed as a zero-crossing) occurs, the continuous variables are reset to the last value they had in state `Q`.
`z then do ... in Q`	Switches to mode `Q` when condition `z` (expressed as a zero-crossing) occurs, the continuous variables are reset to the last value they add in state `Q`, except if they are assigned a value in the `...` block.
`E_1 and E_2`	Puts the two expressions `E_1` and `E_2` in parallel.

2.3 Example of MiniZélus program

Our example of a MiniZélus program modelizes a possible evolution in time of the water tank hybrid system presented in Section 1.3 ("Hybrid systems", p. 4). In this instance, the water tank is initially empty, and we decide that the tap is turned on and off when the water level is mid-way respectively between $r2$ and $r2'$, and $r1$ and $r1'$. The corresponding MiniZélus code is presented in Code 2

Code 2 Modelof the water tank system when initially empty

```
    init x = 0.0
and automaton
    | On -> do
        der x = w - v
      until up(x - ((r1 + r1') / 2.0)) continue Off
      done
    | Off -> do
        der x = - v
      until up(((r2 + r2') / 2.0) - x) continue On
      done
```

3 Static analysis of cascades of discrete zero-crossings

During the execution of a ZÉLUS program, when a watched zero-crossing occurs, the integration phase stops (we exit the continuous mode), and a discrete step is executed. Time does not advance at the discrete step: it is supposed to be instantaneous as in classical synchronous programming.

Thus, it has been decided in ZÉLUS that only one discrete step can occur between two continuous phases. This choice is a way of guaranteeing that time always flows. However, this constraint could be relaxed if we could guarantee that there are only a finitie number of executed discrete steps between any two continuous phases.

This relaxed constraint could be useful for the design of programs: for instance if one is watching zero-crossings of a piecewise continuous signal, one may also want to catch "instantaneous" zero-crossings of the signal, happening at a discrete step in which the value of the signal suddenly becomes positive after having been negative at the end of the integration phase.

Such a relaxation of our necessary condition leads to greater expressivity, however it is only suitable if it maintains the safety property that the previous version achieved. The safety property we are concerned with is not having infinitely many discrete steps between two integration phases so that time keeps flowing (*time-must-flow* constraint).

In this section we suppose that we authorize watching for discrete zero-crossings in ZÉLUS. In this setting we try to find a way to statically determine if the *time-must-flow* constraint is still respected in the programs we can write.

3.1 Discrete/continuous control-flow graph

Since ZÉLUS programs are static in the sense that they cannnont dynamically build state at runtile, the *time-must-flow* constraint can't be violated if no variables depend instantly on itself. Since an integration phase takes time, it induces a delay that breaks the instant dependency cycle. So the idea is to do a dependency analysis of the variables. To this end, we build a graph representing the control-flow of the hybrid program. This graph have two types of vertices: continuous vertices, that correspond to the integration phases; and discrete vertices which correspond to discrete steps. The edges between vertices correspond either to zero-crossings, or to the resumations of integration after a series of discrete steps. When drawing such graphs, continuous vertices are represented by round boxes and discrete vertices by rectangular boxes.

If discrete zero-crossings are forbidden there will be exactly one edge coming out of each discrete vertex and it will go back to a continuous vertex (i.e., the next integration phase starts after a single discrete step). This means that every cycle in the graph trivially goes through a continuous vertex, since the obtained graph is bipartite.

When discrete zero-crossings are allowed, there can be egdes which come out of a discrete vertex and go to another discrete vertex. What is wanted then is to verify the absence of cycles that only go through discrete vertices in the graph, ensuring that there will never be an infinite number of discrete steps between two integration phases.

3.2 Building the discrete/continuous control-flow graphs

Figures 2 to 5 show the translation of `MiniZélus` equations to discrete/continuous control-flow graphs. For the sake of clarity, the `automaton` construct will be treated later. These graphs do not take discrete zero-crossings into account, they correspond to the second paragraph of the previous subsection. The changes that are needed to take discrete zero-crossings into account will be presented after basic composition has been explained.

Putting these pieces of `MiniZélus` code together is done using the "`and`" operator, as we have seen in Section 2.2 ("`MiniZélus` ", p. 6). The corresponding operation on graphs is a composition with the following rules:

- As we do not treat `automaton`, each continuous variable is uniquely defined (there is only one mode), so we can merge all the continuous vertices into one continuous vertex.
- Since all variables are initialized only once, the `init` graphs (Figure 2b) can be merged into one single discrete vertex and we connect its outgoing edge to the continuous vertex.

- The edge of each discrete vertex in the `present` graphs (Figure 4b) connects with the continuous vertex.

In order to take discrete zero-crossings into account, we have (at least) two solutions.

We can decide that ZÉLUS' `up` also watches for discrete zero-crossings. In this setting, each time we have an edge going from continuous vertex to a discrete vertex (i.e., which corresponds to a zero-crossing), we need to add edges to this discrete vertex from each other discrete vertex that assigns a value to a variable on which the zero-crossing depends (the variables appearing in an `up` expression). In practice, the edge count of the resulting graphs is so high that the existence of a discrete vertex cycle is almost guaranteed (and was obvious in the practical examples that we tested). This suggests that this approach cannot lead us to useful conclusions.

The other approach is to split `up` into two operators: one watching for continuous zero-crossing and the other watching for discrete zero-crossings, we will denote them respectively as `upc` and `upd`:

```
z ::= "upc" <e>
    | "upd" <e>
```

The language is then still as expressive as with the other choice (it is still easy to watch for both discrete and continuous zero-crossing of an expression) while allowing to avoid a considerable amount of useless edges in the graphs we are building. The following changes to the graphs are necessary:

- For each of the edges corresponding to an `upd` and only for them we do the same operation of adding edges between discrete vertices as we would be doing in the case of `up` handling both discrete and continuous zero-crossings.
- The edges from the continuous vertex to a discrete vertex have to be removed if the edge corresponds to an `upd`.

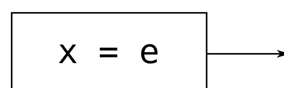
3.3 Examples of discrete/continuous control-flow graphs

An example of a hybrid program with discrete zero-crossings is presented in Code 3. What is happening in this example is the following:

- The variables are initialized: x is set to -3.0, y and z are set to -1.0 (this is considered a discrete step).
- The integration phase starts, y is constant, the derivatives of x and z are both 1.0.
- When z crosses zero (`upc(z)`), the integration phase stops and we enter a discrete step in which y is set to 3.0.
- Since y crosses zero during a discrete step (`upd(y)`), another discrete step is triggered in which x is set to 1.0.

```
init x = e
```

(a) `MiniZélus` code

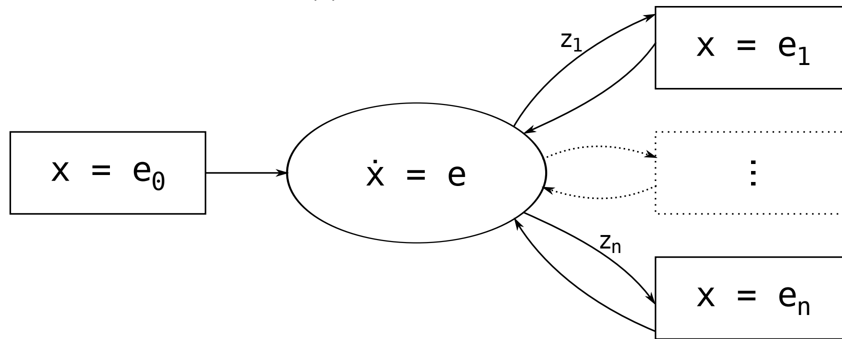


(b) Corresponding graph

Figure 2: `init`

```
der x = e init e_0 reset z_1 -> e_1 | ... | z_n -> e_n
```

(a) `MiniZélus` code

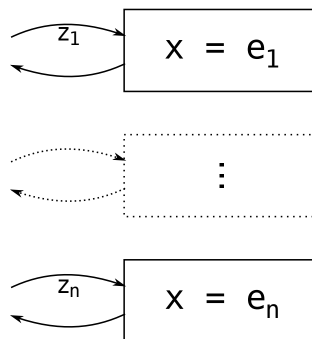


(b) Corresponding graph

Figure 3: `der`

```
x = present z_1 -> e_1 | ... | z_n -> e_n
```

(a) `MiniZélus` code

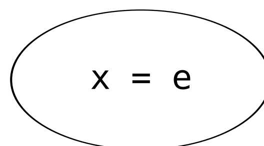


(b) Corresponding graph

Figure 4: `present`

```
x = e
```

(a) `MiniZélus` code



(b) Corresponding graph

Figure 5: `assignment`

- ◦ The integration phase resumes, starting with the new values of x and y.
- ◦ When $x-2$ crosses zero (`upc(x-.2.0)`), the integration phase stops and we enter a discrete step in which y is set to -2.0.
- ◦ Since $-y$ crosses zero during a discrete step (`upd(-.y)`), another discrete step is triggered in which x is set to -2.0.
- ◦ Since $-x$ crosses zero during a discrete step (`upd(-.x)`), another discrete step is triggered in which z is set to -1.0.
- ◦ The integration phases starts again, with the new values of x, y, and z.
- When z crosses zero (`upc(z)`), ...

The corresponding discrete/continuous control-flow graph is shown in Figure 6 and the evolution of the values of x, y, and z is plotted in Figure 7. In this example we can statically determine that there **will not** be any infinite cascade of discrete zero-crossings.

Code 3 Finite cascade of zero-crossings

```
    der x = 1.0 init -3.0 reset upd(y) -> 1.0
                                | upd(-. y) -> -2.0
and der y = 0.0 init -1.0 reset upc(z) -> 3.0
                                | upc(x -. 3.0) -> -2.0
and der z = 1.0 init -1.0 reset upd(-. x) -> -1.0
```

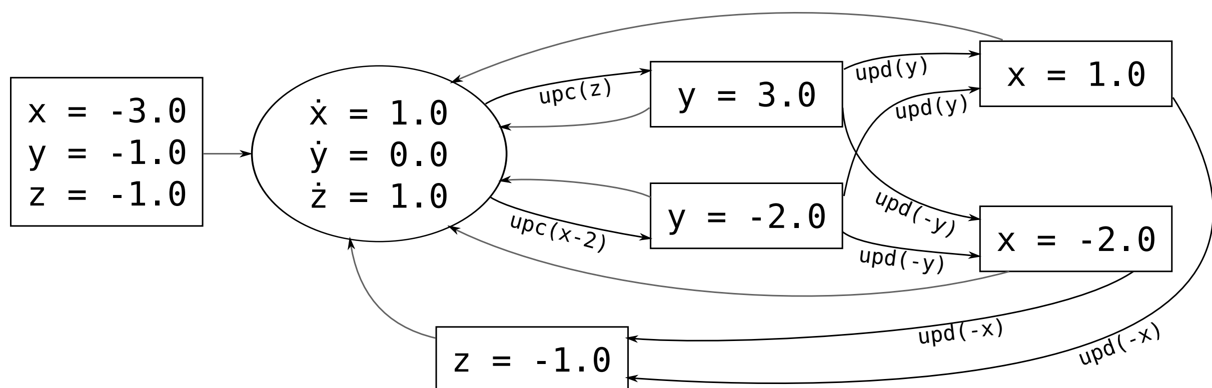


Figure 6: Discrete/continuous control-flow graph of Code 3

Another hybrid program, in which there is an infinite cascade of zero-crossings, is visible in Code 4. What is happening in this example is the following:

- ◦ The variables are initialized: x, y and z are set to -1.0.
- ◦ The integration phase starts, x and y are constant, the derivatives of z is 1.0.
- ◦ When z crosses zero (`upc(z)`), the integration phase stops and we enter a discrete step in which x is set to 1.0.
- Since x crosses zero during a discrete step (`upd(x)`), another discrete step is triggered in which y is set to 1.0.
- ◦ Since y crosses zero during a discrete step (`upd(y)`), another discrete step is triggered in which x is set to -1.0.
- ◦ Since $-x$ crosses zero during a discrete step (`upd(-.x)`), another discrete step is triggered in which y is set to -1.0.

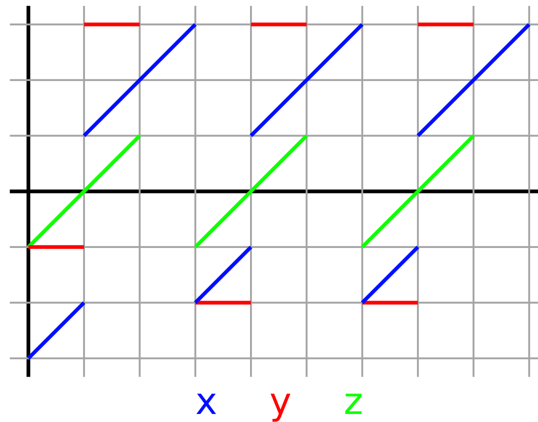


Figure 7: Plot of Code 3

- ◦ Since $-y$ crosses zero during a discrete step (`upd(-.y)`), another discrete step is triggered in which x is set to 1.0.
- Since x crosses zero during a discrete step (`upd(x)`), . . .

The corresponding discrete/continuous control-flow graph is shown in Figure 8. In this example we can statically determine that there **can** be an infinite cascade of discrete zero-crossings.

Code 4 Infinite cascade of zero-crossings

```
    der x = 0.0 init -1.0 reset upd(y) -> -1.0
                              | upd(-. y) -> 1.0
                              | upc(z) -> 1.0
and der y = 0.0 init -1.0 reset upd(x) -> 1.0
                              | upd(-. x) -> -1.0
and der z = 1.0 init -1.0
```

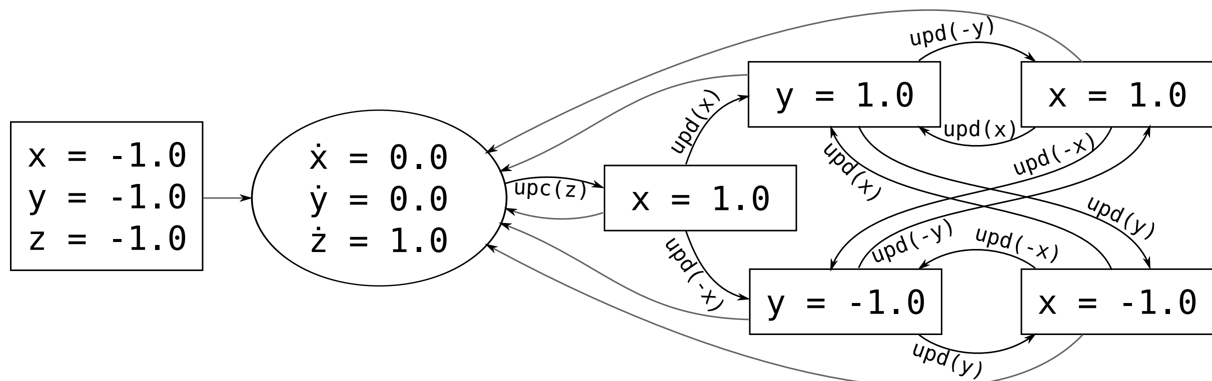


Figure 8: Discrete/continuous control-flow graph of Code 4

3.4 Adding support for automata

This static analysis can be extended to add support for Zélus' `automaton` feature. Each state of an automaton is independant and is internally (the `<E>` part of the definition of `u` in the BNF in Code 1) treated as we did with `MiniZélus` before adding automata.

What needs to be done is to connect to the states of an automaton between them and with the other equations at the same level as the automaton. The graphs for each state are connected using the following rules:

- For each state, we add a new discrete vertex called its "guard" discrete vertex. This vertex is empty (i.e., it does not correspond to any assignments) and is always where we enter the state. We add an edge from it to the continuous vertex of the state.
- The global (outside of the automaton) initialization discrete vertex is connected only to the "guard" discrete vertex of the first state of the automaton (which in ZÉLUS is implicitly the initial state).
- In each state, for each `unless` clause (see `t` in the BNF given in Section 2.2) we add an edge from the "guard" discrete vertex of the state to the "guard" discrete vertex of the state to which the clause would go (see `q` in the BNF).
- In each state, for each `until` clause (see `t` in the BNF):
  - If it is a `continue` transition or a simple `then` transition (the first case of `s` in the BNF), we add an edge from the continuous vertex of the state to the "guard" discrete vertex of the state to which the clause would go (`q` in the BNF).
  - In the other case (`then do ... in`), we create a discrete vertex corresponding to the assignments done in the transition and add two edges: one from the continuous vertex to the state to the new discrete vertex, and one from the new discrete vertex to the "guard" discrete vertex of the state to which the clause would go.

    If there are equations composed (using "`and`") with the automaton that watch for discrete zero-crossings that depends on the variables that are assigned a new value in the automaton transition, we also add an edge for each as we did before adding automata.

The same analysis of the presence of cycles of discrete vertices can be done on the obtained graph. It enables us to authorize more than one `unless` transition to be taken in a row before entering a new integration phase. This would for instance allow an automaton which can choose its initial state depending on the initial values of the variables, by making a finite non-cycling cascade of initial conditions check, stopping in the first state for which the initial condition is satisfied.

3.5 Results

Our static analysis of discrete zero-crossings is a sound over-approximation. In some cases it can guarantee that our *time-must-flow* constraint will be respected (the number of discrete steps between two integration phases is always finite). In other cases, it can only say that the constraint *may* not be respected. Indeed, there might be a cycle of discrete vertices in the graph while the numerical values in the program are such that there will not actually be an infinite number of discrete steps between two integration phases when the program is executed. For instance it would be the case for Code 4 ("Infinite cascade of zero-crossings ", p. 12) if all the reset values of y or x were of the same sign.

What we can do in the compiler is then to print a warning when there is a cycle of discrete vertices in the discrete/continuous control-flow graph, instead of stopping the compilation on an error. This warning could be turned into an error by an option when the compiler is called.

This analysis could be enhanced by taking statically known values, or at least their signs, in consideration. Doing this would help accepting more programs in which the dependency cycle exists but will never lead to an infinite cascade of discrete zero-crossings during the execution.

4 Working with hybrid automata in ZÉLUS

Hybrid automata [14] are a mathematical model for describing hybrid systems such as defined in Section 1.3 ("Hybrid systems", p. 4). A hybrid automaton is a finite state machine with a finite set of continuous variables whose values are defined by ODEs.

A lot of work regarding hybrid systems has been done within this particular formalism. For instance HyTech [15], an automatic tool for the analysis of hybrid systems using model checking, uses hybrid automata for hybrid systems specifications. Also, hybrid automata have been implemented in tools such as Scicos [19], a graphical dynamical system modeler and simulator.

Thus, it is interesting to see if any hybrid automaton can be translated into ZÉLUS in an automatic manner.

This work is not finished yet at the time this writing. The ultimate goal is to have an automatic way to translate any given hybrid automaton into ZÉLUS code. Here we present hybrid automata and then present which boolean expressions can be expressed in terms of zero-crossings, which is one of the problems raised by the translation of hybrid automata to ZÉLUS code. After that, a roadmap of what is left to do to acheive our goal is proposed.

4.1 What is a hybrid automaton

The purpose of a hybrid automaton is to modelize all the possible behaviors of a hybrid system. The discrete state of the system is modelized by the states (called *control modes*) of the automaton, its continuous state is modelized by variables in $\mathbb{R}$. Discrete changes of the system state happen during the transitions (called *control switches*) of the automaton. Continuous changes happen in the states of the automaton and are modelized by differential equations.

4.1.1 Formal definition

A **hybrid automaton** H consists of the following components (definition derived from the one given by Henzinger in [14]):

Variables. A finite set $X = \{x_1, ..., x_n\}$ of real-numbered variables. Let $\dot{X}$ be the set $\{\dot{x}_1, ..., \dot{x}_n\}$ of dotted variables that represent first derivatives during continuous change. and let X' be the set $\{x'_1, ..., x'_n\}$ of primed variables that represent values at the conclusion of discrete change.

Control graph. A finite directed multigraph (V, E). The vertices in V are called *control modes*. The edges in E are called *control switches*.

Initial, invariants, and flow conditions. Three vertex labeling functions $init$, inv, and $flow$ that assign to each control mode $v \in V$ three predicates. Each initial condition $init(v)$ is a predicate whose free variables are from X. Each invariant condition $inv(v)$ is a predicate whose free variables are from X. Each flow condition $flow(v)$ is a predicate whose free variables are from $X \cup \dot{X}$.

Jump conditions. An edge labeling function $jump$ that assigns to each control switch $e \in E$ a predicate. Each jump condition $jump(e)$ is a predicate whose free variables are from X.

Events. An edge labeling function $event$ that assigns to each control switch $e \in E$ a finite set of assignments to the variables in X'. The values which are assigned may depend on the values of variables in X.

4.1.2 Semantics

The execution of a hybrid automaton is an alternance of continuous phases, in control modes, and discrete steps, at control switches.

For each control mode $v \in V$, the $init(v)$ predicate must be satisfied if v is the initial control mode of the automaton ($init(v) = false$ for control mode which cannot be initial).

In a control mode v, the continuous state evolves according to differential equations defined by the predicate $flow(v)$. During this phase the predicate $inv(v)$ must be satisfied.

In a control mode v, any control switch $e = (v, w) \in E$ to another control mode w might be taken at any time while the predicate $jump(e)$ is satisfied. In that case, the variables in X' are assigned values according to $event(e)$ and then the w control mode is entered with an initial value x_i' for each variable x_i.

We call **trajectory** the evolution in time of the state of a hybrid system. A trajectory is said to be *valid* with respect to a hybrid automaton if it is a possible execution of the hybrid automaton:

- It must satifies the initial condition described by the $init$ predicate of the control mode corresponding to the initial discrete state of the trajectory ;
- The continuous state must evolve according to the ODEs described by the $flow$ predicate of the control mode corresponding to the current discrete state ;
- The invariant condition described by the inv predicate of a control mode must be satisfied while the discrete state of the trajectory corresponds to this control mode ;
- Control switches must be taken when the corresponding $jump$ condition is satisfied ;
- Discrete changes at discrete steps must respect the assignments done in the $event$ of the corresponding control switch.

4.2 An example of hybrid automaton

As a simple example of hybrid automaton we will model the water tank (this example is inspired from the *Lecture Notes on Hybrid Systems* [17] by Lygeros) from Section 1.3 ("Hybrid systems", p. 4).

We can model this system, which is represented on Figure 1 ("Illustration of the water tank hybrid system ", p. 4), using the following hybrid automaton:

- The set of real-numbered variables: $V = \{x\}$ (so we have $\dot{V} = \{\dot{x}\}$ and $V' = \{x'\}$).
- The control graph: $G = (\{On, Off\}, \{(On, Off), (Off, On)\})$.
- The inititial conditions:
  - $init(On) = x \leq r1' \wedge x \geq r2'$
  - $init(Off) = false$
- The invariant conditions:
  - $inv(On) = x \leq r1'$
  - $inv(Off) = x \geq r2'$
- The flow conditions:
  - $flow(On) = \dot{x} = w - v$
  - $flow(Off) = \dot{x} = -v$
- The jump conditions:
  - $jump((On, Off)) = x > r1$
  - $jump((Off, On)) = x < r2$
- The events:
  - $event((On, Off)) = x' \leftarrow x$
  - $event((Off, On)) = x' \leftarrow x$

The corresponding hybrid automaton is drawn in Figure 9.

4.3 Expressing boolean expressions using zero-crossings

In ZÉLUS, execution of a discrete step is always triggered by a zero-crossing, as it is the only discrete event the solver observes during the integration phase. This means that the transition of a ZÉLUS automaton from one mode to another is also triggered by a zero-crossing.

In classical synchronous programming languages, the transitions between an automaton's modes depend on boolean conditions. In hybrid systems formalisms such as hybrid automata [14], the transitions between control modes are boolean predicates over the values of the real variables representing the continuous part of the state of the system.

Thus, it is interesting to study which boolean expressions we can express in terms of zero-crossings.

4.3.1 ZÉLUS' `up`

The ZÉLUS operator to watch for zero-crossing is `up`. The `up` operator takes a continuous signal and emits the value `zero` when the signal crosses zero.

The obvious idea of translation the boolean expression b in `if` b `then 1 else -1` is not practically usable, since solver such as Sundials are not designed to detect such steep signals. They are however very efficient with smooth curves using numerical methods, such as Newton's method, to compute the precise instant at which the zero-crossing occured.

If x and y are two continuous variables, $x - y$ is negative when $x > y$, crosses zero exactly when $x = y$, and positive when $x < y$.

Thus, the ZÉLUS expression `up(x -. y)` will emit a `zero` when the boolean expression $x = y$ is true. At the same moment, the boolean expression $x < y$ *becomes* true, and stays true until $y < x$ which will be detected the same way by the ZÉLUS expression `up(y -. x)`.

4.3.2 Disjunction

In ZÉLUS it is easy to watch for multiple zero-crossings in parallel, but in case we want to compose the disjunction with other zero-crossings, we need to be able to emit `zero` when any of two `up`s emit `zero`. This can be done using the ZÉLUS code visible in Code 5.

In Code 5 we use `up(0.0)` as an `up`-expression which will never emit `zero`. The keywords `let hybrid` are the ZÉLUS way of declaring an hybrid node. The `rec` keyword signals that the definitions are mutually recursive.

4.3.3 Conjunction

One difficulty we encounter here is that the "and" needs a memory: when a zero-crossing of one of the two operands happens, we need to be able to check whether the boolean expression represented

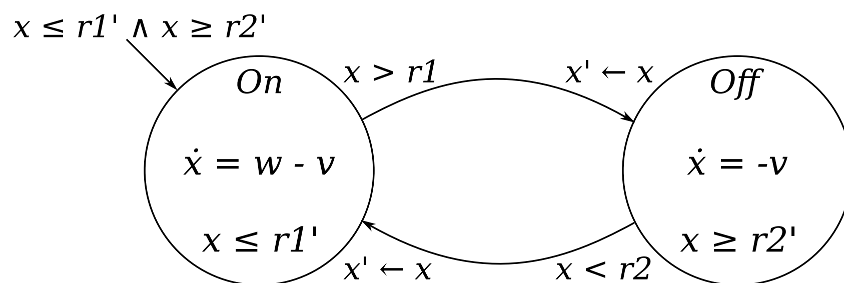


Figure 9: The water tank system

Code 5 $c1 \vee c2$

```
let hybrid zc_or (c1, c2) = zc where
  rec t1 = present c1 -> true
              else false
  and t2 = present c2 -> true
              else false
  and zc = if t1 then c1
      else if t2 then c2
      else up(0.0)
```

by the other one is true. This means that we also need to be able to know when the boolean expressions that the operands are representing become false.

When the boolean expressions that are represented by both operands stay true for a period of time (such as $x < y$), this can be done using the ZÉLUS code visible in Code 6. If one of them is only true at a precise instant (such as $x = y$), then we can use Code 7

Code 6 $c1 \wedge c2$ when c1 and c2 are true for a period of time

```
let hybrid zc_and (c1beg, c1end, c2beg, c2end) = ok where
  rec t1 = present c1beg -> true
                 | c1end -> false
           init false
  and t2 = present c2beg -> true
                 | c2end -> false
           init false
  and ok = if t1 then c2beg
      else if t2 then c1beg
      else up(0.0)
```

Code 7 $c1 \wedge c2$ where c1 is true for a period of time and c2 is true at a precise instant

```
let hybrid zc_and2 (c1beg, c1end, c2) = ok where
  rec t1 = present c1beg -> true
                 | c1end -> false
           init false
  and ok = if t1 then c2
      else up(0.0)
```

In both Code 6 and 7 the arguments `c1beg` and `c2beg` are the up-expressions detecting the moments when the boolean expressions $c1$ and $c2$ become true, and `c1end` and `c2end` are the ones when they become false.

4.3.4 Conclusion

The boolean expressions we can express using ZÉLUS' up obey the grammar described in Code 8.

We do not need to be able to express the moment when an expression such as $c1 \wedge c2$ or $c1 \vee c2$ becomes false because every boolean formula can be expressed as a disjunction of conjunctions.

An example result of zero-crossing encoding of a boolean expression is provided in Code 9.

Code 8 Boolean expressions expressable using zero-crossings

```
x ::= Variables (x, y, u, v, ...)
v ::= Values (42, 13.37, ...)
o ::= Operators (+, -, *, ...)

e ::= x
    | v
    | <e> o <e>

C ::= <e> < <e>
    | <e> > <e>
    | <e> = <e>
    | <C> ∧ <C>
    | <C> ∨ <C>
```

Code 9 Encoding of $(x < 1.0 \wedge y > 2.0) \vee x > 3.0$

```
let hybrid bool_expr (x, y) = ok where
  rec c1 = up(x -. 1.0)
  and nc1 = up(1.0 -. x)
  and c2 = up(2.0 -. y)
  and nc2 = up(y -. 2.0)
  and c3 = up(3.0 -. x)
  and ok = or_cond(and_cond(c1, nc1, c2, nc2), c3)
```

4.4 From hybrid automata to ZÉLUS code: propositions

The translation of hybrid automata into ZÉLUS code raises several problems. The expression of boolean conditions in terms of zero-crossings has been addressed in the previous subsection. The non-determinism of hybrid automata is another hurdle.

Contrary to hybrid automata the purpose of ZÉLUS is not to modelize all possible behaviors of a hybrid system. Since it is a programming language, ZÉLUS is used to program hybrid systems: the continuous parts of the system are still modelized but the discrete part is *controlled*. In other words, the non-deterministic aspect of hybrid automata, which is essential to their semantics and their purpose, does not exist in ZÉLUS. However, it would be "dishonest" to make this translation a mere choice of a valid trajectory with respect to the hybrid automaton being translated.

For this reason the ZÉLUS code resulting from the translation should depend on an oracle. The oracle would be used to be able to execute all the possible valid trajectories for a given hybrid automaton. By choosing any valid instant to take a control switch, it could select a trajectory completely randomly or using a particular strategy. For instance a possible strategy would be to always leave a control mode as soon as a jump condition is satisfied. Using an oracle that we can control would allow the study of all possible evolutions of an hybrid system. It would make ZÉLUS a powerful tool to actually execute and study possible trajectories of a given hybrid automaton, in order to deduce properties of different evaluation strategies.

Our static analysis of cascades of discrete zero-crossings can also help the translation of hybrid automata since it allows to chain `unless` transitions in ZÉLUS' automata to choose the right initial state according to the initial values of the real-numbered variables. Instead of choosing and hard-coding one of the initial states of the hybrid automaton, the ZÉLUS automaton can choose its initial state depending on the initial values of the variables, by making a finite non-cycling cascade of initial

conditions check, stopping in the first state which for which the initial conditions described by the corresponding $init$ predicate are satisfied.

5 Conclusion

We found a discrete analysis of ZÉLUS code which permits verifying that time will not get stuck when we relax constraints on the language to allow multiple discrete steps to happen between two continuous phases.

We also started to investigate the translation of hybrid automata into ZÉLUS code with the purpose of enabling ZÉLUS as a tool to study hybrid automata evaluation startegies and properties of hybrid systems described using the hybrid automata formalism.

In addition to these two works, this internship was the occasion to dive into the research going on around hybrid systems modelers. This research field is still in very early stage, and there are plenty of questions that need answers. Regarding ZÉLUS specifically, several points would need to be addressed:

- *Typing discrete and continuous-time signals.* The mix of continous-time and discrete-time signals must be done such that the behavior does not depend on internal decisions made by the solver. It essentially amounts at separating expressions into three kinds: combinatorial, discrete or continuous. An interesting question is the definition of a more expressive expressive type system able to distinguish piece-wise continuous, piece-wise constant, discrete time signals and periodically sampled signals. The existing clock calculus of synchronous languages (e.g., the one of Lucid Synchrone used in SCADE 6) is a good basis to start with.
- *Combination of solvers.* Existing modelers use a single numerical solver for approximating continuous trajectories. Combining several solvers is the right way to achieve both precision and fast simulation for the whole system. The problem is reminiscent to the problem of automatic code distribution for synchronous languages. We propose to consider language annotations in order to define parts which are approximated by the same black-box solver, to study causality constraints that must be verified between them and to propose a semantics that deal with multi-solvers.
- *Semi explicit DAEs.* Tools like Modelica can manage more general implicit (or acausal) models defined by Differential Algebraic Equations (DAEs). A particular class are semi-explicit DAE made of an ODE and an algebraic constraint. The goal is to study the combination of semi-explicit DAE with control structures (hierarchical automata) to express modes. This raises semantical issues (what is the semantics of the whole language), compilation issues (how to prepare the code so that it can be linked with existing black-box solvers) and optimization issues.

References

[1] G. Kahn. *The Semantics of Simple Language for Parallel Programming*, IFIP Congress, 1974

[2] P. Caspi, M. Pouzet. *Synchronous Kahn networks*, ICFP 1996

[3] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice. *Lustre: A Declarative Language for Programming Synchronous Systems*, POPL 1987.

[4] P. Caspi, M. Pouzet. *Lucid Synchrone: une extension fonctionnelle de Lustre*, JFLA 1999

[5] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. *Non-Standard Semantics of Hybrid Systems Modelers*, JCSS 2012

[6] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. *A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code*, EMSOFT 2011

[7] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. *Divide and recycle: types and compilation for a hybrid synchronous language*, LCTES 2011

[8] A. Benveniste, B. Caillaud, and M. Pouzet. *The Fundamentals of Hybrid Systems Modelers*, CDC 2010

[9] E. A. Lee, and H. Zheng. *Operational Semantics of Hybrid Systems*, HSCC 2005

[10] E. A. Lee, and H. Zheng. *Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems*, EMSOFT 2007

[11] J.-L. Colaço, B. Pagano, and M. Pouzet. *A conservative extension of synchronous data-flow with state machines*, EMSOFT 2005

[12] P. Caspi, and M. Pouzet. *A Co-iterative Characterization of Synchronous Stream Functions*, ENTCS 1998

[13] P. Cuoq, and M. Pouzet. *Modular Causality in a Synchronous Stream Language*, ESOP 2001

[14] T.A. Henzinger. *The Theory of Hybrid Automata*, LICS 1996

[15] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. *HyTech: A Model Checker for Hybrid Systems*, Software Tools for Technology Transfer 1:110-122, 1997

[16] G. Berry. *The constructive semantics of pure Esterel. Draft Version 3*, `http://www-sop.inria.fr/meije/Personnel/Gerard.Berry.html`, 1999

[17] J. Lygeros. *Lecture Notes on Hybrid Systems*, Dept. of Electrical and Computer Engineering, U. of Patras, 2004

[18] E. Ábrahám. *Lecture Notes: Modeling and Analysis of Hybrid Systems*, Faculty of Mathematics, Computer Science, and Natural Sciences RWTH Aachen University, 2012

[19] M. Najafi, and R. Nikoukhah. *Implementation of Hybrid Automata in Scicos*, IEEE Multi-Conference on Systems and Control, 2007