Hybrid Systems and Contracts with Zélus and DynIbex Zeldyn: a Compilation and Verification Toolchain

François Pessaux, Julien Alexandre Dit Sandretto, Alexandre Chapoutot

ENSTA Technical Report

April 2022

Hybrid Systems and Contracts with Zélus and DynIbex

Zeldyn: a Compilation and Verification Toolchain

François Pessaux Julien Alexandre dit Sandretto Alexandre Chapoutot

April 8, 2022

Abstract

Modeling cyber-physical systems is known to be a challenging task. Model-based programming frameworks for CPS are an appealing approach to tackle this challenge, by providing languages close to engineers. However, these tools usually lack formal semantics, causing simulation results to vary from one version to another. A second particularity in the design of CPS is the presence of uncertainties due to parameters not well known at design-time. Monte-Carlo methods are classically used to circumvent this issue but they do not provide a complete coverage. Another solution is to rely on set-based simulation methods. On the one hand, control command systems find an elegant programming solution in synchronous languages. Zélus is such a language extended to deal with CPS, with a well-defined semantics. On the other hand, many set-based simulation libraries exist with a lack of proper front-end to make them suitable for industrial use. We propose to bridge the gap between set-based simulation and development frameworks for CPS by designing a new backend and runtime for Zélus, using set-based methods and guaranteed arithmetic. This runtime is described by very simple set-based primitives in order to be implemented in any set-based simulation library. An instance of the runtime is given with the DynIbex library to prove the effectiveness of the proposed approach. Finally, a mechanism of contracts is added to state and verify properties on the simulated system.

1 Introduction

Hybrid systems are commonly defined as dynamical systems mixing discrete events and continuous-time. They are widely present in control command systems, or more generally in cyber-physical systems (CPS), where a continuous physical process is controlled or monitored by software components which run at discrete instants. Designing CPS is a challenging process as the coupling of continuous and discrete components may generate subtle behaviors. Model-based programming frameworks for CPS are an appealing approach to tackle this challenge [12]. Many industrial development frameworks exist (such as Modelica, Simulink/Stateflow or LabVIEW) which bring a solution for end-users by considering an input language close to engineers. An important limitation of these tools is their lack of formal semantics. Hence the simulation behaviors of a model in these tools may vary from one version to another.

A second particularity in the design of CPS is the presence of uncertainties. Indeed, models of physical processes, as differential equations, rely on parameters which are usually not well known at design-time. These parameters may have an influence on the whole CPS behaviors. A classical solution to address these uncertainties is to apply Monte-Carlo methods which may be time consuming and do not provide a complete coverage. Another approach, used in the present work, is to use set-based simulation methods, as in tools such as [21]. Set-based simulation propagates sets of values instead of points in order to collect all the possible system's behaviors at once. This specialized approach makes possible to perform worst-case analysis of systems, which is useful, for example, to prove safety properties. The main drawback of these approaches is that the input language associated to these tools is generally not suitable for engineers as its relies on mathematical modeling such as hybrid automata.

On the one hand, control command systems find an elegant programming solution in synchronous languages [20] but relies on floating-point simulation engine. Synchronous programming approach gives a formal

semantics of embedded software.Zélus [10] is such a programming language, extended to deal with CPS, hence is an interesting candidate for input language. On the other hand, many set-based simulation libraries exist such as [2, 13, 1, 14] with a lack of proper front-end to make them suitable for industrial use.

Our main objective is to develop a complete toolchain, from the (high-level) model of a CPS to the verification of its properties, even if uncertainties occur. For this, we selected Zélus for its formal semantics and its open source compiler, and DynIbex for its set-based approach allowing to verify constraints under uncertainties.

The contribution of this article is then to bridge the gap between set-based simulation and development frameworks for CPS.

In details, the proposed approach extends the Zélus compiler in order to have two runtimes for Zélus programs: one with a floating-point simulation engine and one with a set-based simulation engine. The set-based runtime is described by very simple set-based primitives, as those given in interval analysis, in order to be implemented in any set-based simulation library. A simple instance of the set-based runtime is given with the DynIbex library to prove the effectiveness of the proposed approach, relying on techniques very similar to existing ones in various reachability frameworks. Indeed, DynIbex has the particularity of offering constraint verification and set-based simulation in a single tool, which will be useful for contracts verification on CPS.

Related Work

The mechanisms used for event detection and handling are rather similar to the ones used in other works. Several frameworks exist for reachability analysis of hybrid systems, though few have a high-level programming language in which to directly describe the systems.

JuliaReach [22] is a Julia library performing set-based reachability analysis on both linear and non-linear systems. The description of the system to analyze has to be manually encoded in Julia using the tools provided by the library.

CORA [2] is a toolbox written in MATLAB providing advanced data-structures and reachability algorithms for linear and non-linear systems. It has been extended with intervals and Taylor models [3, 4]. The modeling of a system is manually done in MATLAB, hence considering floating-point arithmetic as exact.

SpaceEx [16] is a verification platform to model linear (or piece-wise linear) hybrid systems and compute the sets of reachable states using different reachability algorithms. It relies on floating-point arithmetic, though it fails to account for rounding errors. Systems are modeled in a dedicated interface (or in a XML file). A graphical WEB interface is available to set parameters, run simulations and visualize the results. A translator from a subset of Simulink to SpaceEx is also available [27].

FLOW* [13] allows one to model non-linear hybrid systems with uncertainties and compute an over-approximation of reachable states using Taylor models and guaranteed floating-point arithmetic.

Hyson [9] allows one to perform set-based simulations of Simulink hybrid models (continuous and discrete, linear and non-linear, using a subset of the language). It provides guaranteed integration based on Runge Kutta methods with handling of floating-point rounding errors. It can be considered as an ancestor of this work, without automata and extensions of Runge Kutta methods provided by DynIbex. The event detection is also performed by checking the sign of the guard. To refine the event location, an interpolation of the guard is used instead, which may be more efficient than our simple bisection mechanism. However, such a technique would require the automated generation of this interpolation at compile-time.

dReal [18, 19] encodes hybrid systems as SMT modulo ODEs problems. A system has to be encoded as a first-order logic formula with the properties it must respect in the SMT-LIB format. It is then able to check if the properties of the system hold. However, it does not provide high-level constructs to write the systems.

The Acumen [34, 24] framework provides ways to express various kinds of hybrid systems in a Domain Specific Language. It allows point-wise simulations to provide very fancy dynamic visual representations. It also provides an enclosure interpreter supporting intervals to handle uncertainties for system verification

purposes. The main differences with our work come from the nature of the input language and the integration mechanism. Using Zélus provides various static analyses and advanced constructs. Though we do not handle some of these constructs here, work is underway to address a larger subset of the Zélus language, including hierarchical automata, and thus model more complex systems. DynIbex was chosen as a simulation framework as it provides guaranteed integration methods which handle floating-point issues.

In [26], the fine detection of event is performed in a very similar way than we do. Especially, it performs a bisection on time for each box crossing the guard. Then a contraction is performed to reduce the boxes according to the guard condition. However, the reset is applied on each box containing an event before running the simulation with the new dynamics.

The notion of tree is also present in [24] to compute enclosures of solutions and initial conditions to restart a simulation over multiple time extents where a guard intersects the dynamics.

[33] develops an assume-guarantee contracts framework to check invariance properties. Contracts are attached to components of a system and can be combined (in cascading or feedback modes). Contracts are given by hypotheses on the inputs and guarantees on both the state and output of the system. Both hypotheses and guarantees are represented by set inclusions. This allows to synthesize the global properties of the system from the ones of its sub-systems. Such a compositional approach is not feasible in our case since the final model obtained after the compilation by Zélus is flat. Hence, we verify at runtime, properties on the whole system at once.

[7] proposes to model hybrid systems using the Heterogeneous Rich Component to achieve contract-based design. The model is not coded in a programming language. Contracts hypotheses and guarantees are represented by constraints (set membership or inequalities). Components are connected by ports and combination rules are given for cascading and feedback connections. The verification of the contract satisfaction of the whole system is achieved by reachability analysis on the obtained state machine using the ARIADNE tool[6]. Contrary to our approach, the verification is performed by an external tool while it is done by the internal runtime on our side.

In [30], extending [29], a component-based modeling and verification formalism with properties expressed in differential dynamic logic are presented. Contracts are set on components which can be combined. Some compatibility proof obligations must be done to ensure that the combined components provide system-wide safety properties if their contracts hold. Two kinds of contracts allow to state interesting safety properties: change contracts to relate previous and current values of a variable ; delay contracts to deal with the delay between information exchange between components. This widens the kind of properties that can be verified. Contracts of modeled components can be verified using KeYmaera X [17].

[32] presents a library extending Modelica to allow verification of requirements on models. The design of this library is based on the FORM-L requirement modeling language [31], featuring temporal logic operators. The obtained framework can be used as a high-level (and graphical) systems modeling tool that can simulate models and check their compliance with stated properties. Thanks to the temporal logic operators, it is possible to express more complex contracts than in our current work. Simulations are performed in a point-wise manner with non-guaranteed arithmetic, so dealing with uncertainties has to be manually carried out by running several simulations.

CE2E [15] is a tool allowing to verify properties on hybrid models with non-linear dynamics, transitions and resets. Properties are bounded time linear invariants from linear bounded initial sets. Models can be built using a GUI or by importing from Stateflow. The verification algorithm builds, from an initial set, a cover and then determines if it is safe or not. If the cover cannot be classified in one of the two classes, it is recursively refined. At each step a simulation is generated, using guaranteed tools (VNODE-LP or CAPD).

Our compilation process shows correct results on several examples, however in the future, it is important to provide a correctness theorem to strengthen the confidence in the tool. This formal topic is obviously important since it determines the confidence one may have in the obtained simulation results. Various approaches may be adopted to get a formal confidence. One possibility is to model, in a theorem prover, the semantics of Zélus, then to implement the runtime in this framework and prove preservation of the semantics between any Zélus program and its simulation through the transforms performed by the compilation. The compiler and runtime could then be automatically extracted if the theorem prover provides such a mechanism.

Such an approach is used in [36] or, in another domain, in [25].

Another approach, used in [5] consists in modeling the physical environment, the physical system, its behavior and the software components in a logical framework. Then proofs of correct behavior can be carried out by the user on the basis of this modeling. In such an approach, a new proof has to made for each system but the code of the controller can automatically be extracted.

Differently, [8] proposes to strengthen the confidence in a controller by applying a pipeline of transforms using several theorem provers or formally proven software bridges between tools (some of them forming a trusted base). In this framework, a CPS is modeled in differential dynamic logic and the various compilation/transformation steps produce *in fine* a verified controller executable guarded by a proven safe fallback controller.

The rest of the paper is organized as follows. In Section 2, we briefly present Zélus and the features we handle. Section 3 provides a quick introduction to DynIbex and demonstrates how to encode and simulate a differential equation with an initial value using the library. Section 4 addresses the issues of simulating automata with intervals, the simulation runtime and its related automaton architecture. The compilation schema is presented in Section 5. In Section 6, some experimental results are given. Contracts are studied in Section 7. Finally, we conclude and comment on possible further works in Sections 8 and 9.

2 Zélus Succinctly, Used Features

Zélus [10] is a synchronous programming language extended with ordinary differential equations (*ODEs*). It provides a wide range of features like synchronous dataflow equations, hierarchical automata, signals, data-types, pattern-matching, functional features, etc. In this paper, we will only address the constructs required to implement simple hybrid systems having several dynamics. Zélus allows to describe hierarchical automata (*i.e.*, nested) which are not considered in this work. Zélus makes it possible to model systems in which there is an interaction between discrete-time and continuous-time dynamics. Currently we do not address arbitrary discrete-time behavior (i.e. reactions to event cannot be arbitrary Zélus code: only jumps involving expressions without side-effects, conditional, pattern-matching, discrete-side variables, etc) so we consider a class of continuous-time dynamical systems with discontinuity.

A program in Zélus is a hierarchy of *nodes*, possibly parameterized, containing equations relating the inputs and outputs of each node. A node can be instantiated in another one to import the equations of the former in the latter, where parameters are replaced by the effective arguments provided at the instantiation point. This mechanism allows the reuse of sets of equations with different parameters. The equations of a node are those defined in it and those imported from instantiated nodes.

A system with several dynamics is represented by an automaton with states containing systems of coupled equations and transitions between states, triggered by events. By convention in Zélus, an event related to a transition arises when its *guard* expression e changes from negative to positive values (*zero-crossing*). Note that a node may contain equations outside states: we call them *toplevel equations*. The initial state of an automaton is the first one in the **automaton** construct.

We consider three kinds of equations. An ODE **der** x $=e_1$ **init** e_2 is defined by $\dot{x} = e_1$ with the initial condition e_2 (only *Initial Value Problems* are considered in Zélus). If **der** x $=e_1$ **init** e_2 belongs to a state s, each time s is entered the value x is set to the value of the expression e_2, causing a jump. A regular dataflow equation x $=e$ binds `x` to an expression. Finally, we handle initialization equations **init** x $=e$ allowing to provide an initial value to an equation, which makes possible to omit later the initial value of an ODE in a mode if it does not need to be reset when entering the mode.

Let us now present the example which will also be used to illustrate the final generated code. This example is intentionally simple to allow one to recognize its structure in the produced code. In particular, it is not a benchmark to compare the accuracy or performances of our framework with other reachability analysis tools. The model of a simplified rocket launched vertically and burning its fuel until exhaustion can be described in Zélus by the automaton of Figure 1, containing three states. The ODE describing the evolution of the power is global to the automaton. In the state `EngOn` the fuel is used to contribute to the elevation. When the power becomes close to 0, within an ϵ (for instance 0.001) representing the inaccuracy

```
let hybrid main () = zpos where
  rec init zpos = 0.0
  and init speed = 0.0
  and der power = -. 2.0 *. power init 100.0
  and g = -9.81
  and automaton
    | EngOn -> do
        der speed = g +. power
        and der zpos = speed
        until up (-. (power -. 0.001)) then EngOff
    | EngOff -> do
        der speed = g
        and der zpos = speed
        until up (-. zpos) then Crashed
    | Crashed -> do
        der speed = 0.0
        and der zpos = 0.0
        done
   end
```

Figure 1: Model of a simple rocket in Zélus

of the sensor (`up (-. (power -. 0.001))` meaning that $-(power - 0.001)$ goes from negative to positive, hence *power* decreases to 0.001), the automaton goes into the state `EngOff` where no more fuel is available. The equation of the speed is then different from the one in the state `EngOn`. Finally, when the altitude reaches 0 the rocket stops moving. In this model, no ODEs in the states are reset when entering their state and no node instantiation is present. Note that floating-point arithmetic operators are suffixed by a dot in Zélus.

3 DynIbex in a Few Words, Used Features

DynIbex [1] is a C++ library that builds on the Ibex library. Ibex provides tools to develop programs for non-linear constraint processing over real numbers using interval arithmetic and affine arithmetic. DynIbex adds validated numerical integration methods (including handling of floating-point rounding errors), internally computing using zonotopes. To describe a dynamical system, one defines objects of predefined classes to represent the initial values and the (possibly non-linear) ODEs of the system, using a vector-valued representation. That is, initial values are a vector of intervals and the ODEs are "merged" into one unique function whose domain and codomain are vectors of intervals. Note that intervals may have infinite endpoints but in practice, to get significant results one prefers to use bounded intervals. Once these objects are defined, a dedicated function is called to perform the simulation with the desired parameters (integration method, duration, precision, etc.). The representation of an automaton has to be manually encoded which will motivate the creation of a generic runtime described in Section 4.

In Figure 2, we show how to encode the simulation of a simple system of two equations $\dot{s} = c$ and $\dot{c} = -s$ with the initial values 0 and 1, whose solutions are sine and cosine.

The variable `dim` represents the number of equations of the system, `y` represents the continuous state of the system, `ydot` encodes the differential equations.

```
#define T0 (0.0)
#define TEND (6.0)
#define TOL (1e-8)
#define METH (HEUN)

int main () {
  const int dim = 2 ;
  Variable y (dim) ;
  IntervalVector yinit (dim) ;
  Function ydot = Function (y, Return (y[1], -(y[0]))) ;
  yinit[0] = Interval (0.0) ; yinit[1] = Interval (1.0) ;
  ivp_ode problem = ivp_ode (ydot, T0, yinit) ;
  simulation simu =
    simulation (&problem, TEND, METH, TOL) ;
  simu.run_simulation () ;
  simu.export_y0 ("export.dat") ;
  return 0 ;
}
```

Figure 2: Code for ODEs simulation in DynIbex

The mapping from the coupled equations s and c to the vector-valued representation assigns s to the dimension 0 (`y[0]`) and c to the dimension 1 (`y[1]`). All the numerical constants are transformed into trivial proper rounded intervals (*i.e.*, the smallest interval containing the translated float). The initial conditions of the problem are stored in `yinit`. An IVP (*Initial Value Problem*) object `problem` is created to group the initial values, the equations and the initial time. A simulation object `simu` is created and run. Finally, the results are exported as plain text, providing for each time interval of the simulation the intervals representing the solution of each equation. Such an encoding will be the basis for automata simulation using DynIbex.

4 Simulation Runtime and Automaton Architecture

4.1 Notations

An interval $[x] = [\underline{x}, \overline{x}]$ defines the set of reals x such that $\underline{x} \leqslant x \leqslant \overline{x}$. Intervals with infinite endpoints are possible but will lead in practice to deteriorated results. We denote by $\mathbb{IR}$ the set of intervals on reals. A *box*, $[\mathbf{x}] \in \mathbb{IR}^n$, is the Cartesian product of n intervals of reals. The interval union of two intervals is defined as $[\underline{x_1}, \overline{x_2}] \cup [\underline{y_1}, \overline{y_2}] = [\min(\underline{x_1}, \underline{y_1}), \max(\overline{x_1}, \overline{y_2})]$. We lift the union on two boxes as the union of the intervals in each dimension of the product.

Let us consider a function $g : \mathbb{R}^n \to \mathbb{R}^k$ and a set $\mathcal{Z} \subset \mathbb{R}^k$. A contractor $\mathcal{C}_c : \mathbb{IR}^n \to \mathbb{IR}^n$, associated to the constraint $c : g(\mathbf{x}) \in \mathcal{Z}$, is a function taking a box $[\mathbf{x}]$ as input and returning a box $\mathcal{C}_c([\mathbf{x}])$ satisfying i) $\mathcal{C}_c([\mathbf{x}]) \subseteq [\mathbf{x}]$ and ii) $g([\mathbf{x}]) \cap \mathcal{Z} = g(\mathcal{C}_c([\mathbf{x}])) \cap \mathcal{Z}$. $\mathcal{C}_c$ provides a box containing the solutions of $g(\mathbf{x}) \in \mathcal{Z}$ included in $[\mathbf{x}]$: i) ensures that the returned box is included in $[\mathbf{x}]$ and ii) ensures that no solution of $g(\mathbf{x}) \in \mathcal{Z}$ in $[\mathbf{x}]$ is lost. Many interval contractors exist [23] such as Forward-Backward for algebraic constraints or Picard operator for ODE constraints.

A dynamics of an hybrid system is given by $\dot{\mathbf{y}} = f(\mathbf{y})$ with the initial condition $\mathbf{y}(0) = \mathbf{y}_0$. The exact solution of this equation is denoted by $\mathbf{y}(t; \mathbf{y}_0)$. An interval-based simulation approximates $\mathbf{y}$ by computing a sequence of time instants $t_0 < t_1 < \cdots < t_n$ and a sequence of boxes $[\mathbf{y}_0], [\mathbf{y}_1], \cdots, [\mathbf{y}_n]$ such that $\forall i \in [0, n-1]$, $\mathbf{y}(t_i; [\mathbf{y}_i]) \subseteq [\mathbf{y}_{i+1}]$. A *tube* $\mathcal{X}$ is the sequence of pairs $([\mathbf{y}_0], [\mathbf{y}_1]), ([\mathbf{y}_1], [\mathbf{y}_2] \cdots, ([\mathbf{y}_{n-1}], [\mathbf{y}_n])$. The safe approximation between time instants is $[\tilde{\mathbf{y}}_i]$ such that $\forall t \in [t_i, t_{i+1}]$, $\mathbf{y}(t; \mathbf{y}_0) \subseteq [\tilde{\mathbf{y}}_i]$, obtained with the Picard-Lindelöf operator [28], sometimes called *Picard box*.

A hybrid system is given by the tuple $(\mathcal{Q}, n, \mathcal{T}, \mathcal{D}, \mathcal{R})$. $\mathcal{Q}$ is a set of states. $n : \mathbb{N}^*$ is the dimension of the system. $\mathcal{T} : \mathcal{Q} \to ((\mathbb{IR}^n \to \mathbb{IR}^n) \times \mathcal{Q})$ is a map from states to transitions where a transition contains the guard and the destination state. $\mathcal{D} : \mathcal{Q} \to (\mathbb{IR}^n \to \mathbb{IR}^n)$ is a map from states to dynamics where a dynamics

is represented by a vector function. $\mathcal{R} : \mathcal{Q} \to (\mathbb{IR}^n \to \mathbb{IR}^n)$ is a map from states to reset functions (functions applied to $\mathbf{y}$ when entering automaton states, coming from the `init` statements of `der` equations).

4.2 Point-Wise versus Interval-Based Simulation

During a point-wise simulation, a transition (g, q_2) from a state q_1 to a state q_2 occurs at a precise instant t_z, when the guard g becomes equal to 0 (from negative to positive). Hence, the dynamics $\mathcal{D}(q_2)$ starts at t_z, with precise initial conditions corresponding to $\mathcal{R}(\mathbf{y}(t_z))$.

During an interval-based simulation, the transition from q_1 to q_2 may span over several boxes, making both the instants and the values of the continuous state imprecise. In other words, we do not know anymore when the change exactly occurs and what is the state of the system to start $\mathcal{D}(q_2)$ [35]. Moreover, since g crosses 0 "somewhere" in some boxes, both dynamics $\mathcal{D}(q_1)$ and $\mathcal{D}(q_2)$ must be considered active at the same time. $\mathcal{D}(q_2)$ may start at any instant in these boxes, with the initial condition computed by $\mathcal{R}(q_2)$ applied to any values of $\mathbf{y}$ in these boxes. Finally, to be correct with respect to the semantics of Zélus, when a guard crosses 0, we must ensure that the direction is from negative to positive values. We postpone this issue until Section 4.4.

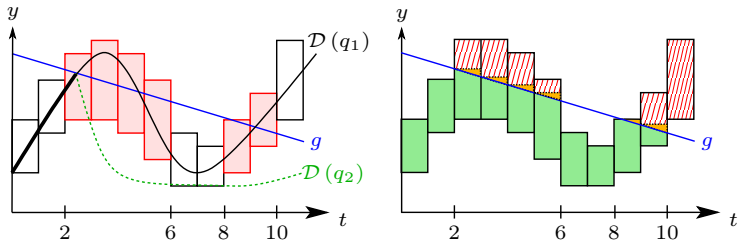

Figure 3: Point-wise simulations vs interval simulation

In Figure 3, to ease the representation, instead of drawing the boxes of g to monitor zero-crossings, we plot g in a point-wise manner and check instants where the boxes of $\mathcal{D}(q_1)$ traverses g from negative to positive. This is simply a change of point of view since g is a function depending on $\mathbf{y}$. In the left part of the figure, the dynamics $\mathcal{D}(q_1)$ is plotted point-wise with the thin section corresponding to the part that would not exist in a point-wise simulation because of the switch to $\mathcal{D}(q_2)$ (drawn dashed). The boxes of the interval-based simulation are filled in red when they cross g. The crossing occurs on several contiguous boxes between $t = 2$ and $t = 6$. Since parts of these boxes are below g, $\mathcal{D}(q_1)$ is considered still valid, leading to other crossing boxes between $t = 8$ and $t = 10$. On the right part of Figure 3, the green parts of boxes are those relevant for the simulation of $\mathcal{D}(q_1)$ while the dashed red ones must be eliminated because their values are for sure not reachable due to the change of dynamics. Orange regions represent parts of boxes that should have been eliminated but that cannot to preserve the shape of the boxes. In all the boxes crossing g, one may be in both states of the automaton: the simulation must fork to represent the two possible futures. Hence we no longer have one unique linear simulation but a *tree* of simulations. Obviously, such forks may lead to an exponential number of sub-trees which can dramatically slow down the simulation. In this work, we decided to favor accuracy instead of speed. More complex techniques allow to merge several futures in a unique one. They may be explored in a future refinement.

4.3 Switching Dynamics

The problem to address is to simulate $\mathcal{D}(q_2)$ on some time intervals $\mathbb{T}$ where boxes of $\mathcal{D}(q_1)$ cross g and determine the initial condition that must be used once g is totally crossed to run the simulation with only $\mathcal{D}(q_2)$. Having the tube of $\mathcal{D}(q_1)$, we will traverse its boxes to transform the temporal uncertainty represented by $\mathbb{T}$ into the spatial uncertainty representing the initial condition of $\mathcal{D}(q_2)$ after $\mathbb{T}$.

The simulation of $\mathcal{D}(q_2)$ must include all the reachable behaviors independently of the instant of the transition and of the continuous state $[\mathbf{y}]$ at this instant. Indeed, as shown in Figure 4, different behaviors may occur if g is crossed after $t = 2$ or after $t = 3$.

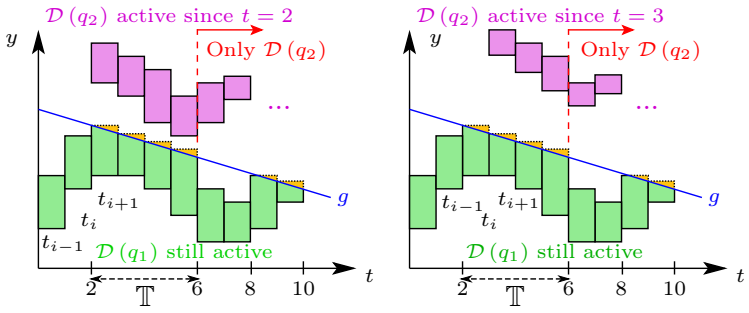

Figure 4: New dynamics starting at different times

For $t_i \in \mathbb{T}$, we denote by $[\mathbf{y}_i]_{|g=0}$ the box $[\mathbf{y}_i]$ restricted to intervals respecting $g = 0$. This restriction, by application of a contractor, is called a *contraction* and formally respects the properties $[\mathbf{y}_i]_{|g=0} \subseteq [\mathbf{y}_i]$ and $g^{-1}(0) \cap [\mathbf{y}_i] = g^{-1}(0) \cap [\mathbf{y}_i]_{|g=0}$. This guarantees that such a filtering reduces intervals in boxes without any solution loss. We lift the notation to the safe approximation on a time interval $[t_i, t_{i+1}]$ as $[\tilde{\mathbf{y}}_i]_{|g=0}$.

For each time interval $[t_i, t_{i+1}] \in \mathbb{T}$, $\mathcal{R}([\tilde{\mathbf{y}}_i]_{|g=0})$ represents the initial condition if $\mathcal{D}(q_2)$ starts in $[t_i, t_{i+1}]$. However, $\mathcal{D}(q_2)$ may also have started in the time interval $[t_{i-1}, t_i]$. Hence, the initial condition must also contain all the values obtained during the simulation of $\mathcal{D}(q_2)$ on $[t_{i-1}, t_i]$.

Indeed, as shown in the left part of Figure 5, since $\mathcal{D}(q_2)$ may have started at any time in $[t_{i-1}, t_i]$, by shifting the evolution depending on the effective beginning instant, any value reachable on this interval may reach the instant t_i . As shown in the left part of Figure 5, this *sub-simulation* process iterates all along the intervals of $\mathbb{T}$, computing the initial conditions of the sub-simulation on $[t_i, t_{i+1}]$ as the union of all the boxes of the sub-simulation on $[t_{i-1}, t_i]$ and the reset applied on $[\tilde{\mathbf{y}}_i]$ at t_i:

$$[\mathbf{y}_0]'_0 = \mathcal{R}(q_2)([\tilde{\mathbf{y}_0}]_{|g=0})$$
$$[\mathbf{y}_0]'_{i+1} = \bigcup_j [\mathbf{y}_j]'_i \mid [\mathbf{y}_j]'_i \ni y(t_i, [\mathbf{y}_0]'_i \cup \mathcal{R}(q_2)([\tilde{\mathbf{y}}_i]_{|g=0}))$$

where a box $[x]$ is obtained by simulating $\mathcal{D}(q_1)$, $[x]'_i$ by simulating $\mathcal{D}(q_2)$ on the interval $[t_i, t_{i+1}]$.

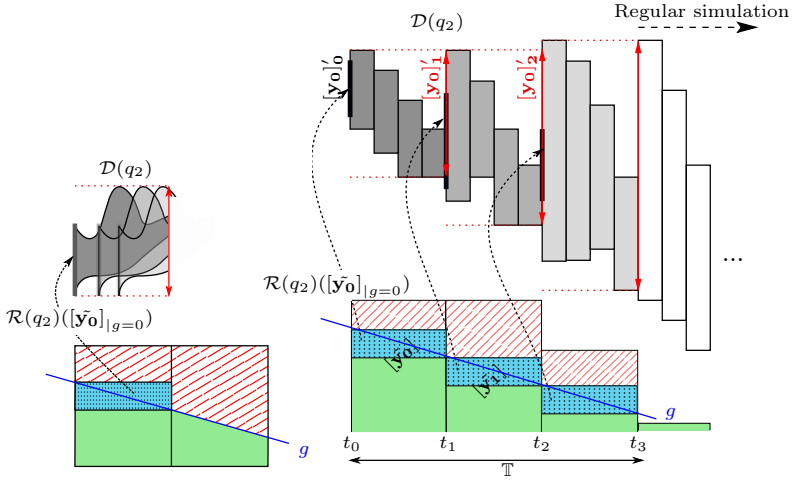

Figure 5: Sub-simulations principle

At the end of $\mathbb{T}$, the regular simulation of $\mathcal{D}(q_2)$ can be ran until the ending time of the global simulation or until a guard of a transition of q_2 is triggered. During each sub-simulation on $\mathbb{T}$, a guard may cross 0, requiring to consider again several dynamics in parallel. Hence, the simulation algorithm detailed in Section 4.5 is naturally recursive, each call creating *nodes* of the aforesaid tree.

The sub-simulation mechanism serves two purposes. First, it allows to compute the initial conditions to

simulate $\mathcal{D}(q_2)$ once g is totally crossed. It also builds the successive tubes representing the state of the system. To obtain a correct tube, all the boxes of a sub-simulation must be extended by union with the initial value of the sub-simulation. Indeed, since the effective transition may have occurred at any time on the sub-simulation interval, the initial value may belong to any box of the tube. Without this extension, the tube of the sub-simulation (which will be recorded in the tree of simulations) would not represent all the reachable values of the system.

A part of simulation with no automaton state change leads to a node containing the tube of this simulation. When a guard crosses 0 on time intervals $\mathbb{T}$, new nodes are created and chained for each sub-simulation in $\mathbb{T}$. Then the last node of these sub-simulations is the parent of the node that will be created for the simulation once the guard totally crossed 0. Because during each sub-simulation a guard can cross 0, nodes of sub-simulations may be themselves roots of other sub-simulations trees.

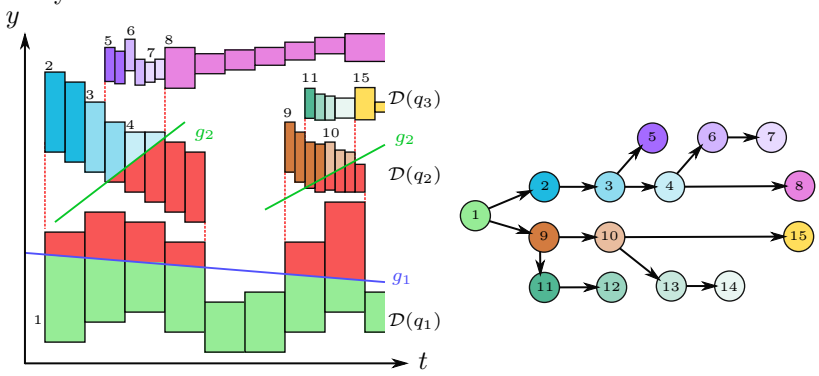

Figure 6: Tree of simulations

An example of simulation with sub-simulations and the related tree is given in Figure 6. We consider an automaton with three states. The state q_1 has a guard g_1 to enter the state q_2 which has a guard g_2 to enter the state q_3. Sub-simulations corresponding to a same interval crossing a guard are drawn in the same color. Dashed vertical lines represent time extents where a guard is crossed, hence where to compute some sub-simulations. Nodes of the tree are colored according to the tube of the sub-simulation they contain. Note that such a tree does not represent temporal relations between siblings for several reasons. First, two siblings may represent different futures, with different integration durations. Second, sub-simulation nodes are produced while computing the new initial condition of a new dynamics once the related guard is strictly above 0: they are not parents of the node hosting the tube of the new dynamics. For instance, the nodes 6 and 7 serve to compute the initial condition to compute the simulation of Node 8 but are not its ancestors. They are children of the Node 4, like Node 8, but they represent a time extent ending before the one of Node 8. However, the tree allows to keep trace of all the possible behaviors.

4.4 Event Detection with Semantics of up

To respect Zélus semantics (see the meaning of up in Section 2), an event has to be detected if a guard crosses 0 from negative to positive values. Simply checking the intersection of the guard with 0 is not sufficient since it does not provide the crossing direction. This would trigger a spurious event if the guard indeed crossed from positive to negative values. Hence, one must evaluate the guard and track instants where it was negative and becomes positive. Using the Picard box (*i.e.*, looking at the sign of $g([\tilde{\mathbf{y}}_i])$) is not accurate enough since knowing that a box crosses 0 does not provide the direction of the crossing. Moreover, the extent of the Picard box may hide real events.

In Figure 7, the evaluation of a guard g is given. Boxes represent the interval-based evaluation while red curves represent possible point-wise simulations. The case (a) leads to an event pretty late in the middle of the second box: one can expect that a simulation with a smaller step size would allow to reduce the size of the box really crossing 0. The case (b) does not trigger any event. Finally, the case (c) also leads to an event despite the global shape of the boxes seems to indicate a crossing from positive to negative values. The guard crosses 0 several times, one being in the expected direction.

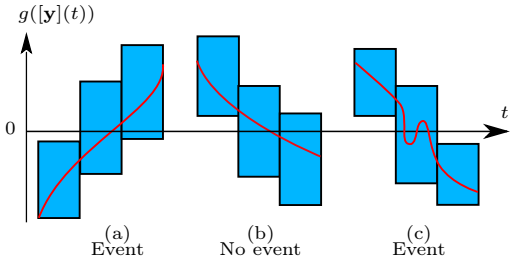

Figure 7: Event detection

The first step of refined event detection performs a dichotomic search on the time interval $[t_i, t_{i+1}]$ for each box such as $0 \in g([\tilde{\mathbf{y}}_i])$ (left side of Figure 8). At each iteration on an interval $[t_n, t_{n+1}]$, we compute the middle time $t_m = (t_i + t_{i+1})/2$. Then we integrate to compute $[\mathbf{y}_m]$ at time t_m. If $g([\mathbf{y}_n]) < 0$ and $g([\mathbf{y}_m]) \geqslant 0$ then the search goes on $[t_n, t_m]$. Otherwise, if $g([\mathbf{y}_m]) < 0$ and $g([\mathbf{y}_{n+1}]) \geqslant 0$ then the search goes on $[t_m, t_{n+1}]$. This process ends when the time interval size reaches a user-defined threshold ε and we make the assumption that only one crossing exists in this interval. As a consequence, this method will fail to accurately detect an event caused by a hump in an interval of size ε (right side of Figure 8) : the whole box will be kept.

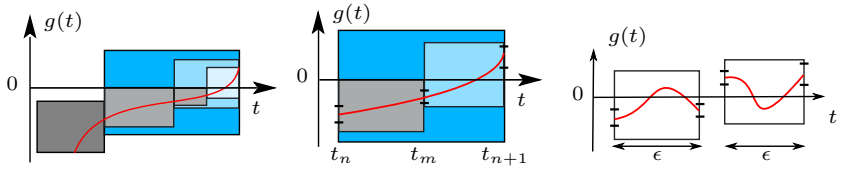

Figure 8: Event detection by refinement on time

Once the refinement process is over, the second step detects events by looking for the leftmost time interval where $g([\mathbf{y}_i])$ was previously strictly below 0 and is now above (or containing 0). The duration of the event lasts until $g([\mathbf{y}_i])$ goes strictly above 0 or strictly below 0. In the first case, the guard totally crossed 0, hence, the old dynamics will be no longer active. In the second case, the new dynamics will be no longer active. The detection algorithm can be represented by the automaton of Figure 9.

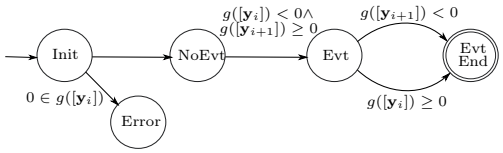

Figure 9: Event detection automaton

If the evaluation of g on the first box of the tube contains 0, it becomes impossible to detect a crossing since g will have never been strictly negative before being positive or null. In this case, the simulation is aborted and an error is raised. This may happen if in the initial or a restart state the guard directly crosses zero. Examining the successive boxes of the tube allows to progress in the automaton until the end of the tube or the end of an event. Depending on the context where event detection is used, the end of an event may lead to different actions (in algorithms 3 and 4).

With the process of detection of beginning/ending of events, it is possible to apply the mechanisms of contraction (to compute $[\mathbf{y}_i]_{|g}$) and sub-simulations presented in Section 4.3 in order to compute the initial condition of the new dynamics. Indeed, these two mechanisms are triggered by the detection of an effective event.

4.5 Simulation Algorithm

We now present the algorithm implementing the simulation with switches of dynamics. We use the notations introduced in the previous section to represent an automaton. This recursive algorithm relies on two mutually recursive main functions: `Run` which performs the simulation of one dynamics and `ComputeBranchingSeeds`

which performs sub-simulations. The other important function is `GetEventRegions` which detects time intervals where a guard crosses 0. Three utility functions are also involved: `BisectOn0` and `Split` to perform the refinement of intervals where a guard crosses 0; `ContractOnEvent` to contract the boxes of a tube when an event is in progress.

We call *branching seed* the information computed during sub-simulations on a contiguous time extent where a guard crosses 0. As shown in Figure 6, several sets of such extents may exist for a same simulation, leading to several crossings of a guard (the two red boxes parts). A branching seed is a tuple $([t], [\mathbf{y}], n_s, n_e)$ containing the global time extent $[t]$ covered by the sub-simulations, the first and last nodes (n_s, n_e) created by these sub-simulations on the time extent and $[\mathbf{y}]$ the box to use as initial condition to carry out the next simulation (*i.e.*, when the guard is totally over 0 and only one dynamics is active).

We make the choice of a top-down presentation of the functions to ease the global understanding of the algorithm. We assume given (by the DynIbex library) a function `DynIbex_Simulation` able to simulate a dynamics on a time extent from an initial value.

Algorithm 1 $Run\,(q, t_0, t_e, [\mathbf{y_0}], union_{[\mathbf{y_0}]})$

```
1: if t0 ⩾ te then return none
2: X ← DynIbex_Simulation (y, D (q), t0, te)
3: if union_[y0] then
4:     for [yi] ∈ X do [yi] ← [yi] ∪ [y0]
5: if T (q) = ∅ then return (X, ∅)
6: else
7:     for (g, _) ∈ T (q) do BisectOn0 (X, D (q), g) end for
8:     changed ← true
9:     while changed do
10:        changed ← false
11:        for (g, _) ∈ T (q) do
12:            changed ← changed ∨ ContractOnEvent (X, g)
13:    n ← node (X, ∅)
14:    for (g, q') ∈ T (q) do
15:        X' ← copy of X
16:        LL_ER ← GetEventRegions (X', g)
17:        L_BS ← ComputeBranchingSeeds (q', LL_ER)
18:        for ([_, t'], [y], ns, ne) ∈ L_BS do
19:            add ns to the children of n
20:            n' ← Run (q', t', te, [y], false)
21:            add n' to the children of ne
22:    return n
```

The function `Run` takes five arguments: the current state q of the automaton, t_0 and t_e the starting and ending times of the simulation, the initial value $[\mathbf{y_0}]$ of the system and a flag telling if the boxes of the computed tube must be extended by union with the initial value of the sub-simulation (*cf.* Section 4.3). It computes the simulation of $\mathcal{D}(q)$ on the time interval $[t_0, t_e]$ with the initial condition $[\mathbf{y_0}]$. If some guards cross 0, it calls `GetEventRegions` to detect effective events (0-crossings), then `ComputeBranchingSeeds` to perform the sub-simulations and recurses with each new dynamics once its related guard is above 0. It returns the node containing the tube of simulation whose children are the trees produced by the sub-simulations. The reset of the initial state must be applied before the initial call to `Run`.

If the end of the simulation time is reached, the empty node (*none*) is returned (lines $1-3$). At Line 2, a simulation of the current dynamics is run using the dedicated function of DynIbex which returns a tube $\mathcal{X}$. Between lines $3-4$, the boxes of the tube are extended with the initial condition if needed (*i.e.*, if we are computing a sub-simulation). At Line 5, if no transitions exit the current state, no change of dynamics is possible until the end of the simulation. Hence a new node is created (and returned) with no children. At Line 7, we refine the simulation's boxes on time intervals where the guard crosses 0 (in whatever direction). Between lines $8-12$, for all the guards of the current state, we contract the tube to keep parts of boxes respecting the guard $\leqslant 0$. This process is applied with each guard until a fixpoint is reached. At this point, parts of the tube not respecting the guards are removed. At Line 13, a new node n is created, recording the contracted tube. Between lines $14-21$, we process all the transitions exiting the current state. At Line 15, we make a copy of the tube since subsequent processing will perform side effects on the boxes. At Line 16, we get the list of sequences of consecutive boxes where an event is spanning. During this process, the tube is contracted to enclose the most precisely boxes $[\mathbf{y}]_\mathbf{i}$ such as $g([\mathbf{y}]_\mathbf{i}) = \mathbf{0}$. At Line 17, for each event, we compute by sub-simulations the time and initial condition to carry on the simulation once the related guard will have totally crossed 0. We recover the starting and ending tree nodes representing the sub-simulations. Between lines $18-21$, we process each branching seed. The first node of the seed is added as child of the node n. Line 20, a new simulation is run on the seed's time extent, with its spatial values as initial condition and the new dynamics (*i.e.*, the guard totally crossed 0, only the new dynamics is now active). This creates a new node n' which is added as child of the seed's last node. Finally, the node n is returned as root of the simulation step.

The function *ComputeBranchingSeeds* takes two arguments: the target state q of the transition whose

guard crossed 0 and the list of sequences of consecutive boxes where an event is spanning. For each sequence of boxes, it computes, by sub-simulations, the time and the initial condition to be used to start the related dynamics once the related guard will be above 0. It returns a list of initial times and conditions that will be used by `Run` to recurse.

Algorithm 2 *ComputeBranchingSeeds* $(q, \mathcal{LL}_{ER})$

```
1: L_BS ← ∅
2: pending ← false  # Ongoing seed?
3: t_0 ← ?  # Pending seed start time.
4: [s_a] ← ?  # Seed restart conditions.
5: t_{-1} ← ?  # Previous box end time.
6: n_s ← none  # Seed's first node.
7: n_e ← none  # Seed's last node.
8: for L_ER ∈ LL_ER do
9:     pending ← false
10:    for ([t_a, t_b], [v]) ∈ L_ER do
11:        if ¬pending then
12:            t_0 ← t_a
13:            [s_a] ← vector of empty intervals
14:            pending ← true
15:        [y_0] = R(q)([v]) ∪ [s_a]
16:        n ← Run (q, [y_0], t_a, t_b, true)
17:        if n_s = none then n_s ← n
18:        if n_e <> none then add n to the children of n_e
19:        n_e ← n
20:        [s_a] ← [s_a] ∪ Collapse(tube of n)
21:        t_{-1} ← t_b
22:    L_BS ← L_BS + ([t_0, t_{-1}], [s_a], n_s, n_e)
23:    n_s ← none
24: return L_BS
```

Line 1 the accumulator of seeds is initialized to the empty list. At Line 2, we signal that no seed is currently under construction. Lines $3-7$, various variables are set to non significant values that will be erased when a seed will be under construction. Between lines $8-23$, we process each sequence of contiguous boxes crossing the guard. At Line 9, since we begin processing a new sequence, no seed is under construction. Between lines $10-23$, we iterate on the consecutive boxes representing the extent of an event. Between lines $11-14$, if no seed is pending we start a new one, recording its initial time. At Line 15, the initial condition of the sub-simulation is computed. Line 16 a sub-simulation is performed on the duration of the current box. The Boolean flag is set to `true` to ensure that the boxes of the tube of the sub-simulation will be extended with the initial condition. Lines $17-18$ we record the starting node of the seed if none is already recorded and add the new node as a child of the currently recorded ending node of the seed. Line 19, we update the ending node as being the new node coming from the sub-simulation. Line 20, the future seed's initial condition is accumulated by union with all the reachable values in the tube of the sub-simulation. The function $Collapse(\mathcal{X}) = \bigcup_{[\tilde{\mathbf{y}}_i] \in \mathcal{X}} [\tilde{\mathbf{y}}_i]$ lifts the union on a tube, making the union of all the boxes of the tube. Line 21, we update the ending time of the future seed as being the upper bound of the time interval of the current box. Line 22, once all the contiguous boxes are processed, we can end the seed and append it to the seeds accumulator. Finally, the seeds accumulator is returned as result.

The function `ContractOnEvent` must detect when a guard g crosses 0 from negative to positive values and contract the boxes of the tube during the event extent. It takes two arguments: the tube and the guard. It returns a Boolean value telling if a contraction occurred, hence the tube changed. This is used by `Run` to detect when the fixpoint is reached.

Algorithm 3 *ContractOnEvent* $(\mathcal{X}, g)$

```
1: crossing, changed ← false
2: if X ≠ ∅ then
3:     let ([y_0], _) = X(0)
4:     if 0 ∈ g([y_0]) then error
5: for ([y_i], [y_{i+1}]) ∈ X do
6:     if g([y_i]) < 0 ∧ g([y_{i+1}]) ⩾ 0 then
7:         [ỹ_i]' ← [ỹ_i]
8:         contract [ỹ_i] with respect to g ⩽ 0
9:         changed ← changed ∨ ([ỹ_i]' ≠ [ỹ_i])
10:        crossing ← true
11:    else
12:        if crossing then
13:            if g([y_{i+1}]) < 0 then crossing ← false
14:            else
15:                if g([y_i]) ⩾ 0 then
16:                    changed ← end of X not reached
17:                    discard the next boxes of X return changed
18:                else
19:                    [ỹ_i]' ← [ỹ_i]
20:                    contract [ỹ_i] with respect to g ⩽ 0
21:                    changed ← changed ∨ ([ỹ_i]' ≠ [ỹ_i])
22: return changed
```

Between lines $2-4$, we ensure that the evaluation of g on the first box of the tube does not contain 0. If so, it will be impossible to detect an event since g will have never been strictly negative before becoming positive. In this case the simulation is aborted. Between lines $5-21$ each pair of consecutive boxes are processed to detect beginnings and endings of events. At Line 6, we check for an event starting, *i.e.*, the guard being below 0 at time t_i and becoming above 0 at time t_{i+1}. Between lines $7-10$ a starting event is detected, hence the contraction is applied. We check if it changed the box and we note that an event is ongoing. Starting from Line 11, an event is ongoing, started earlier. Line 13 we check for the end of the event in case g went back below 0. Between lines $15-17$, we check if g is totally above 0. If so, then the remaining boxes of the simulation

are not reachable since g forbid them. Hence we must delete these boxes from the tube. If the deletion modified the tube we know that the fixpoint of contractions is not yet reached. Finally, between lines $18-21$, the event started earlier is still ongoing and the current box must be contracted.

Algorithm 4 *GetEventRegions* $(\mathcal{X}, g)$

```
1: crossing ← false
2: L_ER, LL_ER ← ∅
3: for ([y_i], [y_{i+1}]) ∈ X do
4:     if g([y_i]) < 0 ∧ g([y_{i+1}]) ⩾ 0 then
5:         contract [ỹ_i] with respect to g = 0
6:         L_ER ← L_ER + ([t_i, t_{i+1}], [ỹ_i])
7:         LL_ER ← LL_ER + L_ER
8:         crossing ← true
9:     else
10:        if crossing then
11:            if g([y_{i+1}]) < 0 then
12:                L_ER ← ∅
13:                crossing ← false
14:            else
15:                if g([y_i]) ⩾ 0 then return LL_ER
16:                else
17:                    contract [ỹ_i] with respect to g = 0
18:                    L_ER ← L_ER + ([t_i, t_{i+1}], [ỹ_i])
   return LL_ER
```

The function `GetEventRegions` computes the list of sequences of consecutive boxes where an event is spanning for a guard. Its result is the input of `ComputeBranchingSeeds`. It takes two parameters: a tube and a guard.

At Line 1, the list representing the ongoing sequence and the list of detected sequences are empty. The algorithm processes each pair of consecutive boxes with a structure very similar to `ContractOnEvent` since it also needs to detect starting/ending events. The major difference is the contraction with respect to $g = 0$ to obtain the boxes of the dynamics at the moment where the new dynamics must start. As long as an event lasts, the safe approximation $[\tilde{\mathbf{y}}_i]$ of two continuous boxes is recorded in the list $\mathcal{L}_{ER}$ with its related time interval. When the event stops, this list is added to the $\mathcal{LL}_{ER}$ accumulator.

The function `BisectOn0` takes two arguments: a tube and a guard g. For each box $[\tilde{\mathbf{y}}_i]$ of the tube, if the evaluation of g on $[\tilde{\mathbf{y}}_i]$ contains 0, then the effective dichotomic refinement is launched by calling `Split`. This refinement will modify the boxes of the tube by side effect.

Algorithm 5 *BisectOn0* $(\mathcal{X}, f, g)$

```
1: for ([y_i], [y_{i+1}]) ∈ X do
2:     if 0 ∈ g([ỹ_i]) then
3:         Split(X, f, g, [t_i, t_{i+1}], [y_i], [y_{i+1}])
```

The function `Split` is in charge of refining the boxes of a tube when a guard crosses 0. It takes six arguments: a tube, a dynamics f, a guard g, the time interval on which the refinement must be done, the boxes obtained by simulating f at times t_i and t_{i+1}.

Algorithm 6 *Split* $(\mathcal{X}, f, g, [t_i, t_{i+1}], [\mathbf{y}_i], [\mathbf{y}_{i+1}])$

```
1: if t_{i+1} − t_i ⩽ a given ε then return
2: t_m ← (t_i + t_{i+1})/2
3: [y_m] ← evaluate f at time t_m
4: insert [y_m] in X
5: if g([y_i]) < 0 ∧ g([y_m]) ⩾ 0 then
6:     Split(X, f, g, [t_i, t_mid], [y_i], [y_m])
7: else
8:     if g([y_m]) < 0 ∧ g([y_{i+1}]) ⩾ 0 then
9:         Split(X, f, g, [t_m, t_{i+1}], [y_m], [y_{i+1}])
```

If the time interval is smaller than a given ε, the refinement process stops. Otherwise, we compute t_m the time in the middle of the time interval and evaluate the dynamics f at this middle time. The obtained box $[\mathbf{y_m}]$ must be inserted in the tube (by side effect) with respect to the temporal ordering. Then a usual dichotomic search is performed depending on which time sub-interval causes the evaluation of guard to be negative then positive or null.

5 Compilation Principle

To be executed by the simulation algorithm, a program written in Zélus must be compiled to create the data-structure representing the automaton. This section presents this compilation process.

5.1 Overview

For several reasons, we cannot simply rewrite Zélus's backend to make it generate C++ code instead of OCaml code and get hybrid systems for free with DynIbex.

First, the generated OCaml code is very dependent on the ODE solver used by Zélus and the solving runtime is extremely different from DynIbex's mechanisms. Second, the runtime simulation code is deeply

mixed with the code related to the hybrid model, making impossible to distinguish between code to be translated into C++, code to be transformed into intervals and code which should be ignored. Finally, as described in Section 4, intervals are strongly incompatible with point-wise simulation.

For these reasons, we need a dedicated compilation scheme to bind Zélus and DynIbex. In the current work, we decide to ignore systems with arbitrary discrete behavior, i.e. with arbitrary discrete code representing the controller part of the system. Taking this part into account clearly requires other techniques (for example abstract interpretation) to obtain an interval-based evaluation mechanism for Zélus general instructions. Moreover, some constructs, even without intervals, require advanced compilation techniques to produce C++ code (pattern-matching, datatypes, etc).

The compilation of a Zélus program requires three steps. The first one builds an intermediate representation of each node of the program. We call this representation a *pre-automaton*. During this pass, a node with no automaton is transformed in an automaton with one unique state. Equations outside the automaton states are pushed into the states, with their initial value removed (if they are ODEs and have an initial value). The redefinition of **init** equations is controlled. The reset of each state and the inits of the pre-automaton are computed.

Once the pre-automaton is obtained, the hierarchy of nodes must be flattened. In each node of the pre-automaton, node instantiation expressions must be replaced by the body of the node where the occurrences of its parameters are replaced by the effective expressions provided at the instantiation point. This process implies a recursive inlining mechanism which terminates since Zélus forbids recursive nodes. Regular and **init** equations are also inlined. The reset of each state is computed from the initial values of ODEs. The output of this pass is an *intermediate automaton* suitable for C++ code production.

Finally, the C++ code generation is in charge of converting the multiple equations into a unique vector-valued function. Each differential equation corresponds to one dimension of the DynIbex `Function` data-structure. During this process, Zélus expressions are compiled to C++ expressions. This process mostly consists of a translation of arithmetic expressions into C++, mapping the identifiers to the appropriate vector component, and converting real constants into trivial intervals. The structure of the intermediate automaton is translated into static C++ structures and arrays to encode $\mathcal{Q}, \mathcal{T}, \mathcal{D}, \mathcal{R}$ introduced in Section 4. These C++ data-structures are used by the generic simulation library implementing the algorithm given in Section 4.5. The generated code and the library are linked together in the final executable.

5.2 Pre-Automaton Generation

The restricted syntax of programs addressed in this work is given in Figure 10.

program	::=	$node^+$
node	::=	`hybrid` $f\,(x^*) = y$ `where` $init^*$ $equation^*$ $automaton$
equation	::=	`der` $x = e_1$ `init` e_2 | `let` $x = e$
init	::=	`init` $x = e$
automaton	::=	`automaton` $state^+$
state	::=	$name$ `->` $equation^+$ $transition^*$
transition	::=	`until up` e `then` $name$
e	::=	$r \mid x \mid$ `op` $e \mid e_1$ `op` $e_2 \mid f\,(e^*)$

Figure 10: Syntax subset of Zélus for automata

A program is a list of nodes. A node is the definition of a parameterized component returning a value y which is the result of one of the equations defined in this node (possibly in its automaton). A node may contain **init** statements providing the initial value of an equation, equations, an automaton. An equation may be a differential equation with an initial value e_2 or a regular dataflow equation binding an expression e to an identifier x. An automaton contains named states containing equations and possibly transitions. A transition is made of a guard expression e allowing to reach a destination state when e changes from negative to positive. Expressions are numeric constants, identifiers, arithmetic expressions or the instantiation of a node named f with expressions. Node instantiations cannot be self-recursive.

The pre-automaton synthesis computes, for each node in the source code, the representation of the

pre-automaton of this node. This intermediate representation is structured as follows.

A node is described by $(f, \mathcal{A}, \{x_1 \dots x_n\}, y)$ where $\mathcal{A}$ is the description of its automaton and the x_i are the node's formal parameters. An automaton $\mathcal{A}$ is a pair $(\mathcal{I}, \mathcal{Q})$ where $\mathcal{I}$ is the set of its inits and $\mathcal{Q}$ is the set of its states. An init is a pair (x, e) describing the initial value of the (differential) equation named x. A state is a tuple $(N, \mathcal{E}, \mathcal{T}, \mathcal{R})$ where N is the name of the state, $\mathcal{E}$ the set of equations running in this state, $\mathcal{T}$ the set of transitions exiting from this state and $\mathcal{R}$ is the set of init expressions of the equations when entering the state (hence producing a reset/jump effect). A transition is a pair (N, e) where N is the name of the destination state and e the expression triggering the transition when crossing 0 (from negative to positive values). An equation is either an ODE `der` (x, e) or a regular dataflow equation `reg` (x, e).

The compilation rules of the pre-automaton elaboration given in Figure 11 use an environment E recording the toplevel equations of the current node. The judgment $E \vdash c_1 \rightsquigarrow c_2$ means that, in the environment E, the construct c_1 is transformed in the construct c_2. The relations $\rightsquigarrow$ are indexed to reflect the syntactical class of the constructs they handle. When it is not needed, we omit the environment. In the rules, we use a set notation to denote lists and the meta-variable `_` to denote non meaningful parts of expressions.

$$(\textsc{Let})\ \texttt{let}\ x = e \rightsquigarrow_{\mathcal{E}} \texttt{reg}\,(x, e_1) \qquad (\textsc{Der})\ \texttt{der}\ x = e_1\ \texttt{init}\ e_2 \rightsquigarrow_{\mathcal{E}} \texttt{der}\,(x, e)$$

$$(\textsc{Tra})\ \texttt{until up}\ e\ \texttt{then}\ N \rightsquigarrow_{\mathcal{T}} (N, e) \qquad (\textsc{Ini})\ \texttt{init}\ x = e \rightsquigarrow_{\mathcal{I}} (x, e)$$

$$(\textsc{St})\ \frac{\begin{array}{c} eq_1 \rightsquigarrow_{\mathcal{E}} eq'_1 \quad \cdots \quad eq_n \rightsquigarrow_{\mathcal{E}} eq'_n \qquad tr_1 \rightsquigarrow_{\mathcal{T}} tr'_1 \quad \cdots \quad tr_m \rightsquigarrow_{\mathcal{T}} tr'_m \\ \mathcal{R} = \{\bigcup_{i=1}^{n} (x, e) \mid eq_i = \texttt{der}\ x = \_\ \texttt{init}\ e\} \quad \mathcal{R}_E = \{\bigcup (x, x) \mid \texttt{der}\ x = \_\ \texttt{init}\ \_ \in E\} \\ \mathcal{E}_E = \{\bigcup \texttt{der}\,(x, e) \mid \texttt{der}\ x = e\ \texttt{init}\ \_ \in E\} \cup \{\bigcup \texttt{reg}\,(x, e) \mid \texttt{let}\ x = e \in E\} \end{array}}{E \vdash N\ \texttt{->}\ eq_1 \cdots eq_n\ tr_1 \cdots tr_m \rightsquigarrow_{\mathcal{Q}} (N, \{eq'_1 \cdots eq'_n\} \cup \mathcal{E}_E, \{tr'_1 \cdots tr'_m\}, \mathcal{R} \cup \mathcal{R}_E)}$$

$$(\textsc{Aut})\ \frac{E \vdash q_1 \rightsquigarrow_{\mathcal{Q}} q'_1 \cdots E \vdash q_n \rightsquigarrow_{\mathcal{Q}} q'_n \qquad WF_I\,(\mathcal{I}, \{q'_1 \cdots q'_n\}) \qquad \mathcal{I}_E = \{\bigcup (x, e) \mid \texttt{der}\ x = \_\ \texttt{init}\ e \in E\}}{E, \mathcal{I} \vdash \texttt{automaton}\ q_1 \cdots q_n \rightsquigarrow_{\mathcal{A}} (\mathcal{I}_E, \{q'_1 \cdots q'_n\})}$$

where:

$$\begin{array}{l} WF_I\,(\mathcal{I}, \mathcal{Q}) = \forall\,(x, \_) \in \mathcal{I}, \\ \quad (\forall\,(\_, \mathcal{E}, \_, \_) \in \mathcal{Q}, \exists\, eq \mid eq = \texttt{der}\,(x, \_) \vee eq = \texttt{reg}\,(x, \_)) \\ \quad \vee \\ \quad (\forall\,(\_, \mathcal{E}, \_, \_) \in \mathcal{Q}, \nexists\, eq \mid eq = \texttt{der}\,(x, \_) \vee eq = \texttt{reg}\,(x, \_)) \end{array}$$

$$(\textsc{Nod})\ \frac{\begin{array}{c} in_1 \rightsquigarrow_{\mathcal{I}} in'_1 \quad \cdots \quad in_m \rightsquigarrow_{\mathcal{I}} in'_m \\ eq_1 \rightsquigarrow_{\mathcal{E}} eq'_1 \quad \cdots \quad eq_p \rightsquigarrow_{\mathcal{E}} eq'_p \qquad \mathcal{I} = \{in'_1 \cdots in'_m\} \qquad E = \{eq'_1 \cdots eq'_p\} \qquad E, \mathcal{I} \vdash aut \rightsquigarrow_{\mathcal{A}} \mathcal{A} \end{array}}{\begin{array}{r} \texttt{hybrid}\ f\,(x_1 \cdots x_n) = y\ \texttt{where}\ in_1 \cdots in_m\ eq_1 \cdots eq_p\ tr_1 \cdots tr_q\ aut \rightsquigarrow_{\mathcal{N}} \\ (f, \mathcal{A}, \{x_1 \cdots x_n\}, y) \end{array}}$$

Figure 11: Pre-automaton construction rules

The rules for equations, transitions and inits are pretty straightforward. One only note that initial conditions of ODEs are discarded. Let us remark that even if the rules Tra and Ini both return a pair, they do not contain the same kind of information since in Tra the first component is a state name although in Ini it is a regular identifier.

The rule St handles states. It is in charge to compute the equations and the reset of the state. The equations of the state are those explicitly defined inside it, extended by those defined at toplevel. The reset of the state is the union of the initial conditions of the ODEs defined in the state and the identity for those defined at toplevel (belonging to E). Indeed, ODEs defined at toplevel are initialized once before entering in the automaton and are never reset when changing of state. Hence the reset of a state must not touch the values of these equations.

The rule Aut processes an automaton. It performs the translation of the states and computes the init of the automaton as the union of the initial conditions of the toplevel ODEs. This rule relies on the predicate $WF_I\,(\mathcal{I}, \mathcal{Q})$ ensuring that each init in $\mathcal{I}$ is either redefined in all the states in $\mathcal{Q}$ or in none of them.

```
rec init a = 1
and automaton
| A -> der x = a init ...
       until up (...) then B
| B -> der x = a init ...
       and a = 2
       until up (...) then A
```

Figure 12: Rejected redefinition

This well-formedness condition is required to ensure that in the future inlining pass of the inits, they will have a predictable value in all the states. If we consider the automaton of Figure 12, when in the state `A`, it is impossible to know if we previously went in `B`, hence updated `a` with the value 2 or if it is still 1. One solution would be to consider the interval $[1, 2]$ in both states which would dramatically add pessimism and mix two unrelated dynamics. This mix would possibly lead to behaviors impossible in the real dynamics.

Finally, the rule Nod handles nodes. It initializes the environment E with the translation of the toplevel equations and builds the representation of the pre-automaton of the node.

5.3 Intermediate Automaton Generation

The structure of the intermediate automaton is pretty close to the one of the pre-automaton. The only changes are in expressions where there is no more node instantiations and in equations where there only remains ODEs, hence described by pairs (x, e). A set of such ODEs is denoted $\mathcal{D}$. A node is now described by the triplet $(\mathcal{A}, \{x_1 \dots x_n\}, y)$ where its name does not appear compared to the representation in the pre-automaton.

The compilation rules for creating the intermediate automaton rely on an environment E mapping a node name to its pre-automaton.

5.3.1 Node Instantiation Inlining

As sketched in Section 5.1, an inlining mechanism is required to flatten the hierarchy of nodes. The rules for inlining in expressions are given in Figure 13. The judgment $E \vdash e \hookrightarrow_e (e', \mathcal{E}, \mathcal{R})$ means that, in the environment E, the expression e is transformed into the expression e' and produces the set of equations $\mathcal{E}$ (differential or regular) and the set of reset expressions $\mathcal{R}$.

$$\text{(Cst)}\ E \vdash r \hookrightarrow_e (r, \emptyset, \emptyset) \qquad \text{(Id)}\ E \vdash v \hookrightarrow_e (v, \emptyset, \emptyset)$$

$$\text{(Op)}\ \frac{E \vdash e_1 \hookrightarrow_e (e_1', \mathcal{E}_1, \mathcal{R}_1) \qquad E \vdash e_2 \hookrightarrow_e (e_2', \mathcal{E}_2, \mathcal{R}_2)}{E \vdash e_1\ \texttt{op}\ e_2 \hookrightarrow_e (e_1'\ \texttt{op}\ e_2', \mathcal{E}_1 \cup \mathcal{E}_2, \mathcal{R}_1 \cup \mathcal{R}_2)}$$

$$\text{(Inst)}\ \frac{\begin{array}{c} E \vdash e_1 \hookrightarrow_e (e_1', \mathcal{E}_1, \mathcal{R}_1) \quad \cdots \quad E \vdash e_n \hookrightarrow_e (e_n', \mathcal{E}_n, \mathcal{R}_n) \qquad \varphi = [x_i \leftarrow e_i'] \quad E(f) = (\mathcal{A}, \{x_1 \cdots x_n\}, y) \\ \mathcal{A} = (\mathcal{I}, \{(N, \mathcal{D}, \mathcal{R}_\mathcal{A})\}) \qquad \mathcal{E}_\mathcal{A} = \{\texttt{der}\,(x, \varphi(e)) \mid (x, e) \in \mathcal{D}\} \qquad \mathcal{R} = \varphi(\mathcal{I} \ltimes \mathcal{R}_\mathcal{A}) \end{array}}{E \vdash f\,(e_1, \dots, e_n) \hookrightarrow_e (y, \mathcal{E}_1 \cup \cdots \mathcal{E}_n \cup \mathcal{E}_\mathcal{A}, \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_n \cup \mathcal{R})}$$

where: $\mathcal{I} \ltimes \mathcal{R} = \mathcal{I} \cup \{(x, e) \in \mathcal{R} \mid (x, e') \notin \mathcal{R}\}$

Figure 13: Instantiation inlining in expressions

The rule Cst handles a real numeric constant and produces the same constant with empty sets of equations and resets. The rule Id handles identifiers the same way.

The rule Opp recursively processes the two sub-expressions to rebuild an arithmetic expression. The equations and resets sets are the unions of those obtained for each sub-expression.

The rule Inst handles the node instantiation and is in charge of the effective inlining. The name f is expected to be bound in the environment to a node description. This rule assumes that the automaton of f has only one state because we do not handle hierarchical automata in this work. This allows to instantiate nodes syntactically containing no automaton for whose the compilation will have created a pre-automaton with one unique dummy state. The expression returned by the rule Inst is the identifier naming the output of f. The equations set is the one of f in which all the occurrences of the formal parameters x_i of f have been substituted by the corresponding effective argument expressions e_i'. Note that by construction, the set $\mathcal{D}$ coming from the intermediate automaton bound to f only contains differential equations. The set of reset expressions is built by a non-symmetric union (written $\ltimes$) of the inits and the resets of the automaton of the inlined node. If a binding exists in both, we keep the one coming from the inits. Indeed, in the node to inline, one may have a non-empty intersection between these two sets since its toplevel equations, pushed in its dummy state, have their initial values set as inits of the automaton and the resets of its dummy state are

set to the identity. The replacement of the formal parameters by the effective argument expressions is also applied on the obtained resets set. Note that in this rule, to simplify the presentation, we omit the renaming of all the identifiers of the inlined node by fresh variables (*i.e.*, not appearing anywhere in the program). This renaming is mandatory to avoid variable capture.

The compilation rules for equations are given in Figure 14. The judgment $E \vdash eq \hookrightarrow_{\mathcal{E}} (eq', \mathcal{E}, \mathcal{R})$ means that, in the environment E, the equation eq is transformed into the equation eq' and produces the sets of equations and resets $\mathcal{E}$ and $\mathcal{R}$. They are pretty straightforward.

$$(\textsc{Der})\ \frac{E \vdash e \hookrightarrow_e (e', \mathcal{E}, \mathcal{R})}{E \vdash \texttt{der}\,(x, e) \hookrightarrow_{\mathcal{E}} (\texttt{der}\,(x, e'), \mathcal{E}, \mathcal{R})}$$

$$(\textsc{Reg})\ \frac{E \vdash e \hookrightarrow_e (e', \mathcal{E}, \mathcal{R})}{E \vdash \texttt{reg}\,(x, e) \hookrightarrow_{\mathcal{E}} (\texttt{reg}\,(x, e'), \mathcal{E}, \mathcal{R})}$$

$$(\textsc{Eqs})\ \frac{E \vdash eq_1 \hookrightarrow_e (eq'_1, \mathcal{E}_1, \mathcal{R}_1) \quad \cdots \quad E \vdash eq_n \hookrightarrow_e (eq'_n, \mathcal{E}_n, \mathcal{R}_n)}{E \vdash \{eq_1 \cdots eq_n\} \hookrightarrow_{\mathcal{E}} (\{eq'_1 \cdots eq'_n\}, \mathcal{E}_1 \cup \cdots \mathcal{E}_n, \mathcal{R}_1 \cup \cdots \mathcal{R}_n)}$$

Figure 14: Instantiation inlining in equations

5.3.2 From Pre-automaton to Intermediate Automaton

We now address the rules transforming the pre-automaton of each node into an intermediate automaton. We assume given a function $TopoSort\,()$ performing topological sorting of the equations and a function $CanonSort\,()$ sorting equations or inits by their name.

As presented in Section 5.1, inits also have to be inlined. However, some can depend on other ones (non-recursively). A first step is then to inline inits in themselves. This inlining is performed by the function $InlInit\,(\mathcal{I})$ defined by:

$$\begin{array}{lcl} InlInit\,(\emptyset) & = & \emptyset \\ InlInit\,((x, e) + \mathcal{I}) & = & (x, e) + InlInit\,(\mathcal{I}[x \leftarrow e]) \end{array}$$

Note that there is no node instantiation in inits. The Zélus typechecking mechanism allowing to separate discrete and continuous computations forbids to use an hybrid node in an init equation. Only continuous nodes may be used, which are not considered in this work. For this reason we process inits before and apart the node instantiations inlining.

The regular equations will also have to be inlined to let only differential equations in the final automaton. This operation is achieved by the function $InlReg\,(\mathcal{E})$ defined by:

$$\begin{array}{lcl} InlReg\,(\emptyset) & = & (\emptyset, id) \\ InlReg\,(\texttt{der}\,(x, e) + \mathcal{E}) & = & \texttt{der}\,(x, e) + InlReg\,(\mathcal{E}) \\ InlReg\,(\texttt{reg}\,(x, e) + \mathcal{E}) & = & \text{let } (\varphi, \mathcal{E}') = InlReg\,(\mathcal{E}[x \leftarrow e]) \text{ in } (\mathcal{E}', [x \leftarrow e] \circ \varphi) \end{array}$$

The rules for the translation of the pre-automaton to the intermediate automaton are given in Figure 15. They rely on an environment E mapping a node name to its intermediate automaton. Some rules also require a set $\mathcal{I}$ representing the init equations of the processed pre-automaton.

The rule Sta handles a state of a pre-automaton. It builds the set of equations $\mathcal{E}'$ by applying node instantiations inlining on the equations of the state. It then sorts the regular equations of $\mathcal{E}'$ to ensure that an equation eq_1 used in eq_2 will appear before eq_2 in the sorted set. This ensures that $InlReg\,()$ will properly inline the regular equations together. The set $\mathcal{E}_3$ represents the conversion into regular equations of the inits that are not redefined by equations in the state. The list $\mathcal{E}_4$ stacks in head all the regular equations followed by the differential ones obtained during instantiations inlining and those explicitly defined in the state. The inlining of regular equations is then performed on $\mathcal{E}_4$ to finally obtain the set $\mathcal{D}_5$ only containing differential equations which is also sorted using the aforementioned $CanonSort\,()$. This sorting ensures that the equations are ordered the same way in all the states, hence making easy a consistent mapping to a vector-valued representation of the functions. The substitution φ obtained during regular equations inlining is applied on the transitions and the resets. The resulting state contains the set of only differential equations $\mathcal{D}_6$ and its resets are those explicitly defined in this state plus those obtained by instantiations inlining. The substitution φ is also returned for a consistency verification in the rule Nod.

$$\text{(Sta)}\ \frac{\begin{array}{c} E \vdash \mathcal{E} \hookrightarrow_{\mathcal{E}} (\mathcal{E}', \mathcal{D}', \mathcal{R}') \\ \mathcal{E}_2 = TopoSort\,(\{eq \in \mathcal{E}' \mid eq = \texttt{reg}\,(x,e)\}) \qquad \mathcal{E}_3 = \{\texttt{reg}\,(x,e) \mid (x,e) \in \mathcal{I} \wedge \nexists\, \texttt{reg}\,(x,\_) \in \mathcal{E}' \wedge \nexists\, \texttt{der}\,(x,\_) \in \mathcal{E}'\} \\ \mathcal{E}_4 = \mathcal{E}_3 \cup \mathcal{E}_2 \cup \{eq \in \mathcal{E}' \mid eq = \texttt{der}\,(x,e)\} \cup \mathcal{D}' \\ (\mathcal{D}_5, \varphi) = InlReg\,(\mathcal{E}_4) \qquad \mathcal{D}_6 = CanonSort\,(\mathcal{D}_5) \qquad \mathcal{T}' = \varphi(\mathcal{T}) \qquad \mathcal{R}'_r = \varphi(\mathcal{R}' \cup \mathcal{R}) \end{array}}{E, \mathcal{I} \vdash (N, \mathcal{E}, \mathcal{T}, \mathcal{R}) \rightsquigarrow_{\mathcal{Q}} ((N, \mathcal{D}_6, \mathcal{T}', \mathcal{R}'_r), \varphi)}$$

$$\text{(Aut)}\ \frac{\begin{array}{c} \mathcal{I}' = TopoSort\,(\mathcal{I}) \\ \mathcal{I}_I = InlInit\,(\mathcal{I}') \qquad E, \mathcal{I}_I \vdash q_1 \rightsquigarrow_{\mathcal{Q}} (\underbrace{(N_1, \mathcal{D}_1, \mathcal{T}_1, \mathcal{R}_1)}_{q'_1}, \varphi_1) \quad \cdots \quad E, \mathcal{I}_I \vdash q_n \rightsquigarrow_{\mathcal{Q}} (\underbrace{(N_n, \mathcal{D}_n, \mathcal{T}_n, \mathcal{R}_n)}_{q'_n}, \varphi_n) \\ |\mathcal{D}_1| = \cdots = |\mathcal{D}_n| \qquad \varphi = \varphi_1 \circ \cdots \circ \varphi_n \\ \mathcal{I}_1 = \varphi(\mathcal{I}_I) \qquad \mathcal{I}_2 = \{(x,e) \in \mathcal{I}_1 \mid \texttt{der}\,(x,\_) \in \mathcal{D}_1 \qquad \mathcal{I}_3 = CanonSort\,(\mathcal{I}_2) \end{array}}{E \vdash (\mathcal{I}, \{q_1 \cdots q_n\}) \rightsquigarrow_{\mathcal{A}} ((\mathcal{I}_3, \{q'_1 \cdots q'_n\}), \{\varphi_1 \cdots \varphi_n\})}$$

$$\text{(Nod)}\ \frac{E \vdash \mathcal{A} \rightsquigarrow_{\mathcal{A}} (\mathcal{A}', \{\varphi_1 \cdots \varphi_n\}) \qquad \forall i\ \exists e, \varphi_i(y) = e \qquad \varphi = \varphi_1 \circ \cdots \circ \varphi_n}{E \vdash (f, \mathcal{A}, \{x_1 \cdots x_n\}, y) \rightsquigarrow_{\mathcal{N}} (f, (\mathcal{A}', \{x_1 \cdots x_n\}, \varphi(y))) + E}$$

$$\text{(Prg)}\ \frac{E \vdash n_1 \rightsquigarrow_{\mathcal{N}} E_1 \quad \ldots \quad E_{n-1} \vdash f_n \rightsquigarrow_{\mathcal{N}} E_n}{E \vdash \{f_1 \cdots f_n\} \rightsquigarrow_{\mathcal{P}} E_n}$$

$$\text{(Top)}\ \frac{\emptyset \vdash p \rightsquigarrow_{\mathcal{P}} E \quad \exists\, (\texttt{main},\ (\mathcal{A}, \{x_1, \ldots, x_m\}, y)) \in E \quad \{(z_1, \_) \cdots (z_m, \_)\} = \mathcal{A}.st_1.\mathcal{D} \qquad \sigma = [z_i \mapsto i]}{t_0, t_f \vdash (\mathcal{A}, \{x_1, \ldots, x_m\}, y), t_0, t_f, \sigma}$$

Figure 15: Rules for constructing the intermediate automaton

The rule Aut handles a pre-automaton. It topologically sorts the inits of the pre-automaton then inlines them together to build the set $\mathcal{I}_I$. Each state representation is built and we ensure that they contain the same number of equations. This is mandatory to ensure that all the dynamics are well-defined in all the states. As inits of the resulting automaton, we only keep the ones related to differential equations. Since all the states contain equations with same names we can filter inits by looking in the equations of the first state. Once filtered, inits are sorted using $CanonSort\,()$ to ensure a consistent ordering with the equations.

The rule Nod handles nodes. It computes the automaton representation then ensures that if the node output is the result of a regular equation y, then y has been inlined with the same value in all the states otherwise we would not know by which expression we should substitute the output in the resulting node representation. As result, this rule returns the environment extended by a binding of the node name to the node representation.

The rule Prg simply iterates on the nodes making a program, extending the environment.

Finally, the rule Top processes all the nodes of the program p, then looks for a node named `main`. It builds a substitution σ mapping each equation identifier to an integer representing the rank of the equation. It will be used during the C++ code generation. Since all the equations in the states have been sorted according to the same order and since the number of equations is the same in all the states, we can build σ by only looking at the equations of the first state. We note $\mathcal{A}.st_1.\mathcal{D}$ the set of differential equations of the first state of the automaton $\mathcal{A}$. The final result is a tuple containing the intermediate automaton as defined in Section 5.3, the initial (t_0) and final (t_f) dates of the simulation and the substitution σ.

5.4 C++ Code Generation

There remains to produce the C++ structures representing the automaton and the initial call to the function of the runtime implementing the function `Run` described in Section 4.5. The C++ type definitions of the automaton are given in Figure 16. The type `StateId` is an `enum` generated from the names of the states. The class `Function` is the DynIbex construct representing the syntax tree of a vector-valued function from several arithmetic expressions.

The code generation process iterates on the components of the intermediate automaton to generate the final automaton in C++ by emitting the corresponding definitions. Arithmetic expressions representing dynamics, guards and resets are mostly translated trivially in C++, DynIbex providing overloaded operators

```
struct tr {
  StateId next_st ;
  Function guard ;
};

struct trs_set {
  int nb_trans ;
  struct tr *trs ;
};

struct automaton {
  Function *dyn_of_state ;
  struct trs_set **trs_by_state ;
  Function **reset_of_state ;
};
```

Figure 16: C++ type definition representing an automaton

dealing with intervals. The substitution σ returned by the rule `Top` allows to replace identifiers by accesses in the arrays used for DynIbex's vector-valued representation of the equations.

We illustrate the generated code, on the basis of the example given in Section 2. Note that the resets of the automaton are the identity for each state since this system contains no ODE local to a state with an initial value. The description of the `main` node is $(\mathcal{A}, \emptyset, zpos)$ where:

$\mathcal{A} = (\{ (power, 100), (speed, 0), (zpos, 0) \}, \mathcal{Q})$
$\mathcal{Q} = \{ (EngOn, \mathcal{E}_1, \mathcal{T}_1, \mathcal{R}_1), (EngOff, \mathcal{E}_2, \mathcal{T}_2, \mathcal{R}_2), (Crashed, \mathcal{E}_3, \emptyset, \mathcal{R}_3) \}$
$\mathcal{E}_1 = \{ (power, -2 * power), (speed, -9.81 + power), (zpos, speed) \}$
$\mathcal{E}_2 = \{ (power, -2 * power), (speed, -9.81), (zpos, speed) \}$
$\mathcal{E}_3 = \{ (power, -2 * power), (speed, 0), (zpos, 0) \}$
$\mathcal{T}_1 = \{ (EngOff, -(power - 0.001)) \}$
$\mathcal{T}_2 = \{ (Crashed, -zpos) \}$
$\mathcal{R}_1 = \mathcal{R}_2 = \mathcal{R}_3 = \{ (zpos, zpos), (speed, speed), (power, power) \})$

Instead of giving formal code generation rules, we prefer to discuss on the basis of the generated code. This makes easier the global understanding of the structure of the code and gives the intuition of the easy translation from the intermediate automaton structure to C++ code. Some parts of the code are omitted and replaced by the ellipsis `(...)` to shorten the listing. In the following listing, the mapping (σ) from the coupled equations *zpos*, *speed*, *power* to the vector-valued representation assigns *zpos* to the dimension 0, *speed* to the dimension 1 and *power* to the dimension 2.

```
1  enum StateId { EngOn, EngOff, Crashed };
2
3  int main () {
4    const int dim = 3 ;
5    Variable y (dim) ;
6
7    Function EngOn_dynamics =
8      Function (y, Return (-2 * y[0], -9.81 + y[0], y[1])) ;
9    (... Idem with dynamics of states EngOff, Crashed)
10
11   Function dyn_of_state[] = { EngOn_dynamics, EngOff_dynamics,
12     Crashed_dynamics };
13
14   struct tr tra_EngOn[] = { { EngOff,
15     Function (y, - (y[0] - 0.001)) } };
16   struct tr tra_EngOff[] = { { Crashed,
17     Function (y, -y[2]) } };
18   struct tr tra_Crashed[] = { };
19   struct trs_set trs_EngOn = { 1, tra_EngOn } ;
20   struct trs_set trs_EngOff = { 1, tra_EngOff } ;
21   struct trs_set trs_Crashed = { 0, NULL } ;
22   struct trs_set* trs_by_state[] = { &trs_EngOn, &trs_EngOff,
23     &trs_Crashed };
24
25   Function reset_EngOn = Function (y, Function (y[0], y[1], y[2])) ;
26   (... Idem with resets of states EngOff, Crashed)
27
28   Function *reset_of_state[] = { &reset_EngOn, &reset_EngOff,
29     &reset_Crashed };
30   struct automaton automaton = { dyn_of_state, trs_by_state,
31     reset_of_state  };
32
33   IntervalVector yinit (dim) ;
34   yinit[0] = Interval (100.) ; yinit[1] = Interval (0.) ;
35   yinit[2] = Interval (0.) ;
36   if (reset_of_state[EngOn])
37     yinit = (autom->reset_of_state[state])->eval_vector (yinit) ;
38   SimuNode *root =
39     run_state (&automaton, EngOn, dim, yinit, 0., GLOBAL_T_END) ;
40   return 0 ;
```

```
41 }
```

At Line 1, an `enum` is defined from the names of the states. The dimension of the system is defined at Line 4. At Line 5, `y` represents the continuous state of the system. It is a vector of intervals that will be accessed in accordance with the aforementioned mapping. At Line 7, the data-structure representing the dynamics of the state `EngOn` is created as a Dynlbex `Function` object which represents the equations. This code is also emitted for the dynamics of the other states. Line 11, all the dynamics are grouped in an array indexed by states (values of the `enum` which are integers starting from 0). Lines $14-18$, the transitions are defined. The one from `EngOn` goes to `EngOff` with a `Function` object representing the guard $-(power-0.001)$. Lines $19-21$, sets of transitions are created for each state. This is mostly a utility data-structure to record the number of transitions in each array. Line 22, transitions sets are also grouped in an array indexed by states. Line 25, the resets of the `EngOn` state are created in a way similar to the creation of the dynamics. One especially sees that resets are identity in this system since for each dimension the `Function` returns the continuous state of this dimension. This code is also emitted for the dynamics of the other states. Line 28, resets are grouped in an array indexed by states. Line 30, the complete automaton is created by grouping all the previously created data-structures. Lines $34-35$, the initial condition of the automaton is created. It is a vector of intervals initialized according to the aforementioned mapping. Lines $36-37$ the reset of the initial state is applied. Finally, lines $38-39$ the call to the simulation function of the runtime is performed, returning the root of the tree of simulations.

Aside this core code, a few additional lines are generated, some in a separate header file, to specify some external settings given when compiling the Zélus program (integration method, starting and ending simulation dates, number of noise symbols, integration precision, etc.).

6 Experimental Results

We extended the Zélus compiler to implement the described compilation process. This new backend operates on the intermediate representation obtained after type, causality and initialization analyses and does not interfere with the standard compilation.

Based on the rocket example given in Section 2, the first experiment is to simulate the system with Zélus native point-wise computation and with our generated code, then to compare the results. Despite the internal computation is performed using zonotopes, the final display is rendered using rectangular boxes.

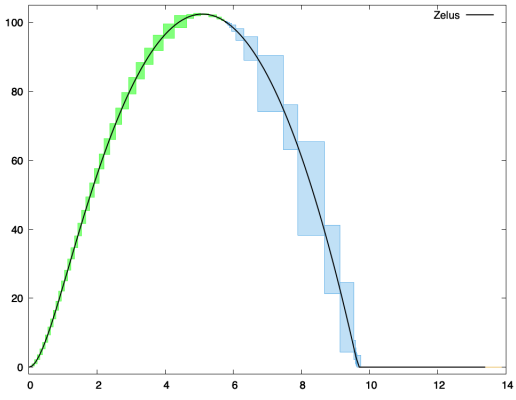

Figure 17: Simulations with/without intervals

In Figure 17, the Zélus simulation until $t = 15$ is represented by the black curve and the simulation obtained using intervals is shown by the colored boxes. Both simulations behave consistently. The results obtained with the standard integration runtime of Zélus always remain inside the boxes obtained using the intervals mechanism. This suggests that the native integration runtime of Zélus is precise enough in this example to avoid inaccuracies that could be caused by float rounding errors. Note that we draw Picard boxes which look quite large. However the boxes at each t_i are really smaller. Plotting them instead would have rendered a less clear visual effect, with small rectangles scattered at each discretization point.

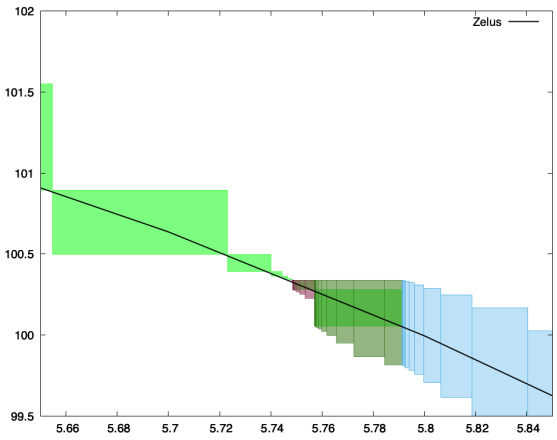

Figure 18: Zooming on sub-simulations

Figure 18 makes a zoom on the extent where the guard of fuel exhaustion crosses 0. It is possible to see, by transparency, the part of the initial simulation (green) where the guard crosses 0. During this time interval, different sub-simulations are performed, each one on a green box crossing the guard. Once the guard totally crossed 0, the new dynamics (`EngineOff`) can start with the initial conditions being the union of reachable values during the sub-simulations. Note that a similar behavior occurs when going in the state `Crashed`, however this is not visible on the picture because *zpos* becomes constant and null, which makes very thin boxes covered by the plot of the point-wise simulation.

The tree of simulations is shown in Figure 19. Since no guard is crossed during the sub-simulations, the tree does not branch and remains linear. The time interval of the simulation is displayed in each node. We can remark that the 11^{th} node (marked with an asterisk) corresponds to a simulation with the new dynamics once the guard is totally above 0 since its time interval spans until the end of the global simulation.

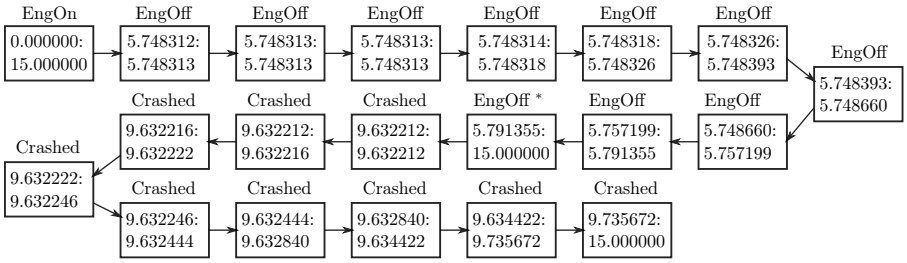

Figure 19: Tree of simulations for the rocket example

We extended the syntax of Zélus to specify an alternative interval value for any float value. This interval is only taken into account when compiling toward Dynlbex. It is then possible to add uncertainty on the initial value of *zpos* to make it belonging to $[0; 20]$ by modifying its definition into **rec init** zpos =0.0 [0.0; 20.0].

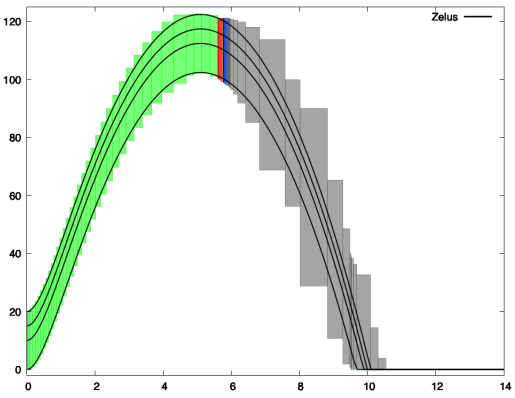

Figure 20: Point-wise simulations vs interval simulation

The simulation obtained is displayed in Figure 20. The effects of uncertainty are visible as soon as the simulation begins since the initial altitude is now a box ranging from 0 to 20. This uncertainty also affects the future of the simulation with taller boxes indicating that the rocket may reach a higher altitude.

Despite a loss of precision, simulating with intervals to model uncertainties allows one to run one unique simulation instead of several point-wise simulations to try to cover the entire range of uncertainty (and possibly miss an important point such as a singularity). The simulation result is less accurate but safe and guaranteed. Running several point-wise simulations raises the question of how many to run and which values to chose, with no guaranty that the chosen

strategy covers all the possible behaviors. Figure 20 shows the inclusion of several point-wise simulations in a single interval-based simulation. We plot several point-wise simulations in the range of uncertainty of *zpos* (0, 10, 15 and 20). As one expects for safety purposes, the interval-based simulations cover all the possible ones.

7 Contracts

Visualization of the tree of simulations becomes complicated as soon as there are branches. It is difficult to plot several disjoint futures in a linear way. However, an interesting use of the tree is the verification of properties since it contains all the possible evolutions of the modeled system.

We extended the syntax of Zélus to allow two simple forms of contracts on nodes. A contract is a property that must hold during all the simulations: it is implicitly an "always" property. Contracts must appear above the node definition they are related to. If a node has several contracts, they must all hold: it is implicitly a conjunction.

contract	::=	`safe` $(x$ `in` $[b_1, b_2])^*$	Interval belonging
	|	`constraint` a	Constraint < 0
a	::=	$r \mid x \mid$ `op` $a \mid a_1$ `op` a_2	Arithmetic expression
b	::=	$r \mid -$`oo` $\mid +$`oo`	Bounds

Figure 21: Syntax for contracts

The syntax of contracts is given in Figure 21. A contract may be a belonging constraint, giving for each relevant equation of the program the interval it must stay in. Otherwise, it may be an arbitrary arithmetic expression that must implicitly be lower than 0. A bound b of interval is a float r or two constants representing infinities. An arithmetic expression a can be a float, an identifier or the application of an unary or binary arithmetic operator (including standard math functions).

7.1 Overview of Contracts Compilation

Compiling contracts does not change the compilation principle described in Section 5. We do not provide the formal rules for the handling of the contract in the pre-automaton and intermediate automaton since the compilation is pretty trivial.

Expressions in contracts are handled the same way as in other parts of a program. The various substitutions built during the compilation are also applied to the identifiers in contracts to keep consistency with the remaining of the code. When a node with a contract is instantiated, its contracts are added to those of the instantiating node (with the right renaming of its identifiers).

During the C++ code generation, if contracts are detected, an extra function `check` is generated. This function is a recursive traversal of the tree of simulations, checking if the compiled properties hold on each box of the tubes of the tree nodes. Contracts are compiled differently according to their shape. They lead to Boolean results that are finally combined by a logical "and" to obtain the global truth value.

`safe` contracts rely on the function `stayed_in` of DynIbex. This function takes a tube t, a vector v of intervals and returns a Boolean value indicating if all the boxes of t are included in v. The vector v is built from the `in` clauses of the contract. Since the user only specifies safety bounds for the equations he is interested in, v must be filled with $[-\infty; +\infty]$ for the dimensions of equations not appearing in the contract.

`constraint` contracts rely on the method `eval_vector` of a `Function` object f of DynIbex. This method takes a vector v of intervals and returns the result of f applied to v. The object f is created by translating the arithmetic expression of the contract, the same way than other expressions. There simply remains to check if the upper bounds of the resulting intervals are lower than 0. This verification is performed by a function `stayed_below_zero` of the simulation runtime.

To illustrate the translation of contracts, we consider the simple Zélus program in Figure 22. It defines four trivial ODEs and states three contracts on the system.

```
{| safe x1 in [0.0, +oo] x2 in [0.0, +oo] ;
   safe x4 in [0.5, 100.0] ;
   constraint x2 -. x3 -. 1. ; |}
let hybrid main () = x1 where
  rec der x1 = 1.0 init 1.0 and der x2 = 2.0 init 1.0
  and der x3 = 3.0 init 1.0 and der x4 = 3.0 init 1.0
```

Figure 22: Contracts in Zélus

The first `safe` contract involves only 2 of the 4 variables. The second contract shows that is it possible to have several `safe` contracts, even if it could have been merged with the first one. Due to node instantiation inlining, it is possible to get several distinct contracts. The third contract illustrates the `constraint` construct. The function `check` obtained after the compilation follows, in which the variable mapping σ to vector-based representation is [x1 ← 0; x2 ← 1; x3 ← 2; x4 ← 3].

```
void check (SimuNode *node, int dim) {
  if (node != NULL) {
    std::cout << "Check on [" << node->t0 << " : " << node->tend << "]" << std::endl ;
    IntervalVector safe_0 (dim) ;
    safe_0[3] = Interval (NEG_INFINITY, POS_INFINITY) ;
    safe_0[2] = Interval (NEG_INFINITY, POS_INFINITY) ;
    safe_0[0] = Interval (0., POS_INFINITY) ;
    safe_0[1] = Interval (0., POS_INFINITY) ;
    bool contractp_0 = stayed_in (node->tube, &safe_0) ;
    IntervalVector safe_1 (dim) ;
    safe_1[2] = Interval (NEG_INFINITY, POS_INFINITY) ;
    safe_1[1] = Interval (NEG_INFINITY, POS_INFINITY) ;
    safe_1[0] = Interval (NEG_INFINITY, POS_INFINITY) ;
    safe_1[3] = Interval (0.5, 100.) ;
    bool contractp_1 = stayed_in (node->tube, &safe_1) ;
    Variable y_2 (dim) ;
    Function constraint_2_fun =
      Function (y_2,
                Return (y_2[1] - y_2[2] - 1, Interval (NEG_INFINITY),
                        Interval (NEG_INFINITY), Interval (NEG_INFINITY))) ;
    bool contractp_2 =
      stayed_below_zero (dim, node->tube, constraint_2_fun) ;
    if (!(contractp_0 && contractp_1 && contractp_2))
      std::cout << "Violated!" << std::endl ;
  }
}
```

The two first contracts are compiled into two separate checks despite they are of the same kind. For each, a vector of intervals representing the safe zone is created, with $[-\infty; +\infty]$ in components not involved in the contract (for `x3` and `x4` in the first contract). The truth value of each contract is obtained by the aforesaid function `stayed_in` of DynIbex. The third contract is compiled into a `Function` expression representing the constraint. Since the dimension of the system is 4, even if the constraint's result is of dimension 1, it must be compiled as a `Function` of dimension 4. The three remaining dimensions are filled with an interval trivially always below 0 (`NEG_INFINITY`). Finally, the conjunction of the truth values of the contracts is tested to emit an error message in case of violation of one of the contracts on the tube of the current node.

8 Future Work

This work is the continuation of [11] (which only handled IVPs without jumps, modes and contracts), with noticeable differences in the compilation process and the handled constructs. Several extension directions are possible. The first one is to consider reset on differential equations as they are supported by Zélus. This would allow, for instance, to easily write the well-known "bouncing ball" model. The second direction is to handle hierarchical automata. This would certainly require deep modifications in the simulation runtime. Currently, we think we can't take benefit from the transformation already performed by Zélus. The third direction is to increase the expressiveness of the contracts, allowing explicit temporal properties. Currently, the property of a contract must be valid on all the boxes of a tube: it implicitly means "always" (or "never"). Finally, a last extension of this work is to address the discrete behavior it is possible to model in Zélus. The

programs currently supported only contain continuous computation, which allows to model the dynamics of a physical system but not a controller coupled to this system.

9 Conclusion

We presented a mechanism to simulate systems having several dynamics, modeled in Zélus using automata, a generic runtime supporting the simulation, the compilation schema translating Zélus programs into C++ code using DynIbex and the extension of the language with some forms of contracts. This shows the possibility to build tools allowing the simulation of programs written in a high-level programming language with interval-based validated numerical integration methods. Various parameters can be set at compile-time to tune the simulation accuracy. This work is fully implemented in the Zélus compiler.

Acknowledgments

This work was partially supported by the "Chair Complex Systems Engineering – Ecole polytechnique, Thales, DGA, FX, Dassault Aviation, Naval Group Research, ENSTA Paris, Télécom Paris, and Fondation ParisTech" and by DGA AID.

References

[1] Julien Alexandre dit Sandretto, Alexandre Chapoutot, and Olivier Mullier. Dynibex lib. `https://perso.ensta-paris.fr/~chapoutot/dynibex/`.

[2] Matthias Althoff. An introduction to CORA 2015. In *Proc. of Applied Verification for Continuous and Hybrid Systems*, 2015.

[3] Matthias Althoff and Dmitry Grebenyuk. Implementation of interval arithmetic in CORA 2016. In *ARCH@CPSWeek*, volume 43 of *EPiC Series in Computing*, pages 91–105. EasyChair, 2016.

[4] Matthias Althoff, Dmitry Grebenyuk, and Niklas Kochdumper. Implementation of taylor models in CORA 2018. In *ARCH@ADHS*, volume 54 of *EPiC Series in Computing*, pages 145–173. EasyChair, 2018.

[5] Abhishek Anand and Ross Knepper. ROSCoq: Robots Powered by Constructive Reals. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, volume 9236, pages 34–50. Springer, 2015.

[6] Andrea Balluchi, Alberto Casagrande, Pieter Collins, Alberto Ferrari, Tiziano Villa, and Alberto Sangiovanni-Vincentelli. Ariadne : a framework for reachability analysis of hybrid automata. 01 2006.

[7] L. Benvenuti, A. Ferrari, E. Mazzi, and A. L. Sangiovanni Vincentelli. Contract-based design for computation and verification of a closed-loop hybrid system. In *Hybrid Systems: Computation and Control*, pages 58–71. Springer Berlin Heidelberg, 2008.

[8] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. Veriphy: Verified controller executables from verified cyber-physical system models. *SIGPLAN Not.*, 53(4):617–630, June 2018.

[9] Olivier Bouissou, Samuel Mimram, and Alexandre Chapoutot. Hyson: Set-based simulation of hybrid systems. In *Proceedings of IEEE International Symposium on Rapid System Prototyping*, pages 79–85, 2012.

[10] Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with odes. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, pages 113–118. ACM, 2013.

[11] Jason Brown and François Pessaux. Interval-based simulation of zélus ivps using dynibex. *Acta Cybernetica*, Aug. 2020.

[12] Luca P. Carloni, Roberto Passerone, Alessandro Pinto, and Alberto L. Angiovanni-Vincentelli. Languages and tools for hybrid systems design. *Found. Trends Electron. Des. Autom.*, 1(1):1–193, 2006.

[13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Proc. of Int. Conference on Computer Aided Verification*, volume 8044 of *LNCS*, pages 258–263. Springer, 2013.

[14] Julien Alexandre dit Sandretto and Alexandre Chapoutot. Validated explicit and implicit Runge–Kutta methods. *Reliable Computing*, 22(1):79–103, 2016.

[15] Parasara Sridhar Duggirala, Sayan Mitra, Mahesh Viswanathan, and Matthew Potok. C2e2: A verification tool for stateflow models. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–82. Springer Berlin Heidelberg, 2015.

[16] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *Proc. of Int. Conference on Computer Aided Verification*. Springer, 2011.

[17] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In *Automated Deduction*, pages 527–538, Cham, 2015. Springer.

[18] S. Gao, S. Kong, and E. M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *International Conference on Automated Deduction*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.

[19] Sicun Gao, Soonho Kong, and Edmund M. Clarke. Satisfiability modulo odes. *CoRR*, abs/1310.8278, 2013.

[20] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[21] Thomas A. Henzinger, Benjamin Horowitz, Rupak Majumdar, and Howard Wong-Toi. Beyond hytech: Hybrid systems analysis using interval numerical methods. In *Hybrid Systems: Computation and Control*, pages 130–144. Springer Berlin Heidelberg, 2000.

[22] Fabian Immler, Matthias Althoff, Xin Chen, Chuchu Fan, Goran Frehse, Niklas Kochdumper, Yangge Li, Sayan Mitra, Mahendra Singh Tomar, and Majid Zamani. Arch-comp18 category report: Continuous and hybrid systems with nonlinear dynamics. In *Proc. of Applied Verification of Continuous and Hybrid Systems*, volume 54 of *EPiC Series in Computing*, pages 53–70. EasyChair, 2018.

[23] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis*. Springer, 2001.

[24] Michal Konečný, Walid Taha, Jan Duracz, Adam Duracz, and Aaron Ames. Enclosing the behavior of a hybrid system up to and beyond a zeno point. In *Proc of IEEE cyber-physical systems, networks, and applications*, pages 120–125, 2013.

[25] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[26] Moussa Maïga, Nacim Ramdani, and Louise Travé-Massuyès. A fast method for solving guard set intersection in nonlinear hybrid reachability. In *IEEE CDC*, pages 1–6, Firenze, Italy, December 2013.

[27] Stefano Minopoli and Goran Frehse. Sl2sx translator: From simulink to SpaceEx models. pages 93–98, 04 2016.

[28] Ramon E. Moore. *Interval Analysis*. Series in Automatic Computation. Prentice Hall, 1966.

[29] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. A component-based approach to hybrid systems safety verification. In *Integrated Formal Methods*, pages 441–456. Springer, 2016.

[30] Andreas Müller, Stefan Mitsch, Werner Retschitzegger, Wieland Schwinger, and André Platzer. Change and delay contracts for hybrid system component verification. In *Fundamental Approaches to Software Engineering*, pages 134–151. Springer, 2017.

[31] Thuy Nguyen. Form-l: A modelica extension for properties modelling illustrated on a practical example. In *Proc of Int. Modelica Conference*, pages 1227–1236, 2014.

[32] Martin Otter, Nguyen Thuy, Daniel Bouskela, Lena Rogovchenko-Buffoni, Hilding Elmqvist, Peter Fritzson, Alfredo Garro, Audrey Jardin, Hans Olsson, Maxime Payelleville, Wladimir Schamai, Eric Thomas, and Andrea Tundis. Formal requirements modeling for simulation-based verification. In *Proc. of Int. Modelica Conference*, pages 625–635, 2015.

[33] A. Saoud, A. Girard, and L. Fribourg. On the composition of discrete and continuous-time assume-guarantee contracts for invariance. In *2018 European Control Conference (ECC)*, pages 435–440, 2018.

[34] Yingfu Zeng, Chad G. Rose, Paul Brauner, Walid Taha, Jawad Masood, Roland Philippsen, Marcia K. O'Malley, and Robert Cartwright. Modeling basic aspects of cyber-physical systems, part II. *CoRR*, abs/1408.1110, 2014.

[35] Fu Zhang, Murali Yeddanapudi, and Pieter J. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. *IFAC Proceedings Volumes*, 41(2):7967 – 7972, 2008. 17th IFAC World Congress.

[36] Liang Zou, Naijun Zhany, Shuling Wang, Martin Fränzle, and Shengchao Qin. Verifying simulink diagrams via a hybrid hoare logic prover. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2013.