Acta Cybernetica **25** (2021) 5–20.

Interval-based Simulation of Zélus IVPs using DynIbex[*]

Jason Brown[a] and François Pessaux[b]

Abstract

Modeling continuous-time dynamical systems is a complex task. Fortunately some dedicated programming languages exist to ease this work. Zélus is one such language that generates a simulation executable which can be used to study the behavior of the modeled system. However, such simulations cannot handle uncertainties on some parameters of the system. This makes it necessary to run multiple simulations to check that the system fulfills particular requirements (safety for instance) for all the values in the uncertainty ranges. Interval-based guaranteed integration methods provide a solution to this problem. The DynIbex library provides such methods but it requires a manual encoding of the system in a general purpose programming language (C++). This article presents an extension of the Zélus compiler to generate interval-based guaranteed simulations of IVPs using DynIbex. This extension is conservative since it does not break the existing compilation workflow.

Keywords: DynIbex, Zélus, compilation, hybrid system, interval, guaranteed integration, simulation

1 Introduction

Hybrid systems are commonly defined as dynamical systems mixing discrete and continuous times. They are widely present in control command systems where a continuous physical process is controlled by software components which run at discrete instants. The implementation of such software components involved in many critical systems has to be verified to ensure that the behavior of the global system does not lead to any critical event. One of the verification techniques is to simulate

[*]This research benefited from the support of the "Chair Complex Systems Engineering - École Polytechnique, THALES, DGA, FX, DASSAULT AVIATION, DCNS Research, ENSTA ParisTech, Télécom ParisTech, Fondation ParisTech and FDO ENSTA" and it is also partially funded by DGA AID. We warmly thank Julien Alexandre dit Sandretto, Alexandre Chapoutot and Olivier Mullier for the many discussions we had to achieve this work.

[a]University of Melbourne, Australia. E-mail: `jason.brown@unimelb.edu.au`, ORCID: `https://orcid.org/0000-0002-1235-3722`.

[b]U2IS, ENSTA Paris, France. E-mail: `francois.pessaux@ensta-paris.fr`, ORCID: `https://orcid.org/0000-0002-4219-3854`.

DOI: 10.14232/actacyb.285246

the global system. In such a simulation process, the continuous physical process is modeled as differential equations whose solutions are approximated by dedicated integration algorithms. The discrete processing is the software components. Both parts of the system have to interact, allowing the discrete process to react to events of the continuous one.

Simulations can be very dependent on the initial conditions of the system. Small variations may have important impacts. Moreover, the initial conditions may not always be accurately known. A solution to address these uncertainties is to compute using intervals, hence to rely on interval-based integration tools [10, 14, 2, 3, 4, 17] possibly with guaranteed arithmetic [5, 8, 13].

Domain Specific Languages and tools exist to ease the modeling, development and verification of hybrid systems (MODELICA, SIMULINK/STATEFLOW, LABVIEW, Zélus and others [7]). These languages provide numerous advantages compared to a manual implementation requiring to explicitly bind the code of the software components with the runtime/library of simulation. They often propose high-level constructs (automata, differential equations, guards) with dedicated static verifications (typechecking, initialization analysis, scheduling, causality analysis) and compile the hybrid model to low-level code (C, C++) to produce an executable simulation.

This work proposes to bind the flexibility of a hybrid programming language, Zélus[6], with the safety of interval-based guaranteed integration using DynIbex[1, 9]. Zélus natively generates imperative OCaml code linked with a point-wise simulation runtime. DynIbex is a plug-in of the C++ Ibex library, bringing various validated numerical integration methods to solve Initial Value Problems (*IVPs*). We do not address the compilation of arbitrary Zélus programs toward DynIbex. We present instead the compilation schema for an IVP described in a subset of Zélus to a C++ simulation code using DynIbex. An example is presented, showing the results obtained with the standard Zélus simulation and with the interval-based one.

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 an IVP using the library. Section 4 addresses the compilation schema. Experimental results are described in Section 5. Section 6 is devoted to related work and we conclude and comment on possible further works in Section 7.

2 Zélus Succinctly, Used Features

Zélus 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 IVPs. Zélus makes it possible to model systems in which there is an interaction between discrete-time and continuous-time dynamics. In this work, we do not address any discrete-time behavior.

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. An IVP is represented by a system of coupled equations coming from the equations defined in a node and those imported from instantiated nodes.

The model of a simple harmonic oscillator with dampening, described by the equation $\ddot{x} + k_2\,\dot{x} + k_1\,x = 0$ with initial values $x(0) = 1,\ \dot{x}(0) = 0$, can be written in Zélus as:

```
let hybrid shm_decay (x0, x'0, k1, k2) = x where
  rec der x = x' init x0
  and der x' = -.k1 *. x -. k2 *. x' init x'0

let hybrid main () = y where
  y = shm_decay (1.0, 0.0, 4.0, 0.4)
```

where `der x` represents $\dot{x}$ and `der x'` is $\ddot{x}$. The node `main` instantiates the node `shm_decay` with specific initial values and k_1 and k_2. Note that floating-point arithmetic operators are suffixed by a dot in Zélus.

3 DynIbex in a Few Words, Used Features

DynIbex is a C++ library that builds on the Ibex library. Ibex provides tools to develop programs for constraint processing over real numbers using interval arithmetic and affine arithmetic. DynIbex adds validated numerical integration methods (including handling of floating-point rounding errors). To describe an IVP one defines objects of predefined classes to represent the initial values and the ODEs of the system, using 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. Once these objects are defined, a dedicated function is called to perform the simulation with the desired parameters (integration method, duration, precision, etc.).

Using the example introduced in Section 2, we show in Figure 1 the expected result of the compilation into C++ code, where the red parts represent code dependent on the compiled IVP. After instantiation of the node `shm_decay` with the effective arguments, the set of equations is `der` $x = x'$ `init` 1 and `der` $x' = -4x - 0.4x'$ `init` 0.

The variable `dim` represents the number of equations of the system, `y` represents the continuous state of the system, `ydot` encodes the differential equations in the `Return` expression. The mapping from the coupled equations `x` and `x'` to the vector-valued representation assigns `x` to the dimension 0 (`y[0]`) and `x'` to the dimension 1 (`y[1]`). All the numerical constants are transformed into single-point intervals taking rounding issues in account. If a float cannot exactly be represented, the obtained interval is the smallest containing this float. The initial conditions of the problem are stored in `yinit`. The number of noise symbols is set using

```
#define T0 (0.0)
#define TEND (6.0)
#define PREC (1e-8)
#define NOISESYMBS (150)
int main () {
  const int dim = 2 ;
  Variable y (dim) ;
  IntervalVector yinit (dim) ;
  Function ydot =
    Function (y, Return (y[1], ((-Interval (4.0)) * y[0]) - (Interval (0.4) * y[1]))) ;
  yinit[0] = Interval (1.0) ;
  yinit[1] = Interval (0.0) ;
  ibex::AF_fAFFullI::setAffineNoiseNumber (NOISESYMBS) ;
  ivp_ode problem = ivp_ode (ydot, T0, yinit) ;
  simulation simu = simulation (&problem, TEND, LC3, PREC) ;
  simu.run_simulation () ;
  simu.export_y0 ("export") ;
  return 0 ;
}
```

Figure 1: IVP encoding in DynIbex (targeted generated C++ code)

`setAffineNoiseNumber`. An IVP 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.

4 Compiling an IVP

4.1 Overview

For several reasons, it is impossible to 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 tightly dependent on the ODE solver used by Zélus and the solving runtime is very different from DynIbex's mechanisms. Second, the runtime simulation code is deeply mixed with the IVP problem code, making impossible to distinguish between code to be translated into C++, code to be transformed into intervals and code which should be ignored. Finally, intervals are strongly incompatible with point-wise simulation. General Zélus programs may contain discrete code running on events, using continuous values. When working with intervals, these values are no longer exact which makes, for instance, conditional constructs fuzzy : a test may be true and false at the same time. In such a situation, a usual computation cannot be carried out anymore.

For these reasons, we need a dedicated compilation schema to bind Zélus and DynIbex and decide to address only IPV in this work.

In this context, compiling the Zélus code requires two steps. First, the hierarchy of nodes must be flattened, harvesting all the differential equations and their initial values. During this process, each node instantiation expression is 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. Since DynIbex simulation only accepts a system of differential equations as input, regular dataflow equations must be transformed into differential equations by symbolic differentiation.

Once the intermediate representation of the flattened system is obtained, the multiple equations have to be aggregated into a unique vector-valued function to finally generate the C++ code. Each differential equation corresponds to one dimension of the DynIbex `Function` data-structure. Initial conditions are also transformed into interval vectors. During this process, Zélus expressions are compiled to C++ expressions. Since nodes are flattened, leading to a list of equations, 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 proper rounded intervals (i.e. the smallest interval containing the translated float).

4.2 Compiling to the Intermediate Representation

The restricted syntax of programs addressed in this work is given in Figure 2.

p	::=	n^+	Program
n	::=	`hybrid` $f\,(x^*)$ `=` y `where` eq^+	Node
eq	::=	`der` x `=` e_1 `init` e_2	Differential equation
	|	$x = e$	Regular dataflow equation
e	::=	r	Real numeric constant
	|	x	Identifier
	|	e_1 `op` e_2	Arithmetic expression
	|	$f\,(e^*)$	Node instantiation

Figure 2: Syntax subset of Zélus for IVP

A program p is a list of nodes. A node n is the definition of a parameterized component returning a value y which is the result of one of the equations eq defined in this node. An equation eq may be a differential equation with an initial value e_2 or a regular dataflow equation binding an expression e to an identifier x. Expressions are numeric constants, identifiers, arithmetic expressions or the instantiation of a node named f with expressions. Node instantiations cannot be self-recursive.

The elaboration of the intermediate representation of an IVP proceeds recursively on the Abstract Syntax Tree of the program. The inlining pass relies on an environment E, a partial function from node names to *node descriptions.* We note $\langle\!\langle\, n, d \,\rangle\!\rangle$ the environment mapping the node name n to its description d and use the + symbol to denote the addition of a new binding to an environment. A node description is a triplet $(\mathcal{I}, y, \mathcal{E})$ where $\mathcal{I}$ is the list of the node's formal parameters, y is the output of the node (i.e. the name of one of its equations), $\mathcal{E}$ is the list of *ivpeqs* describing the ODEs of the node. An *ivpeq* is a triplet $\langle\, x, e_1, e_2 \,\rangle$ where x is the name of the differential equation, e_1 is its expression and e_2 is its initial value.

The node instantiation inlining requires a substitution operation on expressions and *ivpeqs*. Such a substitution is a partial application with a finite domain from identifiers to expressions. We denote by $[x_1 \leftarrow e_1; \ldots; x_n \leftarrow e_n]$ the substitution of the domain $\{x_1, \cdots, x_n\}$ associating the expression e_i to the identifier x_i. We extend a substitution to an application from expressions to expressions and from *ivpeqs* to *ivpeqs* as described in the Figure 3.

$$\begin{array}{lcl}
\langle x, e_1, e_2 \rangle[y \leftarrow e] & = & \langle x, e_1[y \leftarrow e], e_2[y \leftarrow e] \rangle \\
r[y \leftarrow e] & = & r \\
x[y \leftarrow e] & = & x \text{ if } x \neq y \\
x[x \leftarrow e] & = & e \\
(e_1 \; \texttt{op} \; e_2)[x \leftarrow e] & = & e_1[x \leftarrow e] \; \texttt{op} \; e_2[x \leftarrow e] \\
f\,(e_1, \ldots e_n)[x \leftarrow e] & = & f\,(e_1[x \leftarrow e], \ldots e_n[x \leftarrow e])
\end{array}$$

Figure 3: Substitution in an *ivpeq* and expressions

Figure 4 gives an example of the inlining process. In gray is the Zélus code. When processing the node `foo` no changes occur since it contains no instantiation expression. The node `bar` contains two instantiations. For each of them we import the contents of the node where the formal parameters have been replaced by the effective arguments of the instantiation. Then we add this new equation and we replace the instantiation expression by the name of this new equation. Note that to avoid name conflicts, equations are renamed during this process.

```
let hybrid foo (a, b, c) = v where
  der v = -9.8 - c * a init b
```

```
foo: v0, v1, v2 -> v3
  der v3 = -9.8 - (v2 * v0) init v1
```

```
let hybrid bar (y, z, k) = (x1) where
  rec der x2 = foo (x2 - x1, 0, k + 0.5) init 2 + 3
  and der x1 = foo (x2 * x1, z, k) init y
```

```
bar: v0, v1, v2 -> v4
  der v3 = v5 init (2 + 3)
  der v4 = v6 init v0
  der v5 = -9.8 - ((v2 + 0.5) * (v3 - v4)) init 0
  der v6 = -9.8 - (v2 * (v3 * v4)) init v1
```

```
let hybrid main () = x where
  x = bar (1, (1 / 2), 3.14)
```

```
main: -> v0
  der v0 = v3 init 1
  der v1 = v2 init (2 + 3)
  der v2 = -9.8 - ((3.14 + 0.5) * (v1 - v0)) init 0
  der v3 = -9.8 - (3.14 * (v1 * v0)) init (1 / 2)
```

Figure 4: Node instantiation inlining example

Since the `Function` class of DynIbex can only encode ODEs, it is impossible to directly express the equations for `u` or `v` in the following example.

```
hybrid foo () = u where
  rec der z = u init 3.0
  and u = 5.0 +. z +. v
  and v = 1. +. z
```

We cannot simply inline the body of `u` at each occurrence in the equations since `u` is the return identifier of the node. The solution is to symbolically differentiate `5 + z + v`, i.e. create the expressions corresponding to $\delta(\texttt{5}+\texttt{z}+\texttt{v})$ and the initial value of `5 + z + v`. Inside this expression, the identifier `z` represents an ODE, so its derivative is exactly the body of `z`. However, `v` represents a regular dataflow equation. Hence `v` must be replaced by the (recursive) differentiation of its body. Thus, the whole equation is replaced by `der u = 0 + u + 0 + u`. The initial value of this new equation is the original body of `u` in which `z` is replaced by the initial value of `der z` and `v` is replaced by its body in which we recursively apply the transformation. The obtained result is `init 5 + 3 + 1 + 3`. In summary, while processing a node definition, any equation defined by $\texttt{y} = e$ must be replaced by $\texttt{der y = }\mathcal{B}(e)\ \texttt{init}\ \mathcal{I}(e)$ where $\mathcal{B}(e)$ and $\mathcal{I}(e)$ are given in Figure 5.

$$
\begin{array}{lcl}
\mathcal{B}(r) & = & 0 \\
\mathcal{B}(e_1 + e_2) & = & \mathcal{B}(e_1) + \mathcal{B}(e_2) \\
\mathcal{B}(e_1 * e_2) & = & \mathcal{B}(e_1) * e_2 + e_1 * \mathcal{B}(e_2) \\
\mathcal{B}(e_1 / e_2) & = & (\mathcal{B}(e_1) * e_2 - e_1 * \mathcal{B}(e_2)) \,/\, (e_2 * e_2) \\
\mathcal{B}(x) & = & e_1 \quad \textit{if } x \textit{ is defined by } \texttt{der}\ x = e_1\ \texttt{init}\ e_2 \\
\mathcal{B}(x) & = & \mathcal{B}(e) \quad \textit{if } x \textit{ is defined by } x = e \\
\\
\mathcal{I}(r) & = & r \\
\mathcal{I}(e_1 + e_2) & = & \mathcal{I}(e_1) + \mathcal{I}(e_2) \\
\mathcal{I}(e_1 * e_2) & = & \mathcal{I}(e_1) * \mathcal{I}(e_2) \\
\mathcal{I}(e_1 / e_2) & = & \mathcal{I}(e_1) \,/\, \mathcal{I}(e_2) \\
\mathcal{I}(x) & = & e_2 \quad \textit{if } x \textit{ is defined by } \texttt{der}\ x = e_1\ \texttt{init}\ e_2 \\
\mathcal{I}(x) & = & \mathcal{I}(e) \quad \textit{if } x \textit{ is defined by } x = e
\end{array}
$$

Figure 5: $\mathcal{B}(e)$ and $\mathcal{I}(e)$ rules

The compilation rules for expressions are given in Figure 6. The judgment $E \vdash e_1 \rightarrow_4 (e_2, \mathcal{D})$ means that, in the environment E, the expression e_1 is transformed into the expression e_2 and produces the set of *ivpeqs* $\mathcal{D}$.

The rule Num handles a real numeric constant and produces the same constant with an empty set of *ivpeqs*. The rule Id handles identifiers the same way. The rule App 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. The expression returned by the rule App is the identifier naming the output of f and the *ivpeqs* 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. 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. Finally, the rule Opp recursively processes the two sub-expressions to rebuild an arithmetic

$$(\textsc{Num})\,\frac{}{E \vdash r \rightarrow_4 (r,\ \varnothing)} \qquad (\textsc{Id})\,\frac{}{E \vdash x \rightarrow_4 (x,\ \varnothing)}$$

$$(\textsc{App})\,\frac{\begin{array}{c} E(f) = (\{x_1,\ldots,x_n\}, y, \{\langle d_1 \rangle,\ldots,\langle d_m \rangle\}) \\ E \vdash e_1 \rightarrow_4 (e_1',\ \mathcal{D}_1) \quad \ldots \quad E \vdash e_n \rightarrow_4 (e_n',\ \mathcal{D}_n) \\ d_1' = d_1[x_i \leftarrow e_i']_{i=1\ldots n} \quad \ldots \quad d_m' = d_m[x_i \leftarrow e_i']_{i=1\ldots n} \end{array}}{E \vdash f\,(e_1,\ldots,e_n) \rightarrow_4 (y,\ \{\langle d_1' \rangle,\ldots,\langle d_m' \rangle\} \cup \mathcal{D}_1 \cup \ldots \cup \mathcal{D}_n)}$$

$$(\textsc{Op})\,\frac{E \vdash e_1 \rightarrow_4 (e_1',\ \mathcal{D}_1) \quad E \vdash e_2 \rightarrow_4 (e_2',\ \mathcal{D}_2)}{E \vdash e_1\, \texttt{op}\, e_2 \rightarrow_4 (e_1'\, \texttt{op}\, e_2',\ \mathcal{D}_1 \cup \mathcal{D}_2)}$$

Figure 6: Compilation of expressions

expression and the union of the *ivpeqs* obtained for each sub-expression.

The compilation rules for equations are given in Figure 7. The judgment $E \vdash eq \rightarrow_3 \mathcal{D}$ states that the equation eq produces the set of *ivpeqs* $\mathcal{D}$.

$$(\textsc{Eq})\,\frac{E \vdash e \rightarrow_4 \left(e',\ \mathcal{D}\right) \quad e_b = \mathcal{B}\,(e') \quad e_i = \mathcal{I}\,(e')}{E \vdash x\ =\ e \rightarrow_3 \langle\, x, e_b, e_i \,\rangle + \mathcal{D}}$$

$$(\textsc{Der})\,\frac{E \vdash e_1 \rightarrow_4 \left(e_1',\ \mathcal{D}_1\right) \quad E \vdash e_2 \rightarrow_4 \left(e_2',\ \mathcal{D}_2\right)}{E \vdash \texttt{der}\ x\ =\ e_1\ \texttt{init}\ e_2 \rightarrow_3 \langle\, x, e_1', e_2' \,\rangle + \mathcal{D}_1 \cup \mathcal{D}_2}$$

Figure 7: Compilation of equations

The rule Eq handles regular dataflow equations. It transforms the body e of the equation to obtain the *ivpeqs* representing inlined node instantiation expressions of e. The equation is differentiated and added as a new *ivpeq*. The rule Der performs the inlining transformation on the body and initial value of the differential equation, and returns the set made of the *ivpeqs* obtained for each sub-expression extended with the one created for x.

Finally, the compilation rules for nodes then programs are given in Figure 8.

$$(\textsc{Node})\,\frac{E \vdash eq_1 \rightarrow_3 \mathcal{D}_1 \quad \ldots \quad E \vdash eq_n \rightarrow_3 \mathcal{D}_n}{\begin{array}{c} E \vdash \texttt{hybrid}\ f\,(x_1,\ldots,x_m) = y\ \texttt{where}\ eq_1 \ldots eq_n \rightarrow_2 \\ \langle\!\langle\, f,\ (\{\, x_1,\ldots,x_n \,\},\ y,\ \mathcal{D}_1 \cup \ldots \cup \mathcal{D}_n)\,\rangle\!\rangle + E \end{array}}$$

$$(\textsc{Prg})\,\frac{E \vdash nod_1 \rightarrow_2 E_1 \quad \ldots \quad E_{n-1} \vdash nod_n \rightarrow_2 E_n}{E \vdash \{nod_1,\ldots,nod_n\} \rightarrow_1 E_n}$$

$$(\textsc{Top})\,\frac{\begin{array}{c} \varnothing \vdash p \rightarrow_1 E \quad \exists\, \langle\!\langle\, \texttt{main},\ (\{\, x_1,\ldots,x_m \,\}, y, \{\, eq_1,\ldots,eq_n \,\})\,\rangle\!\rangle \in E \\ eq_i = \langle\, x_i, e_i, d_i \,\rangle_{i=1\ldots n} \quad \sigma = [x_i \mapsto i] \end{array}}{t_0, t_f \vdash p \rightarrow (n, \{\, e_1,\ldots,e_n \,\}, \{\, d_1,\ldots,d_n \,\}, t_0, t_f),\ \sigma}$$

Figure 8: Rules for intermediate representation synthesis

The rule Node processes one node, harvesting the *ivpeqs* obtained from its equations. It returns the environment extended with the node description of f.

The rule PRG iterates this process on the list of nodes making the program. The description of each processed node is added to the environment to process the remaining ones. Since Zélus imposes that a node is always defined before any use of it, this ensures that we will always find the node description bound to a node identifier present in an instantiation. The rule TOP processes all the nodes of the program p, then looks for a node named `main`. It builds the representation of the IVP as a tuple containing the size of the problem, the list of ODEs expressions, the list of initial values, the initial (t_0) and final (t_f) dates of the simulation and a substitution σ. This substitution maps each identifier binding an equation to an integer representing the rank of its equation. It will be used during the C++ code generation.

Optimization The combination of the instantiation inlining and the differentiation process can produce duplicated equations which is inefficient since this increases the size of the system. Consider the Zélus program given in Section 2. After the inlining in the node `main`, we get the set of equations :

`der` $x = x'$ `init` 1
`der` $x' = -4 * x - 0.4 * x'$ `init` 0
$y = x$

which is transformed during differentiation of y into :

`der` $x = x'$ `init` 1
`der` $x' = -4 * x - 0.4 * x'$ `init` 0
`der` $y = x'$ `init` 1

where `x` and `y` are redundant. To overcome this inefficiency, when processing equations of the form $y = x$, we drop the equation and replace y by x in the node (i.e. in its equations and its return value).

4.3 Generating C++ Code

The C++ code generation mostly consists of generating a template like the one shown in Figure 1, where the dimension of the system, the `Function` object and the initial conditions are instantiated from the intermediate representation of the compiled IVP. Hence, the code generation relies on the intermediate representation of the node `main` and on some data outside the model (duration, integration method, precision, number of noise symbols) provided at compile-time.

The rule TOP already provides the list of initial values and differential expressions. Most of what remains is the translation of arithmetic expressions into C++. The substitution σ returned by this rule is used while generating the initial values and the `Function` object representing the equations. It allows the replacement of identifiers by accesses in the arrays used for DynIbex's vector-valued representation of the equations. Thus, in the `Function` object, an identifier x is translated into `y[`$\sigma(x)$`]`. In the same way, the initial value of x is set in `yinit[`$\sigma(x)$`]`. A numerical constant n is translated into the trivial interval `Interval (`n`)`.

5 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.

The first experiment was to simulate the system with Zélus and with our generated code, then to compare the results. In Figure 9, the Zélus native simulation is represented by the red line and the simulation obtained using the intervals is shown by the green boxes.

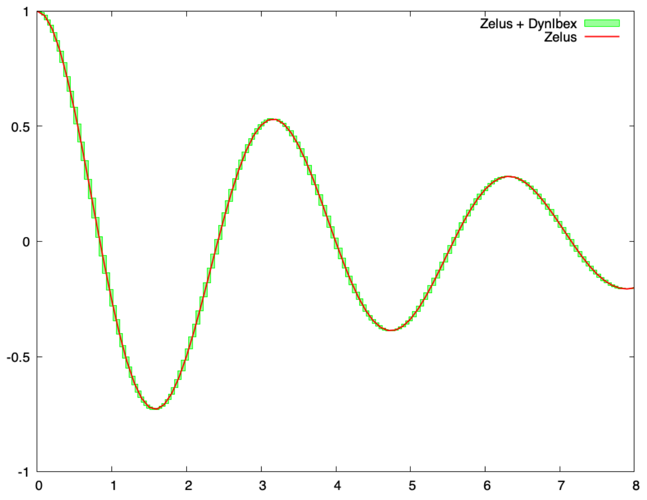

Figure 9: Simulations with/without intervals

The two simulations behave consistently. In particular, 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.

The Zélus syntax has been extended to specify an alternative interval value for any float value. This interval is only taken into account when compiling toward DynIbex. Hence, it is possible to add uncertainty on the initial value of `der x`, by writing `y = shm_decay (1.0 [0.9; 1.0], 0.0, 4.0, 0.4)`. The "default" value `1.0` is then ignored and the interval `[0.9; 1.0]` is considered instead. The simulation obtained after this change, displayed in Figure 10, shows that both simulations continue to behave consistently, but the effect of the uncertainty becomes clearly visible.

It is also possible to add uncertainty on the coefficients of the equations. The examples in Figure 11 are based on a value of k_2 in $[0.3; 0.4]$ instead of 0.4. At

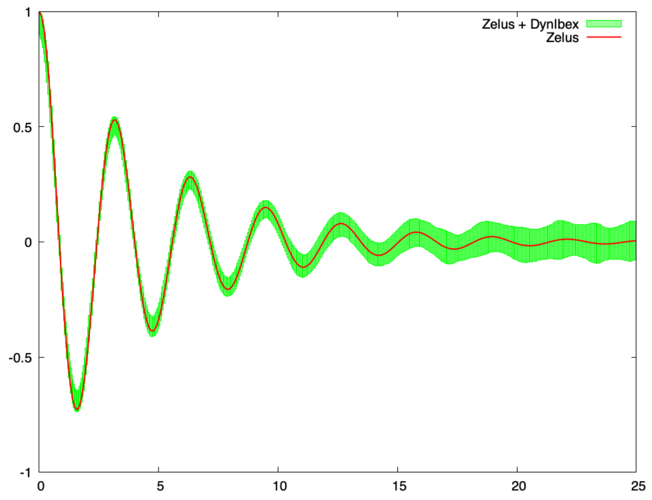

Figure 10: Simulation with initial uncertainty

compile-time, it is possible to select different integration methods, precisions, etc. From left to right, top to bottom, we used third-order Runge-Kutta 10^{-10}, Heun's method 10^{-6}, fourth-order Runge-Kutta 10^{-6} and fourth-order Gauss-Legendre 10^{-12}.

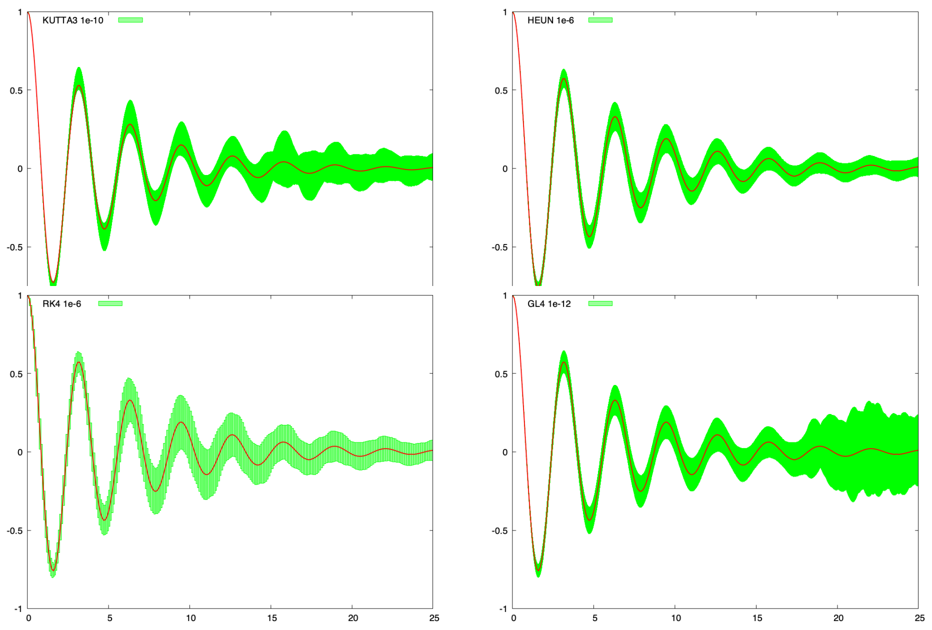

Figure 11: Simulation with parameter uncertainty

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. The simulation result is less accurate but safe and guaranteed. Running several point-wise simulations is not satisfactory: how many must be ran, which values must be chosen? There is no guaranty that the chosen strategy covers all the possible behaviors. The Figure 12 shows the inclusion of several point-wise simulations in a single interval-based simulation. In the top

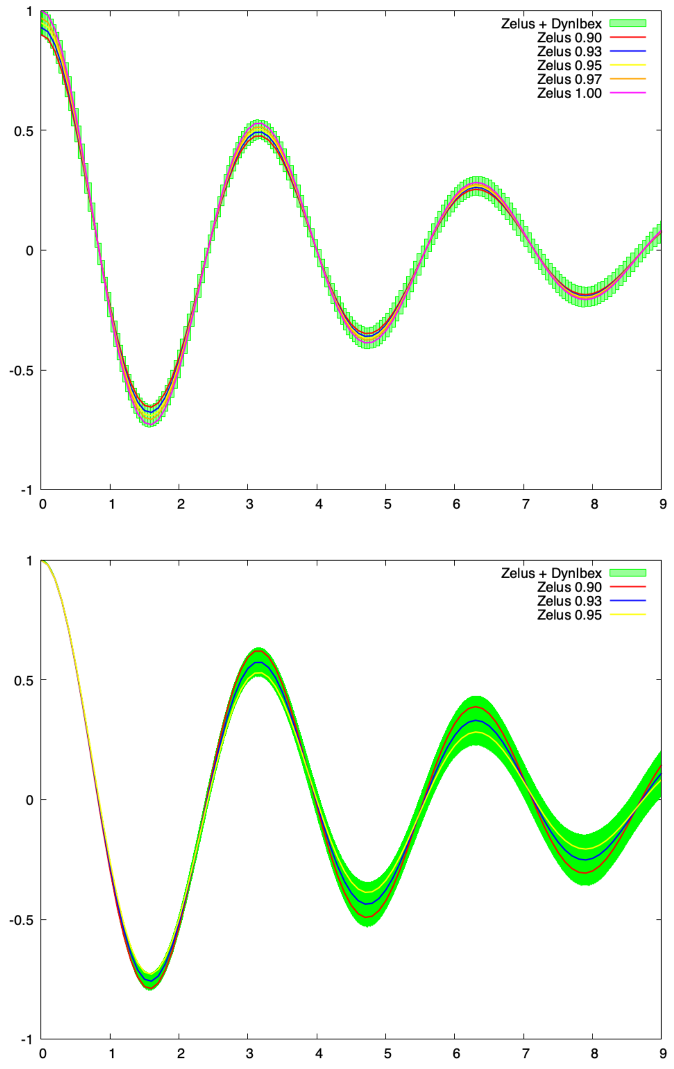

Figure 12: Point-wise simulations vs interval simulation

picture, the interval for the initial condition of `der x` is [0.9; 1]. In the bottom picture, the interval for $k2$ is [0.3; 0.4]. We plot several point-wise simulations in these ranges of uncertainty. As one expects for safety purposes, the interval-based simulations cover all the point-wise ones.

6 Related Work

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 [14] 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 [10] 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 [16].

- FLOW* [8] 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 [5] 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 method with handling of floating-point rounding errors. It can be considered as an ancestor of this work.

- dReal [11, 12] 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 [17, 15] 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 automata, and thus model more complex systems. DynIbex was chosen as a simulation framework as it provides guaranteed integration methods which handles floating-point issues.

7 Future Work

This exploratory work can be extended in three directions. The first direction aims at considering more complex systems. We would like to address IVPs with reset conditions (guards). In such systems, the ODEs are fixed, however the continuous state can be set to particular values on some conditions. To go further, we would like to address systems with several dynamics. In these systems, the ODEs involved in the dynamics may change depending on the state of the system. This will naturally be represented in Zélus by automata. Some obvious issues arise due to the temporal and spacial uncertainties represented by the intervals when changing state in an automaton. Moreover, the compilation schema to implement this will have to remain compatible with the IVPs simulation presented here, while also handling the more elaborate constructs of Zélus.

The second direction is to add contracts in Zélus and to be able to check if they hold during the simulation. First, the form of properties to verify must be chosen since this will have an impact on the logic to use. Then there is the choice of when to verify the contracts: during the simulation or at the end of the simulation. Finally, we need to compile the formulae and represent them with DynIbex constructs.

The last extension of this work is to address the discrete behavior it is possible to model in Zélus. The programs currently supported are restricted to continuous computation. This enables us to model the dynamics of a physical system but not a controller coupled to this system.

8 Conclusion

We presented a mechanism to compile IVPs described in Zélus to C++ code using DynIbex. This allows 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. Extensions to handle more complex systems and to compile contracts verification on programs are in progress.

References

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

[2] Althoff, Matthias. An introduction to CORA 2015. In Frehse, Goran and Althoff, Matthias, editors, *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 120–151. EasyChair, 2015. DOI: `10.29007/zbkv`.

[3] Althoff, Matthias and Grebenyuk, Dmitry. Implementation of interval arithmetic in CORA 2016. In *ARCH@CPSWeek*, volume 43 of *EPiC Series in Computing*, pages 91–105. EasyChair, 2016. DOI: `10.29007/w19b`.

[4] Althoff, Matthias, Grebenyuk, Dmitry, and Kochdumper, Niklas. Implementation of taylor models in CORA 2018. In *ARCH@ADHS*, volume 54 of *EPiC Series in Computing*, pages 145–173. EasyChair, 2018. DOI: `10.29007/zzc7`.

[5] Bouissou, Olivier, Mimram, Samuel, and Chapoutot, Alexandre. HySon: Set-based simulation of hybrid systems. In *Proceedings of IEEE International Symposium on Rapid System Prototyping*, pages 79–85, 2012. DOI: `10.1109/RSP.2012.6380694`.

[6] Bourke, Timothy and Pouzet, Marc. Zélus: A synchronous language with ODEs. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, pages 113–118. ACM, 2013. DOI: `10.1145/2461328.2461348`.

[7] Carloni, Luca P., Passerone, Roberto, Pinto, Alessandro, and Angiovanni-Vincentelli, Alberto L. Languages and tools for hybrid systems design. *Found. Trends Electron. Des. Autom.*, 1(1):1–193, 2006. DOI: `10.1561/1000000001`.

[8] Chen, Xin, Ábrahám, Erika, and Sankaranarayanan, Sriram. Flow*: An analyzer for non-linear hybrid systems. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, pages 258–263, New York, NY, USA, 2013. Springer-Verlag New York, Inc. DOI: `10.1007/978-3-642-39799-8_18`.

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

[10] Frehse, Goran, Le Guernic, Colas, Donzé, Alexandre, Cotton, Scott, Ray, Rajarshi, Lebeltel, Olivier, Ripado, Rodolfo, Girard, Antoine, Dang, Thao, and Maler, Oded. SpaceEx: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan, Shaz Qadeer, editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS, pages 379–395. Springer, 2011. DOI: `10.1007/978-3-642-22110-1_30`.

[11] Gao, S., Kong, S., and Clarke, E. M. 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. DOI: `10.1007/978-3-642-38574-2_14`.

[12] Gao, Sicun, Kong, Soonho, and Clarke, Edmund M. Satisfiability modulo ODEs. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE, 2013. DOI: `10.1109/FMCAD.2013.6679398`.

[13] Henzinger, Thomas A., Horowitz, Benjamin, Majumdar, Rupak, and Wong-Toi, Howard. Beyond HyTech: Hybrid systems analysis using interval numerical methods. In *Hybrid Systems: Computation and Control*, pages 130–144. Springer Berlin Heidelberg, 2000. DOI: `10.1007/3-540-46430-1_14`.

[14] Immler, Fabian, Althoff, Matthias, Chen, Xin, Fan, Chuchu, Frehse, Goran, Kochdumper, Niklas, Li, Yangge, Mitra, Sayan, Tomar, Mahendra Singh, and Zamani, Majid. ARCH-COMP18 category report: Continuous and hybrid systems with nonlinear dynamics. In Frehse, Goran, editor, *ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 54 of *EPiC Series in Computing*, pages 53–70. EasyChair, 2018. DOI: `10.29007/mskf`.

[15] Konečný, Michal, Taha, Walid, Duracz, Jan, Duracz, Adam, and Ames, Aaron. Enclosing the behavior of a hybrid system up to and beyond a Zeno point. In *2013 IEEE 1st international conference on cyber-physical systems, networks, and applications (CPSNA)*, pages 120–125, United States, 2013. IEEE. DOI: `10.1109/CPSNA.2013.6614258`.

[16] Minopoli, Stefano and Frehse, Goran. SL2SX translator: From Simulink to SpaceEx models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, pages 93–98, 04 2016. DOI: `10.1145/2883817.2883826`.

[17] Zeng, Yingfu, Rose, Chad G., Brauner, Paul, Taha, Walid, Masood, Jawad, Philippsen, Roland, O'Malley, Marcia K., and Cartwright, Robert. Modeling basic aspects of cyber-physical systems, part II. In *Procceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications*, 2014. DOI: `10.1109/HPCC.2014.119`.