Interval-Based Simulation of Zélus IVPs Using DynIbex

Jason Brown[1] and François Pessaux*[1]

[1]U2IS, ENSTA Paris, Institut Polytechnique de Paris, 828 boulevard des Maréchaux, 91120 Palaiseau, France

Keywords: DynIbex, Zélus, Compilation, Hybrid System, Interval, Guaranteed Integration, Simulation

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. One of the verification techniques is to simulate 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 guaranteed integration tools [2, 6].

Tools and Domain Specific Languages exist to ease the modeling, development and verification of hybrid systems (Modelica, Simulink/Stateflow, LabVIEW, Zélus and others [4]). 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[3], with the safety of interval-based guaranteed integration using DynIbex[1, 5]. 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 the compilation scheme for an *IVP* described in a subset of Zélus to a C++ simulation code using DynIbex.

1 IVPs in Zélus

An *IVP* in DynIbex is represented by a vector-valued ordinary differential equation (*ODE*) with initial conditions whereas an *IVP* in Zélus is represented by a system of coupled equations. Compilation from Zélus to DynIbex therefore requires a transformation between these representations.

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
```

*Corresponding author.

```
let hybrid main () = x where
  x = 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.

2 Compiling the IVP

Compiling the Zélus code requires two steps. First the hierarchy of nodes must be flattened, harvesting all the differential equations. 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 implies a recursive inlining mechanism which terminates since Zélus forbids recursive nodes.

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 in a vector-valued structure. 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 intervals.

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 code generated for the example given at the beginning of this section is shown in the following listing.

```
#define T0 (0.000000)
#define TEND (6.000000)
int main () {
  const int dim = 2;
  Variable y(dim);
  IntervalVector yinit(dim);
  Function ydot =
    Function
      (y,
       Return
         (y[1],
          ((-Interval(4.000000)) * y[0]) -
           (Interval(0.400000) * y[1]))
          );
  yinit[0] = Interval(1.000000);
  yinit[1] = Interval(0.000000);
  ivp_ode problem = ivp_ode(ydot,T0,yinit)
      ;
  simulation simu =
    simulation(&problem,TEND,GL4,1e-7);
  simu.run_simulation();
  simu.export_y0("export");
  return 0;
}
```

In this generated code, the size of the *IVP* is 2 since we had 2 equations. The interval `y` stores the continuous state of the system. The vector `yinit` contains the initial values. Each equation is translated into an argument of the `Return` constructor. We can see that the compilation mapped the `x'` of the Zélus program to the dimension 1 of the vector-based representation, and `x` to the dimension 0. It is possible to recognize, in the `Return` clause, the translation of `-.k1 *. x -. k2 *. x'` where `k1` has been properly instantiated by 4.0 and `k2` by 0.4.

3 Experimental Results

The first experiment was to simulate the system with Zélus and with our generated code, then to compare the results. In the figure 1, the Zélus native simulation is represented by the red line and the simulation obtained using the intervals is shown by the green boxes.

Both simulations behave consistently. In particular, the results obtained with the standard integration runtime of Zélus always remain inside the boxes obtained using the interval 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.

Although there is not yet syntax extension of Zélus in the current implementation to spec-

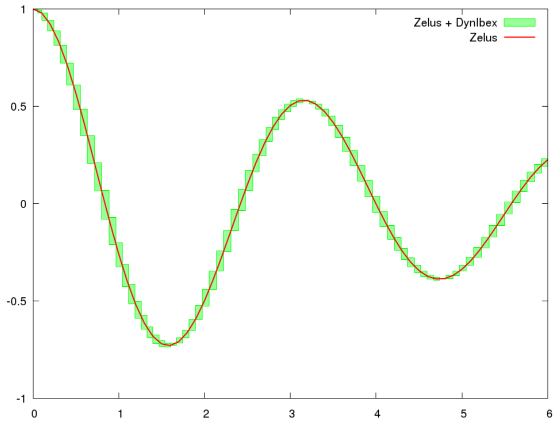

Figure 1: Simulations with/without intervals

ify interval values, it is possible to add uncertainty on the initial value of `der x`, by manually changing the value of `yinit[0]` to `Interval(0.9, 1.0)` in the generated C++ code. The simulation obtained after this change is shown in the figure 2.

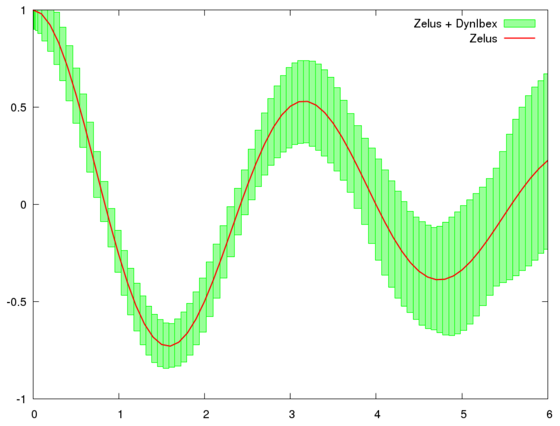

Figure 2: Simulation with initial uncertainty

Both simulations continue to behave consistently, and we see more clearly how the uncertainty increases with time.

4 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. This work has lead to a real implementation in the Zélus compiler. Extensions to handle more complex *IVPs* and to compile contracts verification on programs are in progress.

References

[1] J. Alexandre dit Sandretto, A. Chapoutot, and O. Mullier. The dynibex library. `http://perso.ensta-paristech.fr/~chapoutot/dynibex/`.

[2] O. Bouissou, S. Mimram, and A. Chapoutot. Hyson: Set-based simulation of hybrid systems. In *Proceedings of the 23rd IEEE International Symposium on Rapid System Prototyping, RSP 2012, Tampere, Finland, October 11-12, 2012*, pages 79–85, 2012.

[3] T. Bourke and M. Pouzet. Zélus: A synchronous language with odes. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, pages 113–118, New York, NY, USA, 2013. ACM.

[4] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli. Languages and tools for hybrid systems design. *Found. Trends Electron. Des. Autom.*, 1(1/2):1–193, Jan. 2006.

[5] J. A. dit Sandretto and A. Chapoutot. Validated explicit and implicit Runge–Kutta methods. *Reliable Computing*, 22(1):79–103, Jul 2016.

[6] T. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond hytech: Hybrid systems analysis using interval numerical methods. In N. Lynch and B. H. Krogh, editors, *Hybrid Systems: Computation and Control*, pages 130–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.