Nonlinear Analysis: Hybrid Systems 26 (2017) 168–189

Contents lists available at ScienceDirect

Nonlinear Analysis: Hybrid Systems

journal homepage: www.elsevier.com/locate/nahs

A type-based analysis of causality loops in hybrid systems modelers

Albert Benveniste [a], Timothy Bourke [b,c], Benoit Caillaud [a], Bruno Pagano [e], Marc Pouzet [d,c,b,*]

[a] *Inria Rennes-Bretagne Atlantique, France*
[b] *Inria Paris, France*
[c] *École normale supérieure, PSL Research University, France*
[d] *Sorbonne Universités, UPMC Univ Paris 06, France*
[e] *ANSYS/Esterel Technologies, France*

ARTICLE INFO

Keywords:
Hybrid systems
Synchronous programming languages
Type systems

ABSTRACT

Explicit hybrid systems modelers like *Simulink/Stateflow* allow for programming both discrete- and continuous-time behaviors with complex interactions between them. An important step in their compilation is the static detection of algebraic or *causality* loops. Such loops can cause simulations to deadlock and prevent the generation of statically scheduled code.

This paper addresses this issue for a hybrid modeling language that combines synchronous data-flow equations with Ordinary Differential Equations (ODEs). We introduce the operator `last` x for the left-limit of a signal x. The `last` x operator is used to break causality loops and permits a uniform treatment of discrete and continuous state variables. The semantics of the language relies on non-standard analysis, defining an execution as a sequence of infinitesimally small steps. A signal is deemed *causally correct* when it can be computed sequentially and only changes infinitesimally outside of announced discrete events like zero-crossings. The causality analysis takes the form of a type system that expresses dependencies between signals. In well-typed programs, (i) signals are *provably continuous during integration* provided that imported external functions are also continuous, and (ii) *sequential code can be generated*.

The effectiveness of the system is illustrated with several examples written in Zélus, a Lustre-like synchronous language extended with ODEs.

1. Causality and scheduling

Tools for modeling hybrid systems [1] such as Modelica,[1] LabVIEW,[2] and Simulink/ Stateflow,[3] are now rightly understood and studied as programming languages. Indeed, models are used not only for simulation, but also for test-case

* Corresponding author at: École normale supérieure, PSL Research University, France.
E-mail addresses: Albert.Benveniste@inria.fr (A. Benveniste), Timothy.Bourke@inria.fr (T. Bourke), Benoit.Caillaud@inria.fr (B. Caillaud), Bruno.Pagano@ansys.com (B. Pagano), Marc.Pouzet@ens.fr (M. Pouzet).

1 http://www.modelica.org.
2 http://www.ni.com/labview.
3 http://www.mathworks.com/products/simulink.

http://dx.doi.org/10.1016/j.nahs.2017.04.004
1751-570X/

generation, formal verification and translation to embedded code. This explains the need for formal operational semantics to specify their implementations [2].

The underlying mathematical model is the synchronous parallel composition of stream equations, Ordinary Differential Equations (ODEs), hierarchical automata, and imperative features. While each of these taken separately is precisely understood, real languages allow them to be combined in sophisticated ways. One major difficulty in such languages is the treatment of causality loops.

Causality or *algebraic* loops [3, 2-34] pose problems of well-definedness and compilation. They can lead to mathematically unsound models, prevent simulators from statically ensuring the existence and unicity of a least fixed point, and compilers from generating statically scheduled code (typically sequential code written in C). Statically scheduled code is the usual target for embedded software. But it is also important for efficient simulations of the whole system where continuous-time trajectories are approximated by a numerical solver. The static detection of causality loops, termed *causality analysis*, has been studied and implemented since the mid-1980s in synchronous language compilers [4–7]. The classical and simplest solution is to reject instantaneous cycles or feedback loops, which do not cross a unit delay: at every instant, the value of a signal x only depends on the current value of inputs and possibly some internal state, but not on x itself. For instance, the LUSTRE-like equations[4] :

```
x = 0.0 -> pre y    and    y = if c then x + 1.0 else x
```

define two sequences $(x_n)_{n\in\mathbb{N}}$ and $(y_n)_{n\in\mathbb{N}}$ such that:

$$x(0) = 0 \qquad\qquad y(n) = \texttt{if}\ c(n)\ \texttt{then}\ x(n) + 1\ \texttt{else}\ x(n)$$
$$x(n) = y(n-1).$$

They are causally correct since the feedback loop for `x` contains a unit delay `pre y` ('previous'). Replacing `pre y` with `y` gives non-causal equations that a LUSTRE compiler would reject. Causally correct equations are statically scheduled to produce a sequential, loop-free *step* function. Below is an excerpt of the C code generated by the HEPTAGON LUSTRE compiler [8]:

```
if (self->v_1) {x = 0;} else {x = self->v_2;};
if (c) {y = x+1;} else {y = x;};
self->v_2 = y; self->v_1 = false;
```

It computes current values of `x` and `y` from that of `c`. The internal memory of function *step* is in `self`, with `self->v_1` initialized to true and set to false to encode the LUSTRE operator `->`, and `self->v_2` storing the value of `pre y`.

ODEs with resets: Consider now programs defining continuous-time signals via ODEs and equations only. For example, the program:

```
der y = z init 4.0  and  z = 10.0 - 0.1 * y  and  k = y + 1.0
```

defines the three signals y, z and k, where for all $t \in \mathbb{R}^+$, $\frac{dy}{dt}(t) = z(t)$, $y(0) = 4$, $z(t) = 10 - 0.1 \cdot y(t)$, and $k(t) = y(t) + 1$.[5] This program is causally correct since it is possible to generate a sequential function *derivative*$(y) = \texttt{let}\ z = 10 - 0.1 * y\ \texttt{in}\ z$ that returns the current derivative of y and an initial value 4 for y from which a numeric solver [9] can compute a sequence of approximations $y(t_n)$ for increasing values of time $t_n \in \mathbb{R}^+$ and $n \in \mathbb{N}$. Given a set of mutually recursive equations $\{x_i = e_i\}_{i\in[1..k]}$ and $\{\dot{y}_j = e'_j\}_{j\in[1..m]}$, the compiler has to produce the *derivative* function that defines the current value of $(\dot{y}_j)_{j\in[1..m]}$ from current inputs, discrete state variables and continuous state variables $(y_j)_{j\in[1..m]}$. Thus, for equations between continuous-time signals, integrators break algebraic loops just as delays do for equations over discrete-time signals.

Can we reuse the simple justification we used for data-flow equations to justify that the above program is causal? Consider the value that y would have if computed by an ideal solver taking an infinitesimal step of duration ∂ [10]. Writing ${}^\star y(n)$, ${}^\star z(n)$ and ${}^\star k(n)$ for the values of y, z and k at instant $n\partial$, where $n \in {}^\star\mathbb{N}$ is a non-standard integer, we have:

$${}^\star y(0) = 4 \qquad\qquad {}^\star z(n) = 10 - 0.1 \cdot {}^\star y(n)$$
$${}^\star y(n+1) = {}^\star y(n) + {}^\star z(n) \cdot \partial \qquad\qquad {}^\star k(n) = {}^\star y(n) + 1$$

where ${}^\star y(n)$ is defined sequentially from past values and ${}^\star y(n)$ and ${}^\star y(n+1)$ are infinitesimally close, for all $n \in {}^\star\mathbb{N}$, yielding a unique solution for y, z and k. The equations are thus causally correct.

Troubles arise when ODEs interact with discrete-time constructs, for example when a reset occurs at every occurrence of an event. For example, consider the sawtooth signal $y : \mathbb{R}^+ \mapsto \mathbb{R}^+$ where $\frac{dy}{dt}(t) = 1$ and $y(t) = 0$ when $t \in \mathbb{N}$. One may try with an ODE and a reset:

```
der y = 1.0 init 0.0 reset up(y - 1.0) -> 0.0
```

4 The unit delay initialized to 0 is written $0 -> \texttt{pre}(\cdot)$ or $0\ \texttt{fby}\ \cdot$ ('0 followed by'), or even, notably in SIMULINK, $\frac{1}{z}$.

5 $\texttt{der}\, y = e\ \texttt{init}\ v_0$ is written in SIMULINK as $y = \frac{1}{s}(e)$ with y initialized to v_0.