Symbolic Simulation of Dataflow Synchronous Programs with Timers

Guillaume Baudart, Timothy Bourke, Marc Pouzet

▶ To cite this version:

Guillaume Baudart, Timothy Bourke, Marc Pouzet. Symbolic Simulation of Dataflow Synchronous Programs with Timers. 12th Forum on Specification and Design Languages (FDL 2017), Electronic Chips & System Design Initiative (ECSI), Sep 2017, Vérone, Italy. hal-01575621v1

HAL Id: hal-01575621
https://inria.hal.science/hal-01575621v1

Submitted on 21 Aug 2017 (v1), last revised 11 Jan 2019 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Symbolic Simulation of Dataflow Synchronous Programs with Timers

Guillaume Baudart
IBM Research

Timothy Bourke
Inria Paris
École normale supérieure
PSL Research University

Marc Pouzet
Sorbonne Universités
UPMC Univ Paris 06
École normale supérieure
PSL Research University
Inria Paris

***Abstract*—The synchronous language Lustre and its descendants have long been used to program and model discrete controllers. Recent work shows how to mix discrete and continuous elements in a Lustre-like language called Zélus. The resulting hybrid programs are deterministic and can be simulated with a numerical solver. In this article, we focus on a subset of hybrid programs where continuous behaviors are expressed using timers, nondeterministic guards, and invariants, as in Timed Safety Automata. We propose a source-to-source compilation pass to generate discrete code that, coupled with standard operations on Difference-Bound Matrices, produces symbolic traces that each represent a set of concrete traces.**

***Index Terms*—Programming, simulation, real-time systems, formal specifications.**

I. Introduction

Synchronous languages like Lustre [1] are ideal for programming an important class of embedded controllers. Their discrete model of time and deterministic semantics facilitate the precise expression of reactive behaviors. That said, many systems are naturally modeled using physical timing constraints that almost inevitably involve some 'timing nondeterminism' due to tolerances in requirements or uncertainties in implementations. Conversely, such constraints are readily modeled using Timed Automata [2], [3], and simulated symbolically in Uppaal [4], [5], but large-scale discrete-time behaviors are more cumbersome to express in such tools. In this article we try to have the best of both worlds by incorporating features of Timed Automata into a Lustre-like synchronous language. We focus on programming language design and symbolic simulation rather than verification.

As a simple running example, consider the application shown in Figure 1: two components, P_1 and P_2, are periodically triggered by local clocks c1 and c2. Clocks are subject to jitter and can be modeled as a Timed Automaton as in Figure 2. This model generates a signal c with a nominal period of $(T_{\text{max}} + T_{\text{min}})/2$ and a jitter of $\pm(T_{\text{max}} - T_{\text{min}})/2$. The timing constraints are expressed using a timer variable: t is reset to 0 at every emission of signal c; the transition that emits c may occur whenever $t \geq T_{\text{min}}$; and t must never exceed T_{max}. Putting two or more such clocks in parallel and using their respective clock signals to trigger a program is a standard way to model communicating components [6], [7].

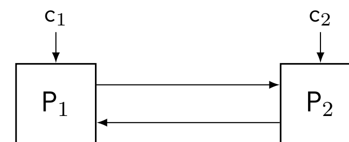


Fig. 1. A simple two-node application with clock inputs c_1 and c_2.

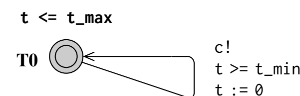


Fig. 2. [7, Figure 4] A simple clock model with clock output c.

Classic synchronous languages are ideal for expressing the dynamic behavior of the components P_1 and P_2, but cannot easily express the continuous-time dynamics of the overall architecture. Zélus (http://zelus.di.ens.fr) [8] is a newer synchronous language that allows mixing both discrete and continuous elements. The timer t of Figure 2 can be simulated by a simple Ordinary Differential Equation (ODE), $\dot{t} = 1$, and jitter can be introduced by resetting the timer to arbitrary values between $-T_{\text{min}}$ and $-T_{\text{max}}$ whenever t reaches 0. But, this approach forces the programmer to make explicit implementation choices that are not part of the specification and it is not modular. Adding constraints on t, like another invariant, requires adapting its defining equation.

We propose expressing *guards* and *invariants* directly as in the following program in a variation of Zélus that we call ZSy.

```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0 reset c() → 0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}
```

As in Zélus, the keyword hybrid declares a continuous-time component clock, parametrized by t_min and t_max, whose output c is defined by three parallel equations. The first equation declares a timer, that is, a variable t where $\dot{t} = 1$, with initial value 0 and reset to 0 at each occurrence of c. We use a dedicated timer syntax to emphasize the focus on timed systems with limited continuous dynamics. The second definition states that the signal c *may* be emitted whenever $t \geq T_{\text{min}}$. The third

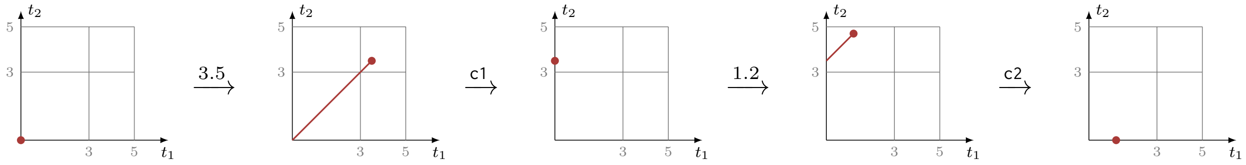


Fig. 3. A concrete simulation trace of `scheduler(3,5)`: t_1 and t_2 denote the values of the two timers, one for each quasi-periodic clock.

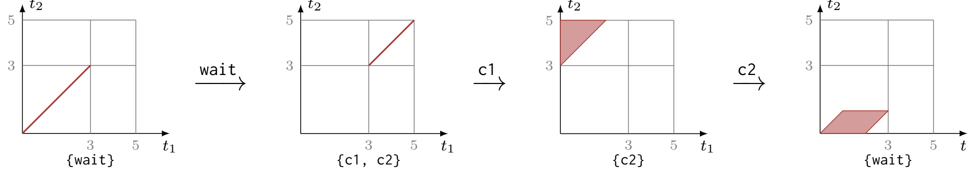


Fig. 4. A symbolic simulation trace of `scheduler(3,5)`. Each step corresponds to a set of timer values and a set of enabled transitions (below).

definition is an invariant stating that the value of t *must* never exceed T_{max}. We use braces to distinguish constraints from boolean conditions.

A model of a simple two-node architecture can be obtained by instantiating the `clock` function twice.

```
let hybrid scheduler(t_min, t_max) = c1, c2 where
  rec c1 = clock(t_min, t_max)
  and c2 = clock(t_min, t_max)
```

Signals produced by the `scheduler` function can then be used to trigger discrete controllers (like $\texttt{P}_1$ and $\texttt{P}_2$) written in the discrete subset of Zélus.

The simulation traces of such systems comprise two kinds of events: time elapsing and discrete transitions triggered by signal emissions. Figure 3 shows a possible execution trace of the two-node architecture of Figure 1 with $T_{\text{min}} = 3$ and $T_{\text{max}} = 5$. Variables t_1 and t_2 denote the values of the two timers, one for each instance of `clock`. Starting from $t_1 = t_2 = 0$, `c1` is triggered when $t_1 = 3.5$. Then `c2` is triggered when $t_2 = 4.7$, that is, 1.2 after `c1`.

For the kind of systems we consider, that is, nondeterministic timed discrete-event systems, an execution is a sequence of discrete events (here, the clock ticks). Rather than simulating one concrete trace that assigns a precise date to each event, we employ an alternative simulation scheme that focuses on the ordering of events. Precise timer valuations are then replaced by symbolic sets called *zones* that encompass the timer valuations giving rise to the same sequences of discrete events.

At each step, the user (or an external 'oracle' program) chooses from a set of possible transitions. A transition means either waiting for a change in the set of enabled guards or firing enabled guards. If the `wait` transition is chosen, we compute the new zone by letting time elapse until the next change in the set of enabled guards as permitted by any invariants. Otherwise, firing guards triggers discrete-time computations, possibly resets some timers, and returns a new initial zone. The new zone is obtained by letting time elapse from this initial zone until the next change in the set of enabled guards.

The symbolic trace that contains the concrete simulation of Figure 3 is presented in Figure 4.

1) The simulation starts with $\{t_1 = t_2 = 0\}$, where no guards are enabled. The first zone is obtained from this initial position by letting time elapse until just before one or more guards become enabled, giving $\{t_1 = t_2 < 3\}$. In this zone, the user has no other choice but to `wait`.
2) The next zone is generated by letting time elapse as long as permitted by the invariants, giving $\{3 \leq t_1 = t_2 \leq 5\}$. In this zone, the user may choose `c1` or `c2` but not `wait`.
3) The user chooses `c1` which resets t_1 to 0. The new initial zone is $\{t_1 = 0 \wedge 3 \leq t_2 \leq 5\}$ and only `c2` is possible.
4) The user chooses `c2` which resets t_2 to 0. The new initial zone is $\{t_2 = 0 \wedge 0 \leq t_1 \leq 2\}$ where no guards are enabled. The only possibility is to `wait` until the next change in the set of enabled guards.

This symbolic trace includes all concrete trace prefixes where `c1` occurs before `c2`. If a program reaches a state with contradictory constraints, or where no transitions are enabled, the simulation becomes stuck—such programs are deemed invalid.

The symbolic representation of timer valuations and zone-based simulation are standard practice for simulating and model checking Timed Automata models, notably in the Uppaal tool [4], [5]. Our simulations differ in that we introduce explicit `wait` transitions between zones with differing sets of enabled transitions. This is natural in our setting where the idea is to 'steer' a synchronous program through time using additional inputs whose simultaneous occurrence is possible and whose absence is significant. In any case, the Uppaal simulation scheme is readily recovered.

Our main contributions are (1) to present a novel Lustre-like language that incorporates features of Timed Automata, (2) to show how to compile programs to generate symbolic simulations using a novel source-to-source transformation, (3) using a 'sweeping' construction to introduce explicit `wait` transitions.

The presentation is structured as follows. Section II recalls the standard data structure used to represent zones and introduces the new construction for `wait` transitions. The formal syntax of ZSy is presented in Section III and we detail the source-to-source compilation pass in Section IV. Related work is discussed in Section V before the conclusion in Section VI.

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \le t_2 \\ 5 < t_3 \le 12 \\ 4 \le t_1 - t_2 \le 8 \end{array}\right\} \qquad \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{cccc} 0 & 1 & 2 & 3 \\ (0,\le) & (0,\le) & (-6,\le) & (-5,<) \\ (20,<) & (0,\le) & (8,\le) & (\infty,<) \\ (\infty,<) & (-4,\le) & (0,\le) & (\infty,<) \\ (12,\le) & (\infty,<) & (\infty,<) & (0,\le) \end{array}$$

Fig. 5. Example of a set of constraints (left) and a corresponding DBM (right).

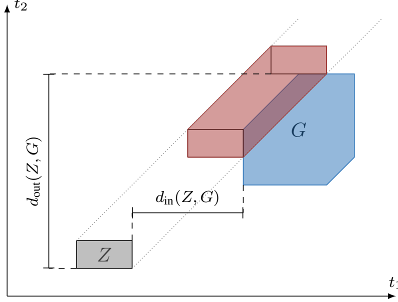


Fig. 6. Activation and deactivation distances of a guard G from an initial zone Z. The dotted line represents the zone obtained by letting time elapse indefinetly from Z. The red region is the zone where the guard is enabled.

II. Difference-Bound Matrices

Difference-bound matrices [9], [10] are a well-known data structure for representing and manipulating zones. DBMs are simple to implement and form a closed set with respect to both discrete transitions (mode changes, resets, intersections) and continuous evolution (time elapsing).

Let $\mathcal{T} = \{t_i\}_{0 \le i \le n}$ be a set of timer variables, with the convention that $t_0 = 0$. A DBM encodes a set of *difference constraints*, each of the form $t_i - t_j \preceq n$ where $\preceq \in \{<, \le\}$ and $n \in \mathbb{Z} \cup \{\infty\}$, by gathering them into a $|\mathcal{T}| \times |\mathcal{T}|$ matrix of pairs of bounds and relations. Each timer variable is assigned a row and column: the row stores the upper bounds on the differences between the timer and all other timers and the column stores lower bounds. Figure 5 shows an example of a set of constraints and its representation as a DBM. We write $(d_{ij}, \preceq^d_{ij})_{0 \le i,j \le |\mathcal{T}|}$ to denote the coefficients of a DBM D.

A. Distances

The simulation scheme involves detecting changes in the set of enabled guards as time passes, that is, as the initial zone moves along the vector $(1, 1, \ldots, 1)$. Each guard G divides the state-space of timer values into three zones: before activation, during activation, and after activation. A guard is enabled whenever the intersection between its activation zone and the current zone is not empty. Given an initial zone Z, we compute two distances for each guard G: $d_{\text{in}}(Z, G)$, the maximum distance before activation becomes possible, and $d_{\text{out}}(Z, G)$, the distance before deactivation. They are illustrated in Figure 6 (which is unrelated to the example in Figure 5).

A distance is a pair $(d, \preceq)$ where $\preceq$ specifies whether the limit is strict or not. Consider, for instance, a guard with activation zone $\{3 < t < 5\}$ and the initial zone $\{t = 0\}$. The distance before activation is $(3, <)$ which means that the guard is only enabled strictly after $t = 3$. The distance before deactivation is $(5, \le)$ since the guard is disabled when $t = 5$.

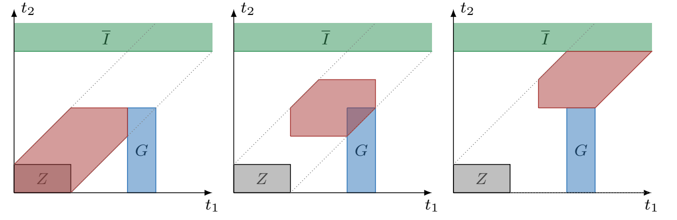


Fig. 7. Succession of zones from an arbitrary initial zone Z, guard G, and invariant I (the complement of I is shown to improved readability).

The distance before activation is obtained by comparing the upper bounds of the initial zone Z, with the lower bounds of the guard zone G (argmin is the index of the minimal value in a set):

$$d_{\text{in}}(Z, G) = (-g_{0j} - z_{j0}, \preceq^g_{0j}) \text{ with } j = \underset{1 \le i \le |\mathcal{T}|}{\operatorname{argmin}} \{-g_{0i} - z_{i0}\}$$

The distance before deactivation is obtained by comparing the lower bounds of the initial zone with the upper bounds of the guard zone; $\overline{\preceq}$ denotes the 'other' relation: $\overline{\le} = <$ and $\overline{<} = \le$.

$$d_{\text{out}}(Z, G) = (g_{j0} + z_{0j}, \overline{\preceq^g_{j0}}) \text{ with } j = \underset{1 \le i \le |\mathcal{T}|}{\operatorname{argmin}} \{g_{i0} + z_{0i}\}$$

B. Sweeping

The sweeping operation generates the succession of zones in a symbolic simulation as time passes, that is, when the user chooses the `wait` transition. Each zone is characterized by a set of enabled guards. The next zone is obtained by *sweeping* the initial zone along the vector $(1, 1, \ldots, 1)$ from its last position until the next change in the set of enabled guards. Figure 7 illustrates two successive `wait` transitions from the initial zone Z relative to a single guard G.

To compute the succession of zones from an initial zone Z, we compute the two distances associated with each guard that would become enabled if time were left to elapse indefinitely. We include the distance $(0, \le)$ for the initial state, and $(\infty, \le)$ for the final state when no more guards are reachable. These distances are ordered lexicographically, with $<$ less than $\le$:

$$(n_1, \preceq_1) < (n_2, \preceq_2) \equiv n_1 < n_2 \text{ or } (n_1 = n_2 \text{ and } \preceq_1 = <).$$

Each pair of successive distances defines a zone obtained by sweeping the initial zone on the corresponding interval. By construction, the succession of zones map the changes in the set of enabled guards as illustrated in Figure 7 (which is not directly related to the example of Figure 6).

The `zsweep` operation sweeps an initial zone Z through the interval defined by distances $(d_1, \preceq_1)$ and $(d_2, \preceq_2)$. The resulting zone C starts from the initial zone delayed by $(d_1, \preceq_1)$, that is, we add d_1 to the lower bounds of Z and take $\preceq_1$ as the limit relation (active after the change):

$$\forall 1 \le i < |\mathcal{T}|,\ C_{0i} = (z_{0i} - d_1, \preceq_1).$$

Zone C ends just before the next change, that is, after a distance $(d_2, \preceq_2)$. We thus add d_2 to the upper bounds of Z and take $\overline{\preceq_2}$ as the limit relation (stop before the next change):

$$\forall 1 \le i < |\mathcal{T}|,\ C_{i0} = (z_{i0} + d_2, \overline{\preceq_2}).$$

For the example with $Z = \{t = 0\}$ and $G = \{3 < t < 5\}$, the set of distances is $\{(0, \leq), (3, <), (5, \leq), (\infty, \leq)\}$, which gives three zones: $\{0 \leq t \leq 3\}$ where the guard is disabled, $\{3 < t < 5\}$ where the guard is enabled, and $\{5 \leq t < \infty\}$ where the guard is disabled again.

Invariants are accounted for by intersecting them with the results of the sweeping mechanism. For instance, adding the invariant `always {t ≤ 10}` to the example, gives the three zones: $\{0 \leq t \leq 3\}$, $\{3 < t < 5\}$, and $\{5 \leq t \leq 10\}$.

C. Zone interface

Our compilation and simulation routines require the small library of zone-manipulating operations summarized below.

- `zall` The complete space (unconstrained zone).
- `zmake(c)` Builds a DBM from a single constraint `c`.
- `is_zempty(z)` Returns *true* if DBM `z` denotes an empty zone.
- `zreset(z,t,v)` Resets a timer `t` to the value `v` in zone `z`.
- `zinter(z1, z2)` Returns the intersection of zones `z1` and `z2`.
- `zinterfold(zv)` Returns the intersection of a list of zones `zv` (fold `zinter` on `zv` starting from `zall`).
- `zup(z)` Lets time elapse indefinitely from zone `z` by removing all upper bounds.
- `zenabled(zc, gv)` Returns a list of booleans characterizing the set of enabled guards in the list `gv`. A guard is enabled if its activation zone $\mathtt{gv}_i$ intersects the current zone `zc`.
- `zdist(zi, g)` Returns the activation and deactivation distances of a guard activation zone `g` from the initial zone `zi`.
- `zdistmap(zi, gv)` Returns the list of distances between an initial zone `zi` and a list of guard activation zones `gv`.
- `zsweep(zi, d1, d2)` Sweeps `zi` between distances `d1` and `d2`.

III. ZSy: A SYNCHRONOUS LANGUAGE WITH TIMERS

We now present the kernel language of ZSy, a single assignment dataflow synchronous language extended with timers and nondeterministic constructs: guards and invariants.

The formal syntax of ZSy is presented in Figure 8. A program is a sequence of *declarations* (d) of n-ary functions. As in Zélus, functions are declared as continuous-time (`hybrid`), discrete-time (`node`), or combinatorial. The *patterns* (p) in function arguments are nested pairs of variables. An *expression* (e) is either a variable x, a constant v, the application of an external operator *op* or a function f, a pair, or an initialized unit delay (e `fby` e),[1] and may refer to a set of locally recursive equations E. Several types of *equation sets* E are supported: simple equations where a variable is defined by an expression, parallel compositions (`and`), piecewise constant variables defined by a list of reset handlers and either an initial expression (`present`/`init`) or a default expression (`present`/`else`), timers with an initial value and a list of reset handlers (`timer`), invariant declarations (`always`) and signals that may be emitted when a constraint is satisfied (`emit`). Compared to Zélus, these last three equations are specific to ZSy. A list of *reset handlers* h comprises a sequence of condition/expression

[1]The declaration `x = e0 fby e` defines a stream x where $x(0) = [\![\mathtt{e0}]\!](0)$ and for all $n > 0$, $x(n) = [\![\mathtt{e}]\!](n-1)$.In other words, it takes its initial value from `e0` and thereafter is equal to `e` delayed by one instant.

```
d ::= let hybrid f(p) = e
    | let node f(p) = e
    | let f(p) = e
    | d d
p ::= x | (p, p)
e ::= x | v | op(e) | f(e) | (e, e)
    | e fby e
    | e where rec E
E ::= x = e
    | E and E
    | x = present h init e
    | x = present h else e
    | timer x init e reset h
    | always { c }
    | emit x when { c }
h ::= e → e | ··· | e → e
c ::= Δ ~ e | c && c
Δ ::= x | x − x
~ ::= < | ≤ | ≥ | >
```

Fig. 8. Syntax of ZSy.

pairs $x_1 \rightarrow e_1$ | $\cdots$ | $x_n \rightarrow e_n$. Conditions must be boolean expressions in discrete-time contexts and signals in continuous-time contexts. When condition x_i is enabled (and $\forall j < i$, x_j is not enabled), the handler h takes the value of the corresponding expression e_i. When no condition is enabled, equations either keep their previous value (`init`) or take a default value (`else`). The form of *constraints* c is limited to allow for its encoding as a DBM.

For simplicity, the only possible action when a guard is fired is to emit a signal. This is not a real restriction since the `present` constructs allow signals to trigger arbitrary discrete-time computations. Since ZSy is a single assignment kernel, no signal can be emitted by two or more distinct guards. After compilation, these signals become inputs of the simulation controlled by the user to handle nondeterministic choices.

IV. COMPILATION

ZSy programs are compiled modularly for symbolic simulation by rewriting the continuous-time components, that is, those marked `hybrid`, into the purely discrete-time subset of the language. A type system inherited from Zélus [11] discriminates between discrete-time and continuous-time computations. The resulting programs have additional inputs and outputs to handle zones and nondeterministic choices. They manipulate abstract DBM values using the small set of operations introduced in Section II. The compilation produces valid discrete Zélus code that can be turned into executables by the Zélus toolchain.

At each step of the simulation scheme described in Section I, a user chooses either to `wait` or to fire enabled guards.

1) If the chosen transition is `wait` we compute the new zone by letting time elapse until the next change in the set of enabled guards (if allowed by the active invariants).
2) Otherwise, firing guards triggers discrete-time computations, possibly resets timers, and returns a new initial zone obtained by letting time elapse from the current initial zone until the next change in the set of enabled guards.

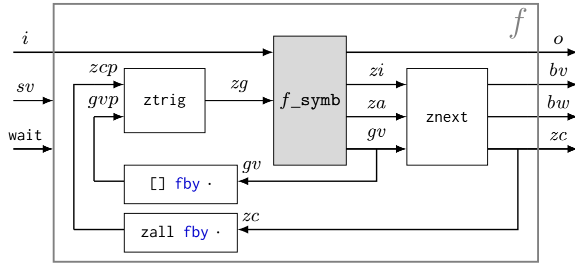
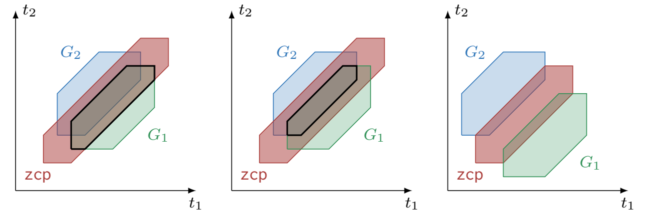


Fig. 9. Compiling a top-level continuous-time function `let hybrid` $f(i)$ = o into a discrete-time one `let node` f(wait, sv, i) = o, bv, bw, zc for simulation. The function f_symb results from the source-to-source transformation of f.



(a) Firing guard G_1 (b) Firing G_1 and G_2 (c) Exclusive guards

Fig. 10. Computing the trigger zone from the current zone zcp.

Compiling the top-level function `let hybrid` $f(i)$ = o gives the result depicted in Figure 9. There are two additional inputs: sv a boolean vector where elements are set to *true* to fire guards, and wait a boolean set to *true* to fire the wait transition. And three additional outputs: bv a boolean vector characterizing the set of enabled guards, bw a boolean set to *true* if the wait transition is enabled, and zc the current zone.

The wait transition and the guard activations are mutually exclusive: if the input wait is set to *true*, other inputs are ignored. Unlike in Timed Automaton models, however, it is possible to simultaneously fire multiple guards. Indeed this is a typical feature of synchronous languages.

The generated function defines three execution phases:

1) Given the current zone zcp and the vector of guard activation zones gvp computed at the previous step, function ztrig computes the trigger zone zg.
2) Function f_symb triggers the discrete-time computations and returns the initial zones zi obtained by applying the resets to zg, the conjunction of active invariants za, and the new vector of guard activation zones gv.
3) Function znext computes the new zone zc by letting time elapse from zi until the set of enabled guards changes.

A. Computing the trigger zone: ztrig

The function ztrig computes the trigger zone of the guards fired by the user by calculating their intersection with the current zone zcp.

```
let node ztrig(sv, zcp, gvp) = zg where
  rec fv = filter(gvp, sv)
  and zg = zinter(zcp, zinterfold(fv))
```

The list of the activation zones of the fired guards fv is obtained by filtering gvp according to sv, whose *true* elements indicate which guards to fire. Multiple guards may be fired simultaneously, as illustrated in Figure 10. Firing mutually exclusive guards gives an empty zone and a blocked simulation.

B. Source-to-source generation of f_symb

We adapt the source-to-source compilation scheme originally developed to compile Zélus programs for simulation with a numerical solver [11, §4]. The translation replaces timers, invariants, and guard definitions. Hybrid declarations in the source program, `let hybrid` $f(i)$ = o become discrete-time declarations in the resulting program,

$$\texttt{let node } f\texttt{_symb}(tv, \texttt{wait}, sv, zg, i) = o, zi, za, gv.$$

There are four new inputs: tv, a vector of timer identifiers; wait, a boolean for indicating a wait transition; sv, a boolean vector indicating the guards to fire; and zg, a trigger zone computed by ztrig. And three new outputs: zi, an initial zone; za, the conjunction of all invariants; and gv, a vector of guard activation zones. At the end of the compilation process, when combining f_symb with ztrig and znext, timer identifiers are replaced by unique values $1, 2, 3, \ldots, n$: the assigned dimensions in the manipulated DBMs. This approach permits the discrimination of timers defined in multiple instantiations of the same node (as in scheduler, for example).

The generation of f_symb is defined by the five mutually recursive functions presented in Figure 11:

$\mathit{TraDef}(d)$ translates declarations. Only continuous-time declarations introduced by `hybrid` are modified.

$\mathit{Tra}(zi, e)$ translates expressions using a variable zi to pass the currently computed version of the initial zone.

$\mathit{TraEq}(zi, E)$ translates equations.

$\mathit{TraZ}(zi, c)$ translates constraints.

$\mathit{TraH}(zi, h)$ translates handlers.

In addition to the resulting expression, constraint, or handler, the last four functions return an intermediate result in the calculation of the new initial zone (zi), and five vector variables to accumulate invariant zones (av), guard activation zones (gv), new signal inputs (sv), new timer inputs (tv), and new equations (E).[2]

We now describe each of the definitions in Figure 11 from the top down.

As described above, the translation of a continuous-time function definition yields a new function with extra inputs and outputs. The translation of the defining expression gives a tuple $\langle e', zi', av, gv, sv, tv, E\rangle$. The initial zone computed incrementally from zg by the translated expression, zi', is only returned when a wait transition is not requested, otherwise the previous value of zi is returned, or zall initially, to be manipulated by the sweeping algorithm in znext (see Figure 9).

[2]We write [] to denote the empty vector and the empty set of equations; $[x_1, \ldots x_n] @ [y_1, \ldots, y_n] = [x_1, \ldots, x_n, y_1, \ldots, y_n]$ to denote the concatenation of two vectors; and $x_0 :: [x_1, \ldots, x_n] = [x_0, x_1, \ldots, x_n]$ to denote the addition of an element at the beginning of a vector.

$TraDef(\texttt{let hybrid } f(p) \texttt{ = } e)$ = let $\langle e', zi', av, gv, sv, tv, E\rangle = Tra(zg, e)$ in
$\texttt{let node } f\texttt{_symb}(tv, wait, sv, zg, p) \texttt{ = } e', zi, za, gv \texttt{ where}$
$\quad\texttt{rec } E \texttt{ and } za \texttt{ = zinterfold}(av)$
$\quad\texttt{and } zi \texttt{ = if } wait \texttt{ then (zall fby } zi\texttt{) else } zi'$
where zi, and za are fresh variables.

$Tra(zi, x)$ = $\langle x, zi, [], [], [], [], []\rangle$ Constants (v) are treated likewise.

$Tra(zi, f(e))$ if f is combinatorial = let $\langle e', zi', av, gv, sv, tv, E\rangle = Tra(zi, e)$ in $\langle f(e'), zi', av, gv, sv, tv, E\rangle$
The rules for $op(e)$, (e_1, e_2), and $\texttt{where rec}$ are defined similarly.

$Tra(zi, f(e))$ if f is continuous-time = let $\langle e', zi', av, gv, sv, tv, E\rangle = Tra(zi, e)$ in
$\langle r, zi_f, za :: av, g :: gv, s :: sv, t :: tv, E \texttt{ and } (r, zi_f, za, g) \texttt{ = } f\texttt{_symb}(t, wait, s, zi', e')\rangle$
where r, zi_f, za, g, s, and t are fresh variables.

$TraEq(zi, x \texttt{ = } e)$ = let $\langle e', zi', av, gv, sv, tv, E\rangle = Tra(zi, e)$ in $\langle zi', av, gv, sv, tv, E \texttt{ and } x \texttt{ = } e'\rangle$

$TraEq(zi, E_1 \texttt{ and } E_2)$ = let $\langle zi_1, av_1, gv_1, sv_1, tv_1, E_1'\rangle = TraEq(zi, E_1)$ in
let $\langle zi_2, av_2, gv_2, sv_2, tv_2, E_2'\rangle = TraEq(zi_1, E_2)$ in
$\langle zi_2, av_1 @ av_2, gv_1 @ gv_2, sv_1 @ sv_2, tv_1 @ tv_2, E_1' \texttt{ and } E_2'\rangle$

$TraEq(zi, x \texttt{ = present } h \texttt{ init } e_0)$ = let $\langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h\rangle = TraH(zi, h)$ in
$\langle zi_h, av_h, gv_h, sv_h, tv_h, E_h \texttt{ and } x \texttt{ = present } h' \texttt{ init } e_0\rangle$
and likewise for $x \texttt{ = present } h \texttt{ else } e$.

$TraEq(zi, \texttt{timer } t \texttt{ init } e_0 \texttt{ reset } h)$ = let $\langle h', zi_h, av_h, gv_h, sv_h, tv_h, E_h\rangle = TraH(zi, h)$ in
let $x_1' \rightarrow e_1 \mid \cdots \mid x_n' \rightarrow e_n = h'$ in
$\langle zi_t, av_h, gv_h, sv_h, t :: tv_h,$
$\quad E_h \texttt{ and } zi_t \texttt{ = present (true fby false)} \rightarrow \texttt{zreset}(zi_h, t, e_0)$
$\qquad \mid x_1' \rightarrow \texttt{zreset}(zi_h, t, e_1) \mid \cdots \mid x_n' \rightarrow \texttt{zreset}(zi_h, t, e_n)$
$\qquad \texttt{else } zi_h\rangle$
where zi_t is a fresh variable.

$TraEq(zi, \texttt{always \{ } c \texttt{ \}})$ = let $\langle c', zi_c, av_c, gv_c, sv_c, tv_c, E_c\rangle = TraZ(zi, c)$ in
$\langle zi_c, za :: av_c, gv_c, sv_c, tv_c, E_c \texttt{ and } za \texttt{ = zmake}(c')\rangle$
where za is a fresh variable.

$TraEq(zi, \texttt{emit } s \texttt{ when \{ } c \texttt{ \}})$ = let $\langle c', zi_c, av_c, gv_c, sv_c, tv_c, E_c\rangle = TraZ(zi, c)$ in
$\langle zi_c, av_c, zs :: gv_c, s :: sv_c, tv_c, E_c \texttt{ and } zs \texttt{ = zmake}(c')\rangle$
where zs is a fresh variable.

$TraZ(zi, \Delta_1 \sim_1 e_1 \cdots \texttt{ \&\& } \Delta_n \sim_n e_n)$ = let $\langle e_i', zi_i, av_i, gv_i, sv_i, tv_i, E_i\rangle = Tra(zi_{i-1}, e_i)$ in
$\langle \Delta_1 \sim_1 e_1' \cdots \texttt{ \&\& } \Delta_n \sim_n e_n', zi_n, av_1 \cdots @ av_n, gv_1 \cdots @ gv_n,$
$\quad sv_1 \cdots @ sv_n, tv_1 \cdots @ tv_n, E_1 \cdots \texttt{ and } E_n\rangle$
where $zi_0 = zi$. $TraH(zi, x_1 \rightarrow e_1 \mid \cdots \mid x_n \rightarrow e_n)$ is treated similarly.

Fig. 11. The source-to-source generation of f_symb.

The conjunction of the invariants in av is calculated with zinterfold and returned as za. The current guard activation zones are returned in gv. The vectors sv and tv are bound as inputs. Finally, any generated equations E are added to the node body.

Variables (x) and constants (v) are passed unchanged. They do not have any effect on the calculation of the initial zone, nor introduce any new zones, inputs, or equations. Neither do combinatorial function applications, external operators, or the tuple constructor: the results of a recursive call to $Tra(zi, e)$ are simply transmitted. Local definitions are handled similarly with additional recursive calls to $TraEq(zi, E)$. There are no rules for discrete-time function applications or fbys as the type system only permits such elements in discrete-time contexts—which are not transformed—and thus not directly within the body of a continuous-time function.

A continuous-time function application $f(e)$ is replaced by a discrete-time one $(r, zi_f, za, g) \texttt{ = } f\texttt{_symb}(t, wait, s, zi', e')$, introduced as a new equation. The fresh variables t and s pass the required timer identifiers and guard inputs, e' is the translation of e, and zi' passes the initial zone calculated by the translated elements. The t and s variables, and those for the returned conjunction of invariants (za), and the vector of guard activation zones are added to the appropriate accumulators. The original outputs of the application are returned in a fresh variable r, as is the updated initial zone, zi_f. The structure of nested function calls is reflected in the tree structure of sv, tv, and gv.

For simple equations (x = e), parallel compositions (and), and present constructs, we apply the translation recursively to compute the initial zone and accumulate inputs, guard activation zones, and invariants. Initial zone calculations are threaded from one side of a parallel composition to the other, which may, regrettably, introduce artificial causality constraints. These can be minimized by sorting the equations prior to their transformation. Otherwise, the Zélus compiler will inline function applications to break causality constraints if necessary. In the present construct, the initial expression e_0 and the handler bodies are discrete and thus do not require translation.

The translation of a timer definition adds a new timer

identifier t to tv and an equation defining the updated initial zone zi_t by a present/else construct. A new handler ensures that the tth dimension of the initial zone is initially reset to the value of e_0, the other handlers are translated from the original construct with calls to zreset introduced to update the initial zone when necessary, otherwise the else branch passes the initial zone without modifying it.

Invariants are translated into an equation defined by zmake, which generates a DBM from the constraint expression.

Finally, the translation of a signal emission adds an activation zone, zs, to gv and a corresponding element to the vector sv of signals controlled by the user. The activation zone is defined by an equation using zmake. Since ZSy is a single assignment kernel, a signal can only ever be defined by a single guard.

C. *Computing the current zone:* znext

The discrete-time function znext computes the current zone zc and the set of enabled transitions (wait or enabled guards) using the information computed by f_symb, namely the initial zone zi, the conjunction of all invariants za, and the vector of guard activation zones gv.

```
let node znext(wait, zi, za, gv) = zc, bv, bw where
  rec dp = if wait then (dzero fby d) else dzero
  and dl = zdistmap(zi, gv)
  and d  = mindist(gv, dp)
  and zn = zsweep(zi, dp, d)
  and zc = zinter(zn, za)
  and bv = zenabled(zc, gv)
  and zm = zinter(zup(zn), za)
  and bw = (zc ≠ zm)
```

When the user chooses to wait, the current zone zc is obtained via the sweeping mechanism described in Section II-B. Sweeping restarts from dzero $= (0, \leq)$ whenever the user fires guards (otherwise f_symb's outputs are unchanged).

We compute dl, the list of distances for reachable guards, from zi and gv. The function mindist(dl, dp) gives the smallest distance greater than dp. It enumerates the distance pairs in order; dp = dzero fby d is the distance reached at the previous step. Each successive distance pair (dp, d) defines a zone zn obtained by sweeping zi between dp and d. The current zone zc is the intersection of zn and the invariant za.

We compute bv, the vector characterizing the enabled guards, from zc and gv. The maximal zone zm is obtained from zn by letting time elapse indefinitely with the same invariant za. A wait transition is possible as long as zc $\neq$ zm (bw).

Returning zm rather than zc gives the Uppaal simulation scheme where a user may choose from all reachable guards.

D. *Example*

Figure 12 presents the result of the compilation of the clock and scheduler nodes defined in Section I. Each continuous-time function is compiled into two discrete-time functions: the result of the source-to-source generation (clock_symb and scheduler_symb), and the wrapper presented in Figure 9. Note that the compilation is modular: scheduler_symb makes two calls to clock_symb. The scheduler function takes as additional inputs the signals wait and (c1, c2) which allows the user to

```
let node clock_symb(t, wait, c, zg, (t_min, t_max))
    = c, zi, za, [zs] where
  rec zit = present (true fby false) → zreset(zg, t, 0)
          | c → zreset(zg, t, 0)
          else zg
  and zs = zmake({t ≥ t_min})
  and zb = zmake({t ≤ t_max})
  and za = zinterfold([zb])
  and zi = if wait then (zall fby zi) else zit

let node clock(wait, c, (t_min, t_max)) = c', bv, bw, zc where
  rec zg = ztrig([c], zcp, gvp)
  and c', zi, za, gv = clock_symb(1, wait, c, zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv

let node scheduler_symb((t1, t2), wait, (c1, c2), zg, (t_min, t_max))
    = (c1', c2'), zi, za, gv1 @ gv2 where
  rec c1', zi1, za1, gv1 = clock_symb(t1, wait, c1, zg, (t_min, t_max))
  and c2', zi2, za2, gv2 = clock_symb(t2, wait, c2, zi1, (t_min, t_max))
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zi2

let node scheduler(wait, (c1, c2), (t_min, t_max))
    = (c1', c2'), bv, bw, zc where
  rec zg = ztrig([c1; c2], zcp, gvp)
  and (c1', c2'), zi, za, gv =
    scheduler_symb((1, 2), wait, (c1, c2), zg, (t_min, t_max))
  and zc, bv, bw = znext(wait, zi, za, gv)
  and zcp = zall fby zc
  and gvp = [] fby gv
```

Fig. 12. Compilation of clock and scheduler.

control the symbolic simulation. Discrete-time functions are not modified. It is, for instance, possible to simulate the application shown in Figure 1 by activating the discrete controllers P_1 and P_2 on the signals c1' and c2' emitted by scheduler.

V. Related work

Our proposal is based on Timed Safety Automata [2], [3], and greatly influenced by the Uppaal (http://www.uppaal.org) [4], [5] simulator where users choose among enabled transitions to navigate the reachable symbolic states of a model. Our approach differs in two main ways. First, we include explicit 'wait transitions' and develop algorithms for them (Sections II-A and II-B). Second, Uppaal models are networks of Timed Automata with a C-like language for manipulating values, whereas we adapt techniques developed for synchronous languages that allow state-based behaviors to be encapsulated in named nodes and arbitrarily composed in parallel and hierarchically. Synchronous languages have efficient, modular compilation schemes and are regularly used to program large-scale embedded controllers. We treat simultaneous signal emission, integrate a rich language for discrete controllers and techniques for its efficient compilation, and give a meaning to timing constructs by translating them into a simpler discrete language. Compared Zélus, we allow a limited form of non-determinism in guard expressions and present a novel source-to-source compilation pass to simulate (some programs) symbolically rather than numerically.

Much has been written about verifying hybrid systems [12], typically using symbolic representations to over-approximate the states and traces of hybrid automata. This article does not address verification, and focuses instead on language design and compilation. Our symbolic representations are exact.

Several works propose adding nondeterministic constraints to synchronous languages. Lutin [13], [14] and the commercial Argosim Stimulus tool (http://argosim.com) are designed for testing discrete-time reactive programs. Inputs are specified nondeterministically and characterized at each simulation step by a set of possible values. Concrete rather than symbolic traces are generated by choosing values randomly. Yo-yo [15], [16] is a tool for the symbolic simulation of discrete-time dataflow programs extended with nondeterministic relations between variables. The simulator generates symbolic states that represent sets of possible variable values. Rather than add nondeterminism to variable values, we add timing nondeterminism to continuous-time constructs—variables still evolve deterministically. Compilation still produces a (discrete) synchronous program, but the discretization is chosen based on the timing constraints.

There have been several propositions for integrating continuous time into synchronous programs, usually for verifying real-time properties. The semantics of Argos programs extended with timeouts and watchdogs [17] was expressed as timed automata for verification in the Kronos model checker [18]. Similarly, Taxys [19] allows the annotation of Esterel programs with real-time characteristics and verifies them with Kronos. The idea is to design applications in discrete time and to show their correctness with respect to implementations modeled in continuous time. The semantics of Quartz programs have been defined using timed Kripke structures for the verification of timing properties [20] and more recently extended to treat hybrid systems [21]. Abstract interpretation techniques have been proposed to verify safety properties of synchronous models involving multiform timing characteristics linked by linear relations [22]. We focus on a different problem: the symbolic simulation of real-time models involving nondeterminism. Instead of adding real-time assumptions to a discrete-time program, we start from a continuous-time model and show how an execution can be discretized.

VI. Conclusion

We combine existing techniques and data structures for Timed Safety Automata with typing and compilation techniques for synchronous languages to develop a novel programming language where discrete reactive logic can be mixed with nondeterministic continuous-time features. In particular, we present an extension of Lustre and a specialization of Zélus for modeling real-time reactive systems, propose a symbolic simulation scheme based on 'sweeping', and show how to implement it via source-to-source compilation. A type system, not presented here, ensures the correct composition of discrete-time and continuous-time elements [23, §6.5].

Our proposal has been implemented using the Zélus compiler and a small library of DBM operations.[3] The addition of valued signals and hierarchical automata to the language, and its application to a standard academic example are described elsewhere [23, §6.7]. Unlike the work around Uppaal, we do not address verification or treat industrial case studies. A future direction could be to verify programs in our 'extended version of Lustre' by either generating C code and using the highly-tuned Uppaal DBM library, or combining symbolic techniques for Lustre programs [24] with those for Timed Automata [25].

[3] https://github.com/gbdrt/zsy/tree/fdl17

Acknowledgment

We thank Louis Mandel for his valuable remarks on the text.

References

[1] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *POPL*, Germany, Jan. 1987, pp. 178–188.

[2] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[3] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, vol. 111, no. 2, pp. 192–244, Jun. 1994.

[4] K. G. Larsen, P. Pettersson, and Y. Wang, "Uppaal in a nutshell," *J. Softw. Tools for Tech. Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.

[5] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. Riverside, California, USA: IEEE Computing Society, Sep. 2006, pp. 125–126.

[6] P. Caspi, "The quasi-synchronous approach to distributed control systems," VERIMAG, Crysis Project, Tech. Rep. CMA/009931, May 2000, *The Cooking Book*.

[7] F. Vaandrager and A. de Groot, "Analysis of a biphase mark protocol with Uppaal and PVS," *Formal Aspects of Computing*, vol. 18, no. 4, pp. 433–458, Dec. 2006.

[8] T. Bourke and M. Pouzet, "Zélus: A synchronous language with ODEs," in *HSCC*, USA, Apr. 2013, pp. 113–118.

[9] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in *AVMFSS*, France, Jun. 1990, pp. 197–212.

[10] J. Bengtsson, "Clocks, DBMs and states in timed systems," Ph.D. dissertation, Uppsala University, 2002.

[11] A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet, "Divide and recycle: types and compilation for a hybrid synchronous language," in *LCTES*, USA, Apr. 2011, pp. 61–70.

[12] R. Alur, "Formal verification of hybrid systems," in *EMSOFT*, Taiwan, Oct. 2011, pp. 273–278.

[13] P. Raymond, Y. Roux, and E. Jahier, "Lutin: A language for specifying and executing reactive scenarios," *EURASIP J. of Emb. Sys.*, 2008.

[14] ——, "Specifying and executing reactive scenarios with Lutin," *Elecronic Notes in Theoretical Computer Science*, vol. 203, no. 4, pp. 19–34, 2008.

[15] C. Mauras, "Symbolic simulation of interpreted automata," in *SYNCHRON*, Germany, Dec. 1996.

[16] D. Garriou, "Symbolic simulation of synchronous programs," *Elecronic Notes in Theoretical Computer Science*, vol. 65, no. 5, pp. 11–18, 2002.

[17] M. Jourdan, F. Maraninchi, and A. Olivero, "Verifying quantitative real-time properties of synchronous programs," in *CAV*, Greece, Jun. 1993.

[18] S. Yovine, "Kronos: A verification tool for real-time systems," *J. Softw. Tools for Tech. Transfer*, vol. 1, no. 1, pp. 123–133, 1997.

[19] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxys = Esterel + Kronos: A tool for verifying real-time properties of embedded systems," in *CDC*. Orlando, Florida, USA: IEEE, Dec. 2001, pp. 2875–2880.

[20] G. Logothetis and K. Schneider, "Extending synchronous languages for generating abstract real-time models," in *DATE*, France, Mar. 2002.

[21] K. Bauer and K. Schneider, "From synchronous programs to symbolic representations of hybrid systems," in *HSCC*. Stockholm, Sweden: ACM Press, Apr. 2010, pp. 41–50.

[22] N. Halbwachs, "Delay analysis in synchronous programs," in *CAV*, Greece, Jun. 1993, pp. 333–346.

[23] G. Baudart, "A synchronous approach to quasi-periodic systems," Ph.D. dissertation, PSL Research University, Mar. 2017.

[24] G. Hagen and C. Tinelli, "Scaling up the formal verification of Lustre programs with SMT-based techniques," in *FMCAD*, USA, Nov. 2008, pp. 109–117.

[25] T. Isenberg and H. Wehrheim, "Timed automata verification via IC3 with zones," in *ICFEM*, ser. LNCS, vol. 8829, Nov. 2014, pp. 203–218.