1.7 Extensions

In this section we discuss two possible extensions of ZSy to improve its expressiveness: valued signals and automata.

Valued signals

In ZSy, signals cannot carry values but it is relatively simple to add this feature to the language by reusing Zélus signals. The type of a signal α *signal* is parametrized by α, the type of its values. A pure signal, without any value, has type *unit signal*. An expression calculating a value to emit on a signal must be of kind D since emissions are discrete computations.

We keep the syntax of Zélus for valued signals:

$$(\textit{emission}) \quad \texttt{emit}\ s\ [\texttt{=}\ e]$$

$$(\textit{reception}) \quad \texttt{present}\ s(v)\ [\texttt{on}\ P(v)] \rightarrow\ e$$

with the particular case `present` $s() \rightarrow e$ for pure signals. The optional condition [`on` $P(v)$] allows to filter the value v received on a signal with a boolean predicate P directly in the branches of the `present` handler.

Automata

A major restriction of ZSy is the absence of state in continuous functions. Conditionals like `if` e_0 `then` e_1 `else` e_2 can be added as an external operator of arity 3, but in that case, the three expressions e_0, e_1, and e_2, and the equations produced by their translations, are computed at every step. It is, however, possible to extend ZSy with hierarchical automata following the compilation technique introduced in [BBCP11b].

Consider the following example:

```
let hybrid auto() = o where
  rec automaton
      | S1 → do o = 1
             and timer t1 init 0 reset c1 → 0
             and emit c1 when {t1 > 3}
             and always {t1 ≤ 5}
        until c1 then S2
      | S2 → do o = 2
             and timer t2 init 0 reset c2 → 0
             and emit c2 when {t2 > 2}
             and always {t2 ≤ 7}
        until c2 then S1
```

val auto: unit $\xrightarrow{\mathrm{C}}$ *int*

An automaton in continuous contexts is translated into a similar discrete automaton where the signals triggering transitions between states are replaced by boolean conditions.

The easiest solution is to duplicate all equations introduced by timers, guards, and invariants during the translation such that, if a variable is defined in one state of the automaton, the same variable returns a dummy value in all other states.

```
let hybrid auto_symb((t1, t2), wait, (c1, c2), zg) = o, zi, za, [zs1; zs2] where
  rec automaton
      | S1 → do o = 1
              and zi1 = present (true fby false) → zreset(zg, t1, 0)
                        | c1 → reset(zg, t1, 0)
                        else zg
              and zs1 = zmake({t1 > 3})
              and za1 = zmake({t1 ≤ 5})
              and zi2 = zall and zs2 = zempty and za2 = zall
          until c1 then S2
      | S2 → do o = 2
              and zi2 = present (true fby false) → zreset(zg, t2, 0)
                        | c2 → reset(zg, t2, 0)
                        else zg
              and zs2 = zmake({t2 > 2})
              and za2 = zmake({t2 ≤ 7})
              and zi1 = zall and zs1 = zempty and za1 = zall
          until c2 then S1
  and za = zinterfold([za1; za2])
  and zi = if wait then (zall fby zi) else zinterfold([zi1; zi2])
```

val auto_symb: (int × int) × bool × (bool × bool) × zone $\xrightarrow{\mathrm{D}}$ *int × zone × zone × zone list*

Each state of the automaton generates a possible initial zone zi. This variable takes the dummy value `zall` in all other states. We gather all these zones into a vector and the global initial zone is the intersection of its elements. The activation zone of a guard defined in one state is empty in all other states (the guard cannot be enabled). An invariant defined in one state becomes `zall` in all other states.

Following [BBCP11b], it is also possible to minimize memory allocations by reusing variables across multiple states. For instance, the pairs of variables (`zi1`, `zi2`) and (`za1`, `za2`) could be merged since they are used in exclusive states. However, we still need to gather all timer identifiers, guard signals, and guard activation zones for interaction with the user.

A complete example: the train gate

We motivated our approach with the example of a quasi-periodic architecture which is both a typical example of a nondeterministic timed system and the main focus of this thesis. But our proposal is more general and ZSy, once extended with valued signals and automata, permits the expression of more complex models. For instance the train gate [BDL06, §4] is a classic example of a system mixing nondeterministic continuous components, the train controllers; with discrete components, the gate controller.

The gate controls access to a bridge for several trains. The bridge can be crossed by only one train at a time and the gate ensures that a train never engages if another is still crossing the bridge. Timing constraints are used to model uncertainty on the speeds of the trains.

Train controller When approaching the bridge, a train waits 10 time units to receive a `stop` signal from the gate controller. If, after this delay, nothing is received, the train starts crossing the bridge. On the other hand, if a signal `stop` is received, the train stops and waits until the gate sends a `go` signal when the bridge is free.

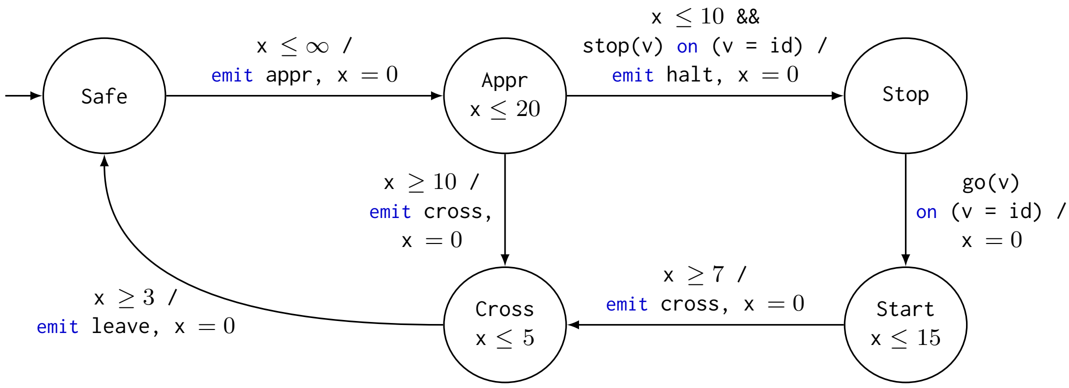

Figure 1.14: [BDL06, Figure 9] Nondeterministic train controller where `x` is a timer reset at each transition and `id` is the train identifier.

The train controller is the five-state automaton illustrated in figure 1.14. Its inputs are the two signals emitted by the gate: `stop` and `go`. The train sends a signal `appr` when approaching the bridge and a signal `leave` when leaving the bridge. A train is characterized by a unique identifier `id`.

```
let hybrid train(id, stop, go) = appr, leave where
  rec timer x init 0
      reset go(v) on (v = id) | halt() | appr() | cross() | leave() → 0
  and automaton
      | Safe  → do emit appr when {x ≤ ∞}
          until appr() then Appr
      | Appr  → do emit halt when {x ≤ 10} && (stop(v) on (v = id))
                 and emit cross when {x ≥ 10}
                 and always {x ≤ 20}
          until halt() then Stop
           else cross() then Cross
      | Stop  → do
          until go(v) on (v = id) then Start
      | Start → do emit cross when {x ≥ 7}
                 and always {x ≤ 15}
          until cross() then Cross
      | Cross → do emit leave when {x ≥ 3}
                 and always {x ≤ 5}
          until leave() then Safe
```

val train: ident $\times$ *ident signal* $\times$ *ident signal* $\xrightarrow{\text{C}}$ *unit signal* $\times$ *unit signal*

In the initial state `Safe`, a train can approach the bridge, sending a signal `appr` with its `id`, at any time. The constraint for leaving this state is thus `{x ≤ ∞}`.

An approaching train takes at most 20 time units to reach the bridge. This is expressed in the invariant `always {x ≤ 20}` in the `Appr` state. If the train receives a message `stop` that corresponds to its `id` within the first 10 time units (`stop(v) on (v = id) && {x ≤ 10}`) it can be stopped before the bridge. The train then enters the `Stop` state. Otherwise, after 10 time units, the train cannot be stopped and starts crossing, that is, enters the `Cross` state.

At $x = 10$ both transitions are possible and can be activated simultaneously. As in Zélus, transitions are treated sequentially in the `until` handler and in our model `halt` takes priority.

In the `Stop` state the train waits for a signal `go` corresponding to its `id` and then enters the `Start` state to resume its crossing. The train takes between 7 and 15 time units to restart, then it begins crossing the bridge. Finally, the crossing takes between 3 and 5 time units and a signal `leave` is emitted when the train leaves the bridge.

The timer `x` is reset whenever the controller enters a new state.

Gate controller The gate controller is a classic discrete controller. It maintains a queue of approaching trains.

```
let node queue(push, pop) = q where
  rec init q = empty()
  and present
      | push(v) & pop(_) → do q = enqueue(dequeue(last q), v) done
      | push(v) → do q = enqueue(last q, v) done
      | pop(_) → do q = dequeue(last q) done
```

val queue: α *signal* $\times$ β *signal* $\xrightarrow{\mathrm{D}}$ α *queue*

The queue is updated whenever a new train sends a signal `appr` or leaves the bridge and a two-state automaton controls the emission of the `stop` and `go` signals.

```
let node gate(appr, leave) = stop, go where
  rec q = queue(appr, leave)
  and automaton
      | Free → do
          unless appr(v) on (size(q) = 1) then Occ
            else (size(q) > 0) then do emit go = front(q) in Occ
      | Occ  → do
          unless leave() & appr(v) then do emit stop = v in Free
            else leave() then Free
            else appr(v) then do emit stop = v in Occ
```

val gate: ident signal $\times$ *unit signal* $\xrightarrow{\mathrm{D}}$ *ident signal* $\times$ *ident signal*

The controller starts in the `Free` state. If a train approaches while the queue is empty (in which case, `size(q) = 1` since the train is added to the queue) it starts crossing and the controller enters the `Occ` state. On the other hand, if no train is approaching but the waiting queue is not empty (`size(q) > 0`), the controller sends a signal `go` to the first train in the queue and enters the `Occ` state.

In the `Occ` state, the controller waits for a train to leave the bridge. Meanwhile, it sends `stop` signals to all approaching trains. If the two events occur simultaneously we combine the two behaviors. This discrete controller is activated whenever a signal `appr` or `leave` is emitted by one of the trains.

Complete model The components can now all be plugged together, combining the output signals of the train controllers to form two global signals `appr` and `leave`. For instance, the complete code of a gate controlling two trains is:

```
let hybrid train_gate() = () where
  rec appr1, leave1 = train(1, stop, go)
  and appr2, leave2 = train(2, stop, go)
  and present leave1() | leave2() → do emit leave done
  and present
      | appr1() → do emit appr = 1 done
      | appr2() → do emit appr = 2 done
  and present appr(_) | leave() → do stop, go = gate(appr, leave) done
```

val train_gate: unit $\xrightarrow{\mathrm{C}}$ *unit*

Note that in this example, there is a mutual dependency between the nondeterministic controllers of the trains and the discrete gate controller.