Symbolic Simulation of Dataflow Synchronous Programs with Timers

Guillaume Baudart[1] Timothy Bourke[2,3] Marc Pouzet[4,3,2]

1. IBM Research
2. Inria Paris
3. DI, École normale supérieure
4. Univ. Pierre et Marie Curie

FDL 2017, Verona, Italy—18–20 September 2017

The synchronous language Lustre

[Caspi, Pilaud, Halbwachs, and Plaice (1987): "Lustre: A Declarative Language for Programming Synchronous Systems"]

- Ideal for programming an important class of embedded controllers.
  - Academic foundation of Scade Suite tool for critical industrial systems.

- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: R_1, R_2, R_3, R_4, R_5, ...

The synchronous language Lustre

[Caspi, Pilaud, Halbwachs, and Plaice (1987): "Lustre: A Declarative Language for Programming Synchronous Systems"]

- Ideal for programming an important class of embedded controllers.
  - Academic foundation of Scade Suite tool for critical industrial systems.

- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: $R_1, R_2, R_3, R_4, R_5, \ldots$

But, 'physical' timing constraints are often required.

The synchronous language Lustre [Caspi, Pilaud, Halbwachs, and Plaice (1987): "Lustre: A Declarative Language for Programming Synchronous Systems"]

- Ideal for programming an important class of embedded controllers.
  - Academic foundation of Scade Suite tool for critical industrial systems.

- Based on a discrete-time abstraction.

```
every trigger:
  read inputs;
  compute;
  write outputs
```

model: $R_1, R_2, R_3, R_4, R_5, \ldots$

But, 'physical' timing constraints are often required.

Timed (Safety) Automata [Alur and Dill (1994): "A Theory of Timed Automata"] [Henzinger, Nicollin, Sifakis, and Yovine (1994): "Symbolic Model Checking for Real-Time Systems"]

- Model the passage of time and timing non-determinism
  - (tolerances in requirements / uncertainties in implementations).
- Verification and Symbolic Simulation in Uppaal [Behrmann, David, and Larsen (2006): *A tutorial on Uppaal 4.0*]

Dataflow synchronous language basics

```
let average(x, y) = (x + y) / 2
```

Dataflow synchronous language basics

```
let average(x, y) = (x + y) / 2
```

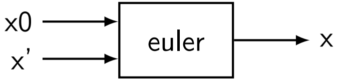
x	0	1	2	5	4	5	6	...
y	4	3	4	2	0	2	2	...
x + y / 2	2	2	3	3	2	3	4	...

Dataflow synchronous language basics

```
let average(x, y) = (x + y) / 2
```

x	0	1	2	5	4	5	6	⋯
y	4	3	4	2	0	2	2	⋯
x + y / 2	2	2	3	3	2	3	4	⋯

```
let h = 10.0
let node euler(x0, x') = x where
  rec nx = x +. (h *. x')
  and x = x0 fby nx
```

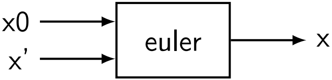


Dataflow synchronous language basics

```
let average(x, y) = (x + y) / 2
```

x	0	1	2	5	4	5	6	⋯
y	4	3	4	2	0	2	2	⋯
x + y / 2	2	2	3	3	2	3	4	⋯

```
let h = 10.0
let node euler(x0, x') = x where
  rec nx = x +. (h *. x')
  and x = x0 fby nx
```

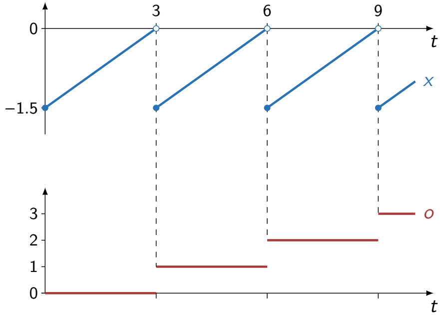


x0	0	1	2	3	4	5	6	⋯
x'	2	1	2	0	2	3	1	⋯
nx	20	30	50	50	70	100	110	⋯
x	0	20	30	50	50	70	100	⋯

- Node: set of causal equations (variables at left).
- Semantic model: synchronized streams of values.
- A node defines a function between input and output streams.

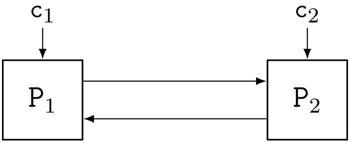
Zélus: synchronous language + ODEs [Bourke and Pouzet (2013): "Zélus: A Synchronous Language with ODEs"]

```
let node nat(v) = y where
  rec y = v fby (y + 1)

let hybrid sawtooth(x', x0) = o where
  rec init o = 0
  and der x = x' init x0 reset z → x0
  and z = up(x)
  and present z → do o = nat(1) done

let hybrid main = sawtooth(0.5, -1.5)
```



- Combine discrete-time and continuous-time behaviours
  - A type system ensures that compositions are well-defined.
  - Align discrete behaviours on 'zero-crossing' events.
- Source-to-source compilation for simulation with a numeric solver.
- Research focus on hybrid programming languages
  - E.g., Simulink/Stateflow, Modelica, Ptolemy...
- Manual and compiler: http://zelus.di.ens.fr

Example: quasi-periodic nodes [Caspi (2000): *The Quasi-Synchronous Approach to Distributed Control Systems*]

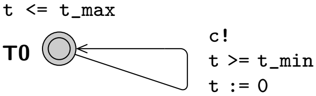


Two network nodes activated on clock inputs c_1 and c_2

- Each node is periodically triggered by a local clock.
- The difference between ticks i and $i + 1$ is bounded:

$$T_{\mathrm{min}} \leq t_{i+1} - t_i \leq T_{\mathrm{max}}$$

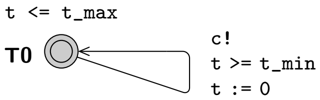
- Easy to model a clock as a Timed Automaton: [Vaandrager and Groot (2006): "Analysis of a Biphase Mark Protocol with Uppaal and PVS"]



- What about combining with discrete controller code?

Clock in Zélus?

```
let hybrid clock(t_min, t_max) = c where
  rec der t = 1.0 init 0.0 reset c() → 0.0
  and present up(t - t_min) → do emit c done
```

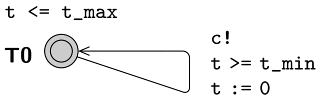


Programming Timed Automaton in Zélus

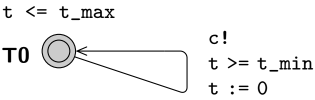
- Very restricted ODEs ($\dot{x} = 1$): no need for a numeric solver.
- Cannot express ‘timing non-determinism’.

- Very appealing to ‘embed’ discrete programs in continuous time.
- The discrete/continuous type system rejects meaningless compositions.

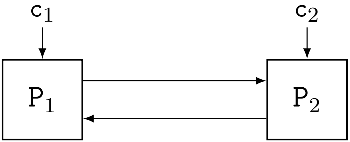
```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0.0 reset c() → 0.0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}
```



```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0.0 reset c() → 0.0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}
```




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




```
let hybrid quasinodes(t_min, t_max) = o1, o2 where
  rec c1, c2 = scheduler(t_min, t_max)
  and o1 = present c → node1(channel(o2)) init oi
  and o2 = present c → node2(channel(o1)) init oi
```

Zsy: syntax

$$
\begin{array}{rcl}
d & ::= & \texttt{let hybrid } f\,(p) \texttt{ = } e \\
 & | & \texttt{let node } f\,(p) \texttt{ = } e \\
 & | & \texttt{let } f\,(p) \texttt{ = } e \\
 & | & d\ d
\end{array}
$$

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid op(e) \\
 & | & (e,\ e) \\
 & | & f\,(e) \\
 & | & e \texttt{ fby } e \\
 & | & e \texttt{ where rec } E
\end{array}
$$

$$
\begin{array}{rcl}
E & ::= & x \texttt{ = } e \\
 & | & E \texttt{ and } E \\
 & | & x \texttt{ = present } h \texttt{ init } e \\
 & | & x \texttt{ = present } h \texttt{ else } e \\
 & | & \texttt{timer } x \texttt{ init } e \texttt{ reset } h \\
 & | & \texttt{always \{ } c \texttt{ \}} \\
 & | & \texttt{emit } x \texttt{ when \{ } c \texttt{ \}}
\end{array}
$$

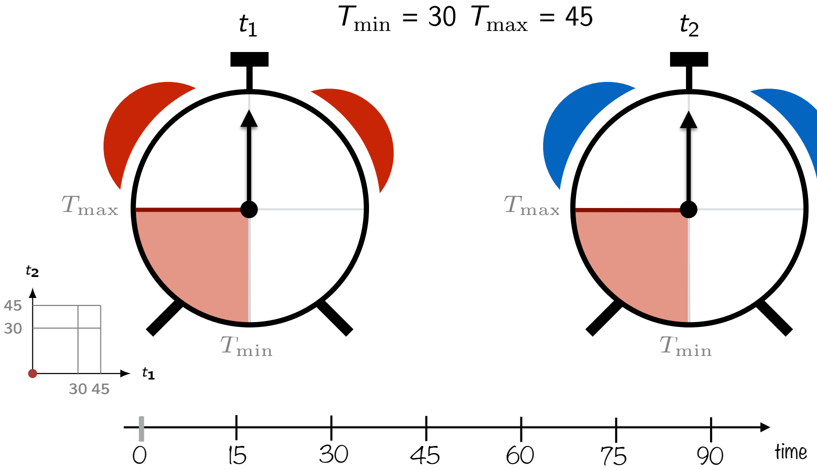
- A program is a list of declarations.
- A node is defined by an expression.
- Expressions refer to sets of equations.

New features

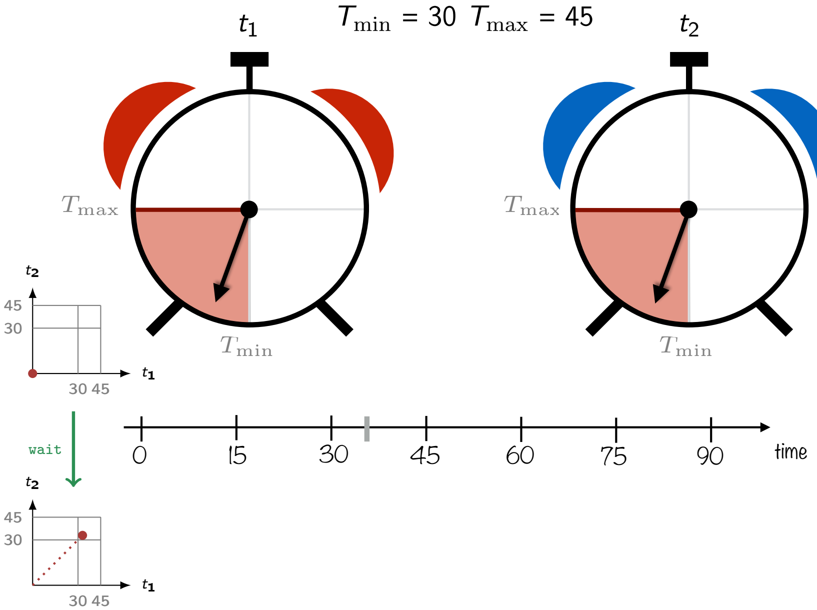
- Timers (time elapsing)
- Invariants (must)
- Guards (may)

$$
\begin{array}{rcl}
p & ::= & x \mid (p,\ p) \\
 & & \\
h & ::= & e \rightarrow e \mid \cdots \mid e \rightarrow e \\
c & ::= & \Delta \sim e \mid c \;\&\&\; c \\
\Delta & ::= & x \mid x - x \\
\sim & ::= & < \mid \leq \mid \geq \mid >
\end{array}
$$

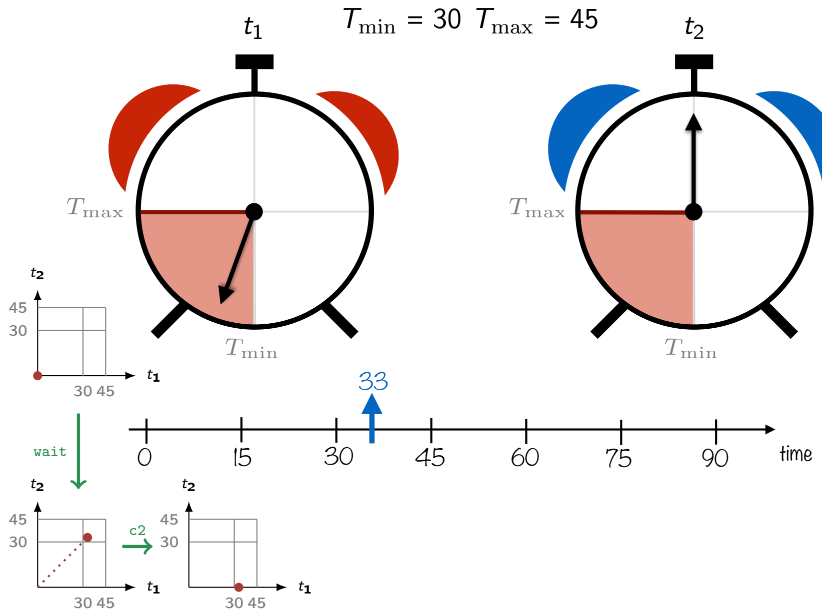
Concrete Simulation Trace



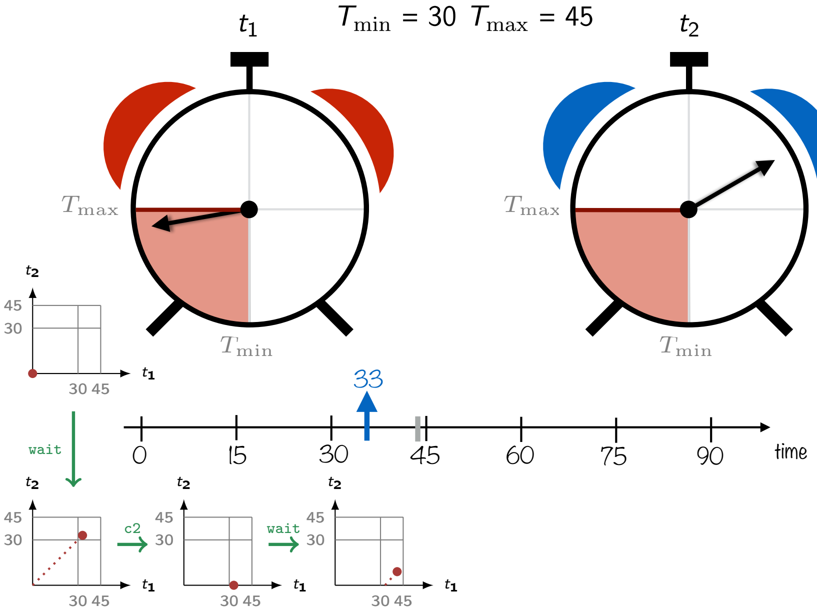
Concrete Simulation Trace



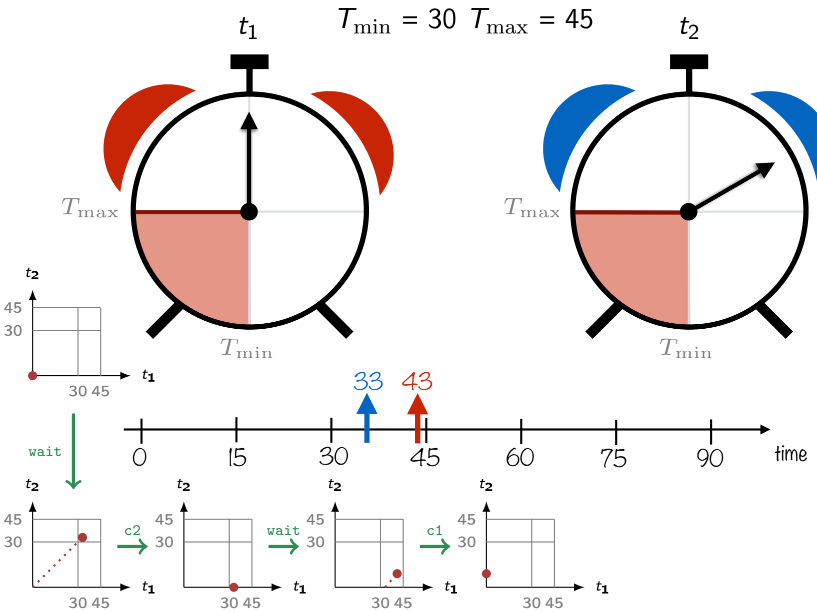
Concrete Simulation Trace



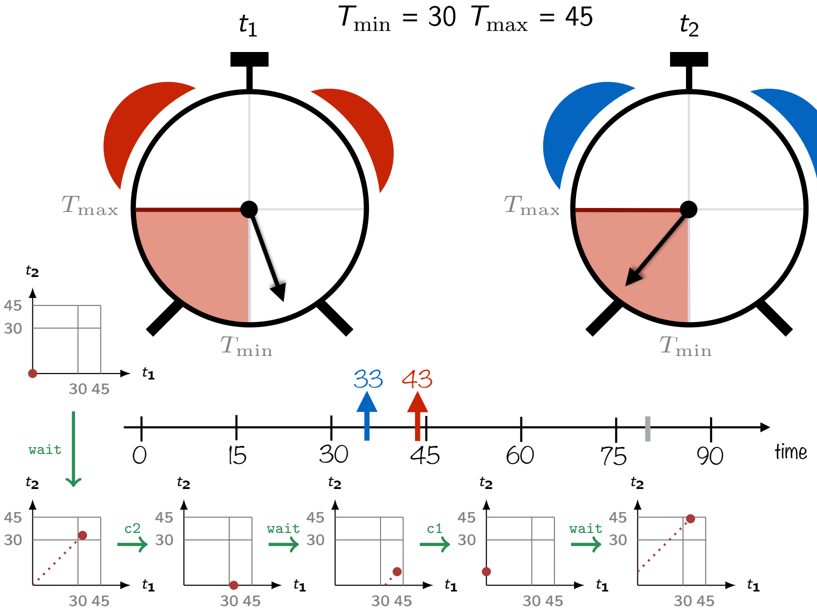
Concrete Simulation Trace



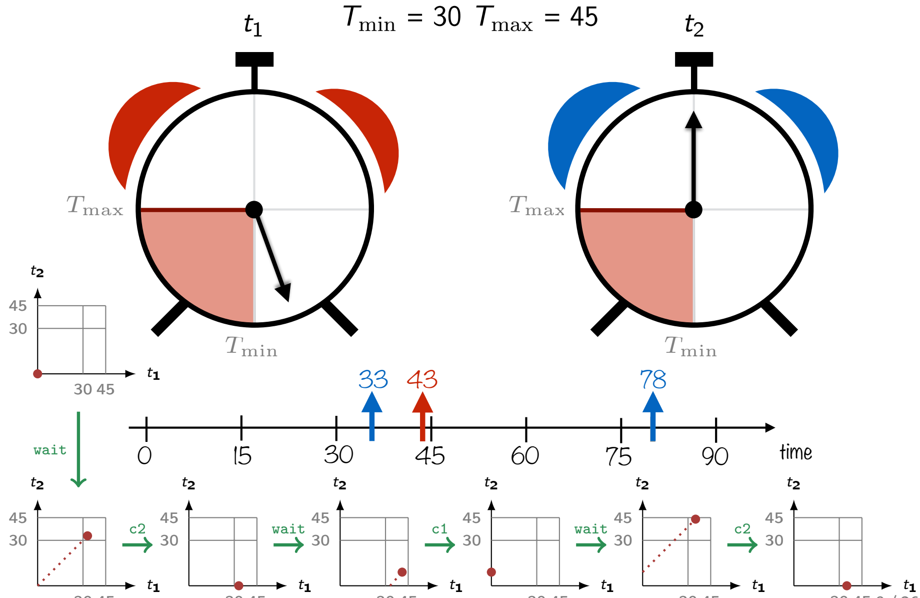
Concrete Simulation Trace



Concrete Simulation Trace

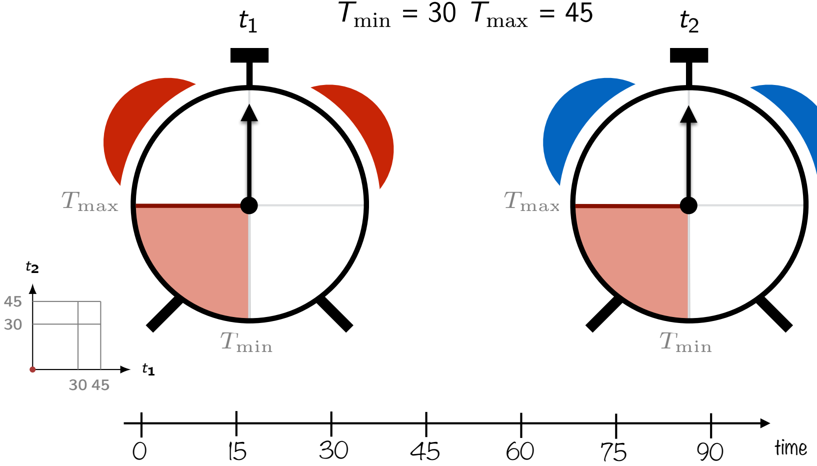


Concrete Simulation Trace

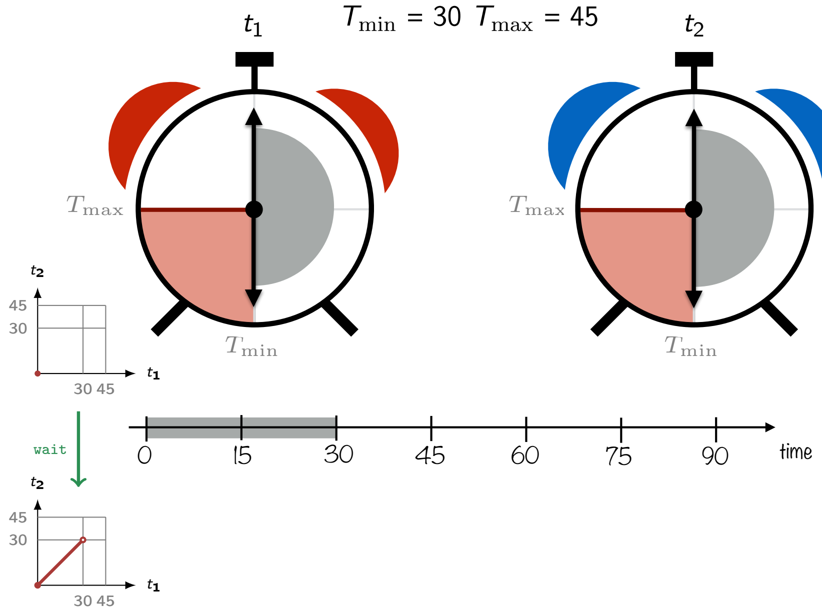


Symbolic Simulation Trace

$T_{\min} = 30$ $T_{\max} = 45$

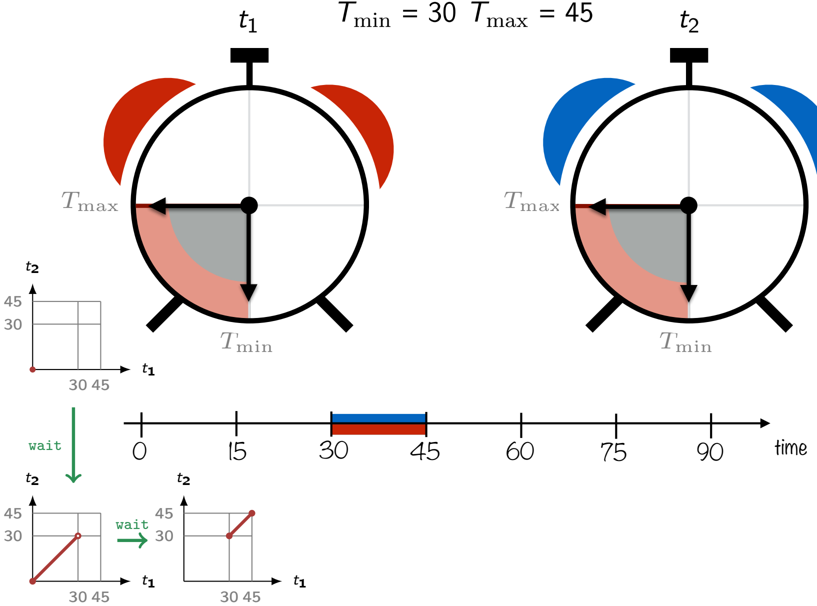


Symbolic Simulation Trace

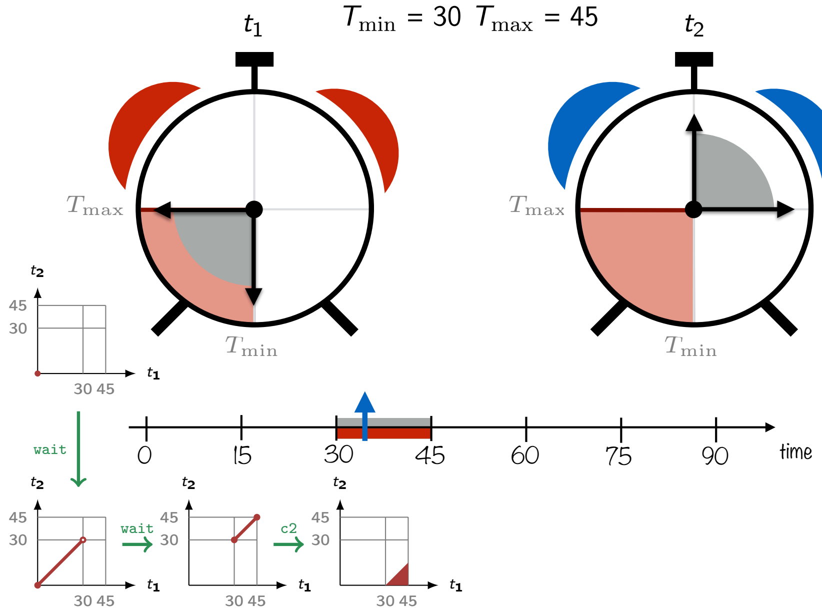


Symbolic Simulation Trace

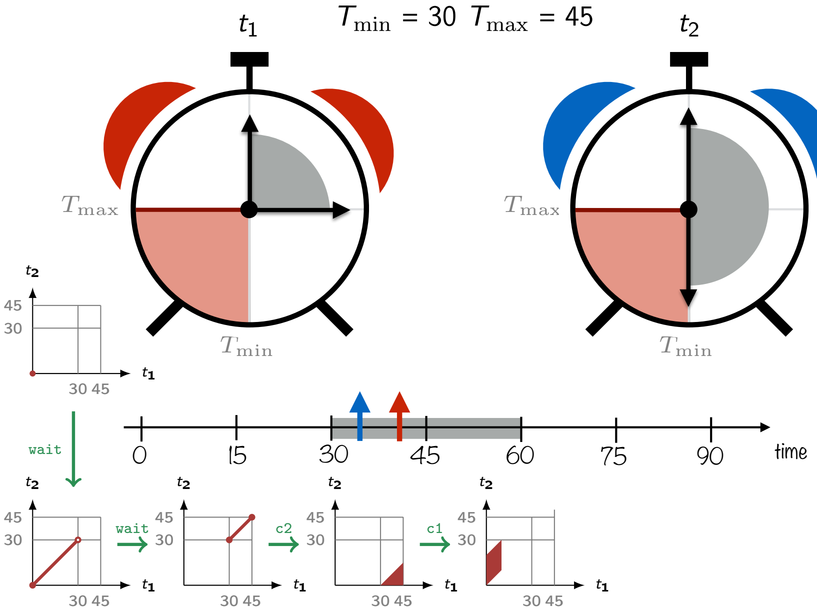
$T_{\min} = 30 \;\; T_{\max} = 45$



Symbolic Simulation Trace

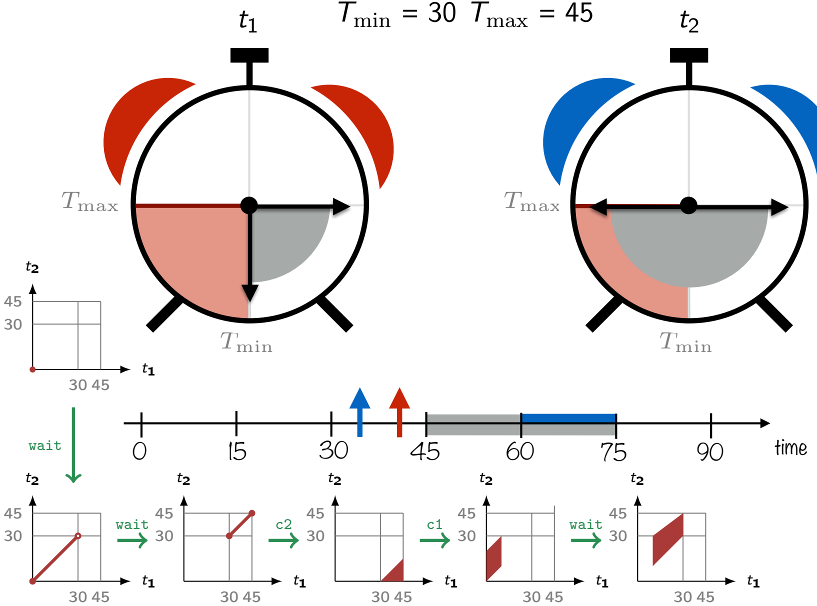


Symbolic Simulation Trace

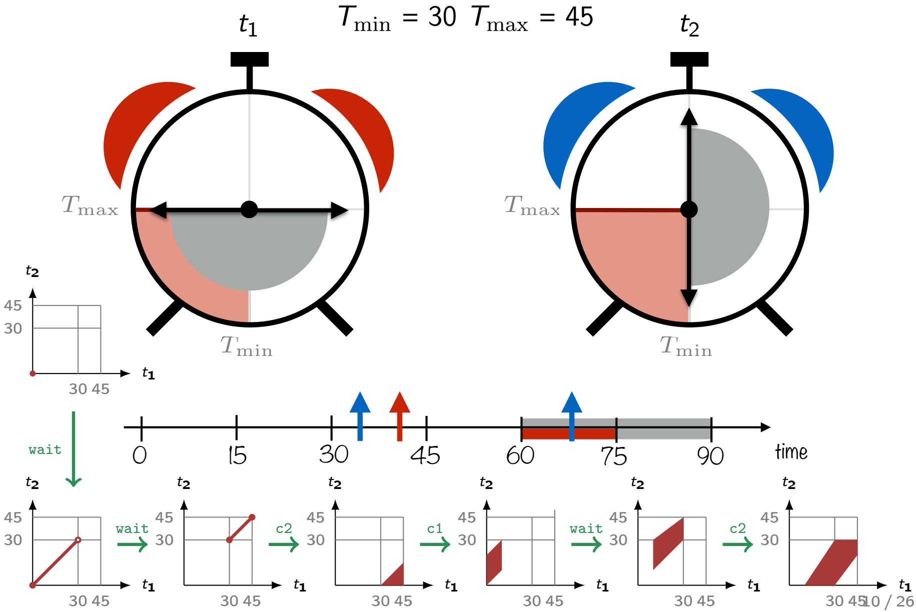


Symbolic Simulation Trace

$T_{\min} = 30 \;\; T_{\max} = 45$



Symbolic Simulation Trace



Difference Bound Matrices [Dill (1990): "Timing assumptions and verification of finite-state concurrent systems"]

Set of constraints

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{array}\right\}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{cccc} (0, \leq) & (0, \leq) & (-6, \leq) & (-5, <) \\ (20, <) & (0, \leq) & (8, \leq) & (\infty, <) \\ (\infty, <) & (-4, \leq) & (0, \leq) & (\infty, <) \\ (12, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right] \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Difference Bound Matrices [Dill (1990): "Timing assumptions and verification of finite-state concurrent systems"]

Set of constraints

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \le t_2 \\ 5 < t_3 \le 12 \\ 4 \le t_1 - t_2 \le 8 \end{array}\right\}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0,\le) & (0,\le) & (-6,\le) & (-5,<) \\ (20,<) & (0,\le) & (8,\le) & (\infty,<) \\ (\infty,<) & (-4,\le) & (0,\le) & (\infty,<) \\ (12,\le) & (\infty,<) & (\infty,<) & (0,\le) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \le\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Difference Bound Matrices [Dill (1990): "Timing assumptions and verification of finite-state concurrent systems"]

Set of constraints

$$\begin{Bmatrix} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{Bmatrix}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0,\leq) & (0,\leq) & (-6,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ (\infty,<) & (-4,\leq) & (0,\leq) & (\infty,<) \\ (12,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Difference Bound Matrices [Dill (1990): "Timing assumptions and verification of finite-state concurrent systems"]

Set of constraints

$$\left\{\begin{array}{c} t_1 < 20 \\ 6 \le t_2 \\ 5 < t_3 \le 12 \\ 4 \le t_1 - t_2 \le 8 \end{array}\right\}$$

The corresponding DBM

$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} (0, \le) & (0, \le) & (-6, \le) & (-5, <) \\ (20, <) & (0, \le) & (8, \le) & (\infty, <) \\ (\infty, <) & (-4, \le) & (0, \le) & (\infty, <) \\ (12, \le) & (\infty, <) & (\infty, <) & (0, \le) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \le\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).

Difference Bound Matrices [Dill (1990): "Timing assumptions and verification of finite-state concurrent systems"]

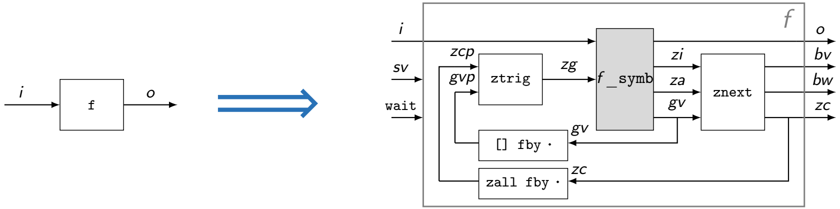
Set of constraints

$$\left\{ \begin{array}{c} t_1 < 20 \\ 6 \leq t_2 \\ 5 < t_3 \leq 12 \\ 4 \leq t_1 - t_2 \leq 8 \end{array} \right\}$$

The corresponding DBM

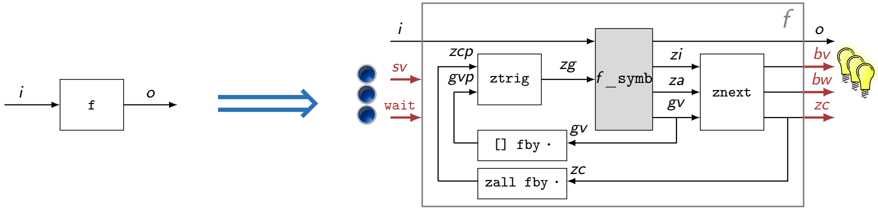
$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} (0,\leq) & (0,\leq) & (-6,\leq) & (-5,<) \\ (20,<) & (0,\leq) & (8,\leq) & (\infty,<) \\ (\infty,<) & (-4,\leq) & (0,\leq) & (\infty,<) \\ (12,\leq) & (\infty,<) & (\infty,<) & (0,\leq) \end{bmatrix} \end{array}$$

- Represents a set of possible clock values.
- Two-dimensional array of *difference constraints*: $t_i - t_j \preceq n$ where $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z} \cup \{\infty\}$.
- One dimension for each clock in the system.
  - row = upper bounds on differences with other clocks.
  - column = lower bounds on differences with other clocks.
- The t_0 clock is always equal to zero (for lower and upper bounds).



Source-to-source transformation of hybrid nodes into discrete ones.

- Replace timers, guards, and invariants.
- Use a small library of *Difference Bound Matrices* (DBMs).



New inputs

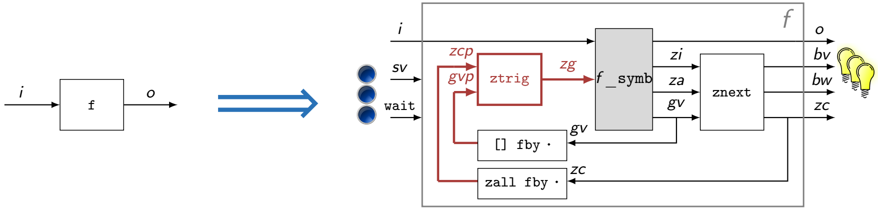
Add ‘buttons’ that push choice (non-determinism) outside the program.

- *sv*: (boolean vector) specifies guards to fire.
- `wait`: (boolean) specifies a wait transition.

New outputs

Add ‘light bulbs’ that show which buttons are valid.

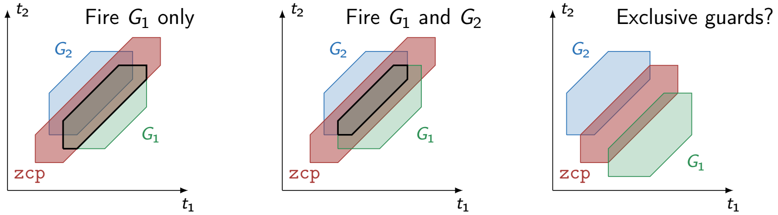
- *bv*: (boolean vector) indicates enabled guards.
- *bw*: (boolean) indicates that wait is possible.
- *zc*: the current symbolic zone.

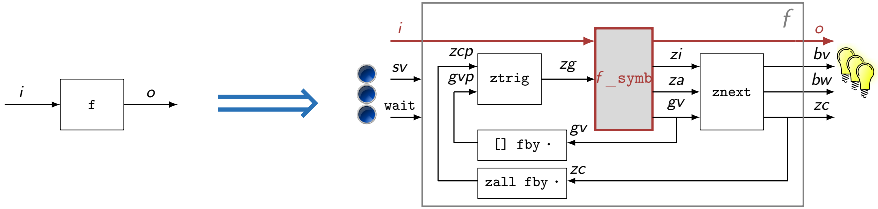


Compute trigger zone of fired guards.

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

- Filter enabled guard zones according to user inputs.
- Intersect them with the previous symbolic state.





Source-to-source transformation

Defined as 5 mutually recursive functions over syntax.

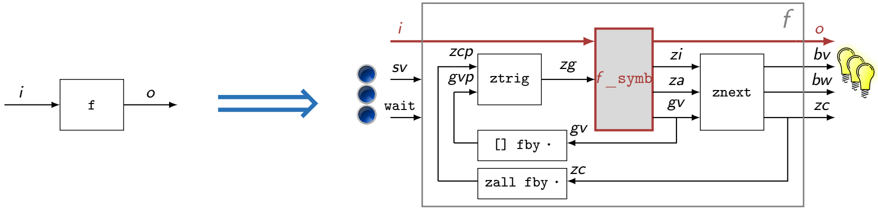
TraDef(d) translates declarations. Only continuous-time declarations introduced by `hybrid` are modified.

Tra(zi, e) translates expressions using a variable *zi* to pass the currently computed version of the initial zone.

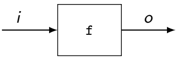
TraEq(zi, E) translates equations.

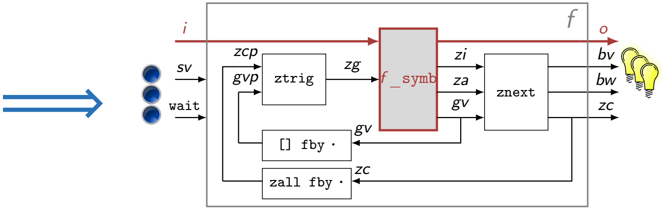
TraZ(zi, c) translates constraints.

TraH(zi, h) translates handlers.

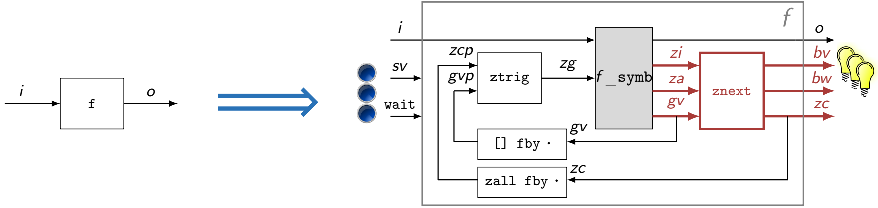



```
let hybrid clock(t_min, t_max) = c where
  rec timer t init 0.0 reset c() → 0.0
  and emit c when {t ≥ t_min}
  and always {t ≤ t_max}


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


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
```

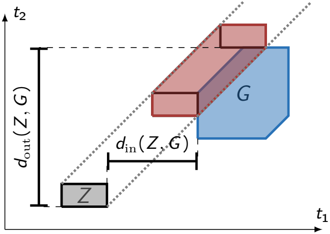


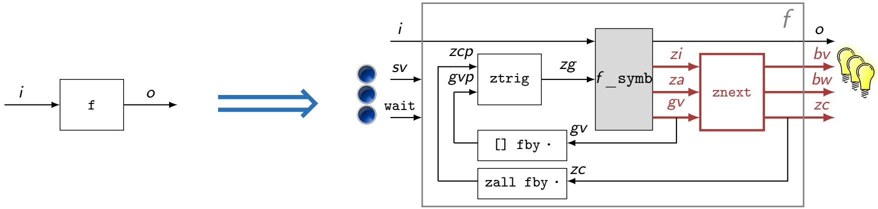
Compute *next* symbolic state and enabled transitions

- Take initial zone zi, invariant conjunction za, and guard zone vector gv.
- Compute the symbolic state and the transition ‘lights’.

```
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(dl, 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)
```

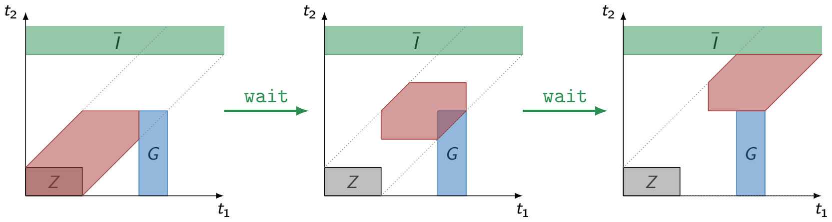
dzero = $(0, \leq)$

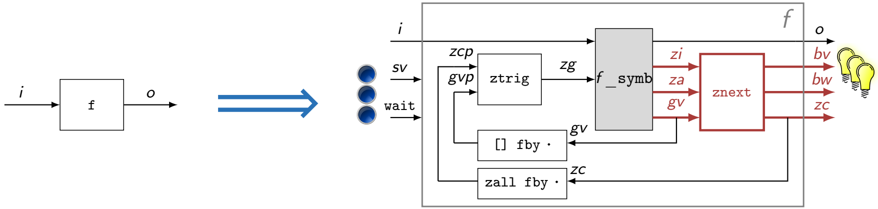




Compute *next* symbolic state and enabled transitions

- Take initial zone `zi`, invariant conjunction `za`, and guard zone vector `gv`.
- Compute the symbolic state and the transition 'lights'.

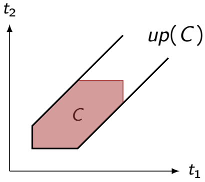


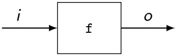


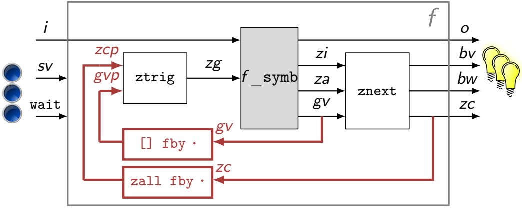
Compute *next* symbolic state and enabled transitions

- Take initial zone `zi`, invariant conjunction `za`, and guard zone vector `gv`.
- Compute the symbolic state and the transition ‘lights’.

```
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(dl, 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)
```

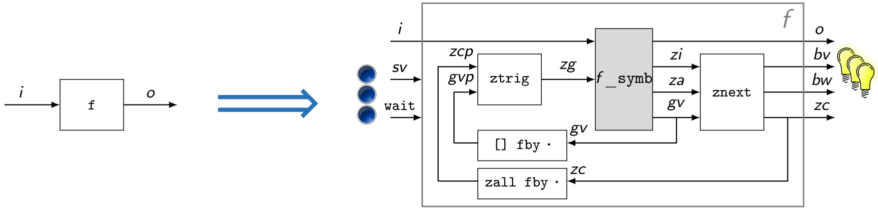






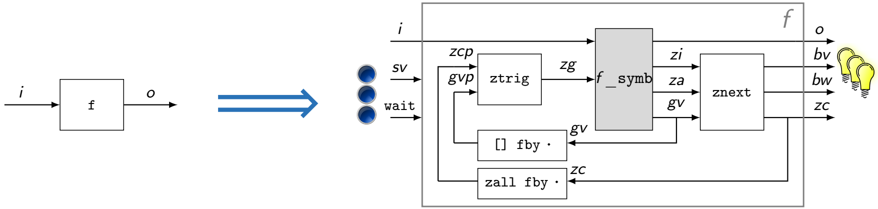
Feedback is key to the scheme's simplicity

- Avoid multiple passes by calculating in one cycle and using in the next.
- Remember the next active guard zones.
- Remember the next active symbolic state.



Express compositions and delays in discrete subset of language

```
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(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
```



Summary of 3 execution phases

1. From current zone *zcp* and vector of guard activation zones *gvp* (from previous step), `ztrig` computes the trigger zone *zg*.

2. *f* _`symb` triggers discrete-time computations and returns *zi* obtained by applying resets to *zg*, the conjunction of active invariants *za*, and the new vector of guard zones *gv*.

3. `znext` computes the new zone *zc* by letting time elapse from *zi* until the set of enabled guards changes.

DBM interface

Prototype implemented in OCaml.

- `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`.
- `zup(z)` Lets time elapse indefinitely from zone `z` (drops 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 $\texttt{gv}_i$ 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`.

Comparison

Uppaal

- First-rate graphical interface and simulator.
- Verification by model-checking.
- Highly-optimized DBM library.
- Single-level of parallel composition of instantiated templates.
- C-like language for combinatorial functions.
- Sophisticated semantics implemented inside tool.

Zsy

- Hierarchical parallel compositions.
- Lustre-like language for stateful functions.
- Semantics encoded by source-to-source transformation.

Conclusion

Contributions

- A novel Lustre-like language with Timed Automaton features.
- Source-to-source compilation schema for symbolic simulations.
- Novel 'sweeping' construct for explicit `wait` transitions
- Prototype implementation: https://github.com/gbdrt/zsy/tree/fdl17

Future directions

- Generate C and link with Uppaal DBM library?
- Incorporate richer domains? [Miné (2006): "The octagon abstract domain"]
- Implement support for state machines? [Baudart (2017): "A Synchronous Approach to Quasi-Periodic Systems"]
- Verification by symbolic model-checking? [Hagen and Tinelli (2008): "Scaling up the formal verification of Lustre programs with SMT-based techniques"] [Isenberg and Wehrheim (2014): "Timed Automata Verification via IC3 with Zones"].

References I

Alur, R. and D. L. Dill (1994). "A Theory of Timed Automata". In: 126.2, pp. 183–235.

Baudart, G. (2017). "A Synchronous Approach to Quasi-Periodic Systems". PhD thesis. PSL Research University.

Behrmann, G., A. David, and K. G. Larsen (2006). *A tutorial on Uppaal 4.0.*

Bourke, T. and M. Pouzet (2013). "Zélus: A Synchronous Language with ODEs". In: USA, pp. 113–118.

Caspi, P. (2000). *The Quasi-Synchronous Approach to Distributed Control Systems*. Tech. rep. CMA/009931. *The Cooking Book*. VERIMAG, Crysis Project.

Caspi, P., D. Pilaud, N. Halbwachs, and J. Plaice (1987). "Lustre: A Declarative Language for Programming Synchronous Systems". In: Germany, pp. 178–188.

Dill, D. L. (1990). "Timing assumptions and verification of finite-state concurrent systems". In: France, pp. 197–212.

Hagen, G. and C. Tinelli (2008). "Scaling up the formal verification of Lustre programs with SMT-based techniques". In: USA, pp. 109–117.

References II

Henzinger, T. A., X. Nicollin, J. Sifakis, and S. Yovine (1994). "Symbolic Model Checking for Real-Time Systems". In: *Information and Computation* 111.2, pp. 192–244.

Isenberg, T. and H. Wehrheim (2014). "Timed Automata Verification via IC3 with Zones". In: vol. 8829, pp. 203–218.

Miné, A. (2006). "The octagon abstract domain". In: 19.1, pp. 31–100.

Vaandrager, F. and A. de Groot (2006). "Analysis of a Biphase Mark Protocol with Uppaal and PVS". In: *Formal Aspects of Computing* 18.4, pp. 433–458.