Real-Time Implementation of BIP: Clocks and Real-Time Constraints

Jacques Combaz

DCS Days – March 27, 2009

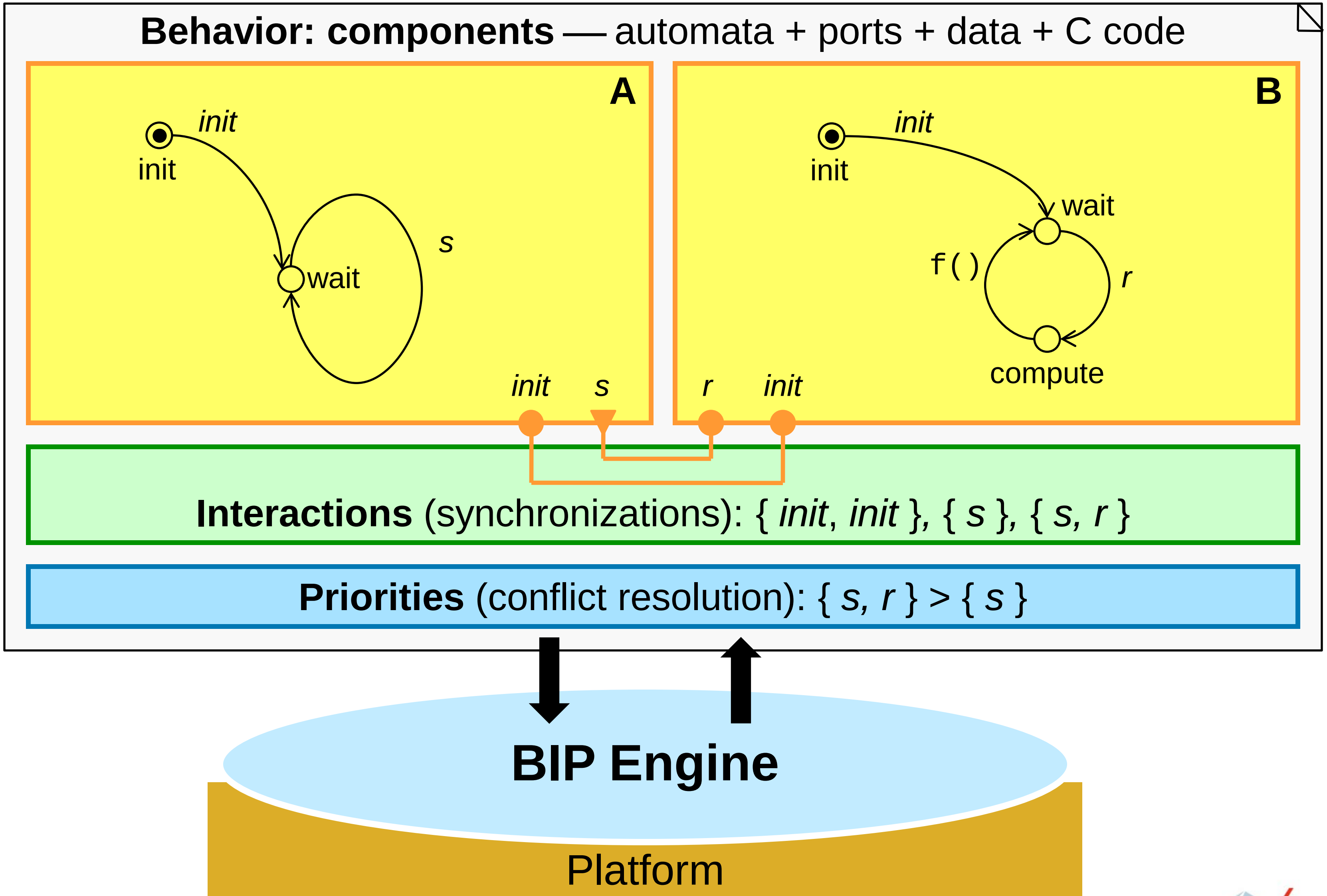
Outline

1. Introduction: (Timed) BIP Model
2. Computing Timed Interactions
3. Model Time vs Real-Time
4. Real-Time Scheduling Policy
5. Future Work

Outline

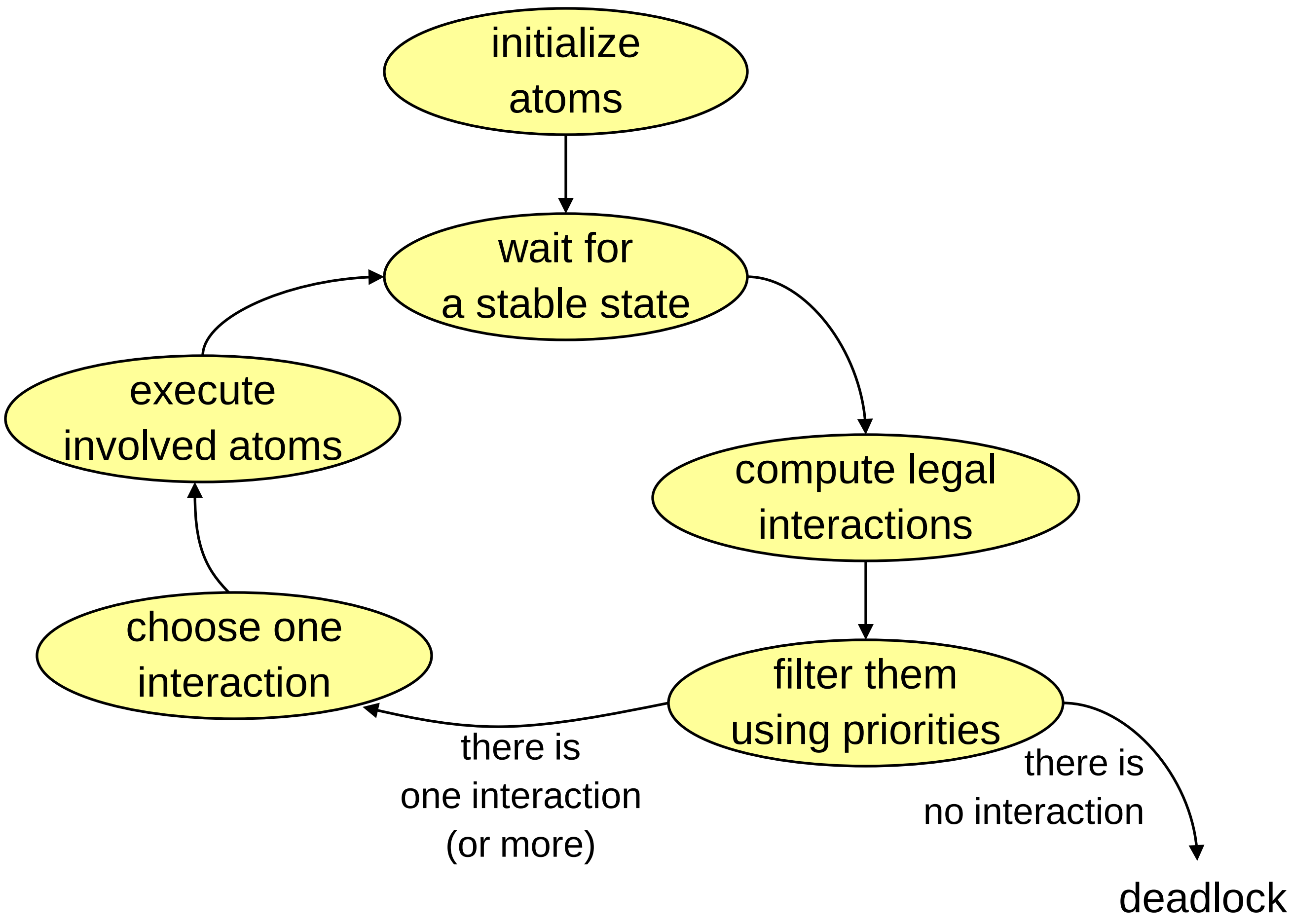
1. **Introduction: (Timed) BIP Model**
2. Computing Timed Interactions
3. Model Time vs Real-Time
4. Real-Time Scheduling Policy
5. Future Work

BIP Model



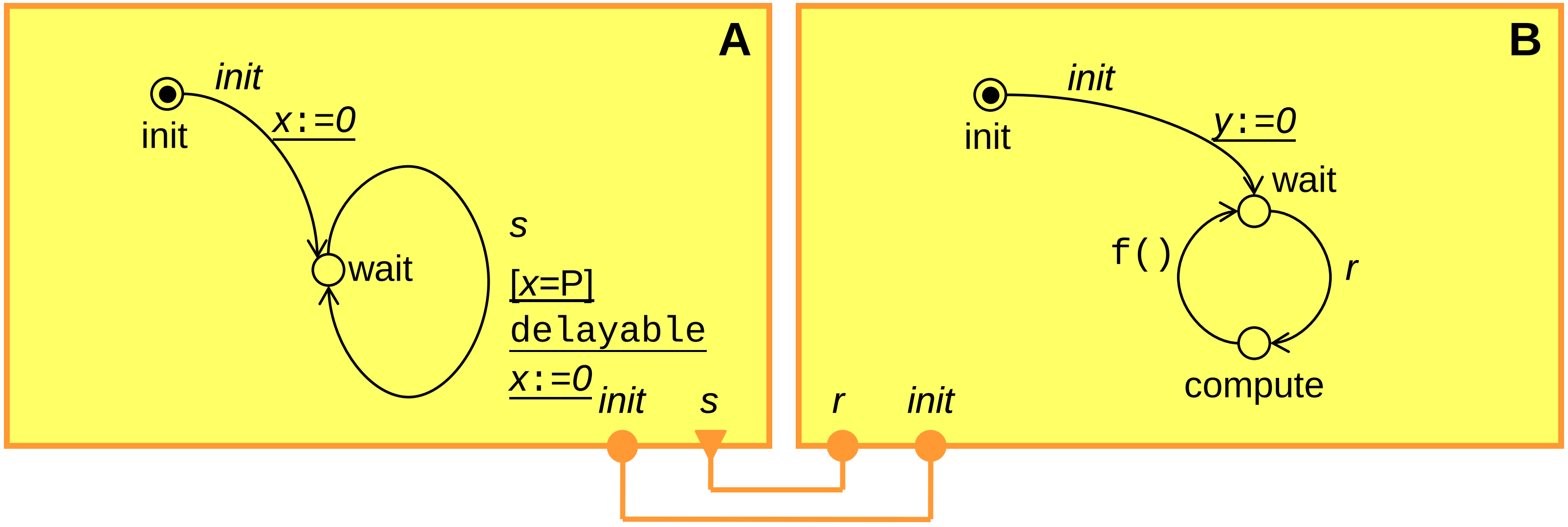
Behavior: components — automata + ports + data + C code
A
init
init
wait
s
init
s
B
init
init
wait
f()
r
compute
r
init
Interactions (synchronizations): { *init*, *init* }, { *s* }, { *s*, *r* }
Priorities (conflict resolution): { *s*, *r* } > { *s* }
BIP Engine
Platform

Centralized Implementation of the BIP Engine



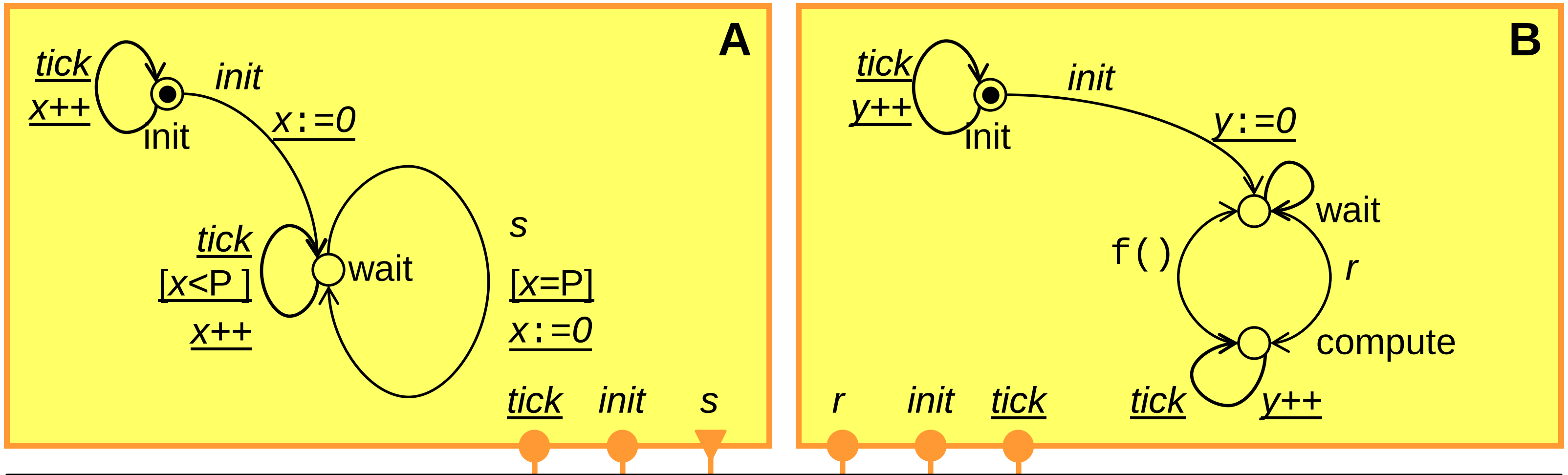
Timed BIP Model

Behavior: timed automata (clocks with discrete semantics + urgency type)



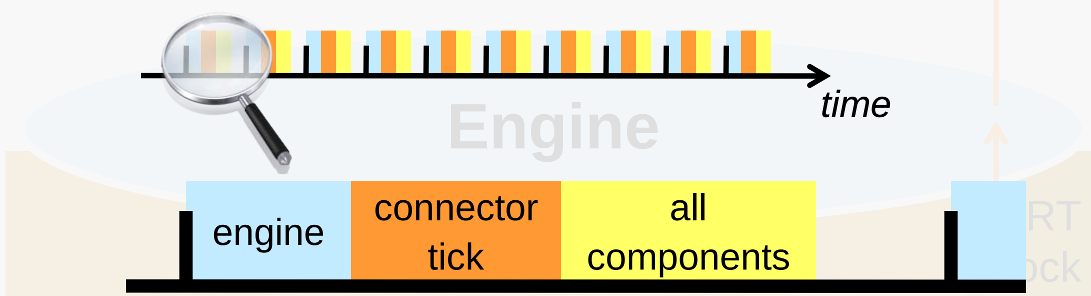
Implementing with Tick Connector

Connector tick is synchronized with the platform clock



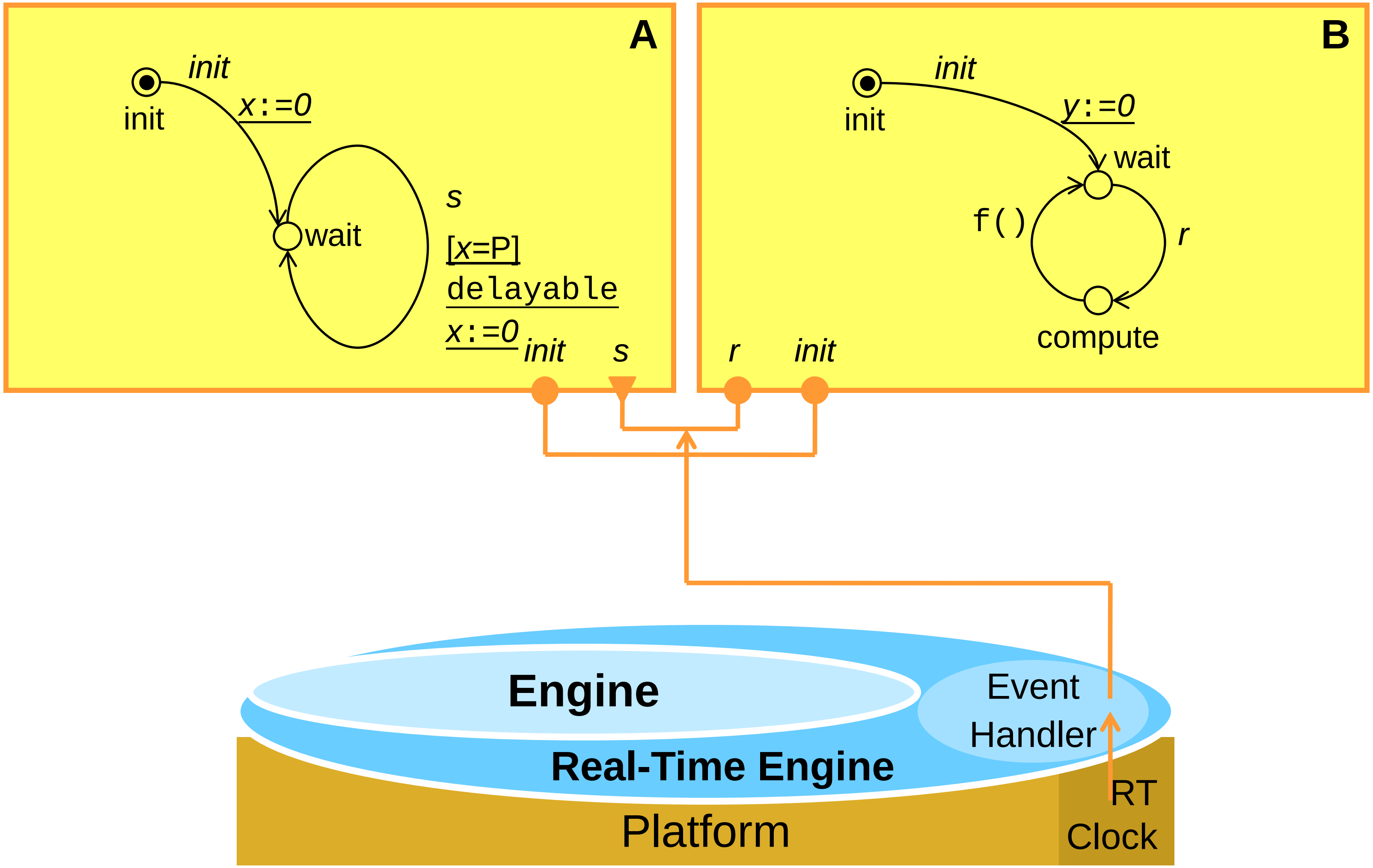
Problem:

- inefficient: synchronous execution
- not implementable if execution times ≥ clock period

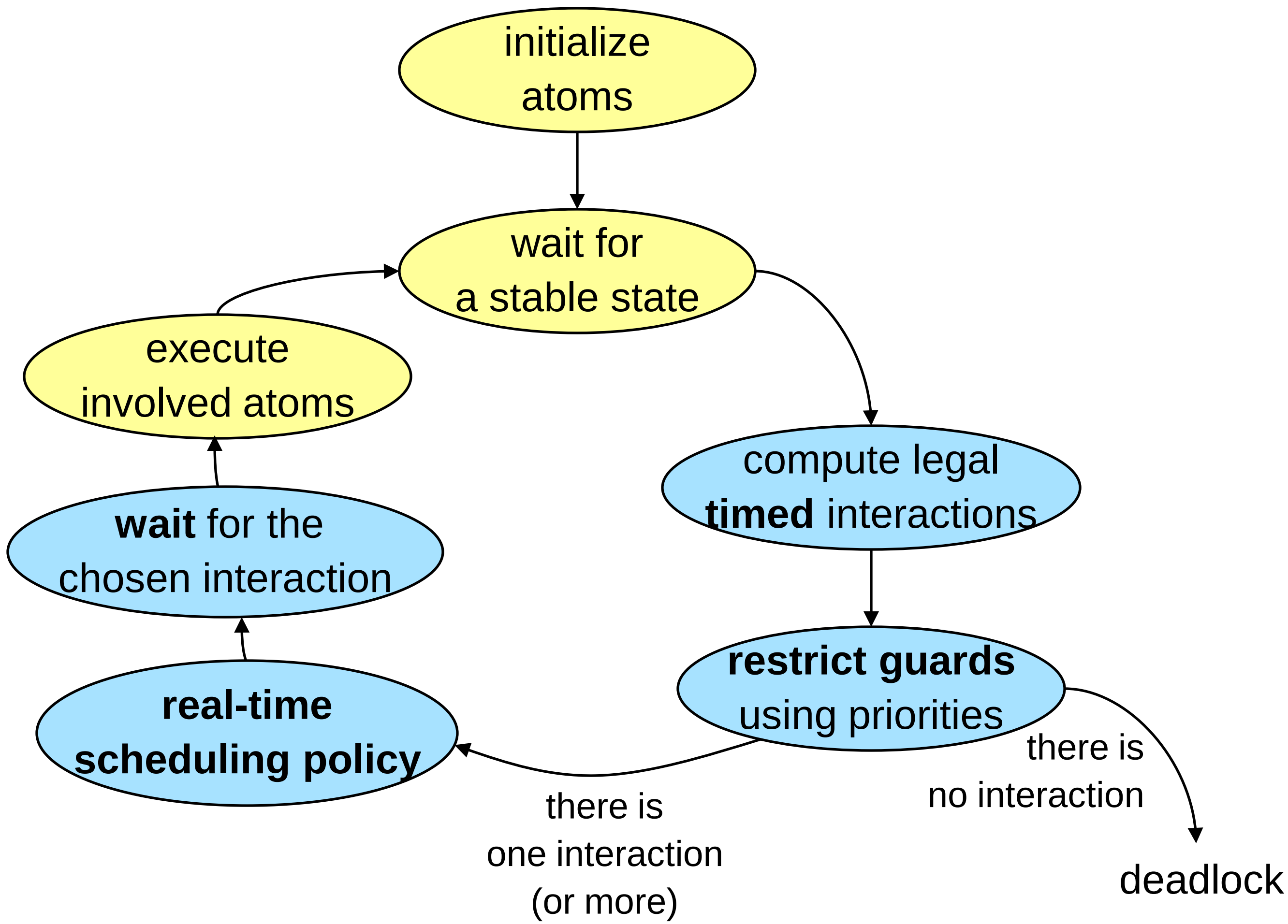


Proposed Engine

Behavior: timed automata (urgency) + ports + data + C code



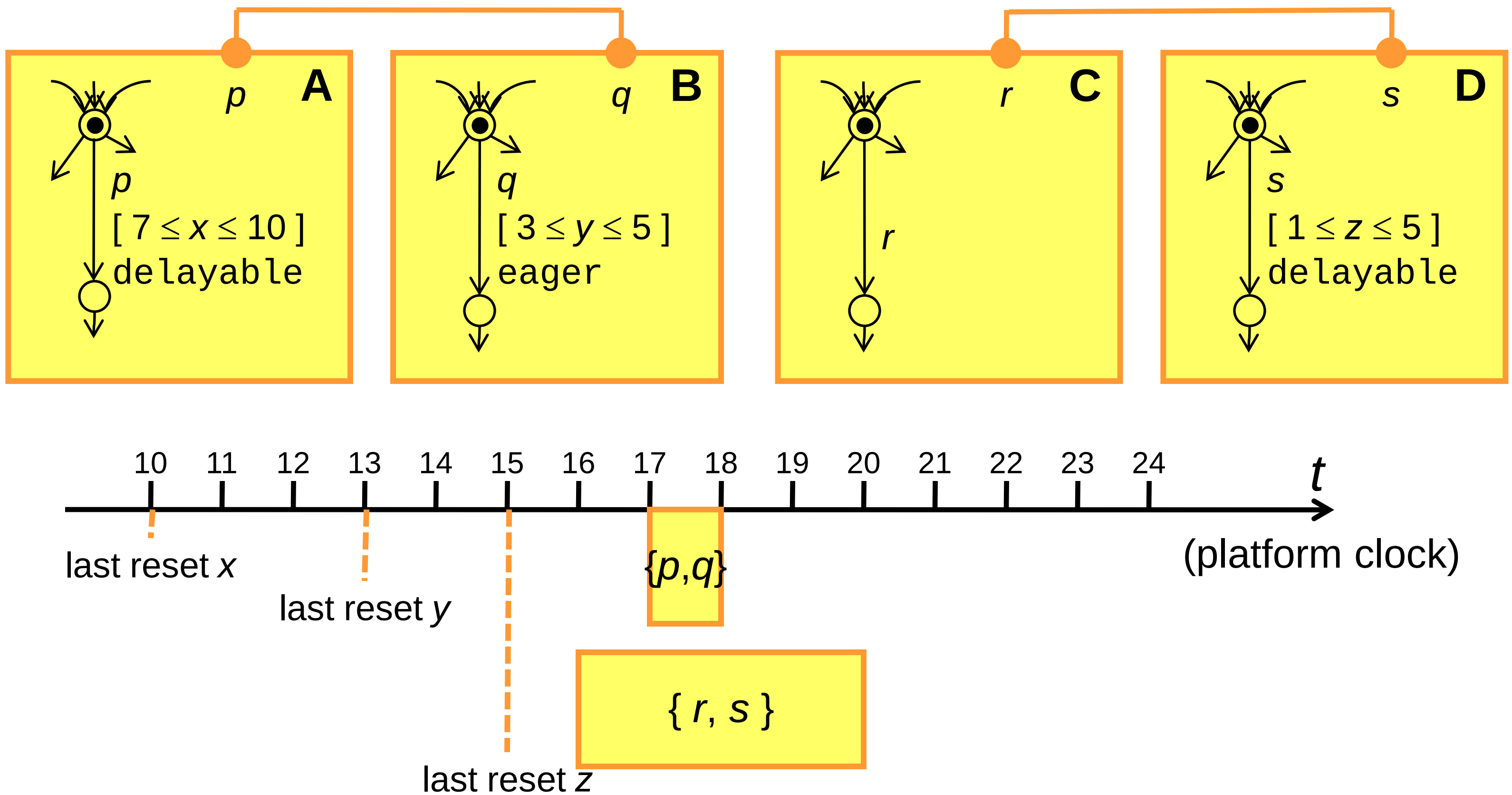
Real-Time BIP Engine (Centralized)



Outline

1. Introduction: (Timed) BIP Model
2. **Computing Timed Interactions**
3. Model Time vs Real-Time
4. Real-Time Scheduling Policy
5. Future Work

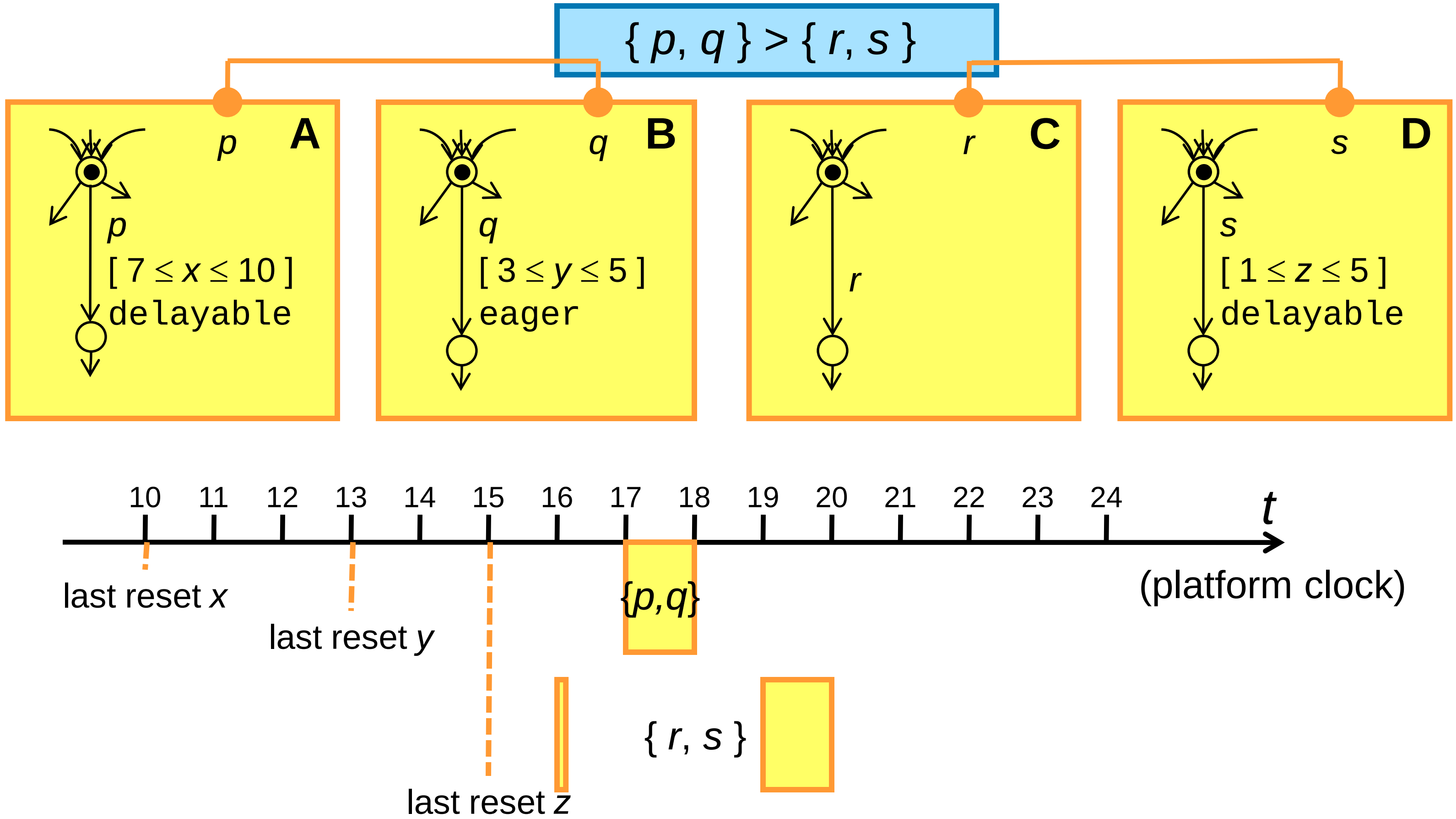
Computing Timed Interactions



$\{ p, q \}$: $[\, 17 \le t \le 20 \,]^{\mathtt{delayable}} \cap [\, 16 \le t \le 18 \,]^{\mathtt{eager}} = [\, 17 \le t \le 18 \,]^{\mathtt{eager}}$

$\{ r, s \}$: $[\, -\infty \le t \le +\infty \,]^{\mathtt{lazy}} \cap [\, 16 \le t \le 20 \,]^{\mathtt{delayable}} = [\, 16 \le t \le 20 \,]^{\mathtt{delayable}}$

Priorities



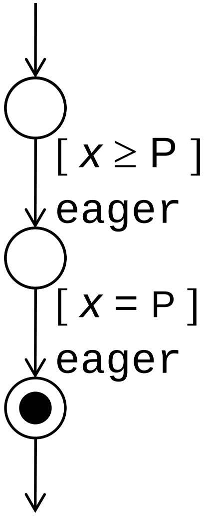
$\{ r, s \}$: $\quad [\, 16 \leq t \leq 20 \,]^{\mathtt{delayable}} \setminus [\, 17 \leq t \leq 18 \,]^{\mathtt{eager}}$

$= [\, t = 16 \,]^{\mathtt{lazy}} \cup [\, 19 \leq t \leq 20 \,]^{\mathtt{delayable}}$

Outline

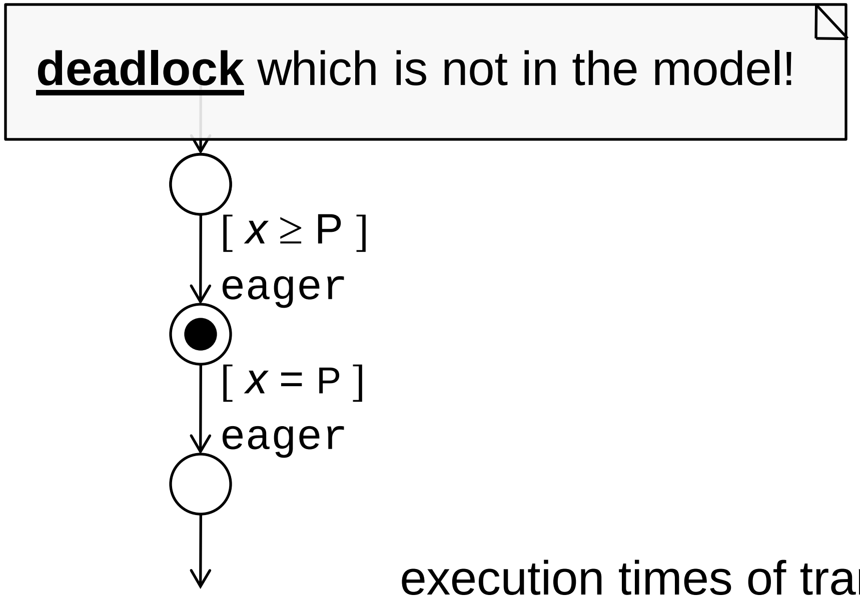
1. Introduction: (Timed) BIP Model
2. Computing Timed Interactions
3. **Model Time vs Real-Time**
4. Real-Time Scheduling Policy
5. Future Work

Example #1 (Model Execution)



x = P

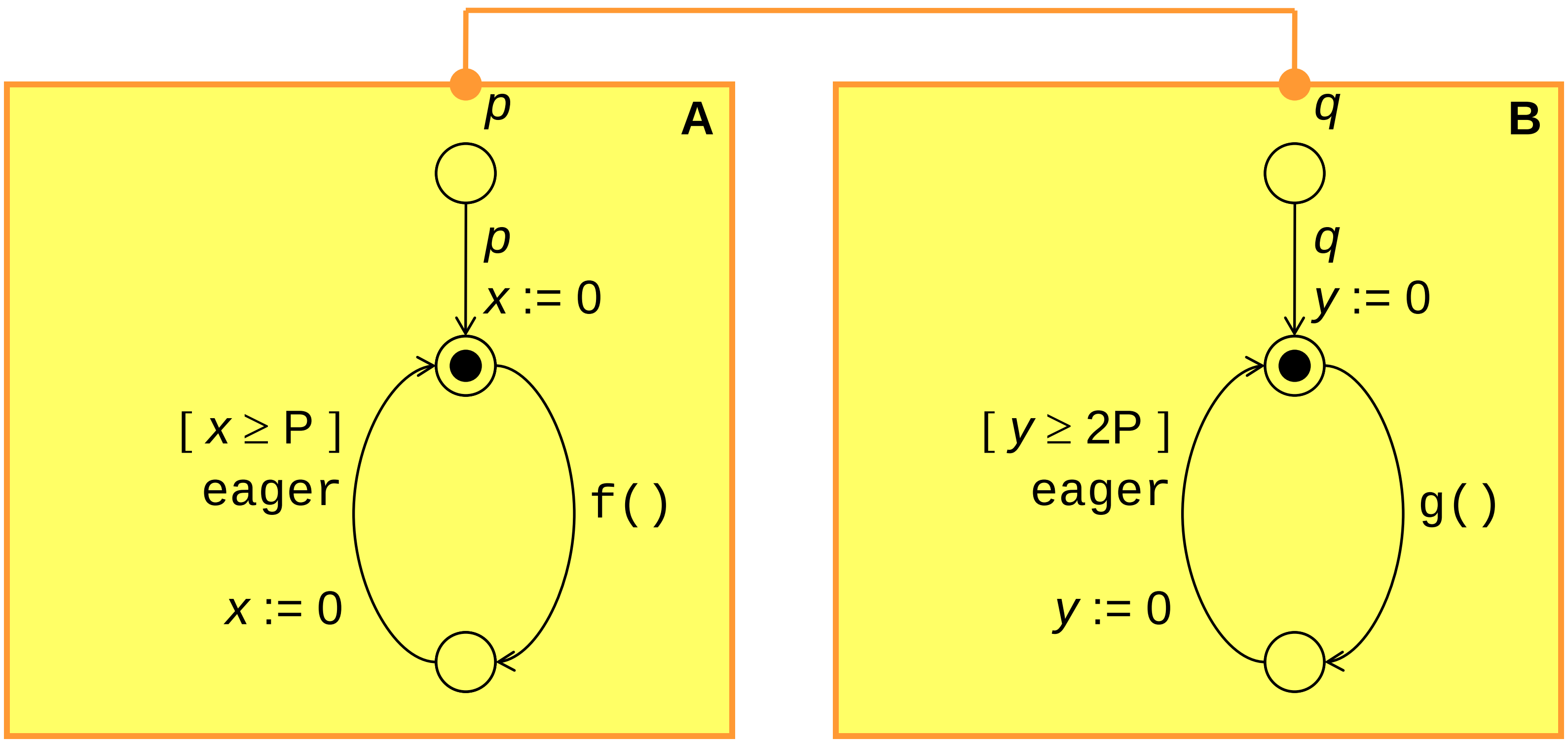
Example #1 (Actual Execution)



x = P+ε

execution times of transitions: ε

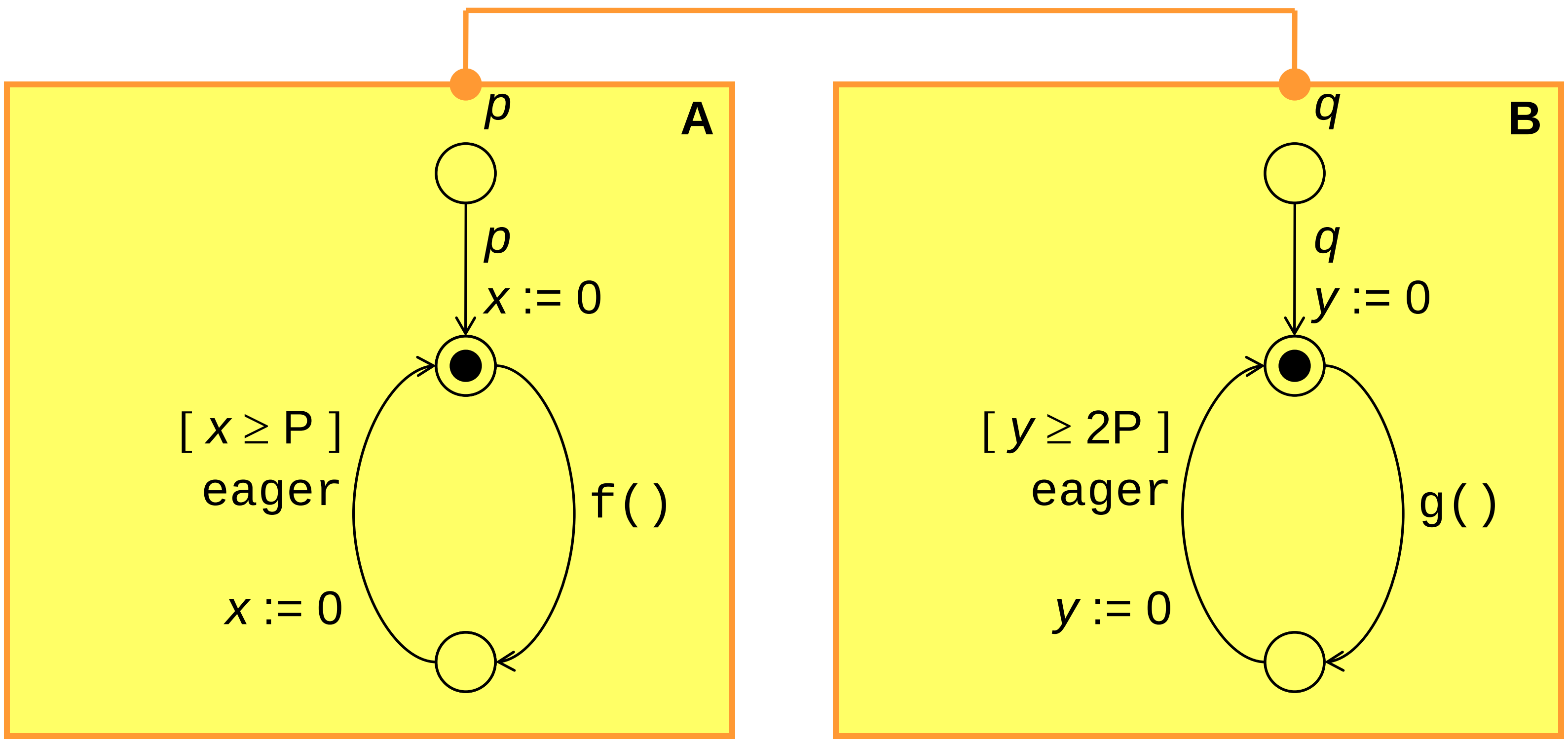
Example #2 (Model Execution)



$x = 0$

$y = 0$

Example #2 (Actual Execution)



$x = 0$
$y = 2\varepsilon$

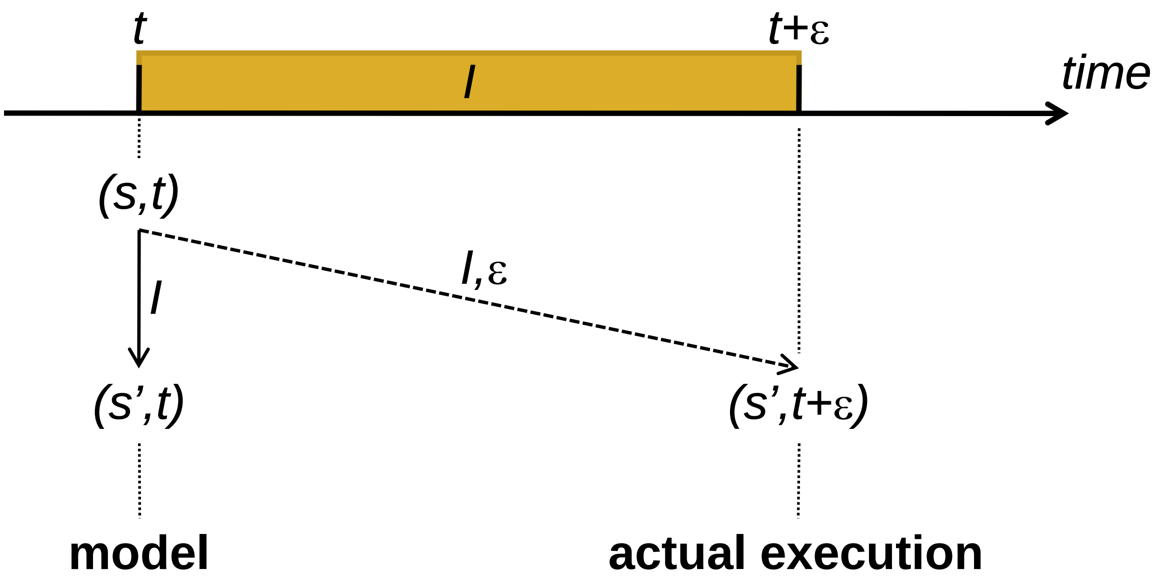
<u>execution times</u> of transitions:

f()	ε_f
g()	ε_g
others	ε

Model Time vs Real-Time

- In **model semantics**, interaction execution is **<u>instantaneous</u>**
- In **actual implementation**, everything **<u>takes time</u>**
- *Model* time and *real-time* cannot coincide at each state of the system

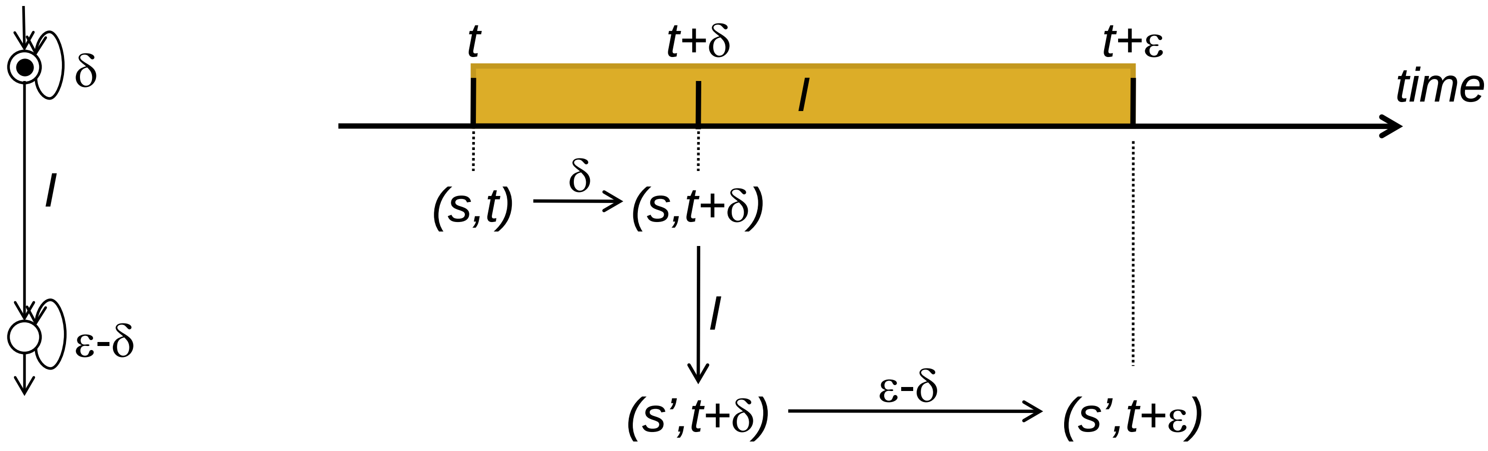
s,*s'*: states
I: interaction



- The Real-Time BIP Engine should be based on "**logical"** or "**model" time**
- **Synchronizations** between model time and real-time are required

Exact Synchronization

- Interaction *I* can be executed for any value of time between *t* and *t*+ε
- Synchronization at each control state

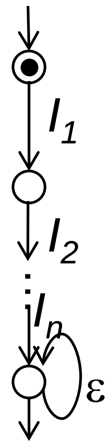


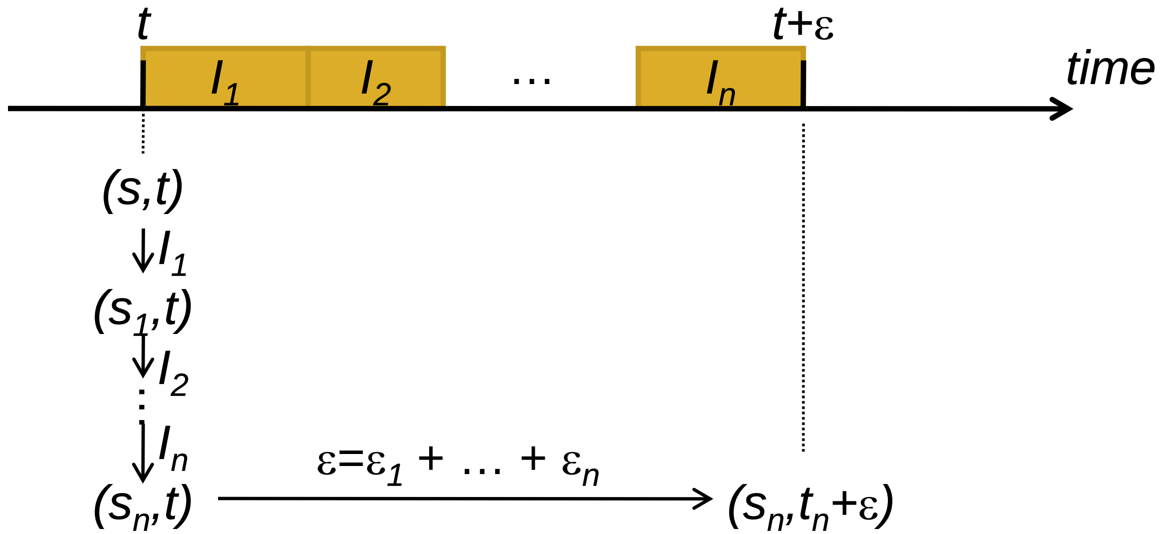
- Actual execution: $(s,t) \rightarrow^{I,\varepsilon} (s',t+\varepsilon)$

- Model: $\forall \delta \in [0;\varepsilon]\ \ (s,t) \rightarrow^{\delta} (s,t+\delta) \rightarrow^{I} (s',t+\delta) \rightarrow^{\varepsilon\text{-}\delta} (s',t+\varepsilon)$

Relaxed Synchronization

- Synchronization can be made after a finite sequence $I_1 \ldots I_n$ of interactions that have to be executed at model time t



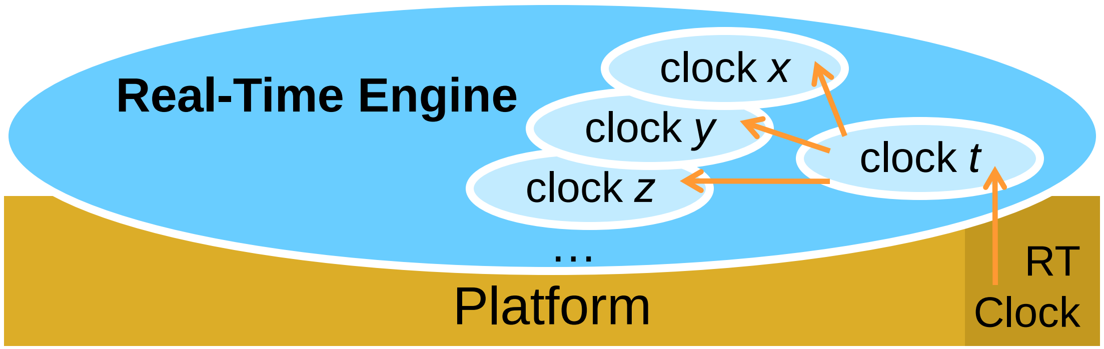


- Actual execution: $(s,t) \rightarrow^{I_1,\varepsilon_1} (s_1,t+\varepsilon_1) \rightarrow^{I_2,\varepsilon_2} \ldots \rightarrow^{I_n,\varepsilon_n} (s_n,t+\varepsilon_n)$

- Model: $(s,t) \rightarrow^{I_1} (s_1,t) \rightarrow^{I_2} \ldots \rightarrow^{I_n} (s_n,t) \rightarrow^{\varepsilon} (s_n,t + \varepsilon_1+\ldots+\varepsilon_n)$

Implementing Clock Synchronization

- Clock *t* encapsulates the platform clock: it represents *logical* or *model* time
- Clock *t* and platform clock are synchronized only when necessary, depending on the synchronization model (*exact* or *relaxed*)
- User clocks *x*, *y*, *z*, … are computed w.r.t. model time *t*



Implementability

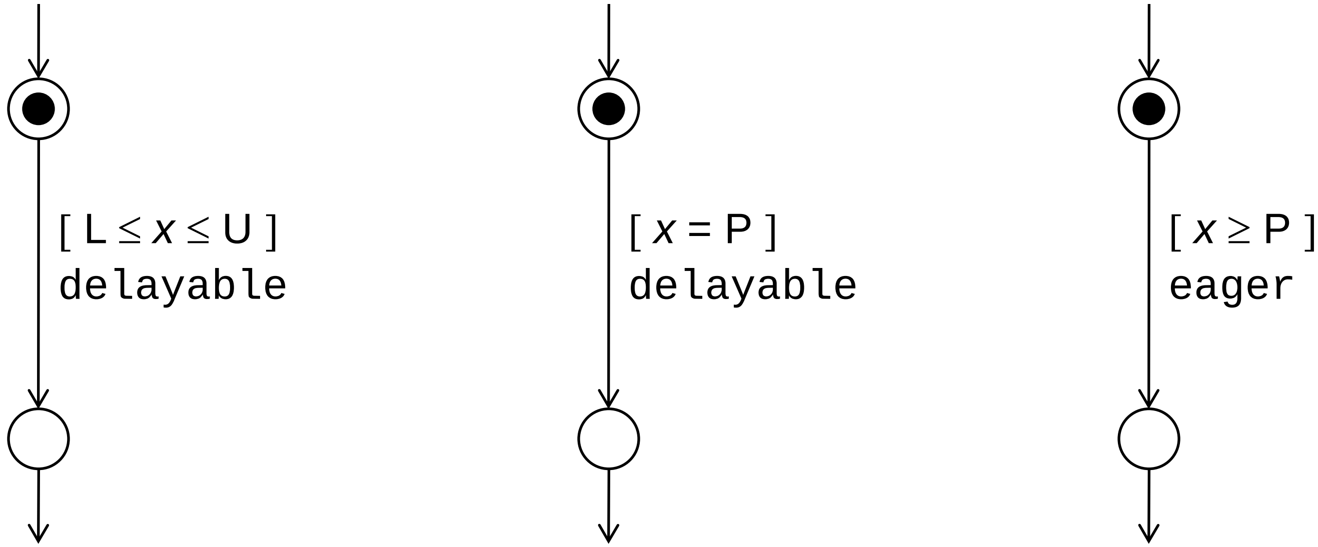
- **Problem:** Given a platform, the *synchronized model* (exact or relaxed) may give traces (sequences of transitions) that are not in the model.

  1. Using a **faster** processor solves the problem.
     $\rightarrow$ **OK**

  2. The model requires an **infinite** processor speed for executing correctly.
     $\rightarrow$ **not implementable** with the considered semantics

- Formally, the model is implementable if there exists execution times $\varepsilon_i > 0$ for transitions such that the set of traces of the synchronized model are included in the set of traces of the model.

Implementability (Examples 1/3)



implementability:

exact ☑
relaxed ☑

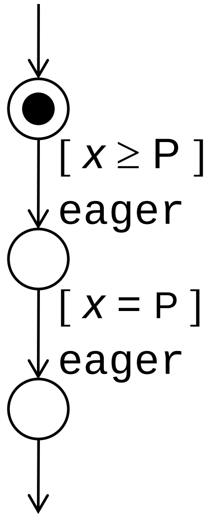
implementability:

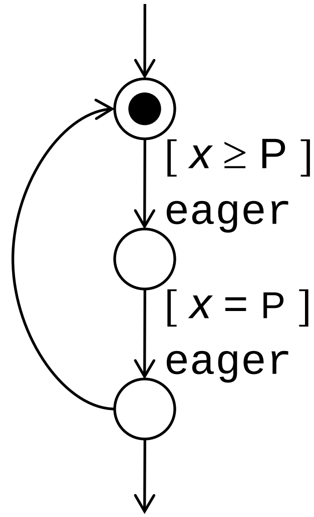
exact ☒
relaxed ☑

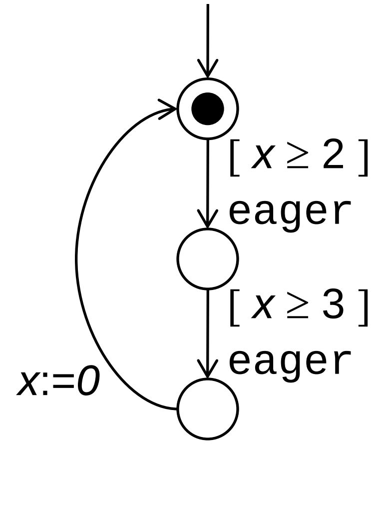
implementability:

exact ☒
relaxed ☑

Implementability (Examples 2/3)







implementability:

exact ☒
relaxed ☑

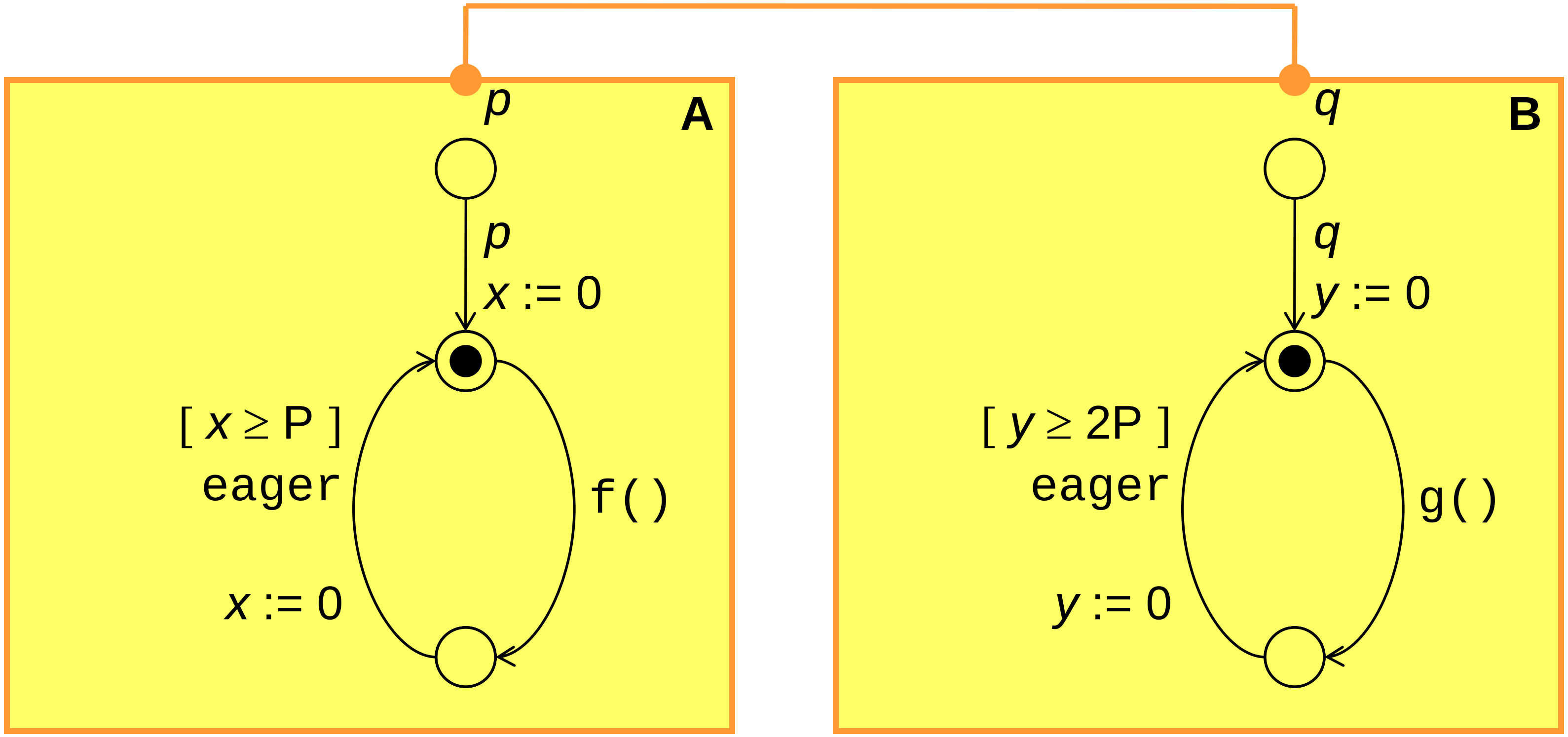
implementability:

exact ☒
relaxed ☒

implementability:

exact ☒
relaxed ☑

Implementability (Examples 3/3)



$x = 2\varepsilon$
$y = 2\varepsilon$

<u>implementability:</u>

exact ☒

relaxed ☑

<u>execution times</u> of transitions:

`f()`	ε_f
`g()`	ε_g
others	ε

Outline

1. Introduction: (Timed) BIP Model
2. Computing Timed Interactions
3. Model Time vs Real-Time
4. **Real-Time Scheduling Policy**
5. Future Work

Computing Deadlines

- deadline(*C*,*T*) : deadline associated to constraint *C*, if the current value of the involved clock is *T*

- deadline($[L \leq t \leq U]^{\mathtt{lazy}}$,*T*) = $+\infty$

- deadline($[L \leq t \leq U]^{\mathtt{delayable}}$,*T*) = **max** $[L;U] \cap [T;+\infty]$ (with **max** $\emptyset = +\infty$)

- deadline($[L \leq t \leq U]^{\mathtt{eager}}$,*T*) = **min** $[L;U] \cap [T;+\infty]$ (with **min** $\emptyset = +\infty$)

- deadline($[L_1 \leq t \leq U_1]^{\mathtt{u1}} \cup \ldots \cup [L_N \leq t \leq U_N]^{\mathtt{uN}}$,*T*) = **min** deadline($[L_i \leq t \leq U_i]^{\mathtt{ui}}$,*T*)

Computing Next Activation

- next(C,T) : next value of the involved clock for which C is enabled, if the current value of the clock is T
- $\text{next}([L \leq t \leq U]^{\texttt{urgency}}, T) = \textbf{min}\ [L;U] \cap [T;+\infty]$ (with $\textbf{min}\ \emptyset = +\infty$)
- $\text{next}([L_1 \leq t \leq U_1]^{\texttt{u1}} \cup \ldots \cup [L_N \leq t \leq U_N]^{\texttt{uN}}, T) = \textbf{min}\ \text{next}([L_i \leq t \leq U_i]^{\texttt{ui}}, T)$

Relaxed Sync. Implementation

```
Engine()
  clk := 0                                             /* reset platform clock */
  T := 0                                               /* logical time := 0 */

  infinite_loop
    Legals := GetLegalInteractions()                   /* interactions and constraints */
    if (Legals = ∅) break                              /* deadlock */
    Legals' := ApplyPriorities(Rules, Legals)          /* restrict constraints with priorities*/

    I := EDF_Scheduler(Legals', T)                     /* real-time scheduler */
    D := deadline(I, T)                                /* deadline for I */

    if ( D > T || clk - T > MAX_DRIFT )
      T := clk                                         /* synchronize T and clk */
      if ( D > T ) break                               /* deadline is missed */

    wait( t ≥ next(I, T ) )                            /* wait for next activation of I */
    T := next ( I, T))                                 /* update logical time  if needed */

    execute(I)                                         /* execute I */
```

Implementing EDF Scheduling Policy

EDF_Scheduler(Legals', T)

D := $\mathbf{min}_{I \in \mathrm{Legals}}$ deadline(I, T)

return I **such that** deadline(I, T) = D

EDF_Δ_Scheduler(Legals', T)

D := $\mathbf{min}_{I \in \mathrm{Legals}}$ deadline(I, T)

N := $\mathbf{min}_{I \in \mathrm{Legals}}$ next(I, T)

if ($D - N > \Delta$) */* enough time to execute non-urgent interactions */*

  return I **such that** next(I, T) = N

else

  return I **such that** deadline(I, T) = D

5. Future Work

- **BIP Toolchain:**

  - a prototype of a (centralized) Real-Time BIP Engine has been done

  $\rightarrow$ optimization of the Real-Time BIP Engine

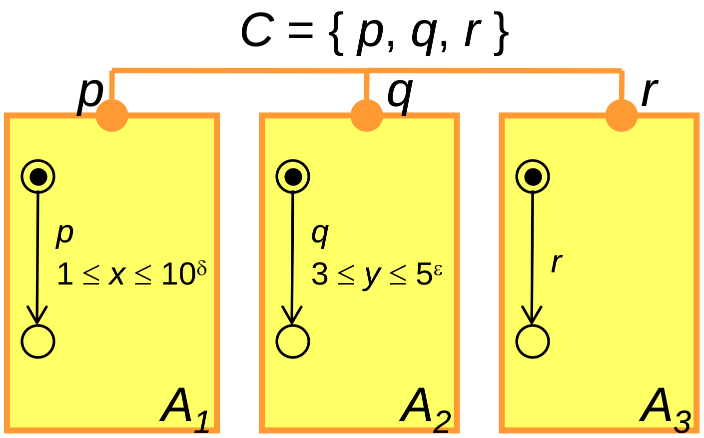
  $\rightarrow$ modification of the parser and the code generator

- **Distributed implementation:**

  $\rightarrow$ distributed clocks (synchronizations)

Backup

BIP Model



- Components $= \{ A_1, A_2, A_1 \}$
- Clocks $= \{ x, y \}$
- Ports $= \{ p, q, r \}$
- Connectors $\subseteq 2^{\text{Ports}}$ $= \{ C \}$

- Real-time: $t \in$ Reals
- Reset function: last_reset_date : Clocks $\rightarrow$ Reals
- x:=0 $\rightarrow$ last_reset_date[x] = t_0 where t_0 is the current value of t

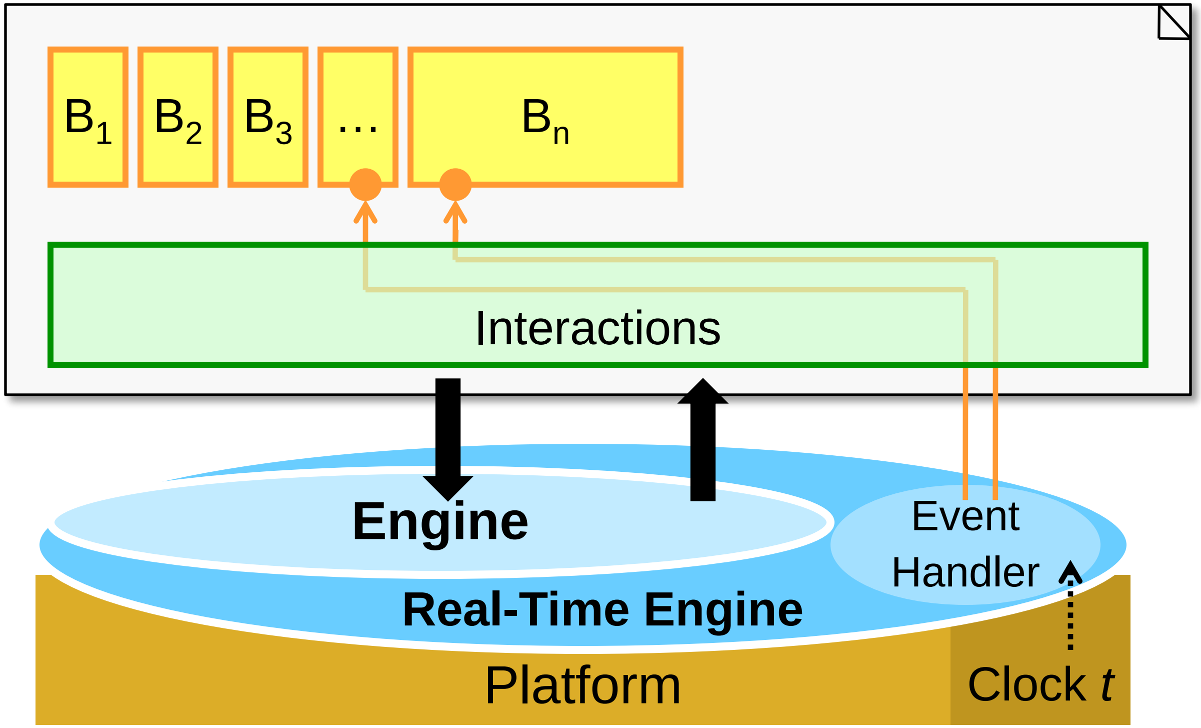
At a given system state :

- Synced $= \{ (p,x,[1;10],\delta),$
  $(q,y,[3;5],\varepsilon),$
  $(r,-,-,-) \}$

ε: eager
δ: delayable
λ: lazy

$\varepsilon > \delta > \lambda$

Clocks and Real-Time Constraints



p

```
on p
provided
delayable
x in [L;U]
```

Real-Time *t* **Engine** enabled test

p

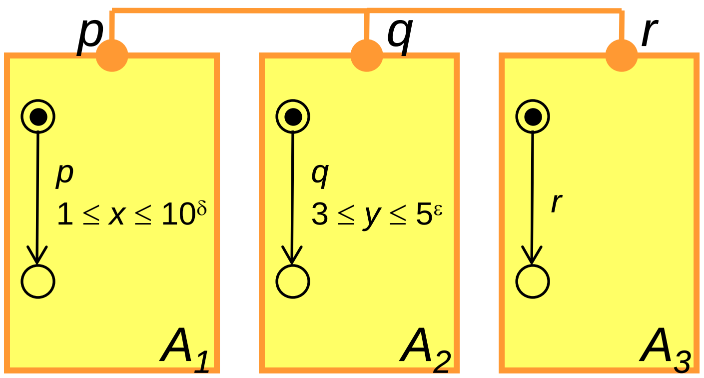
```
on p
provided
delayable
x in [L;U]
```

$L' = L + \text{last\_reset\_date}[x]$

$U' = U + \text{last\_reset\_date}[x]$

$L' \leq t \leq U'$

Timed BIP Model



- Components = { A_1, A_2, A_1 }
- Clocks = { x, y }
- Ports = { p, q, r }
- Connectors $\subseteq 2^{\text{Ports}}$ = { C }

- Real-time: $t \in$ Reals
- Reset function: last_reset_date : Clocks $\rightarrow$ Reals
- x:=0 $\rightarrow$ last_reset_date[x] = t_0 where t_0 is the current value of t

At a given system state :

- Synced = { (p,x,[1;10],δ),
  (q,y,[3;5],ε),
  (r,-,-,-) }

ε: eager
δ: delayable
λ: lazy

$\varepsilon > \delta > \lambda$

Time Conversion

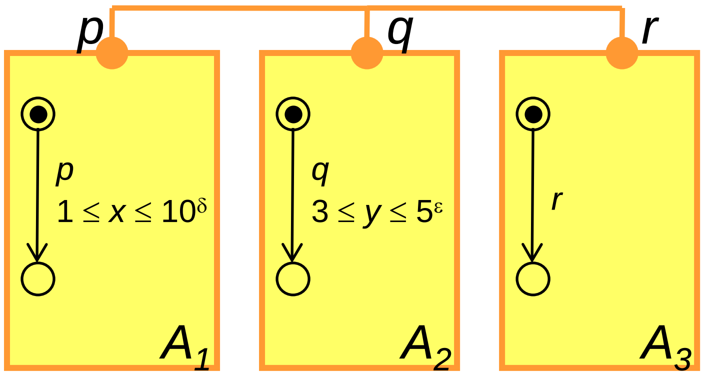
- <u>Platform clock:</u> t

  $(p,\text{-},\text{-},\text{-}) \rightarrow (p,t,[-\infty;+\infty],\lambda)$

  $(p,x,[L;U],\tau) \rightarrow (p,t,[L';U'],\tau)$ where $L' = L + \text{last\_reset\_date}[x]$

  $U' = U + \text{last\_reset\_date}[x]$

$C = \{ p, q, r \}$



- Synced $= \{ (p,x,[1;10],\delta), (q,y,[3;5],\varepsilon), (r,\text{-},\text{-},\text{-}) \}$

  Synced $= \{ (p,t,[1;10],\delta), (q,t,[4;6],\varepsilon), (r,\text{-},\text{-},\text{-}) \}$

  if last_reset_date[x]=0 and last_reset_date[y]=1

- We write

$p \in$ Synced	for $(p,t,[L;U],\tau) \in$ Synced
$p \in \text{Synced}_{\text{untimed}}$	for $(p,t,[-\infty;+\infty],\lambda) \in$ Synced
guard(p)	for $[L;U]$ if $(p,t,[L;U],\tau) \in$ Synced
guard(p)	for $\emptyset$ if $p \notin$ Synced
$\tau(p)$	τ if for $(p,t,[L;U],\tau) \in$ Synced

Computing Legal Interactions
1. Strong Synchronization

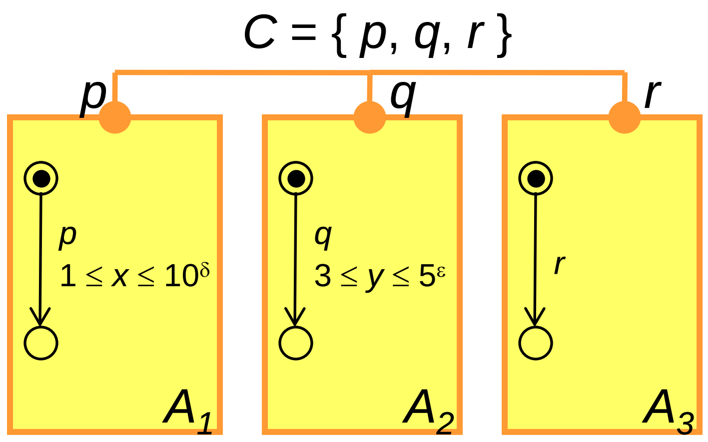
- Let $C \in$ Connectors such that $C = \{ p_1,\ldots,p_n \}$ is a strong synchronization, i.e. C defines the set of interactions $\{ \{ p_1,\ldots,p_n \} \}$

- For $I = \{ p_1,\ldots,p_n \}$ we have $(I,t,\mathrm{guard}(I),\tau) \in$ Legals iff:
  1. $I \subseteq$ Synced
  2. $\mathrm{guard}(I) = [t_0;+\infty] \cap \bigcap_{p\in I} \mathrm{guard}(p)$
  3. $\mathrm{guard}(I) \neq \emptyset$
  4. $\tau = \max_{p\in I} \tau(p)$

- We have $(I,t,\mathrm{guard}(I),\max \tau_j) \in$ Legals iff:
  1. $I \subseteq$ Synced
  2. $\mathrm{guard}(I) = [L;U]$ where $(p_j,t,[L_j;U_j],\tau_j) \in$ Synced and $L = \max_j L_j, t_0$

     $U = \min_j U_j$
  3. $L \leq U$

Computing Legal Interactions
1. Strong Synchronization (example)



If Synced = { $(p,t,[1;10],\delta)$, $(q,t,[4;6],\varepsilon)$, $(r,t,[-\infty;+\infty],\lambda)$ } and $t_0 = 5$

then Legals = { $(\{ p,q,r \},t,[5;6],\varepsilon)$ }

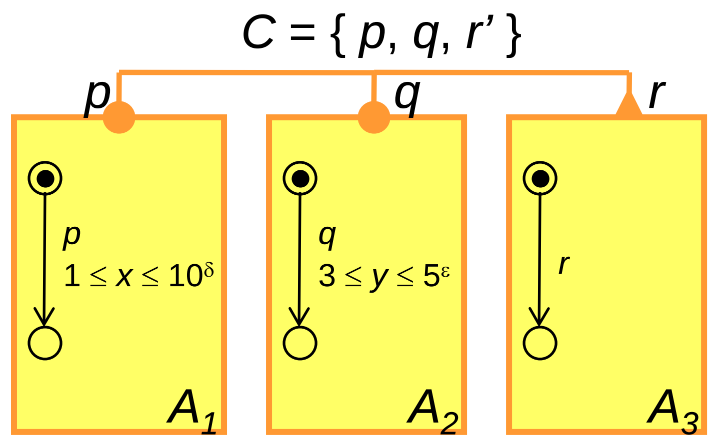
Computing Legal Interactions
2. General Case

- Let $C \in$ Connectors such that $C = \{ p_1,\ldots,p_n \}$
  C defines a set of interactions $\{ I_1,\ldots,I_m \}$

- For each $I \in \{ I_1,\ldots,I_m \}$ we have $(I, t, \text{guard}(I), \tau) \in$ Legals iff:
  1. $\text{Synced}_{\text{untimed}} \subseteq I \subseteq \text{Synced}$
  2. $\text{guard}(I) = [t_0;+\infty] \cap \bigcap_{p \in I} \text{guard}(p) \setminus (\bigcup_{p \in C \setminus I} \text{guard}(p))$
  3. $\text{guard}(I) \neq \emptyset$
  4. $\tau = \max_{p \in I} \tau(p)$

- Notice that $\text{guard}(I)$ will be of the form $\text{guard}(I) = [L_1;U_1] \cup \ldots \cup [L_N;U_N]$
  (we consider the discrete semantics for \)

Computing Legal Interactions
2. General Case (example)



- Synced = { (p,t,[1;10],δ), (q,t,[9;11],ε), (r,t, [-∞;+∞],λ) }
  (last_reset_date[x]=0, last_reset_date [y]=6)

- C defines the set of interactions $\{ I_1, I_2, I_3, I_4 \}$ such that, for $t_0 = 1$, we have:

$I_1 = \{ r \}$	guard(I_1) = $[t_0;+\infty] \setminus ([1;10] \cup [9;11])$	$(I_1, t, [12;+\infty], \lambda) \in$ Legals
$I_2 = \{ r,p \}$	guard(I_2) = $[t_0;+\infty] \cap [1;10] \setminus [9;11]$	$(I_2, t, [1;8], \delta) \in$ Legals
$I_3 = \{ r,q \}$	guard(I_3) = $[t_0;+\infty] \cap [9;11] \setminus [1;10]$	$(I_3, t, [11;11], \varepsilon) \in$ Legals
$I_4 = \{ r,p,q \}$	guard(I_4) = $[t_0;+\infty] \cap [1;10] \cap [9;11]$	$(I_4, t, [9;10], \varepsilon) \in$ Legals

Computing Priorities

- Applying priorities: Legals $\rightarrow$ Legals'

- Let I and $I_1, I_2, \ldots, I_n$ such that $I < I_i$ provided $L_i \leq t \leq U_i$
  then $(I, t, \text{guard}'(I), \tau) \in$ Legals' iff :
  1. $I \in$ Legals and for all i=1..n $I_i \in$ Legals
  2. $\text{guard}'(I) = \text{guard}(I) \setminus (\cup_{i=1..n} [L_i;U_i] \cap \text{guard}(I_i))$
  3. $\text{guard}'(I) \neq \emptyset$

Implementation (Clocks)

```
class GlobalClock : Clock
 + time(),reset()
 + wait()
 + freeze(),go(),update()
```

computed w.r.t. another clock or directly

Centralized Engine (Monothread)

```
Engine() {
  T = new Clock(t);                          /* T: logical time, t: real-time , both freezed */
  t.go();                                    /* start real-time */

  while(true) {
    Synced = GetSyncedPorts();               /* list of synced ports */
    Legals = GetLegalInteractions(Synced);   /* list of legal interactions */
    Legals' = ApplyPriorities(Legals);       /* priorities*/

    I = EDF_Scheduler(Legals', T.time());    /* real-time scheduler */
    if (I == NULL) break;                    /* deadlock */

    if (SyncPoint(I, T.time(), t.time())) {
      old_time = T.time(); T.update();       /* synchronize logical time */
      if (CheckDeadlineMiss(I, old_time , T.time())) break;
    }

    T.wait(next(I, T.time()));               /* wait for next activation of I */
    I.execute();                             /* execute I */
    if (CheckForDeadlineMiss(I, T.time(), t.time())) break;
  }
  DeadlockOrDeadlineMiss();
}
```

Centralized Engine (Monothread)

```
SyncPointExact(I, T, t) {
  return true;
}

CheckDeadlineMissExact(I, T, t) {
  if (deadline(I,T) < t) return true;
  else return false;
}



SyncPointRelaxed(I, T, t) {
  if (deadline(I,T) == T  && t - T < MAX_DRIFT) return false;
  else return true;
}

CheckDeadlineMissRelaxed(I, T, t) {
  if (T < deadline(I,T) < t) return true;
  else return false;
}
```

Implementation (Atom, Ports, ...)

- `class Atom` `class Compound`
  `+ rt_sync()` `+ rt_activate()`

- `class Port`
  `+ constraint` real-time constraint associated to the port by `rt_sync()`
  `+ rt_execute()`

- `class Connector`
  `+ mFeasibleInter` list of feasible interactions (depending on the real-time)
  `+ rt_execute()`

- `class Interaction`
  `+ constraint` associated real-time constraint
  `+ mNext` next interaction in the list
  `+ rt_execute()`

Planning

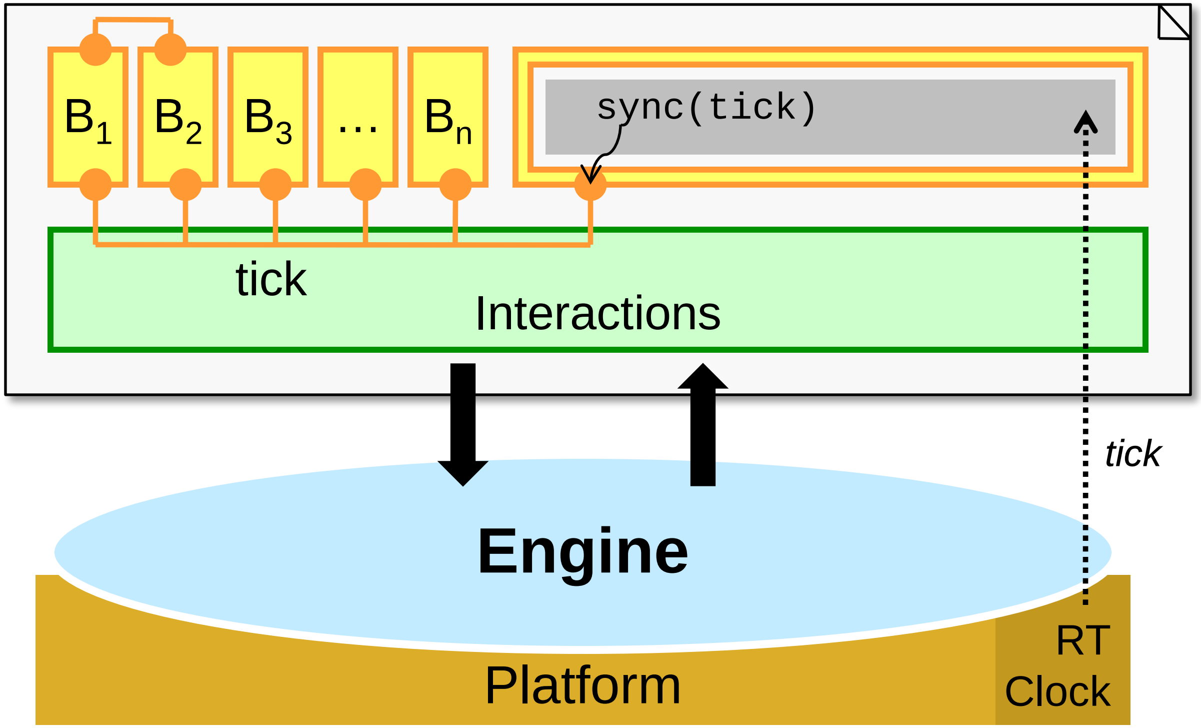
- BIP engine:

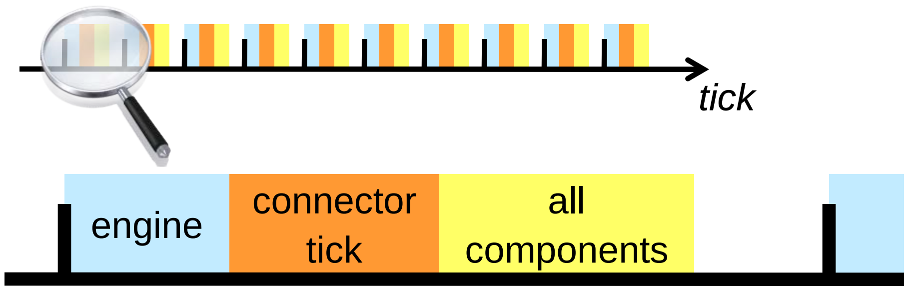
Functionality	Prototype	Tested
clocks	✓	✓
guard and urgency	✓	✓
connectors	✓	✓
hierarchical connectors	✓	✗
priorities	✓	✗
real-time scheduler	✓	✓

- BIP tool chain:

Functionality	Prototype	Tested
parser	✗	✗
code generator	✗	✗

Tick Implementation

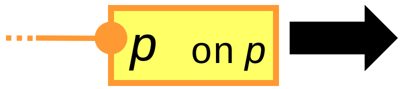




- <u>Synchronous execution:</u>
  - inefficient

Standards Ports / Events

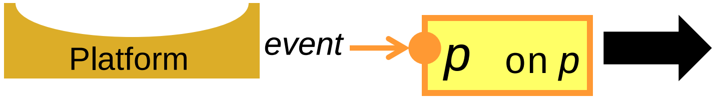
Clock t **Engine** enabled test



$N_p = t_0$
$D_p = +\infty$

$N_p \leq t \leq D_p$

Clock t **Engine** enabled test

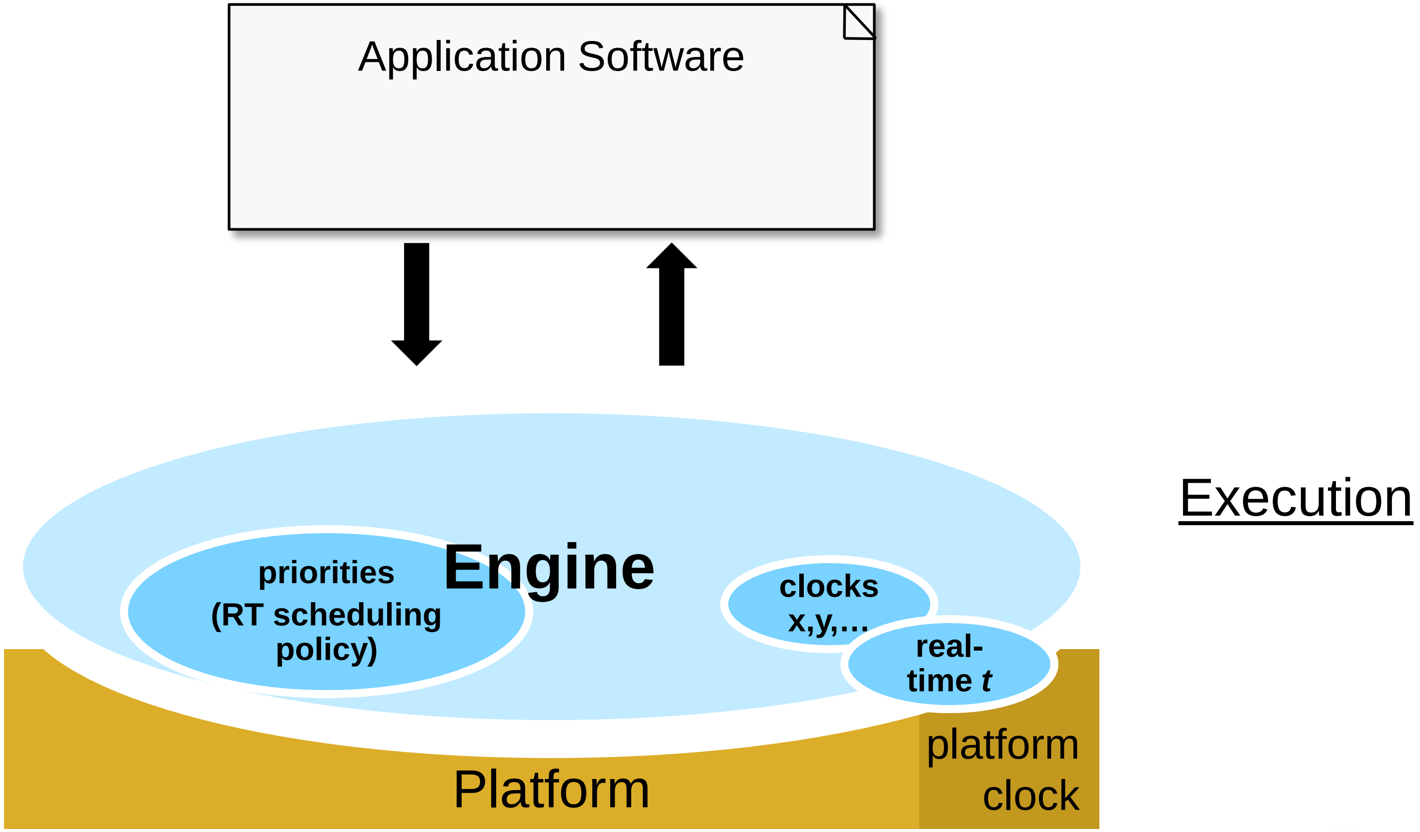


N_p: minimal arrival time
D_p: deadline if present

$N_p \leq t \leq D_p$

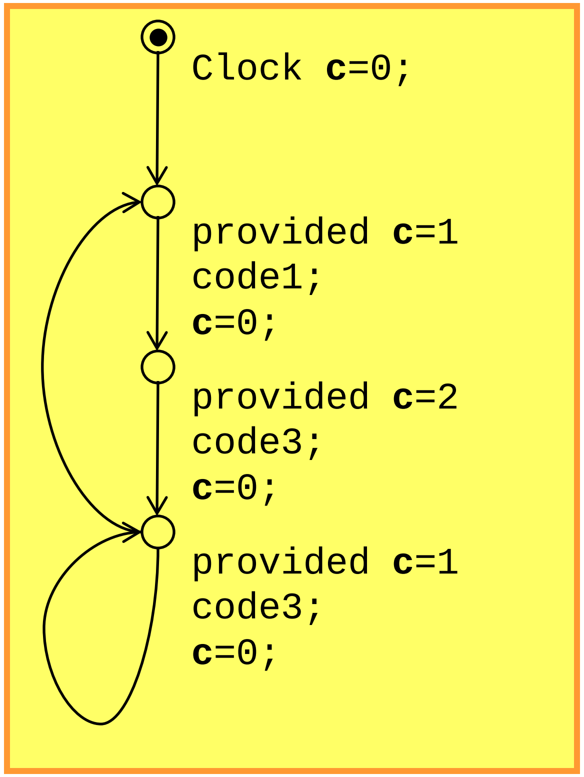
Real-Time Engine Implementation

- One engine for simulation and implementation
- In simulation mode, the real-time *t* is driven by the engine
- In execution mode, the real-time *t* is connected to the platform clock



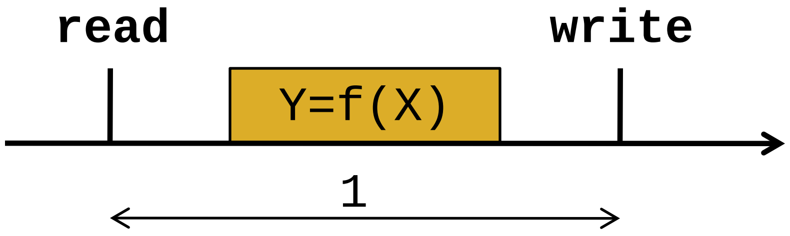
OASIS in BIP: control flow

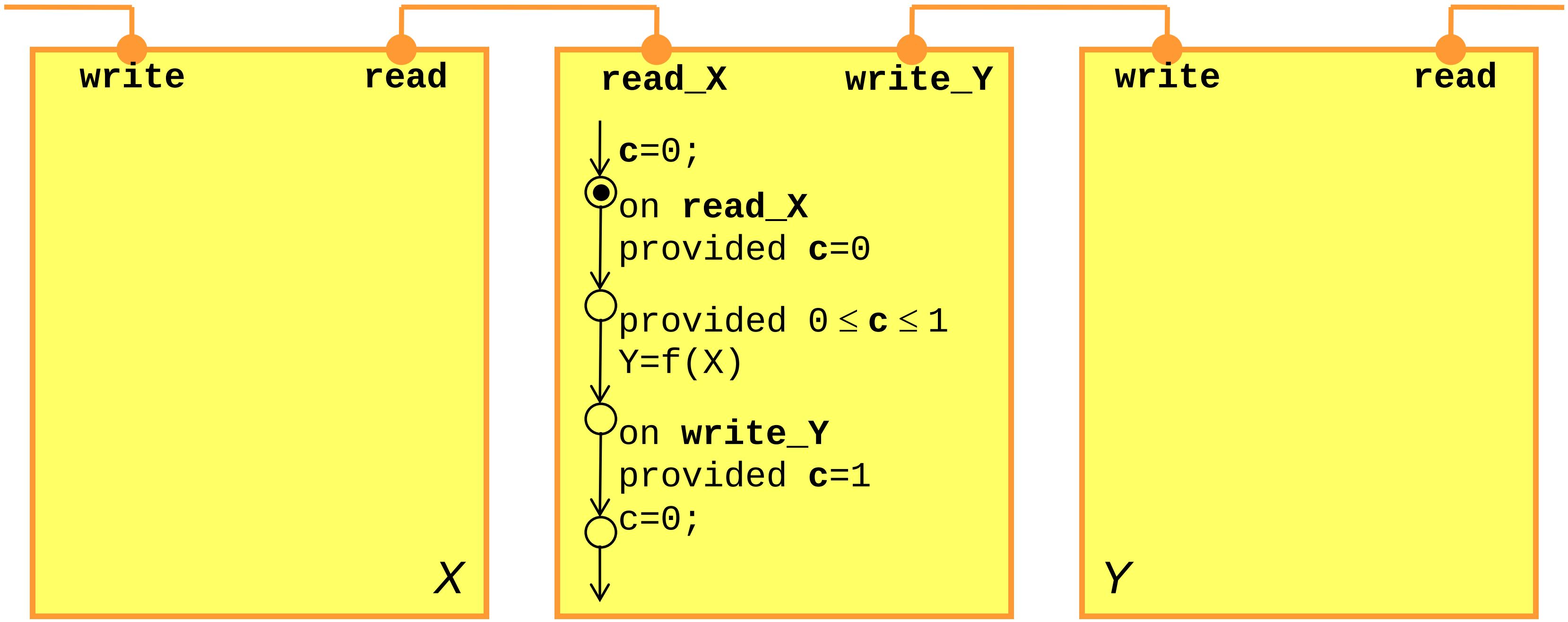
```
body start {
   code1; advance(1);
   code2; advance(2);
   do {
       code3;
       advance(1);
   } while (condition);
}
```



OASIS in BIP: computation

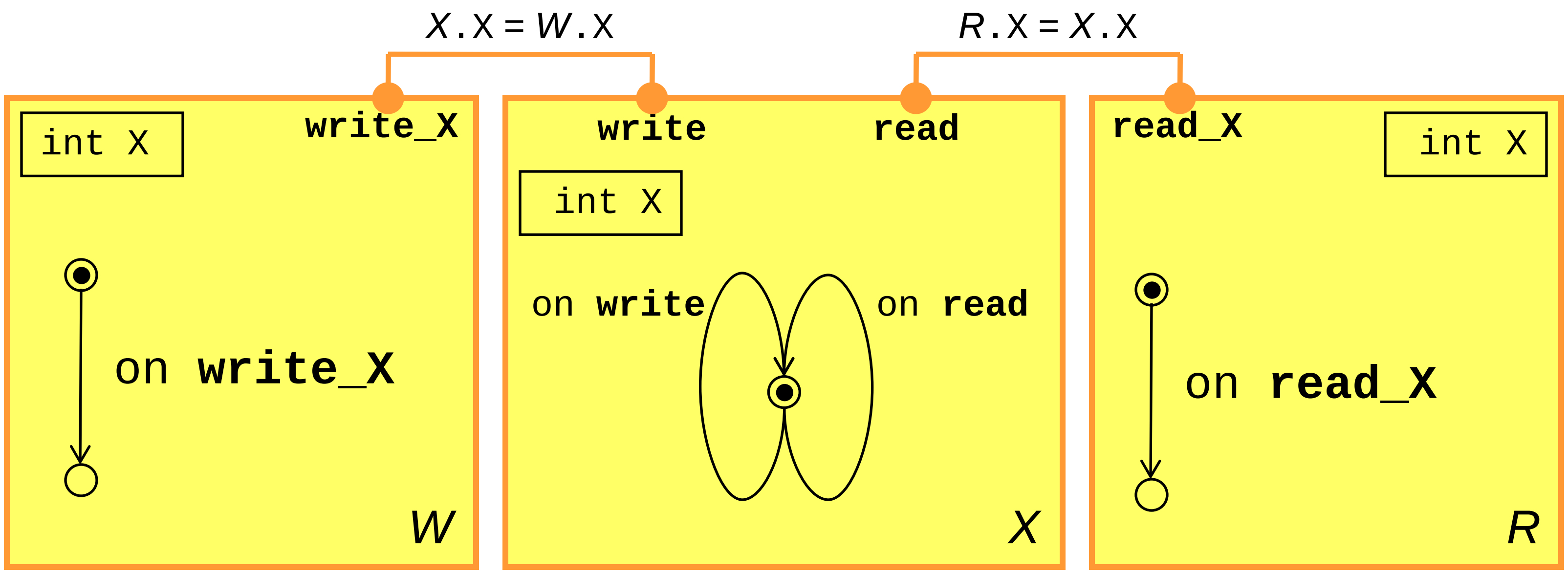
```
body start {
   ...
   Y=f(X); advance(1);
   ...
}
```





OASIS in BIP: coordination

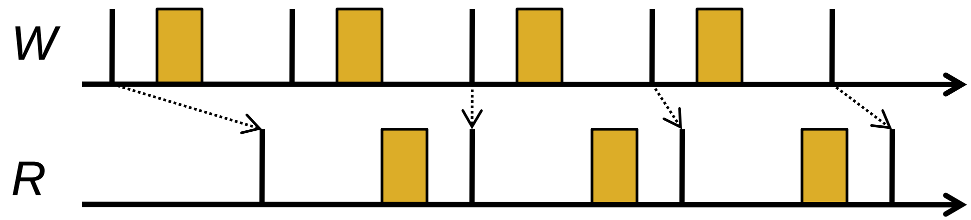
- If the reader doesn't read old values of *X*, we store only the current value in *X*:



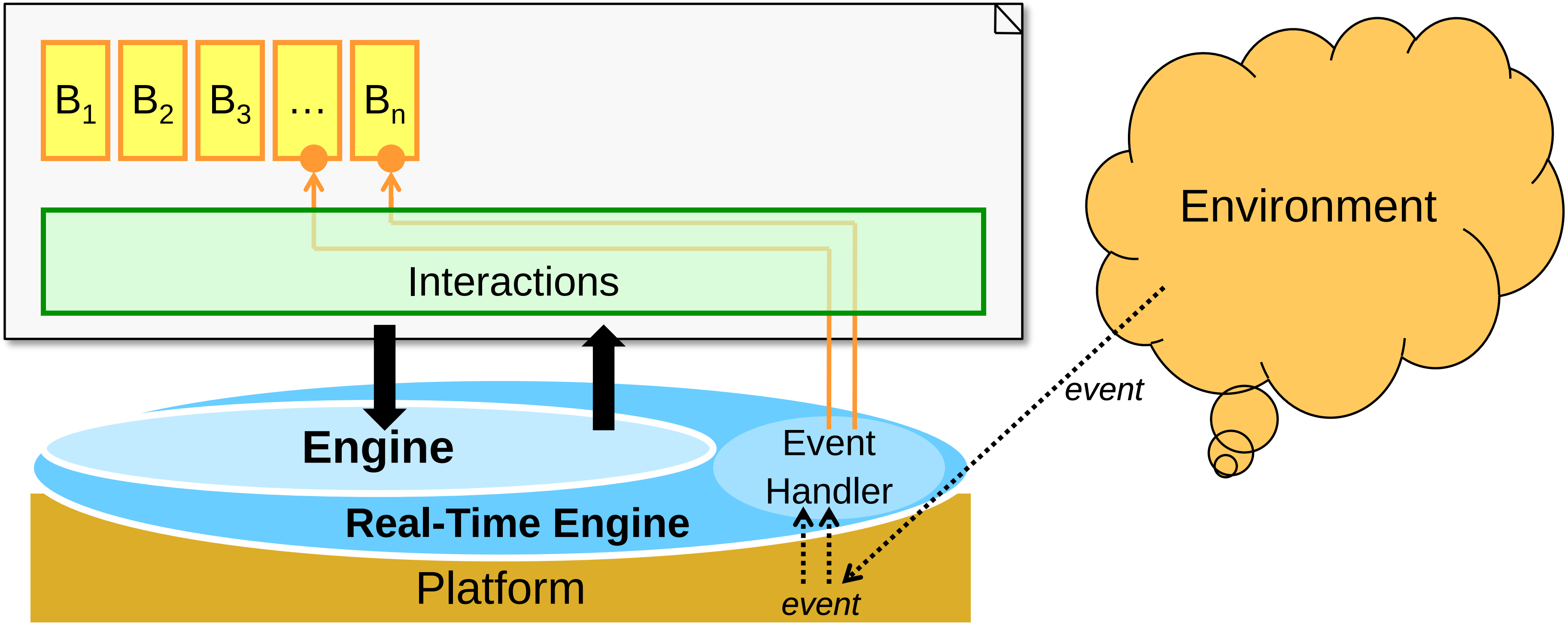
- The priority rule

write|write_X > read|read_X

resolves conflicts when writing and reading are possible at the same time.

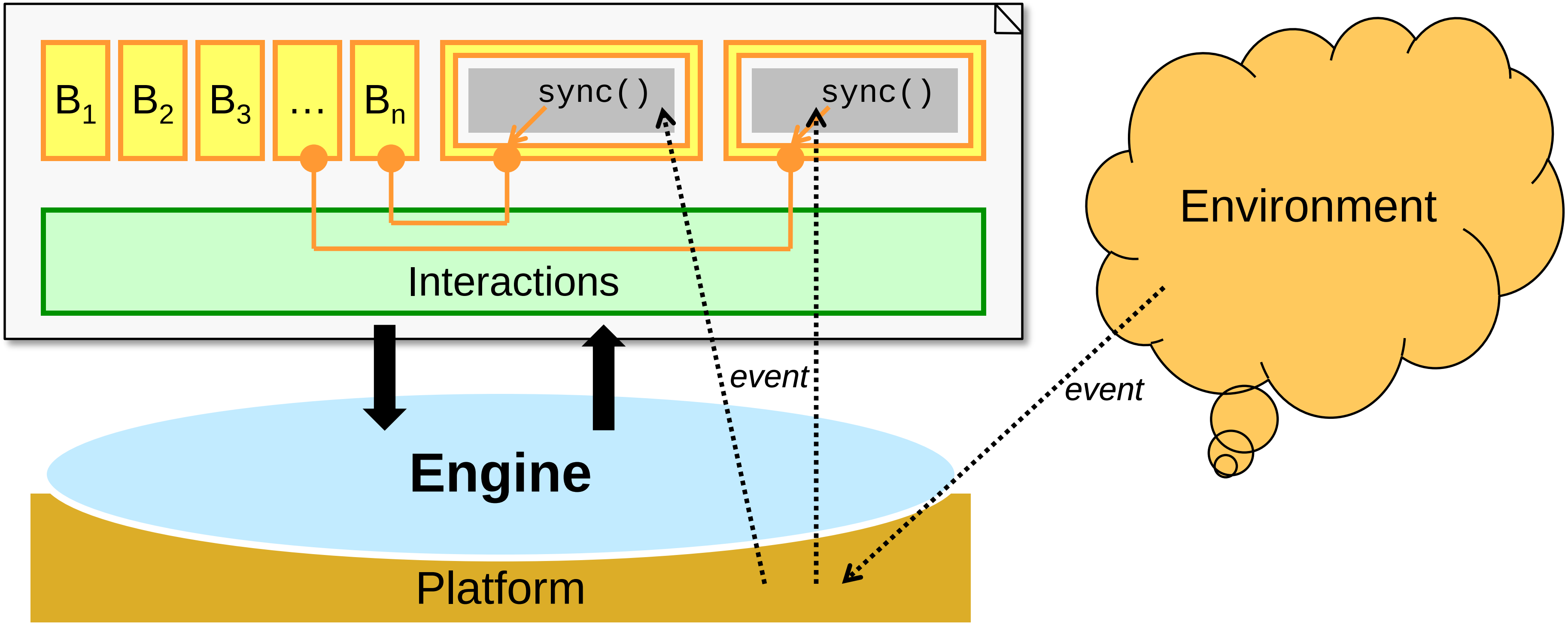


Simulation vs Implementation (2/2)



- Direct implementation:
  - events are directly handled by the engine
  - efficient active wait or interruption mechanisms
  - standard interfaces

Simulation vs Implementation (1/2)



- <u>Implementation by encapsulation:</u>
  - events are handled into components
  - only active waits (no interruption mechanisms)
  - specific interfaces