Model Based Development of Embedded Control Software

Part 2: Real-Time Systems

Claudiu Farcas

Credits: MoDECS Project Team, Giotto
Department of Computer Science
cs.uni-salzburg.at

UNIVERSITÄT SALZBURG

Contents

- Soft and Hard real-time systems
- Event and Time-based interaction model
- Environment vs Software time
- Real-Time Operating Systems

Target problem – Soft real-time

- Typical applications
  - VoIP
  - Video Streaming
  - Video/Computer Games
  - Communication devices (i.e., modem, ATM, GSM)
- No critical resources
- Generally sufficient CPU power
- Dynamic resource allocation (e.g., memory)
- Degraded Quality of Service (QoS) at peak load

Target problem – Hard real-time

- Typical applications
  - Mechanical/Mecatronic/Electronic controllers
  - Safety critical systems
- High temporal accuracy
- Minimal I/O jitter
- Limited resources: CPU, Memory, Battery
- Predictable peak-load performance

UNIVERSITÄT SALZBURG

Interaction Model - Events

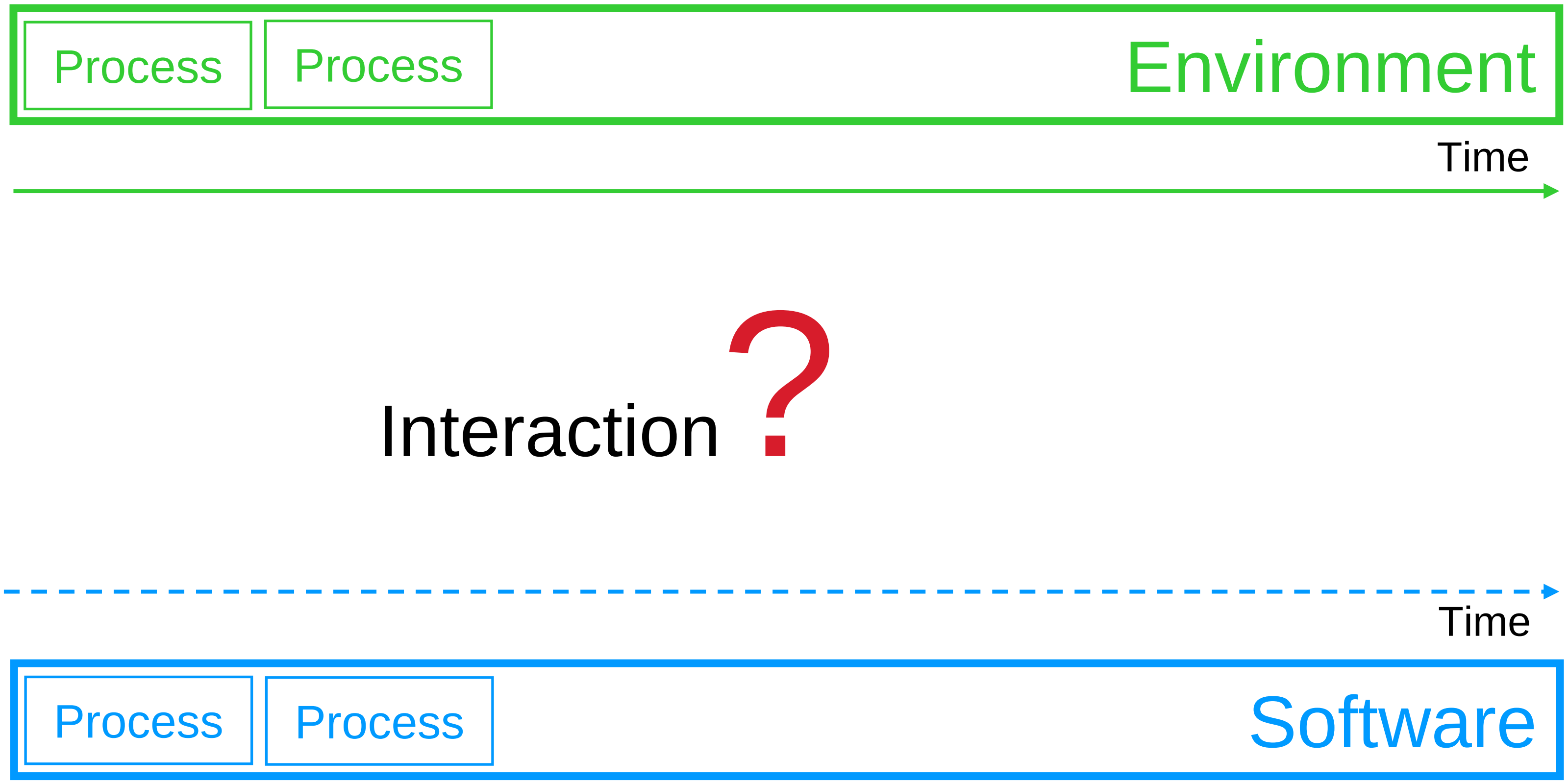
- Advantages
  - Pipeline design
  - Support for aperiodic systems
  - Low CPU utilization
- Disadvantages
  - Unpredictable concurrency
  - Response latency introduce additional jitter
  - Hardly scalable
  - No benefit for periodic events

Interaction Model – Time triggered

- Advantages
  - Support for periodic systems
  - Predictable concurrency
  - Deterministic behavior
  - Minimal jitter
  - Scalable
  - May be distributed
- Disadvantages
  - Emulation for aperiodic systems
  - Higher CPU utilization

Environment vs Software Time

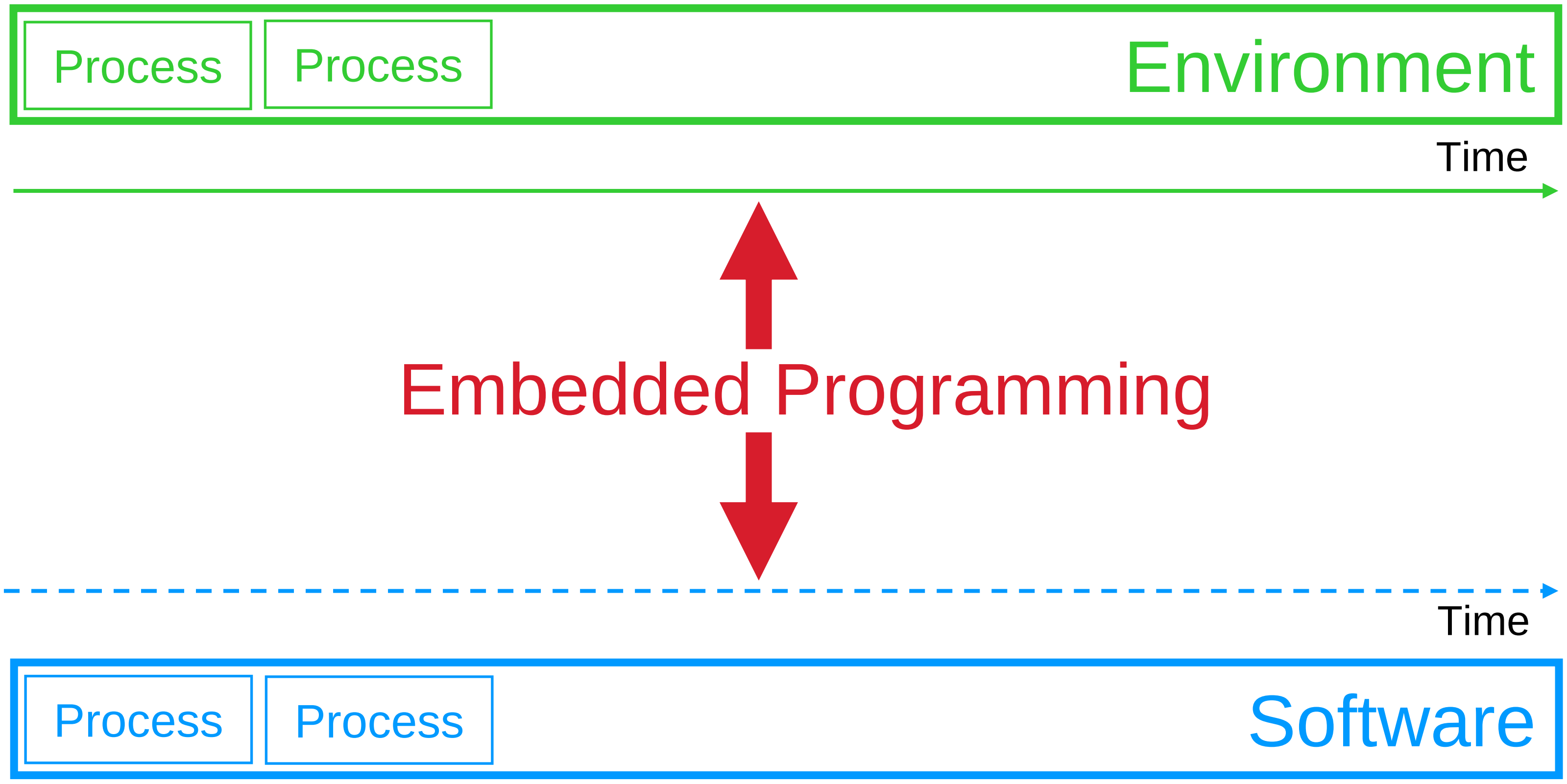
- Continuous vs Discrete time



UNIVERSITÄT SALZBURG

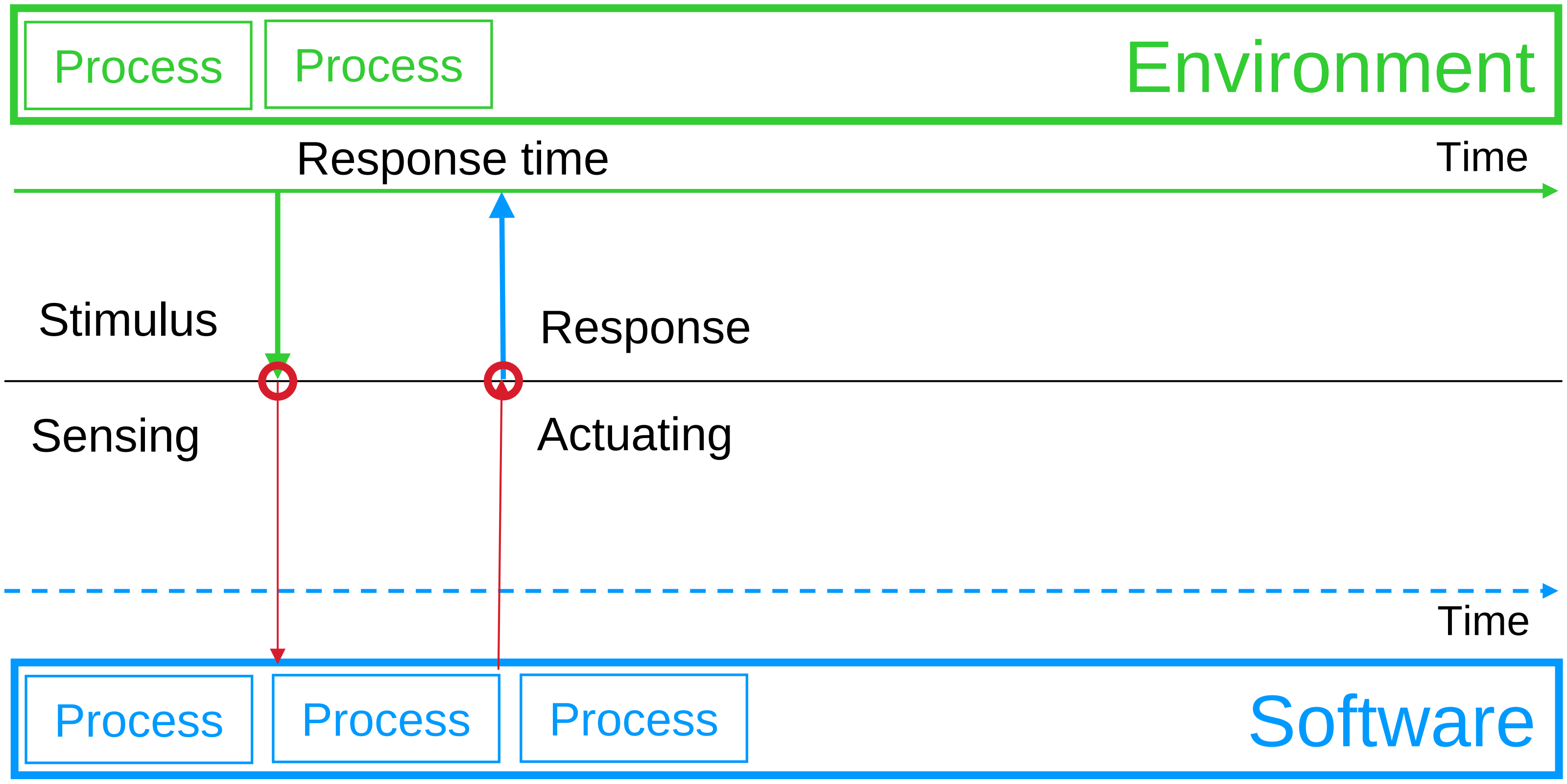
Environment vs Software Time

- Interaction between software and environment



Event triggered computation

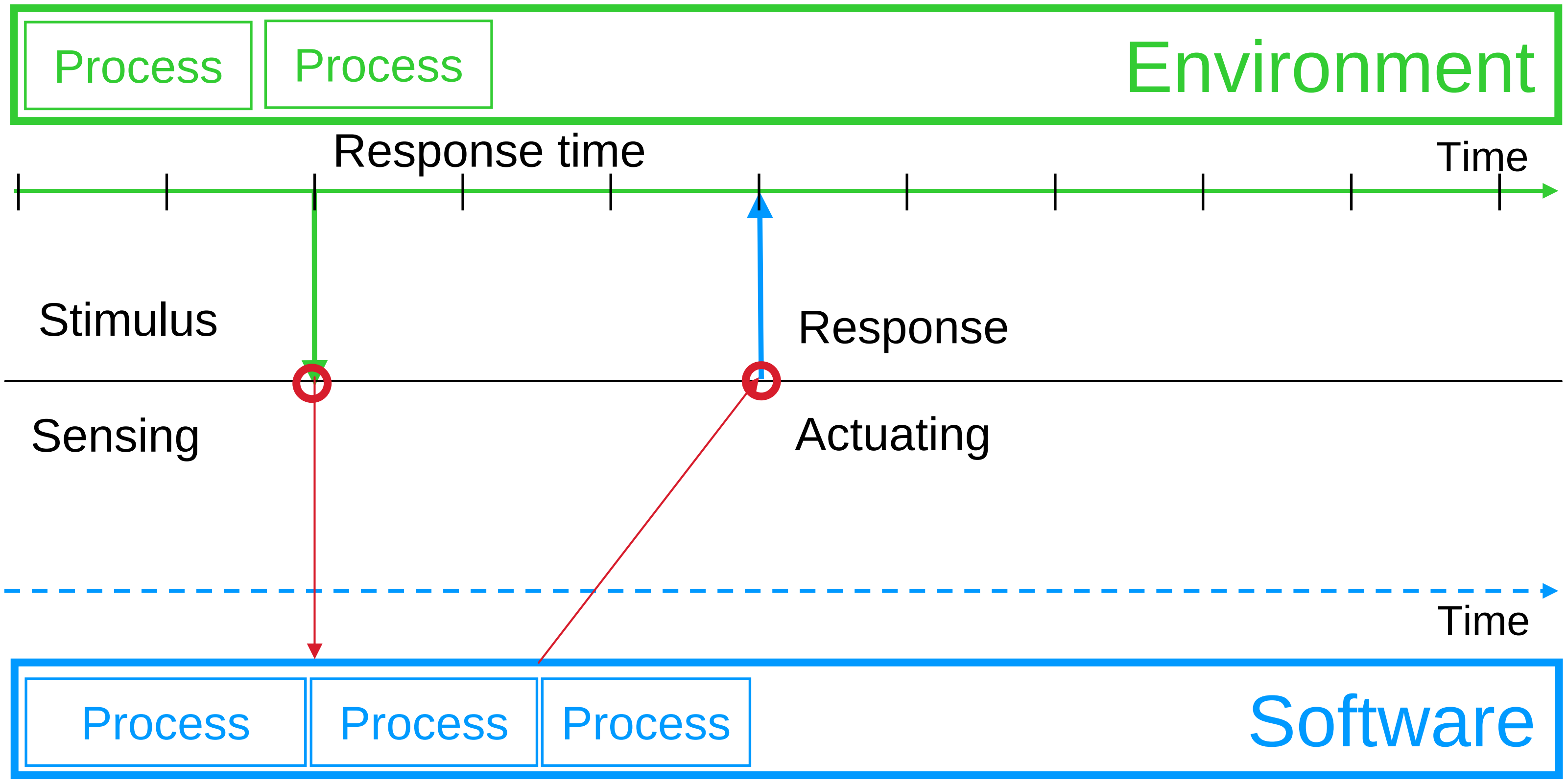
- Response to an event from the environment



UNIVERSITÄT SALZBURG

Time triggered computation

- Response to an event from the environment



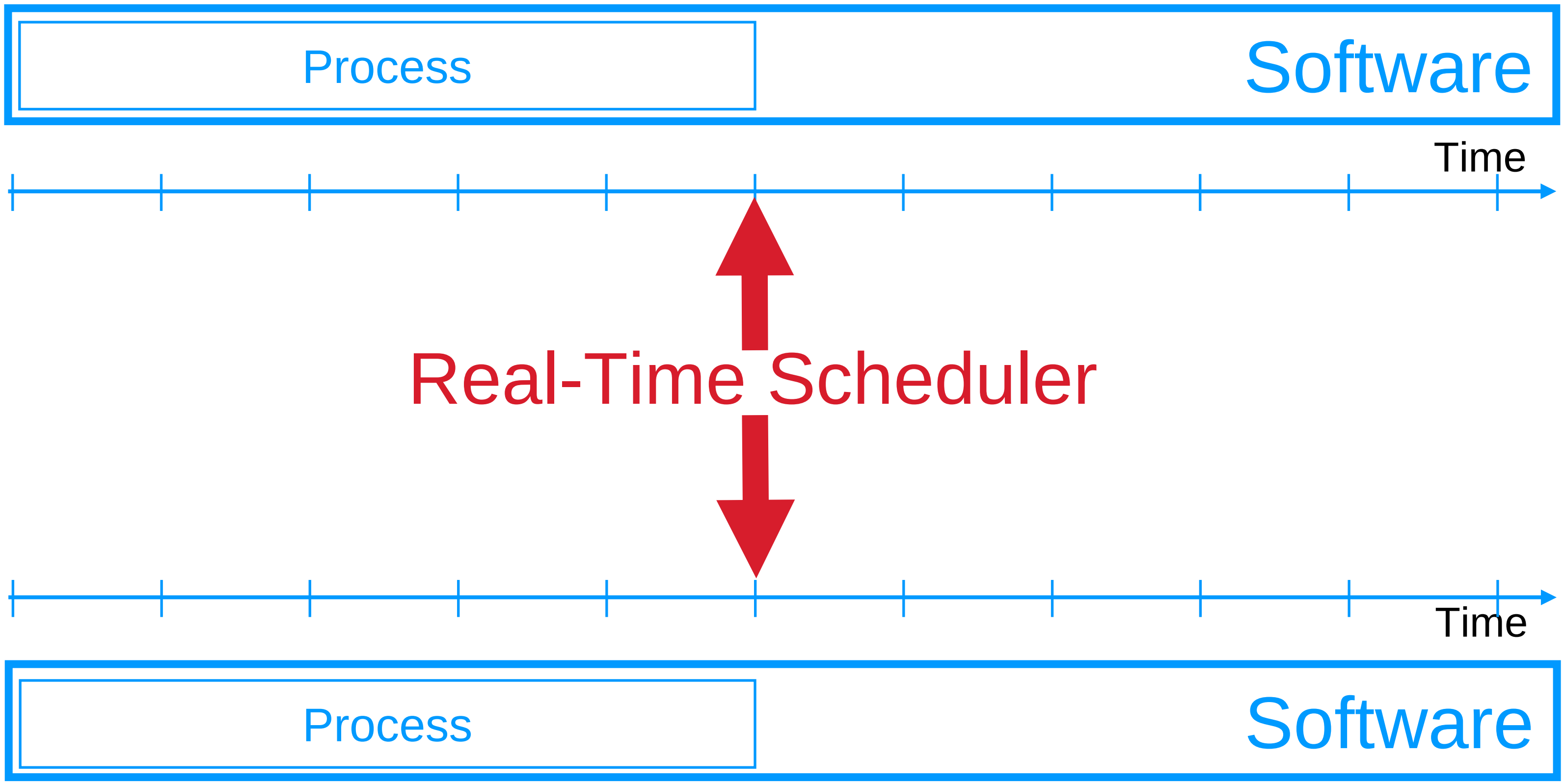
UNIVERSITÄT SALZBURG

Software to software communication

- May use the same event or time model
- Plenty of APIs available but not all adequate for RT systems
- Predictable behavior desired -> required
- Scheduling decisions become important
- Low level API timing not negligible

Software to software interaction

- Real-time software environment



UNIVERSITÄT SALZBURG

Real-Time OS Services

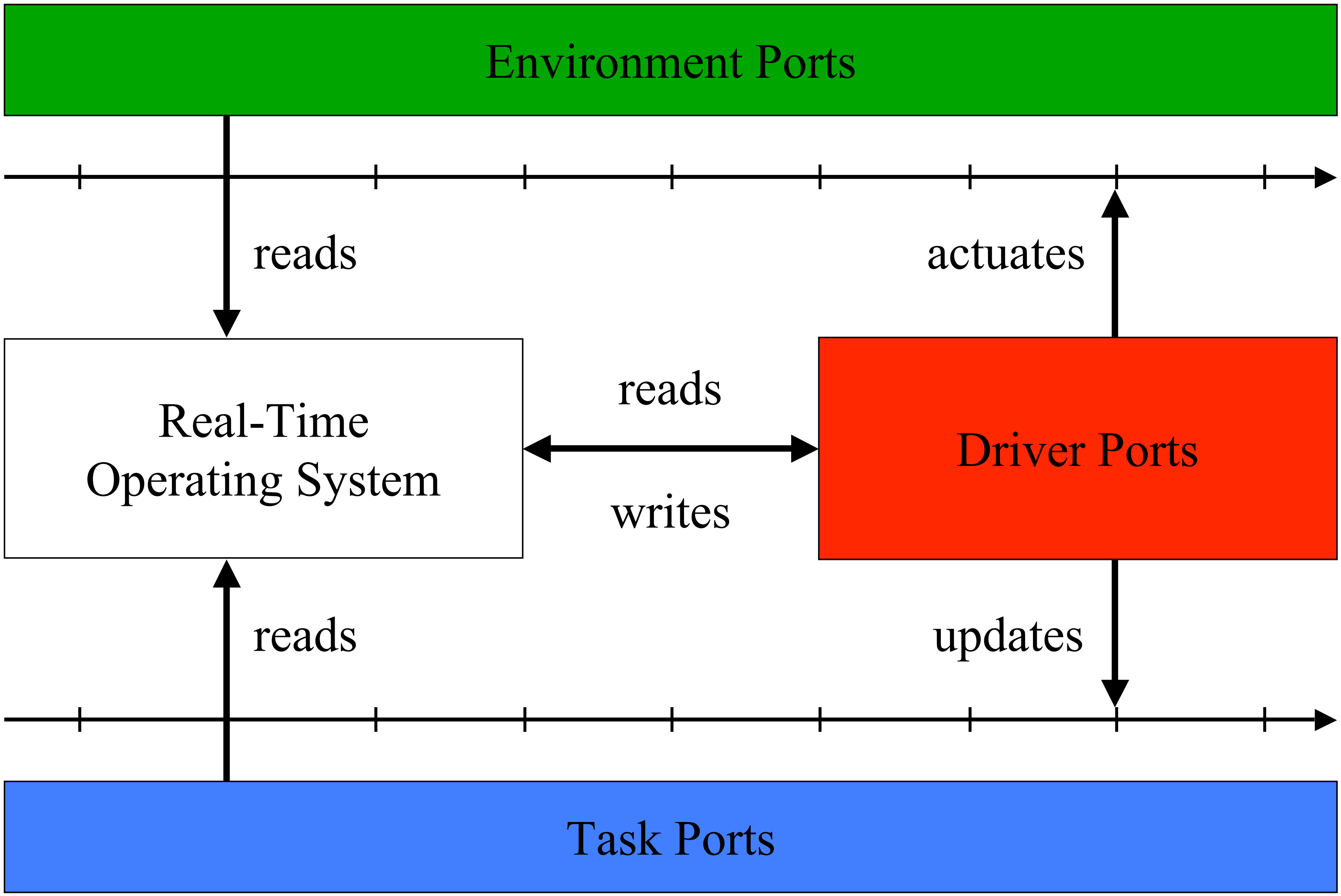
- Environment
  - Sensing / Actuating – Drivers
  - Triggers – Interrupt handlers
- Software
  - intercommunication – Shared memory
  - Triggers – signals (e.g., semaphores, mailboxes)
  - Scheduling – RT Scheduler
  - Concurrency support – CPU multiplexing, SMP, etc

LET, Giotto, ...

- Follow-up
  - Credits: Prof. C. Kirsch slides ESE-RTOS04, pages 12-73

UNIVERSITÄT SALZBURG

Memory Model



Definition: Task

- A task is a *function* from its input and state ports to its output and state ports

- A task *runs to completion* (cannot be killed)
- A task is *preemptable*

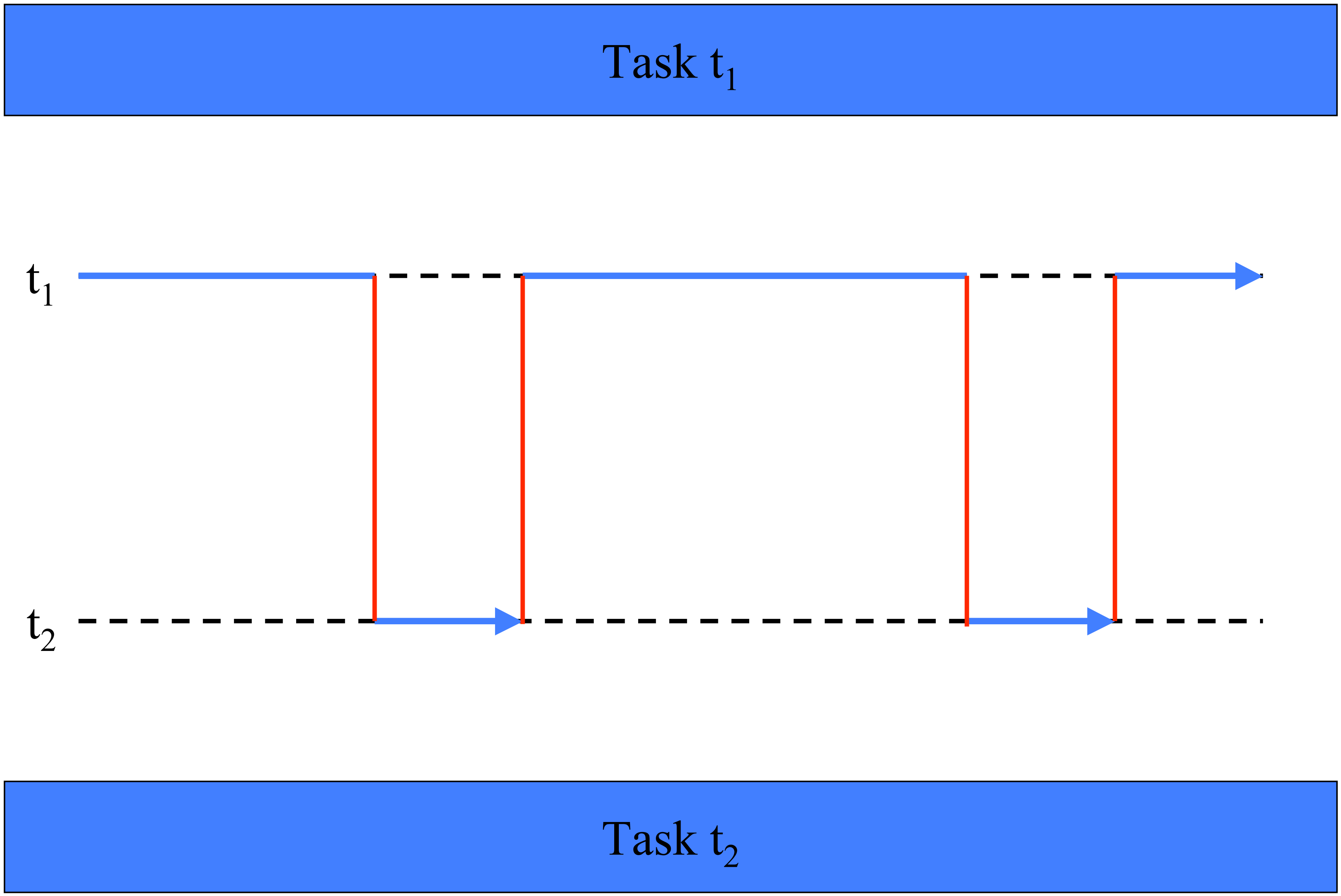
- A task does not use *signals* (except at completion)
- A task does not use *semaphores* (as a consequence)

- API (used by the RTOS):
  - `initialize` {task: state ports}
  - `release` {task}
  - `dispatch` {task: function}

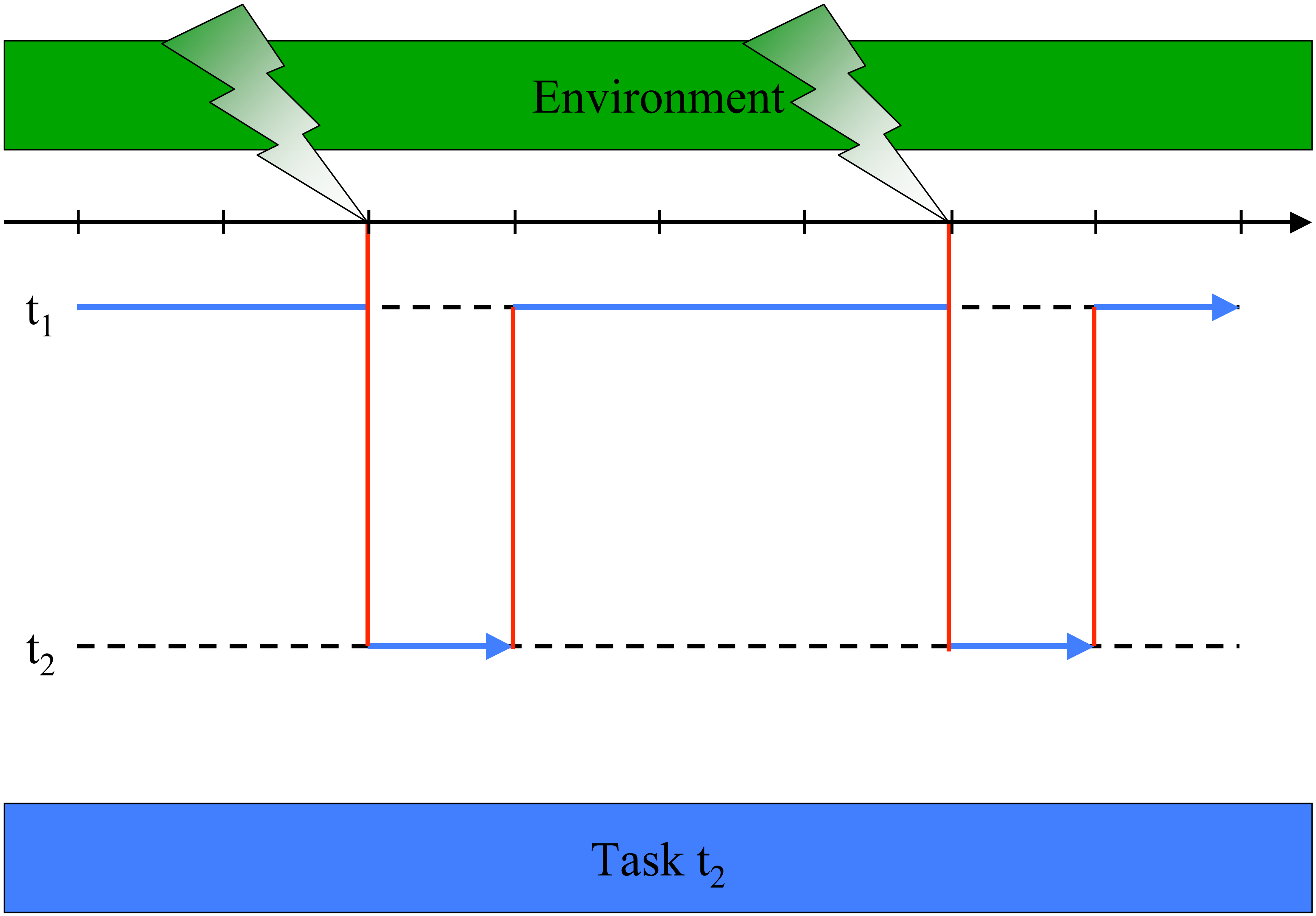
So, what's the difference between a task and a function?

- A task has an operational semantics:
  - A task is implemented by a *subroutine* and a *trigger*
  - A task is either *environment*- or *software-triggered*
  - The completion of a task may trigger another task

Task t_2 Preempts Task t_1



Who Triggers Task t_2?



Definition: Event and Signal

- An event is a *change of state* in some environment ports
- A signal is a *change of state* in some task ports

- A synchronous signal is a *change of state* in some driver ports

Definition: Trigger

- A trigger is a *predicate* on environment, task, driver ports

- A trigger *awaits* events and/or signals
- A trigger is *enabled* if its predicate evaluates to true
- Trigger evaluation is *atomic* (non-preemptable)

- A trigger can be *activated* by the RTOS
- A trigger can be *cancelled* by the RTOS
- A trigger can be *enabled* by an event or a signal

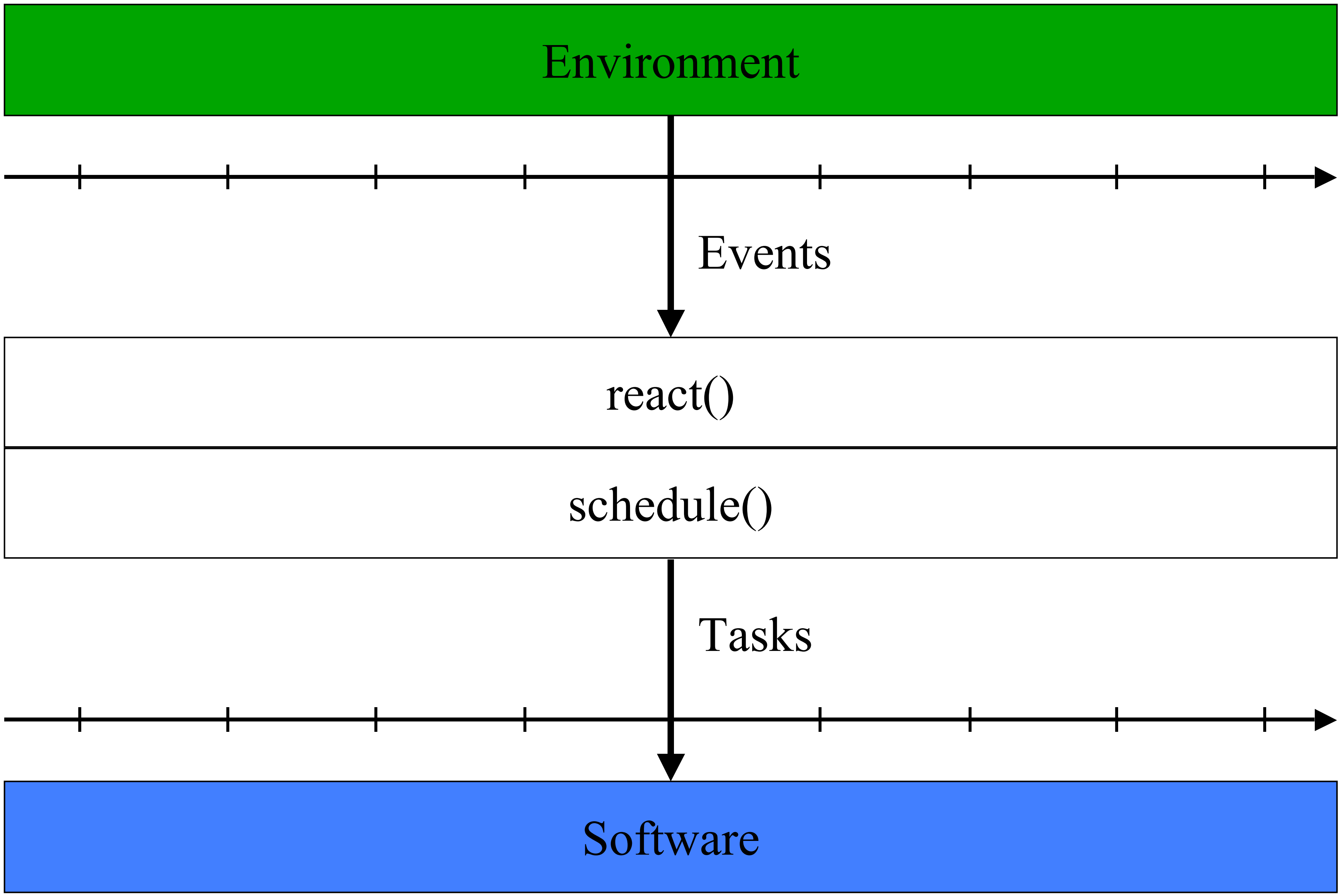
- API (used by the RTOS):
  - `activate` {trigger}
  - `cancel` {trigger}
  - `evaluate` {trigger: predicate}

My First RTOS

```
react() {
 ∀ tasks t: initialize(t);
 ∀ triggers g: activate(g);
 while (true) {
  if ∃ trigger g: evaluate(g) == true then
   released-tasks := ∀ to-be-released-tasks t: release(t);
   schedule();
 }
}
```

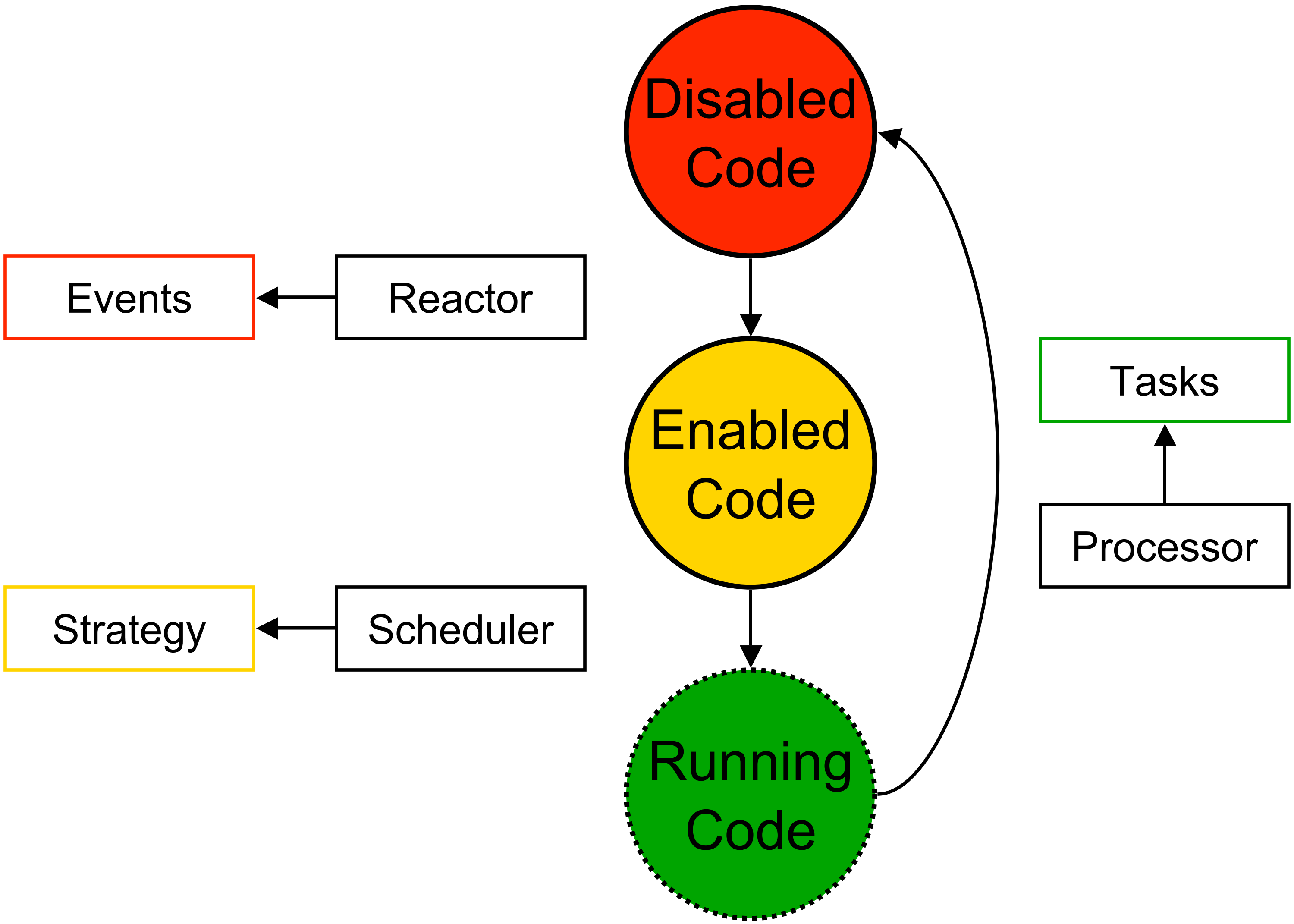
```
schedule() {
  ∀ released-tasks t: dispatch(t);
  released-tasks := {};
}
```

RTOS Model: Reaction vs. Scheduling



Reactor vs. Scheduler vs. Processor

(Kirsch in the Proceedings of EMSOFT 2002)



RTOS with Preemption

```
react() {
 ∀ tasks t: initialize(t);
 ∀ triggers g: activate(g);
 while (true) {
  if ∃ trigger g: evaluate(g) == true then
   released-tasks := ∀ to-be-released-tasks t: release(t);
   schedule_concurrently();
 }
}
```

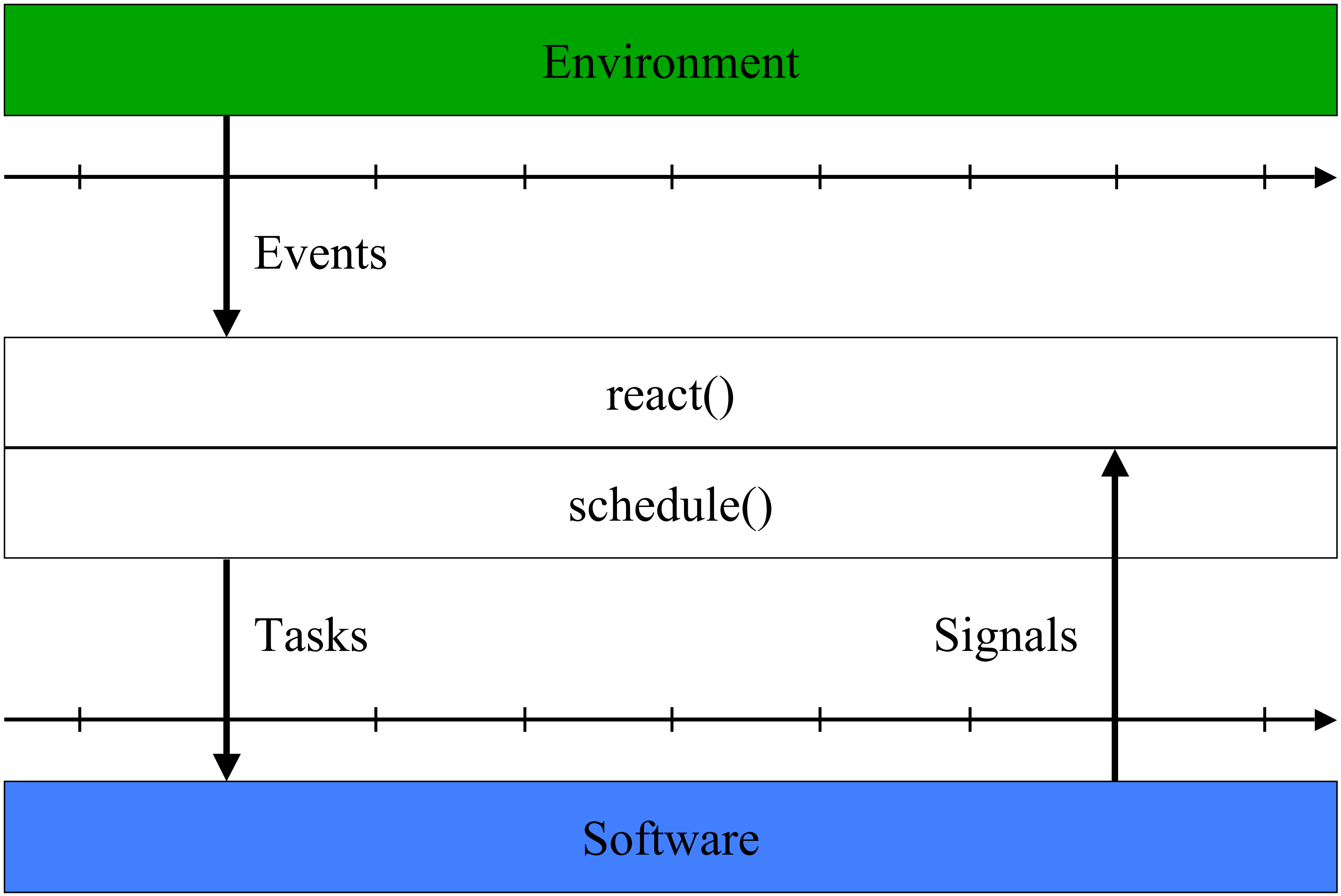
```
schedule_concurrently() {
  ∀ released-tasks t: dispatch(t);
  released-tasks := {};
}
```

Corrected RTOS with Preemption

```
react() {
 ∀ tasks t: initialize(t);
 ∀ triggers g: activate(g);
 while (true) {
  if ∃ trigger g: evaluate(g) == true then
   released-tasks := released-tasks ∪
                ∀ to-be-released-tasks t: release(t); }}
```

```
schedule() {
 while (true) {
  t := select(released-tasks);
  dispatch(t);
  released-tasks := released-tasks \ { t }; }}
```

RTOS Model with Signals



Definition: Thread

- A thread is a *behavioral function* (with a trace semantics)

- A thread *may be killed*
- A thread is *preemptable*

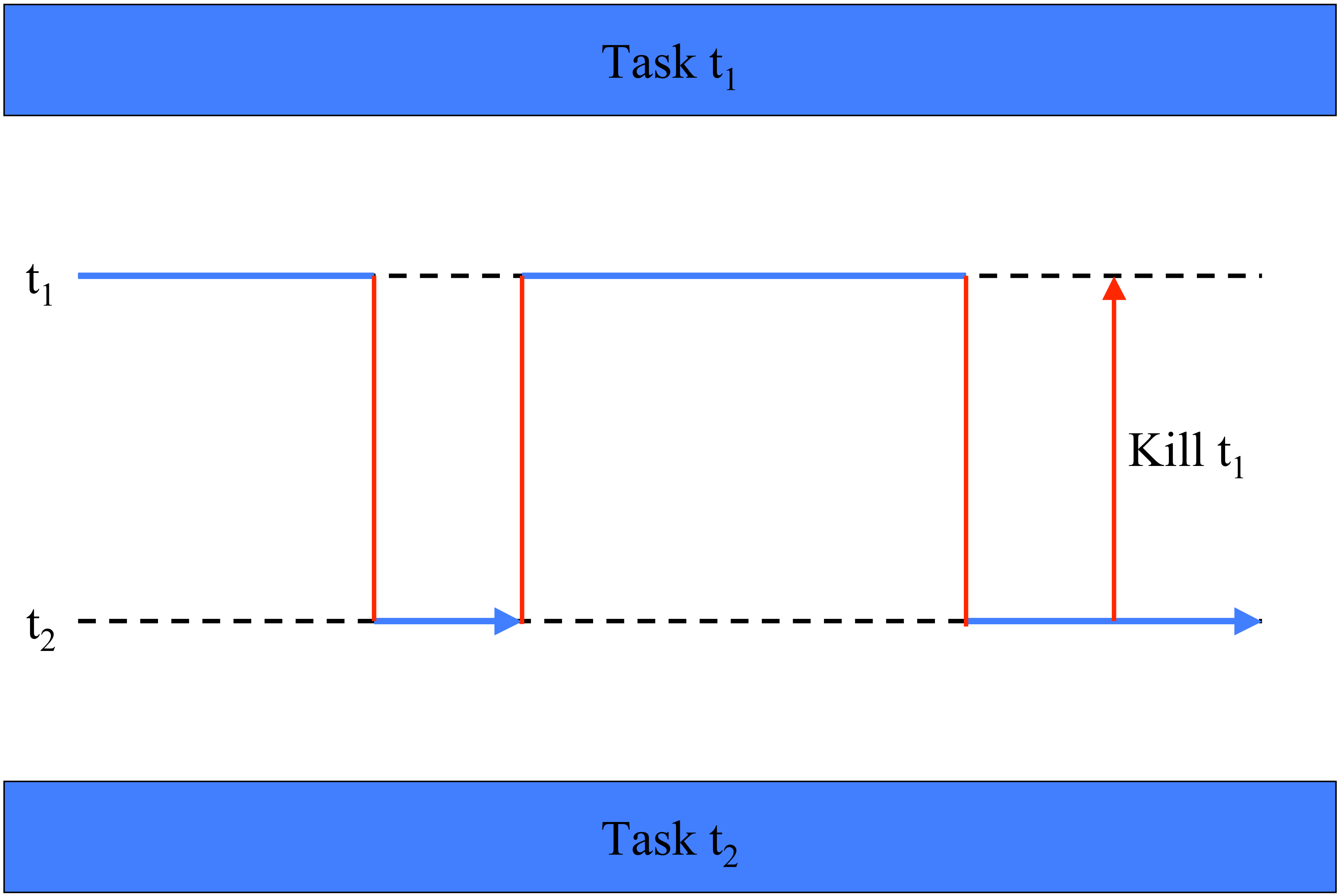
- A thread may use *signals*
- A thread may use *semaphores*

- API (used by the RTOS or threads):
  - `initialize` {thread: ports}
  - `release` {thread}
  - `dispatch` {thread: function}
  - `kill` {thread}

So, what’s the difference between a thread and a task?

- A thread is a *collection* of tasks:
  - A thread is implemented by a *coroutine*
  - A thread requires signals

Task t_2 Kills Task t_1



Signal API

- A signal can be *awaited* by a thread
- A signal can be *emitted* by a thread
- Signal emission is *atomic* (non-preemptable)

- API (used by threads):
  - `wait` {signal}
  - `emit` {signal}

- Literature:
  - emit: send(signal)

Definition: Semaphore

- A semaphore consists of a *signal* and a *port*

- A semaphore can be *locked* by a thread
- A semaphore can be *released* by a thread
- Semaphore access is *atomic* (non-preemptable)

- API (used by threads):
  - `lock` {semaphore}
  - `release` {semaphore}

- Literature:
  - lock: P(semaphore)
  - release: V(semaphore)

Binary Semaphore (Signal)

```
lock(semaphore) {
  if (semaphore.lock == true) then
    wait(semaphore.signal);

  semaphore.lock := true;
}
```

must be atomic

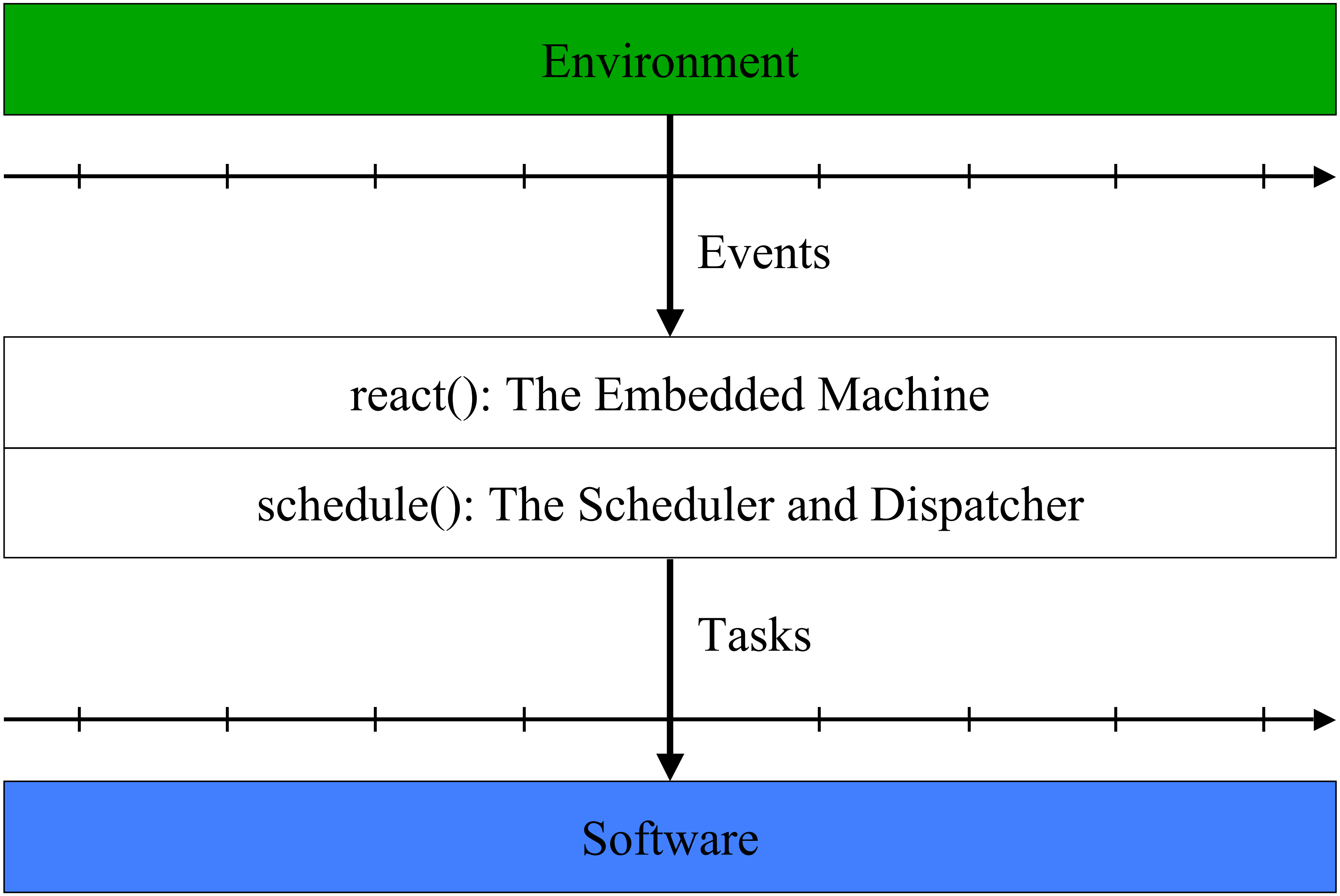
```
release(semaphore) {
  semaphore.lock := false;
  emit(semaphore.signal);
}
```

Binary Semaphore (Busy Wait)

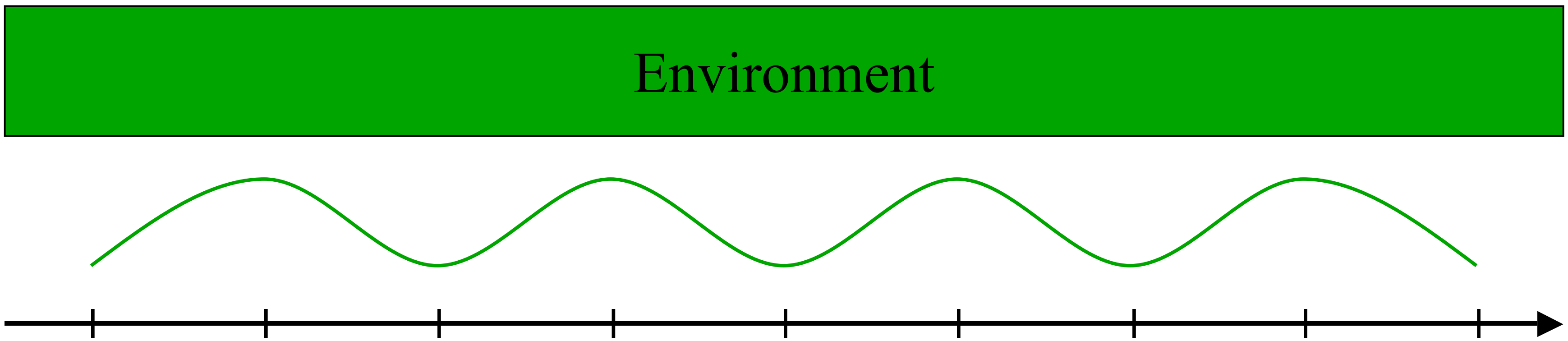
```
lock(semaphore) {
   while (semaphore.lock == true) do {}   ⎫ each round
   semaphore.lock := true;                ⎭ must be atomic
}

release(semaphore) {
   semaphore.lock := false;
}
```

The Embedded Machine

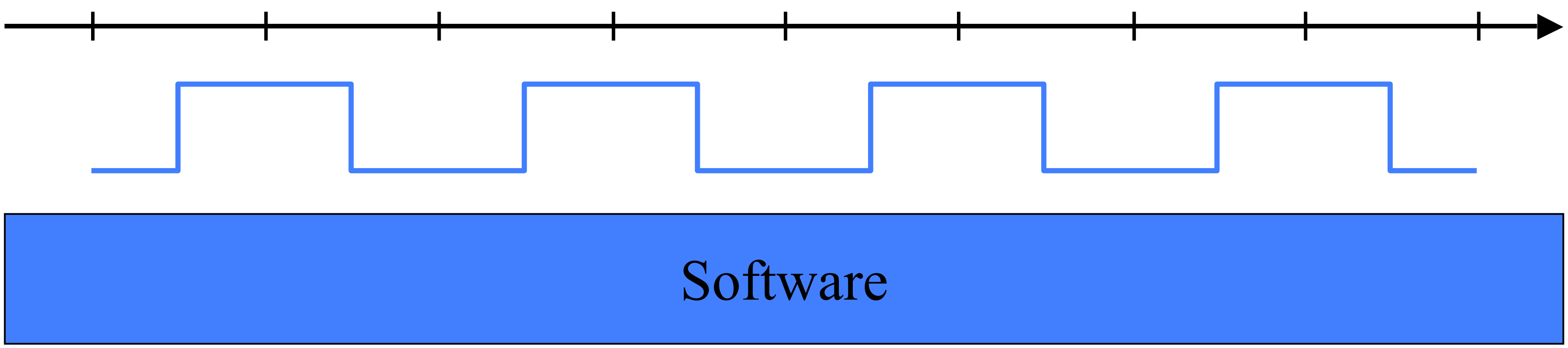


Proposal

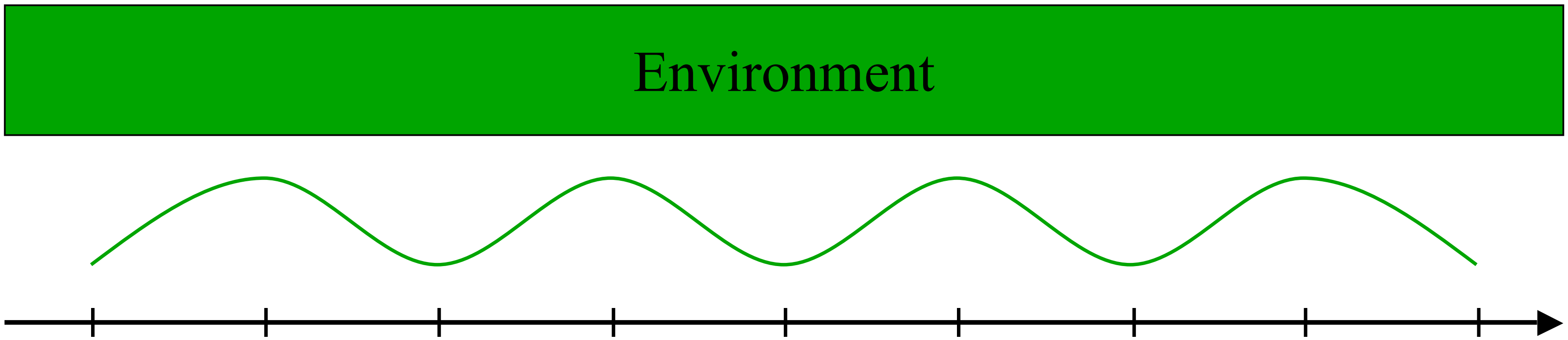


Human: Programming in terms of environment time

Compiler: Implementation in terms of platform time

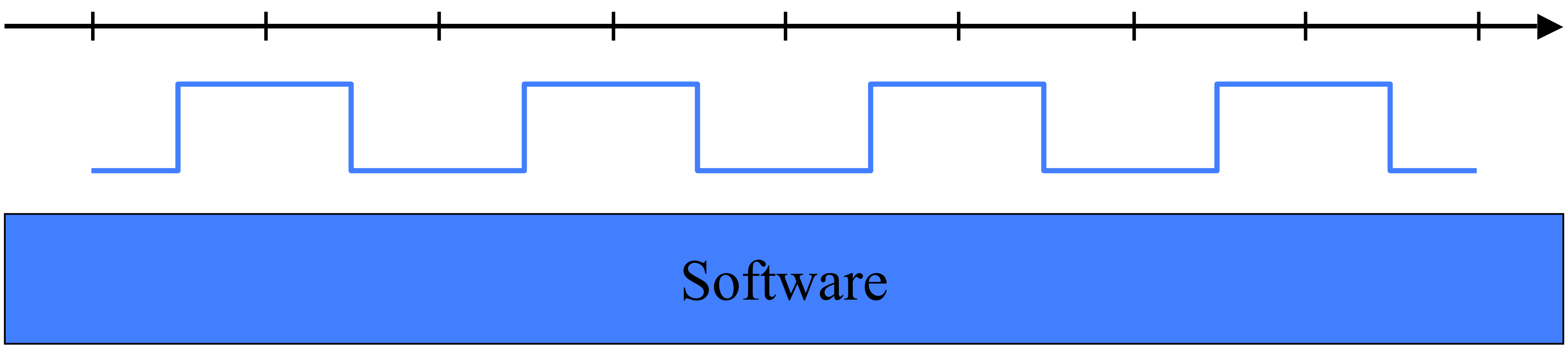


Platform Time is Platform Memory

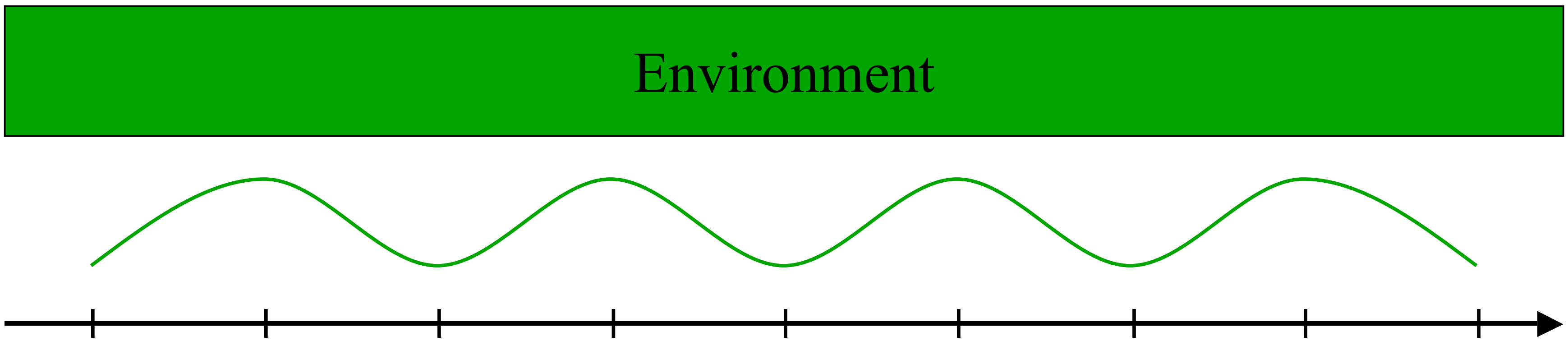


• Programming as if there is enough platform time

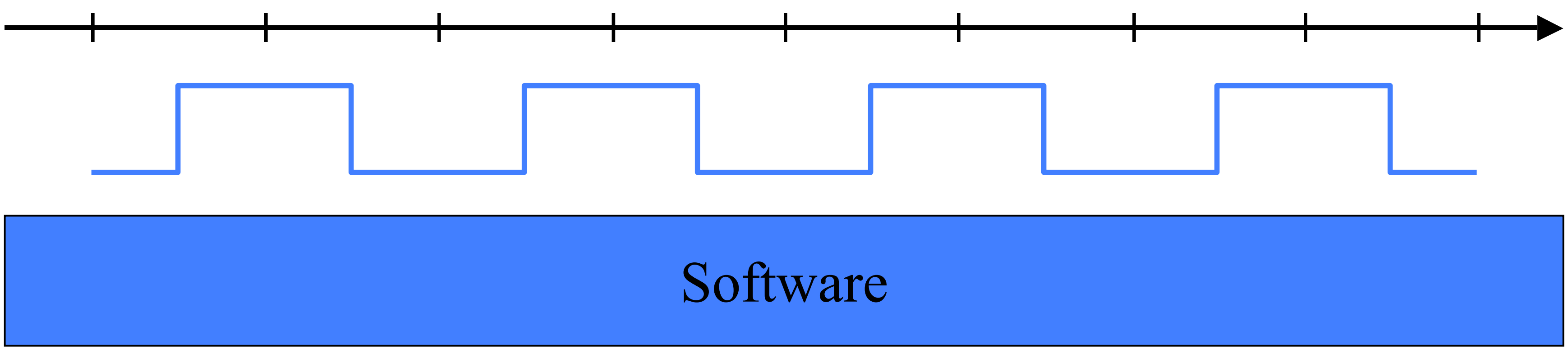
• Implementation checks whether there is enough of it



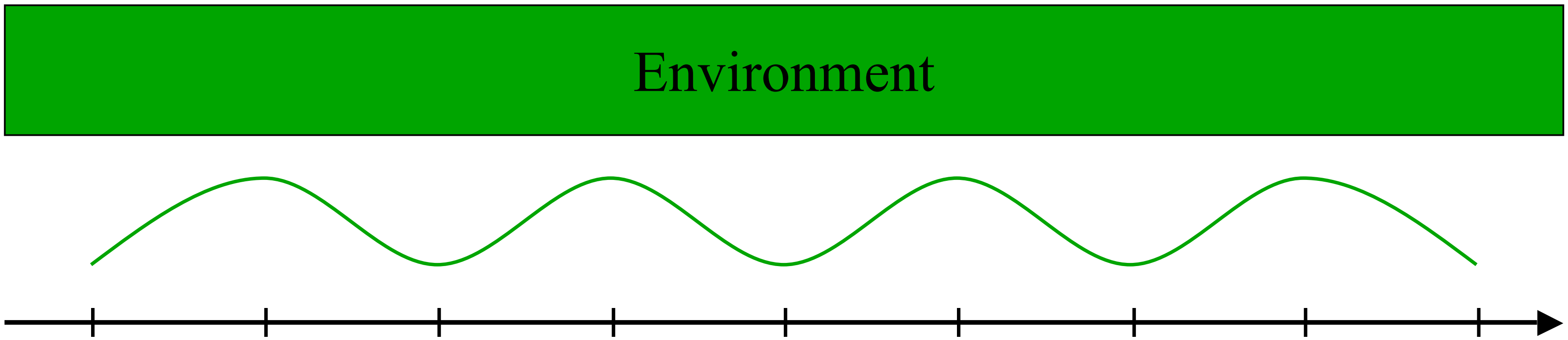
Portability



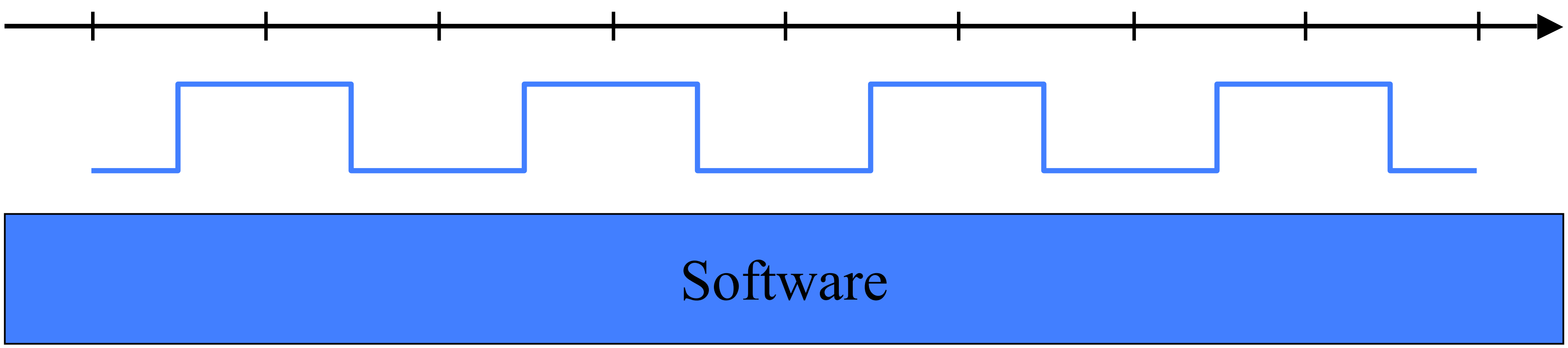
- Programming in terms of environment time yields platform-independent code



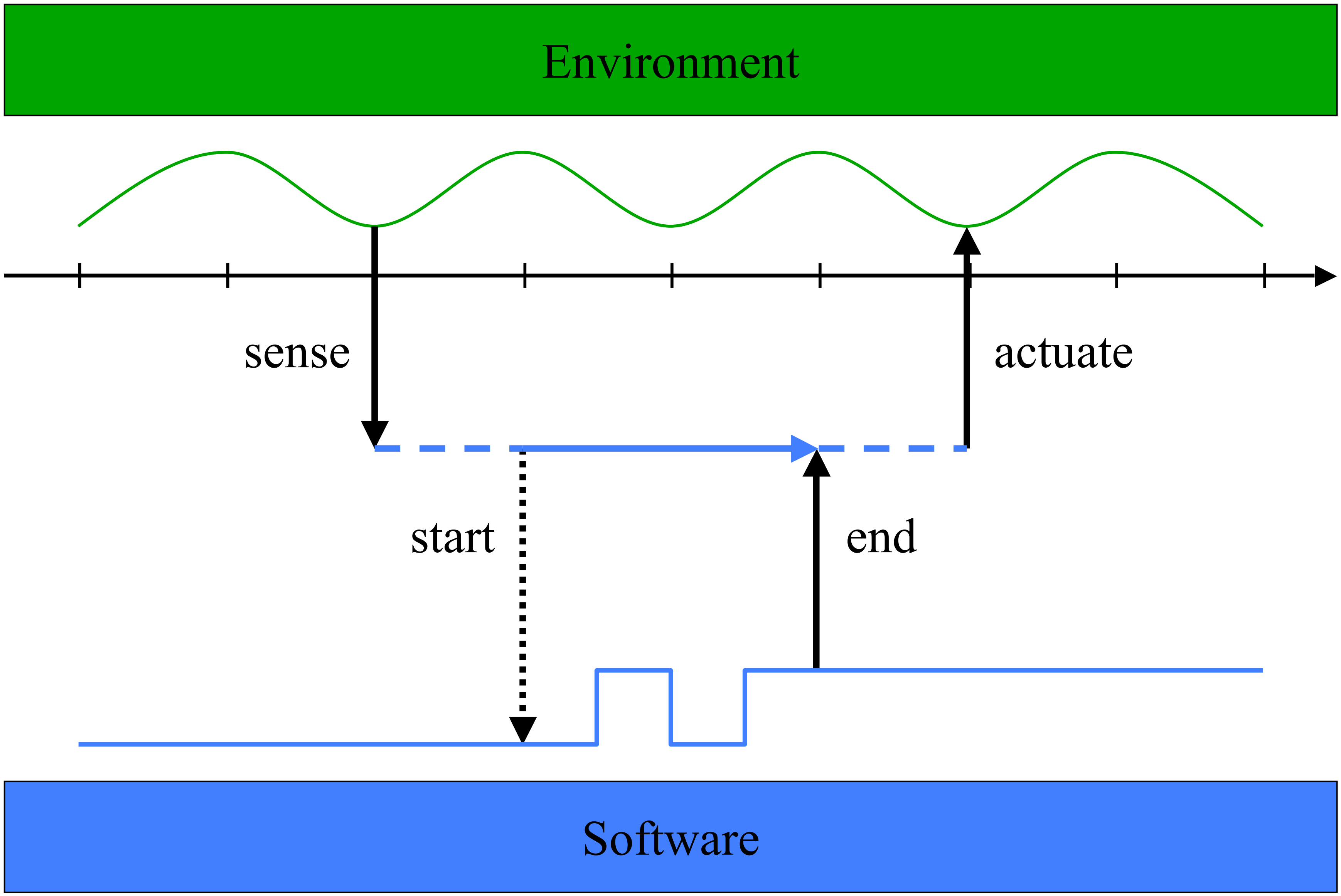
Predictability



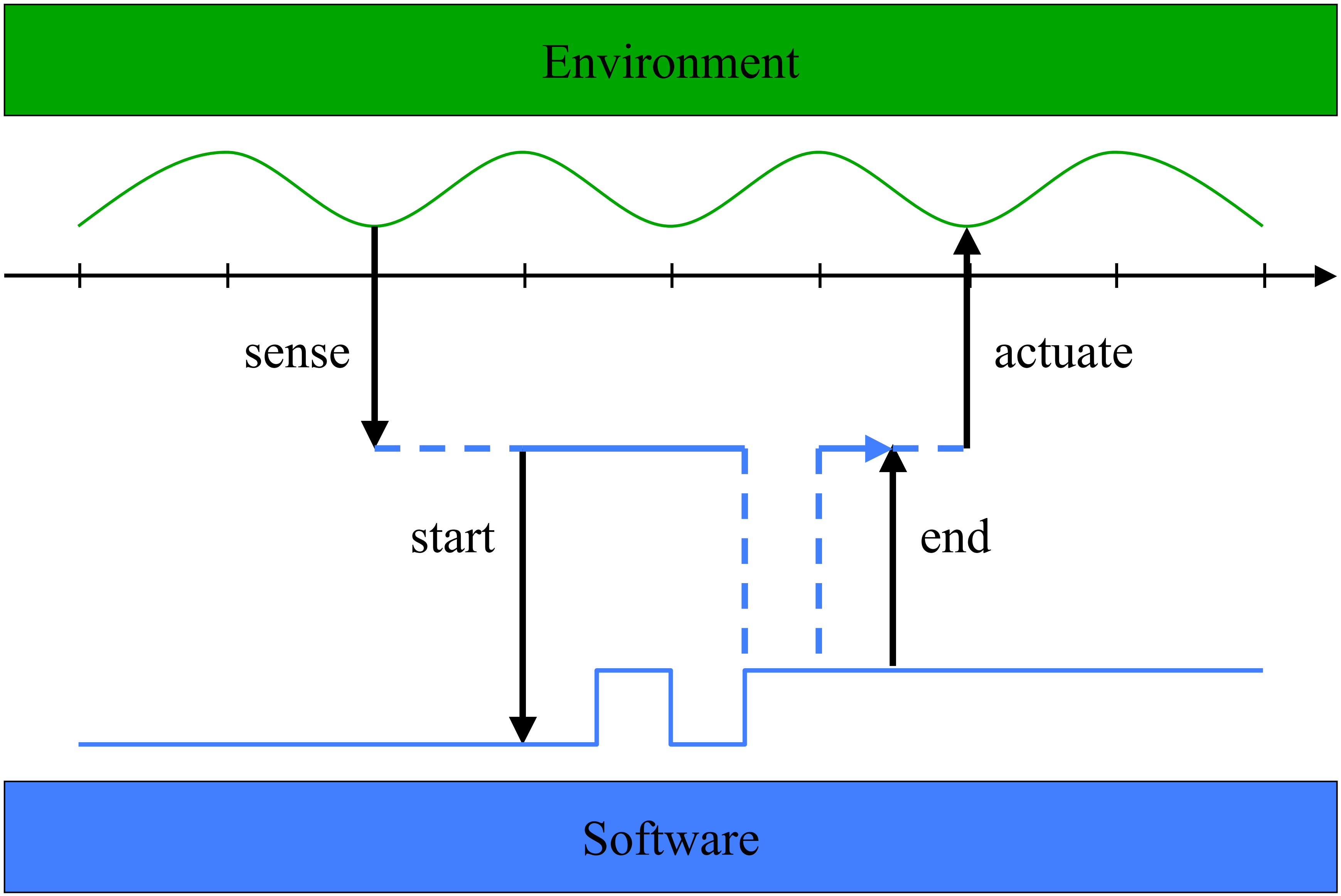
- Programming in terms of environment time yields <u>deterministic</u> code



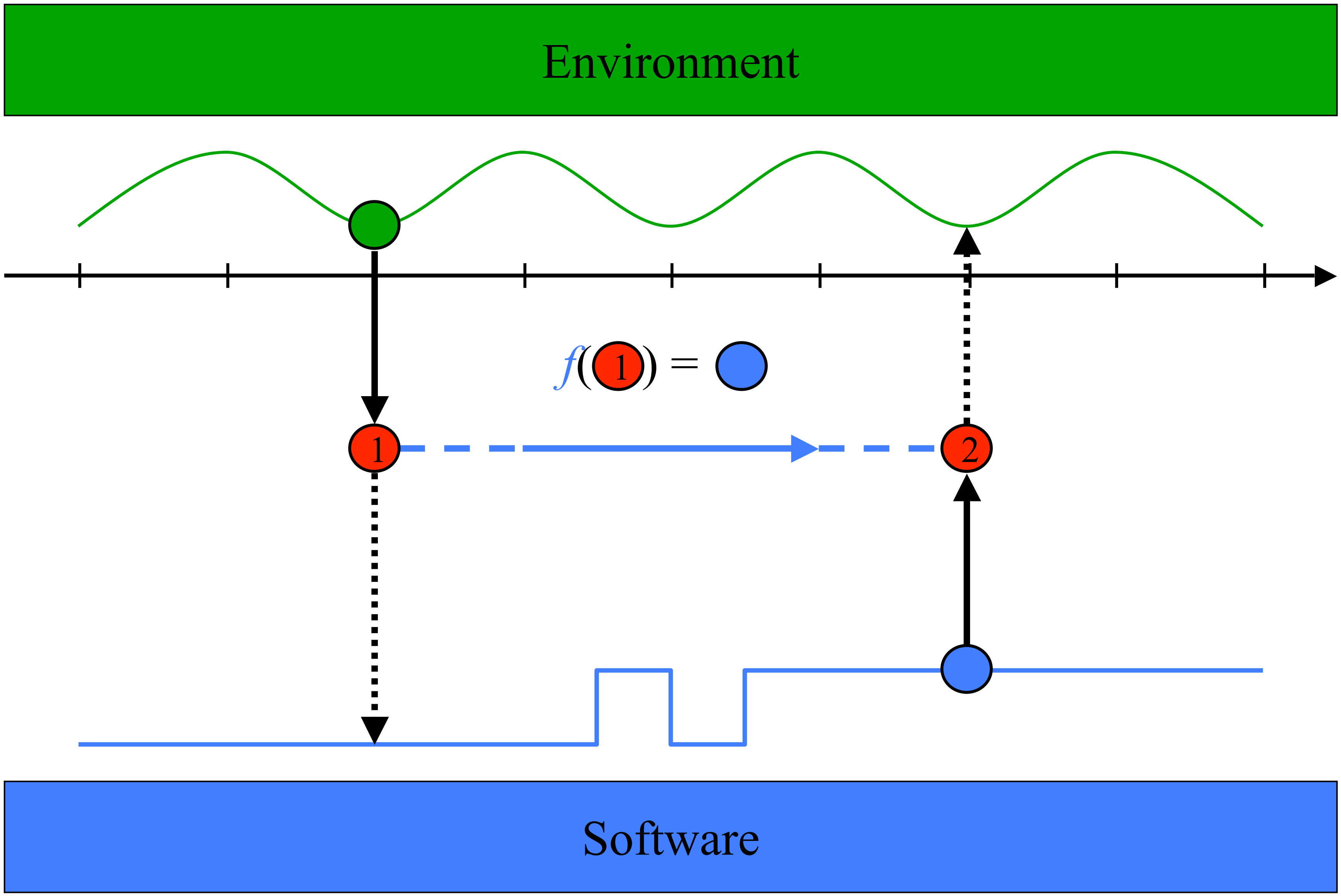
The Task Model



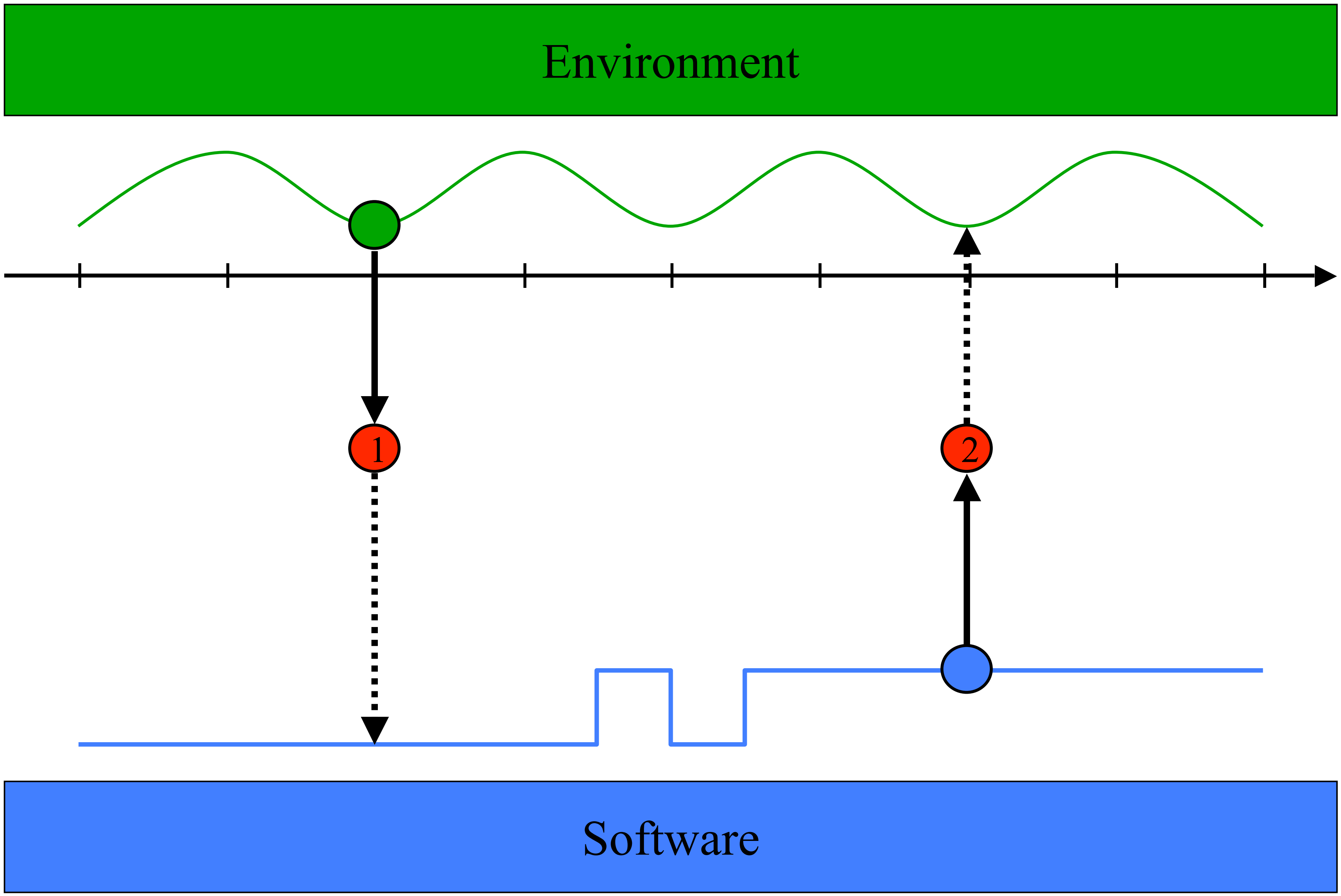
Preemptable…



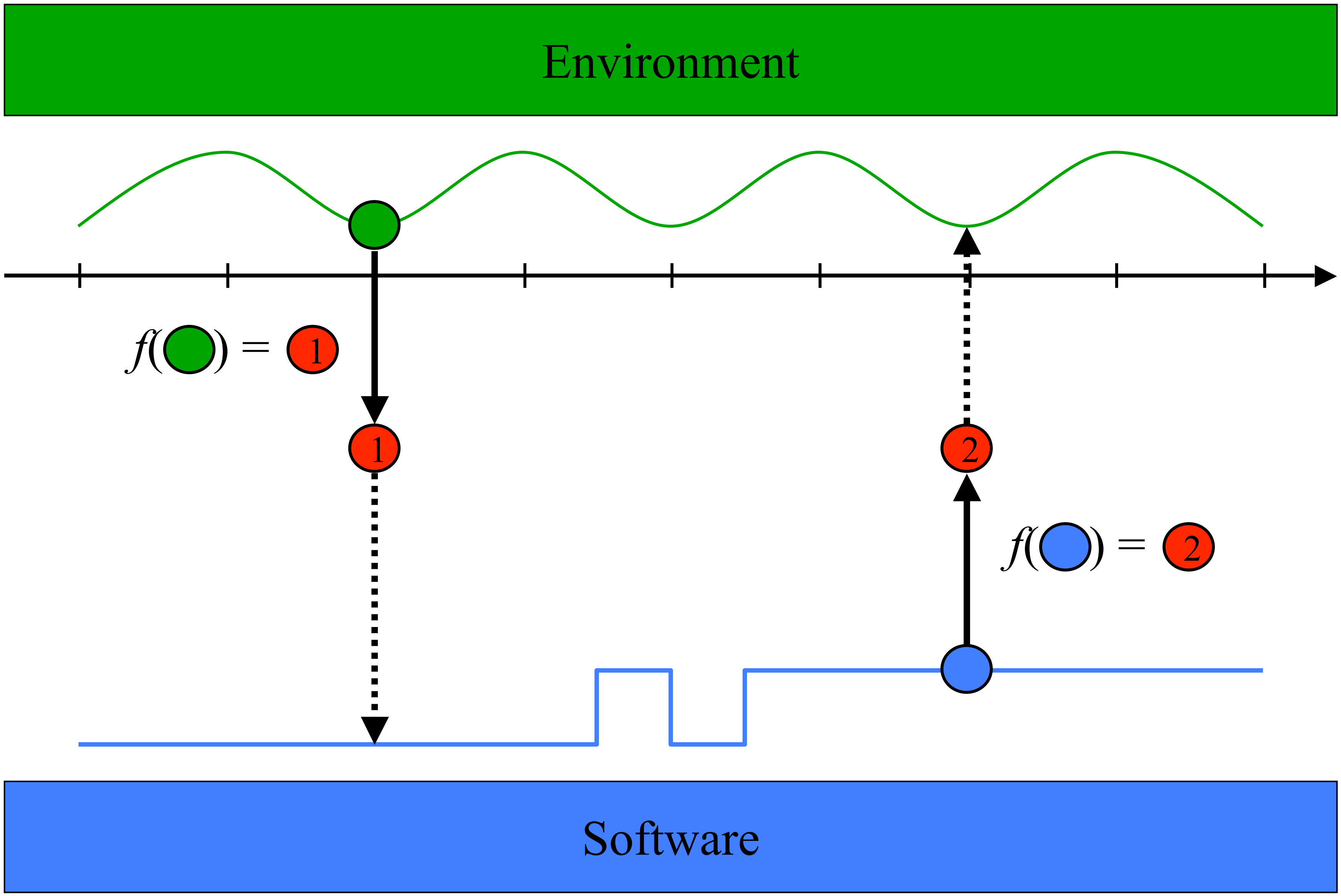
…but Atomic



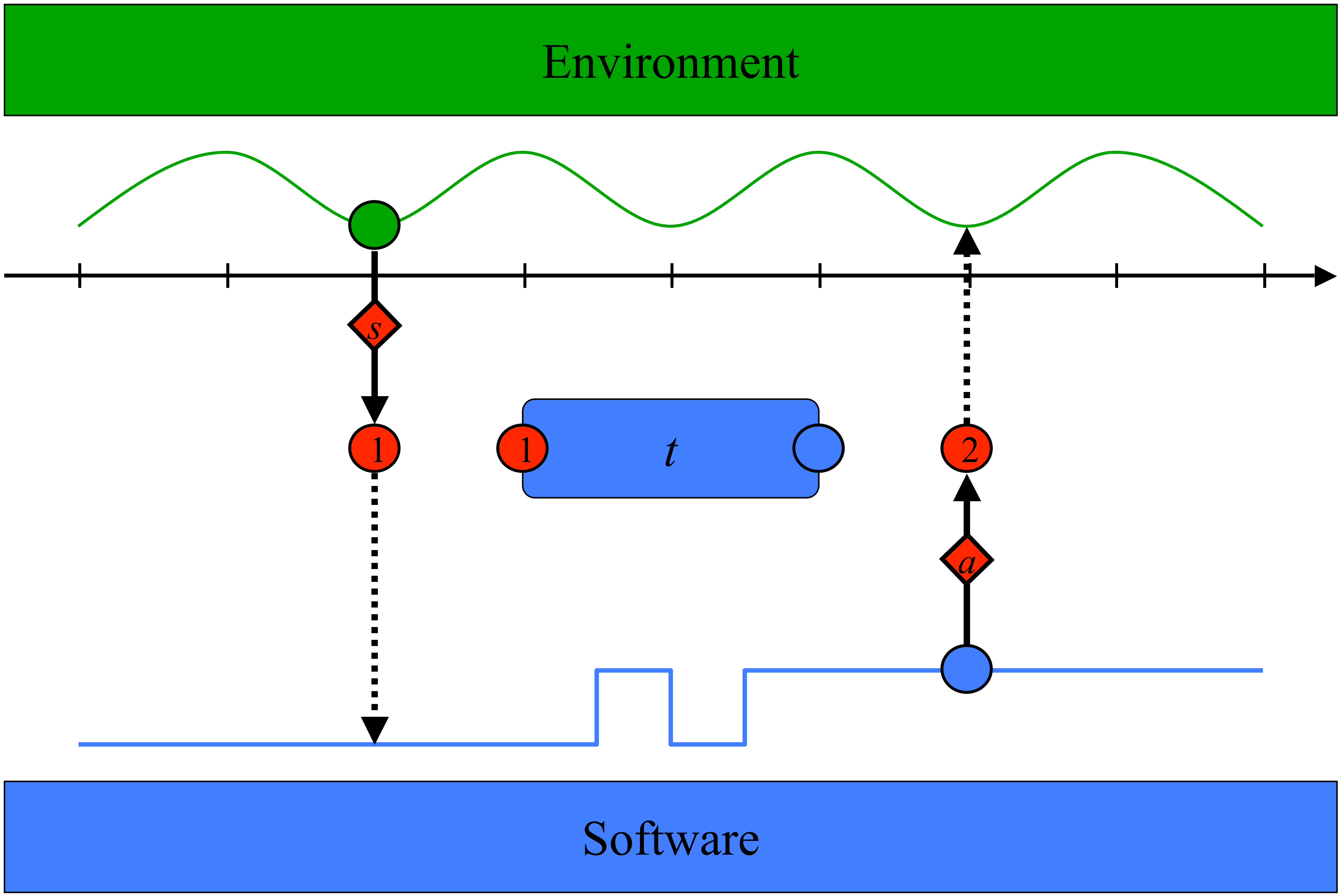
The Driver Model



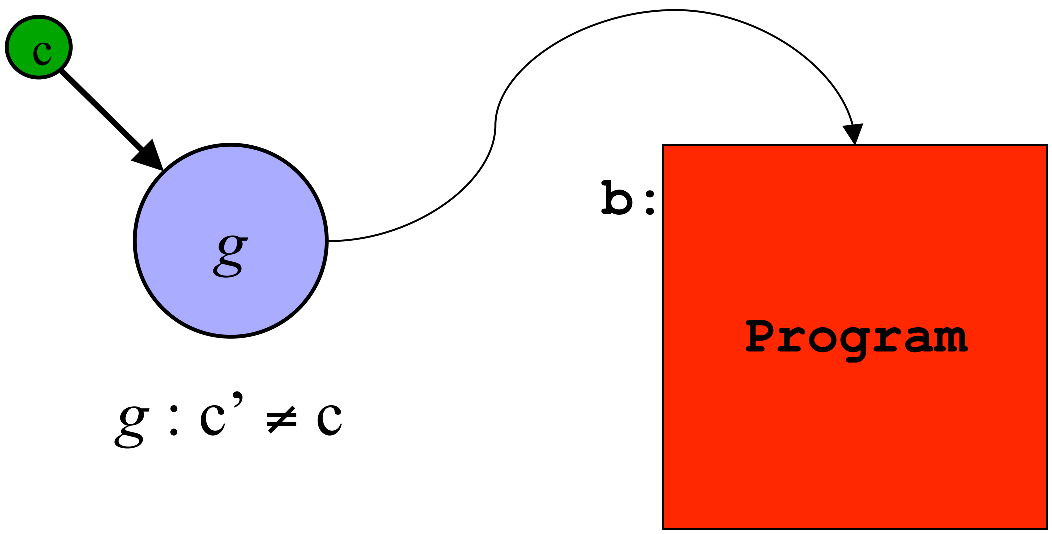
Non-preemptable, Synchronous



Syntax

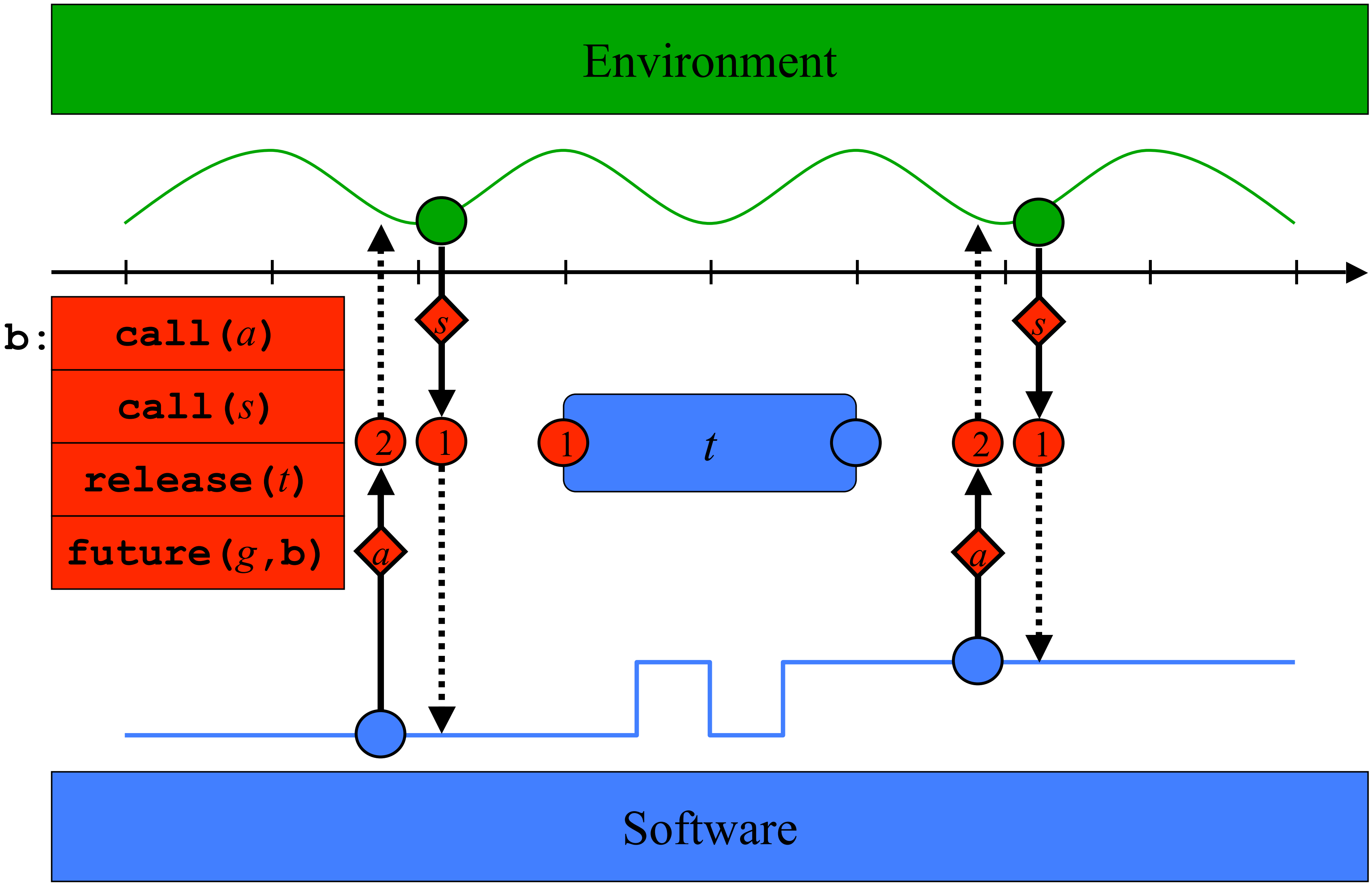


A Trigger *g*

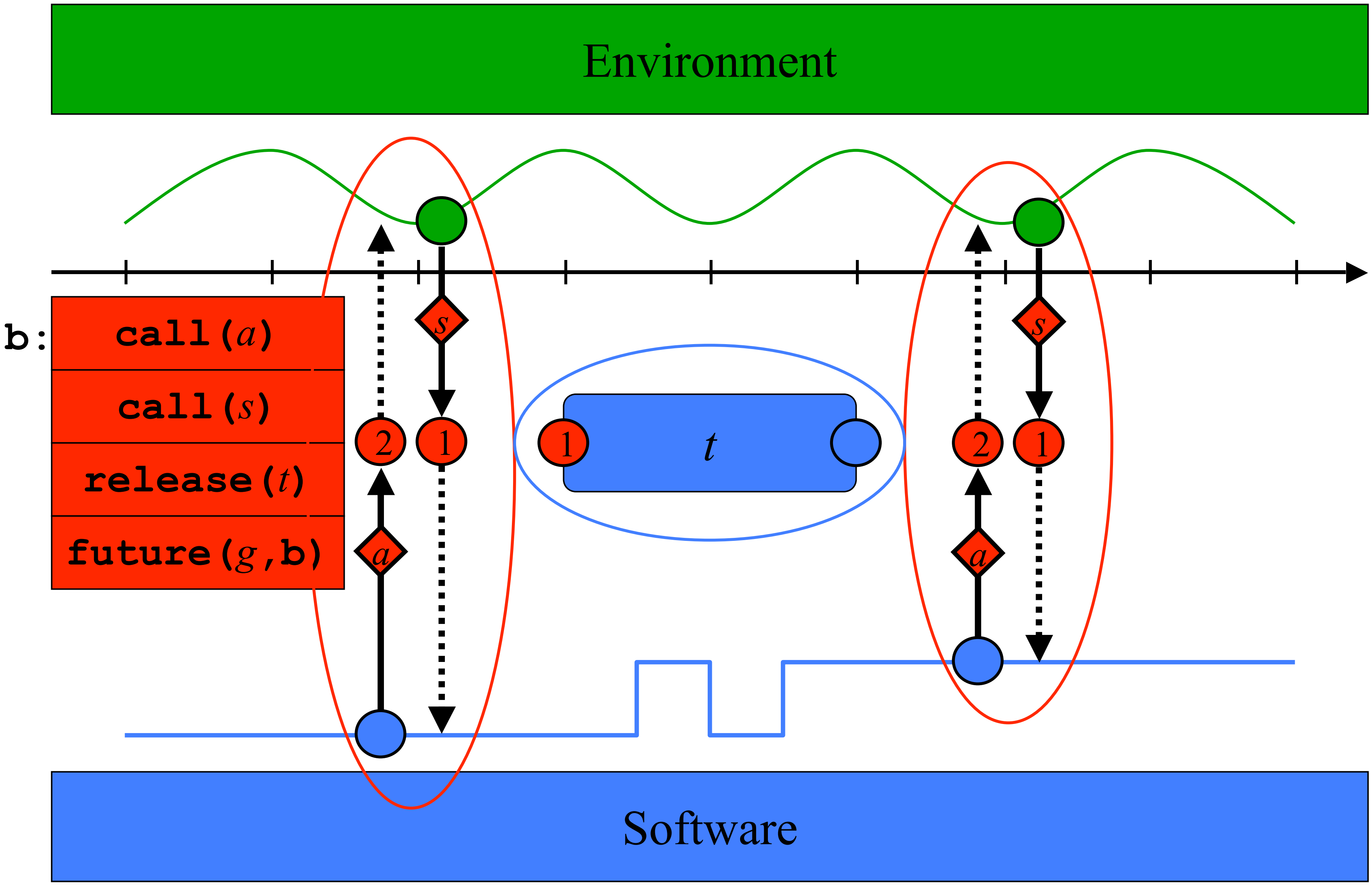


g : c’ ≠ c

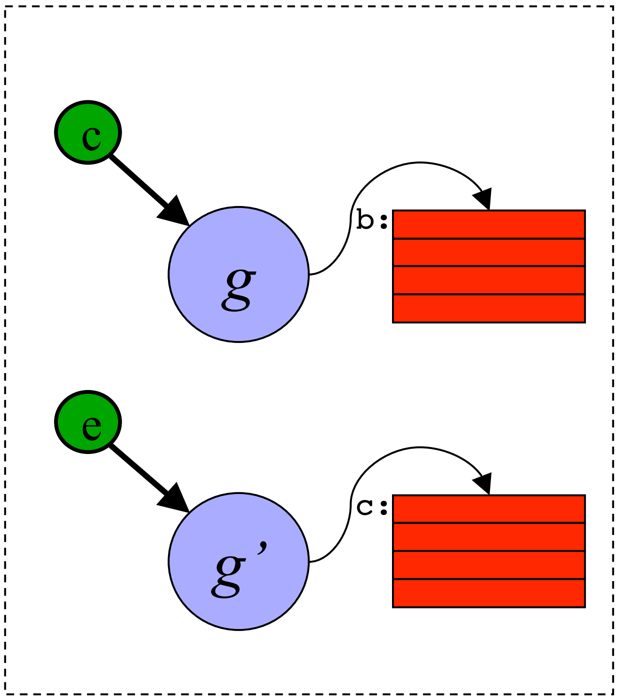
An Embedded Machine Program



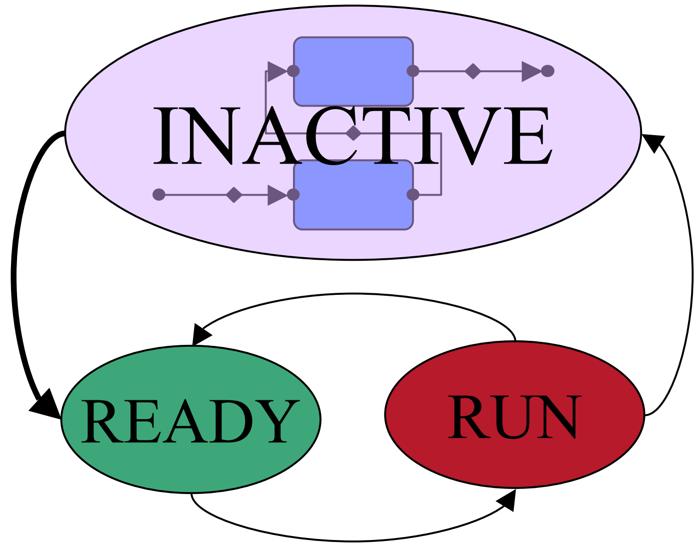
Synchronous vs. Scheduled Computation



Synchronous vs. Scheduled Computation



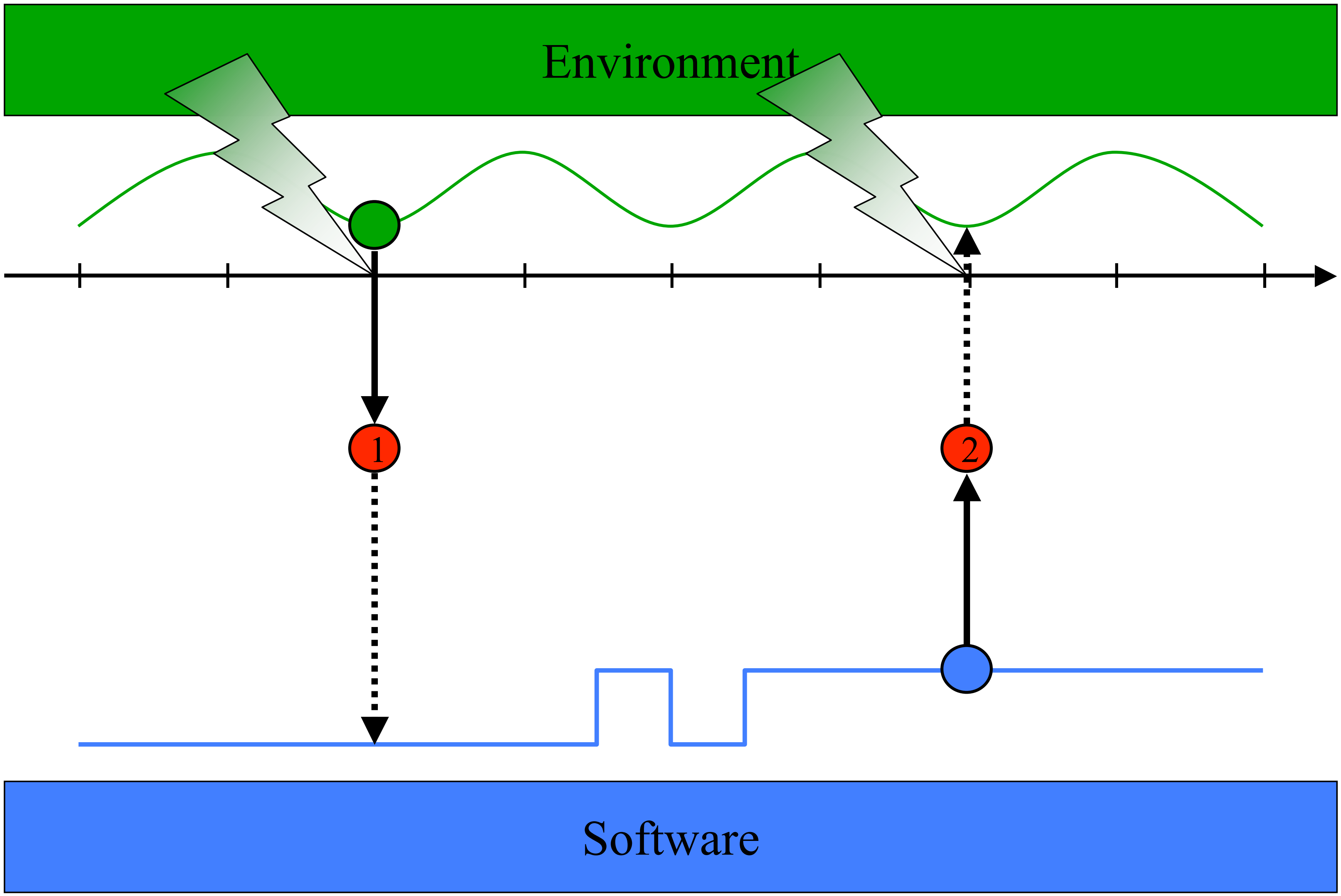
releases



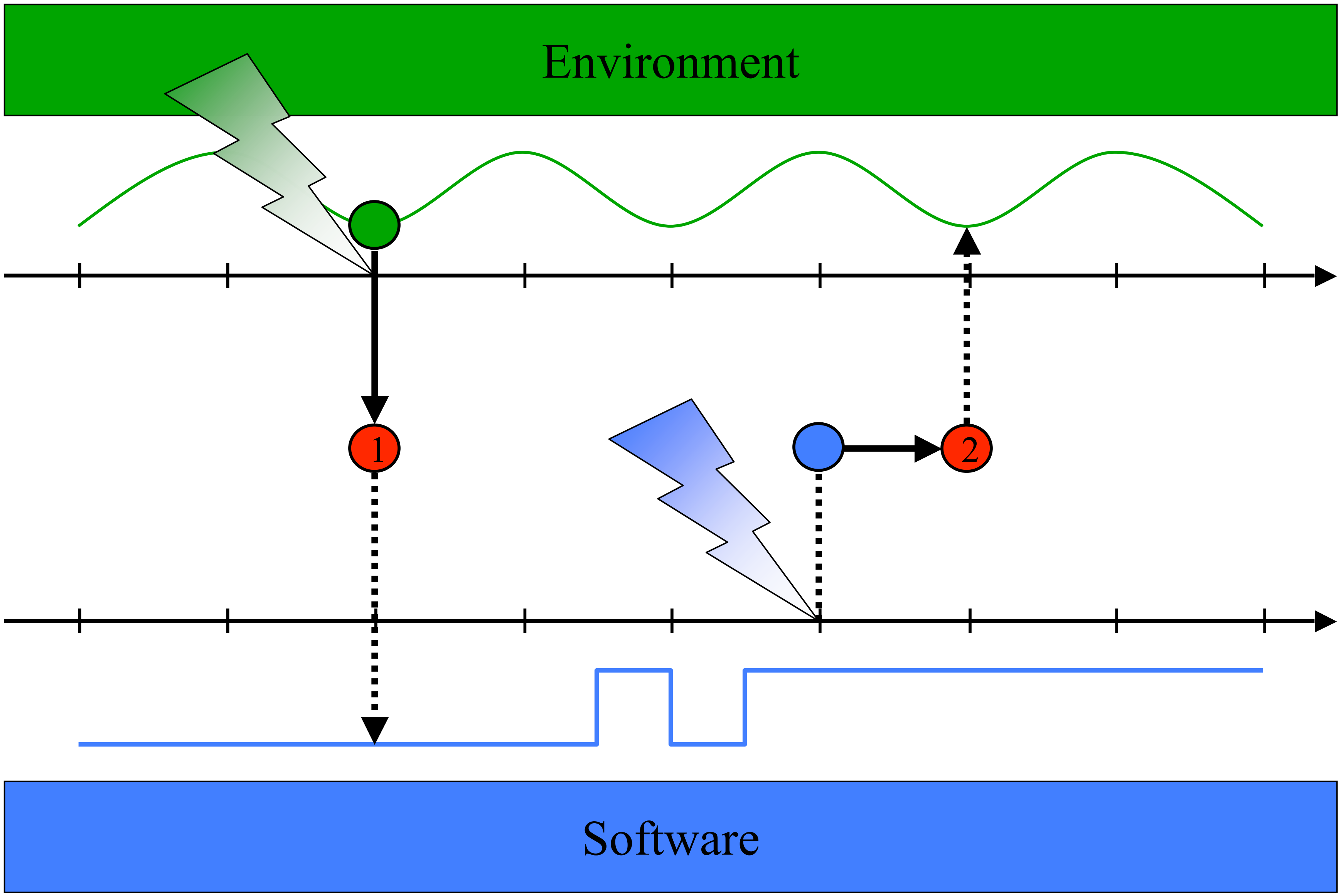
- Synchronous computation
- Kernel context
- Trigger related interrupts disabled

- Scheduled computation
- User context

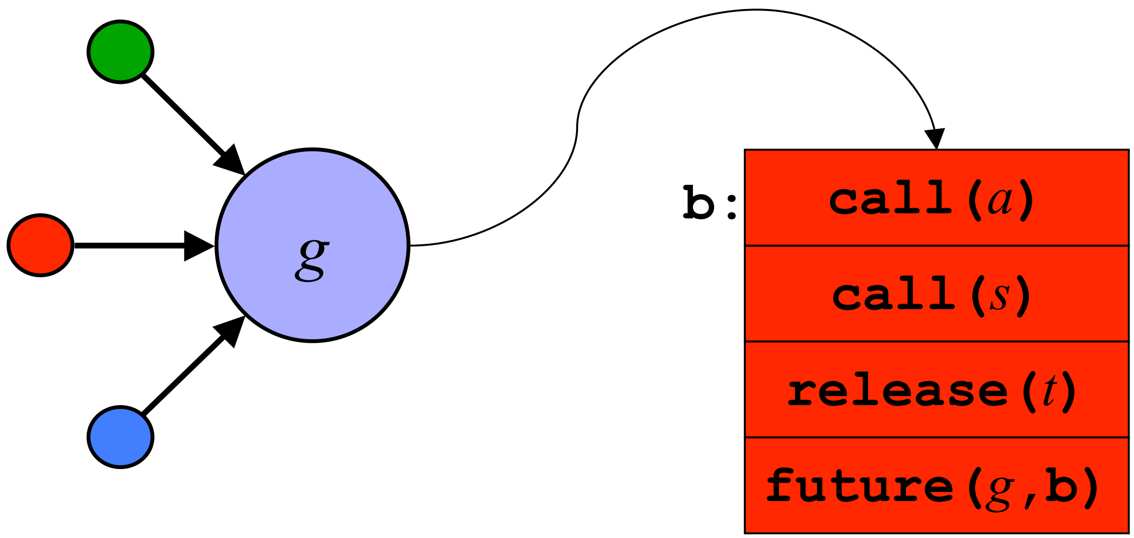
Environment-triggered Code



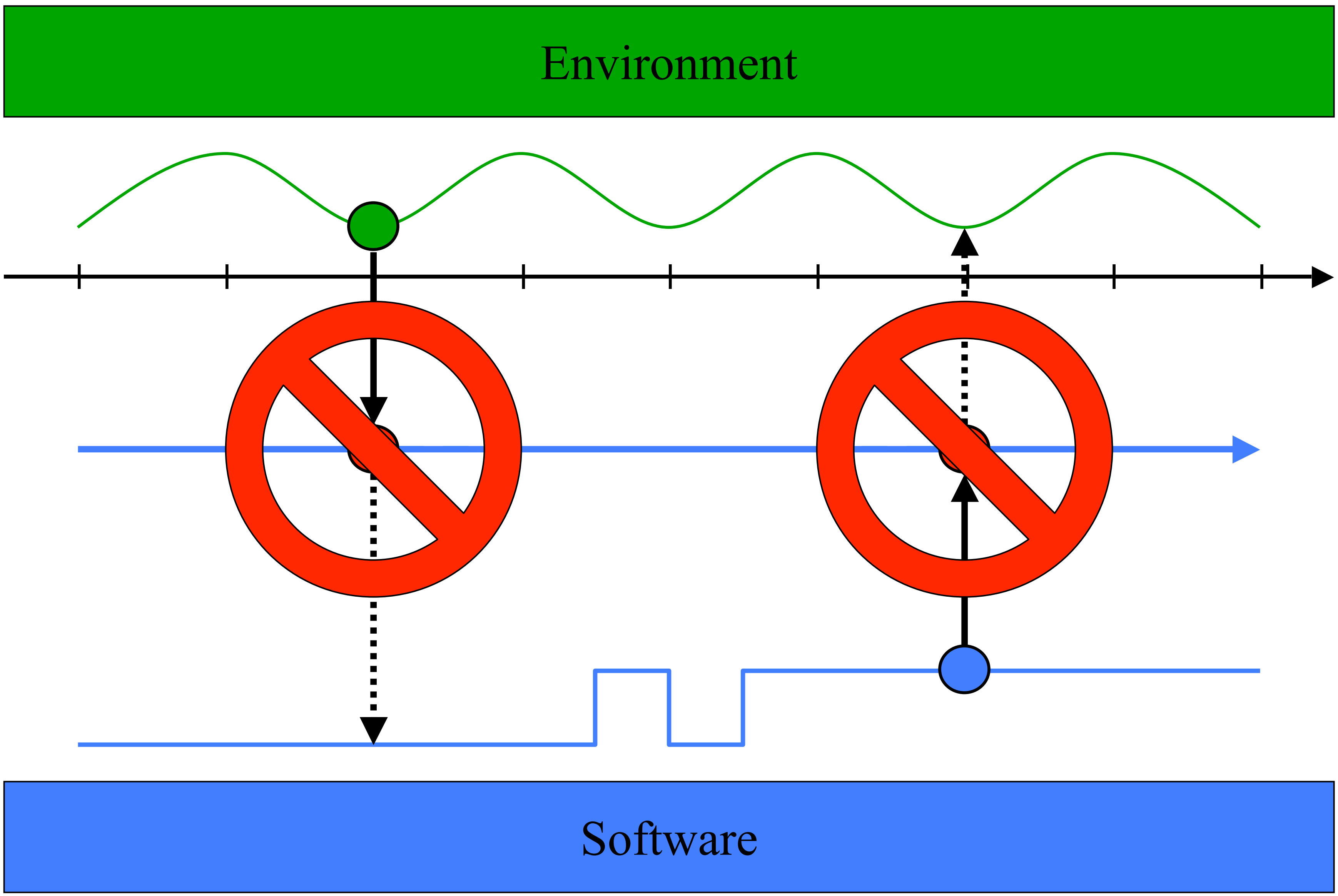
Software-triggered Code



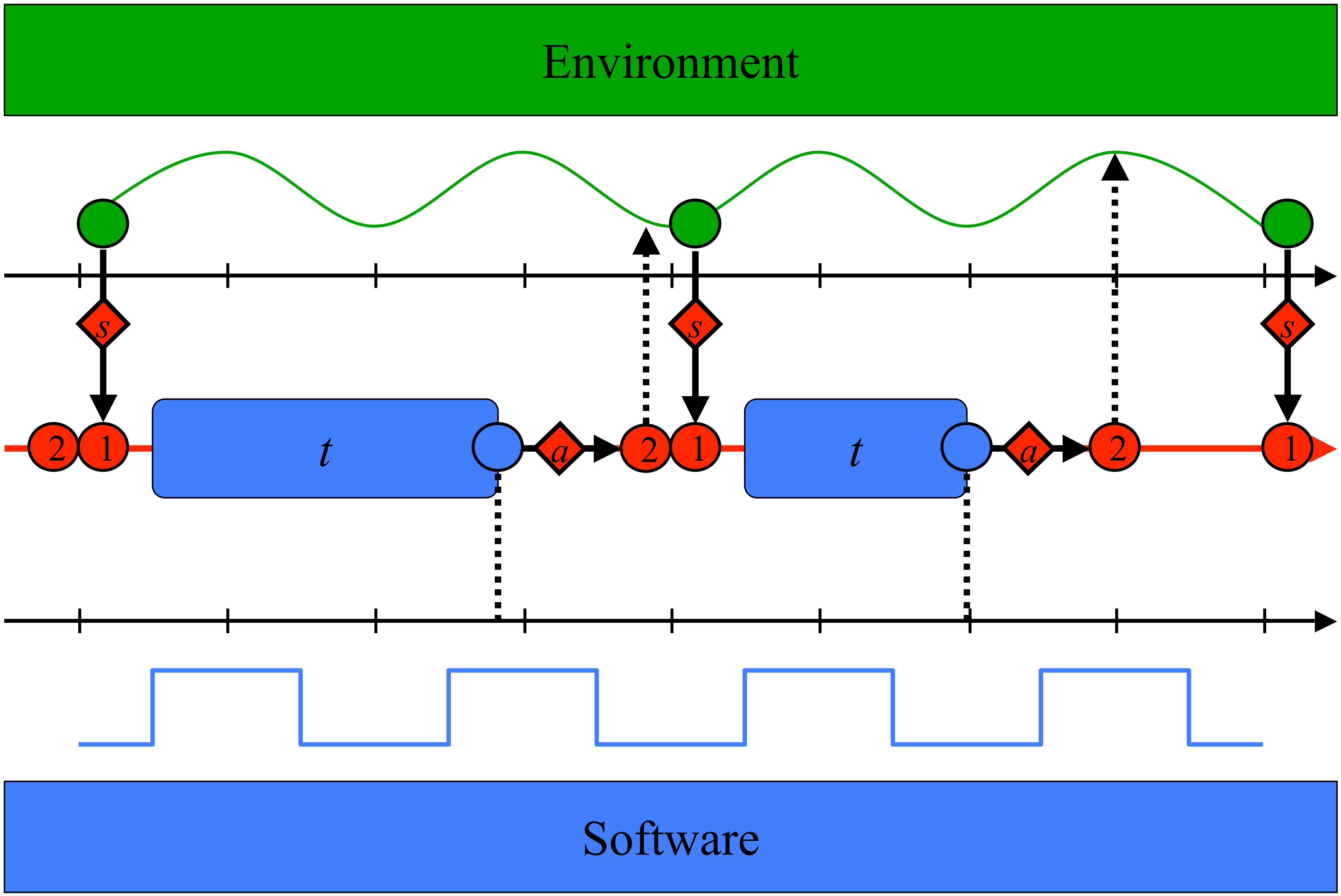
Trigger *g*: Input-, Environment-Triggered



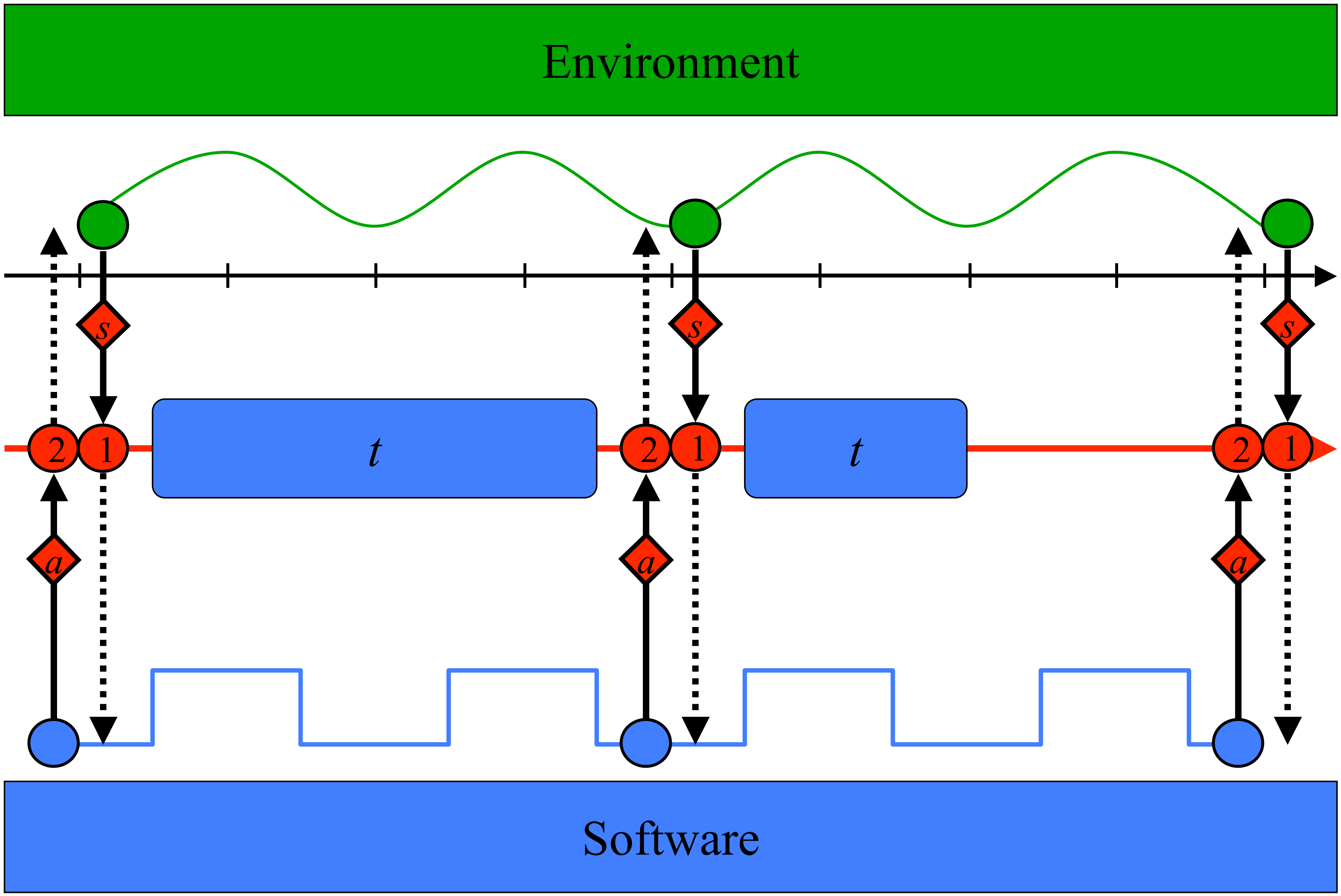
Time Safety



Input-determined If Time Safe

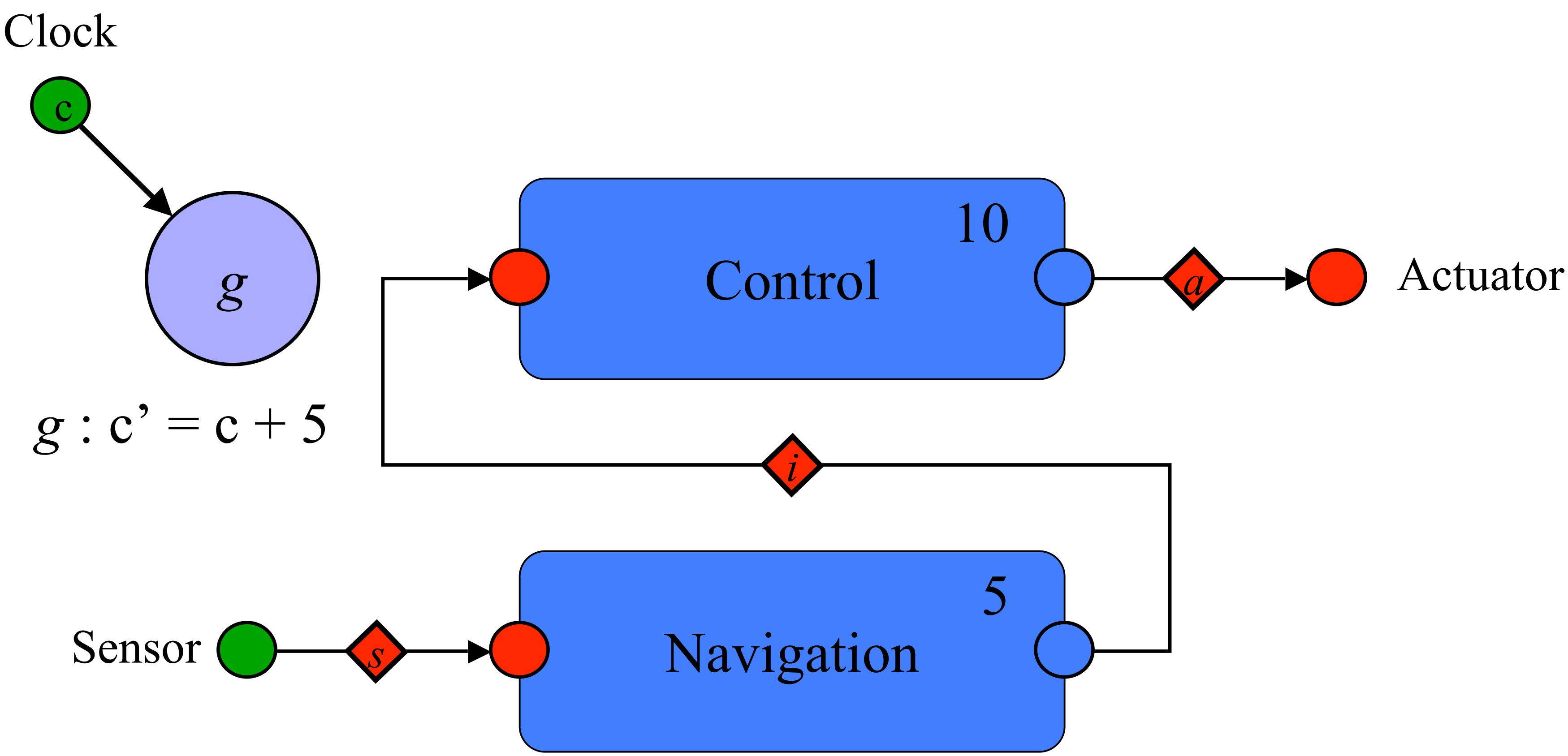


Environment-determined If Environment-triggered

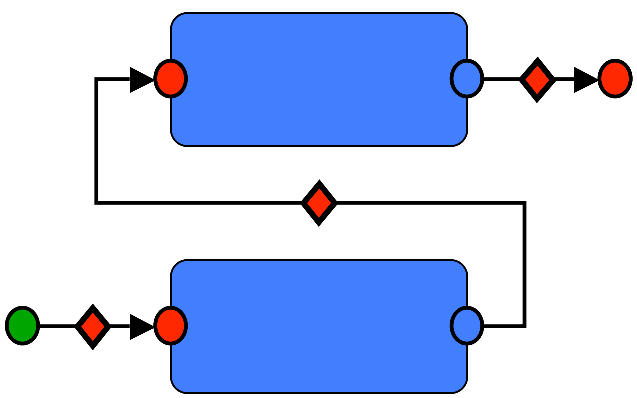


The Zürich Helicopter

Helicopter Control Software



Giotto Syntax (Functionality)



sensor gps_type GPS uses c_gps_device ;

actuator servo_type Servo := c_servo_init
uses c_servo_device ;

output

ctr_type CtrOutput := c_ctr_init ;

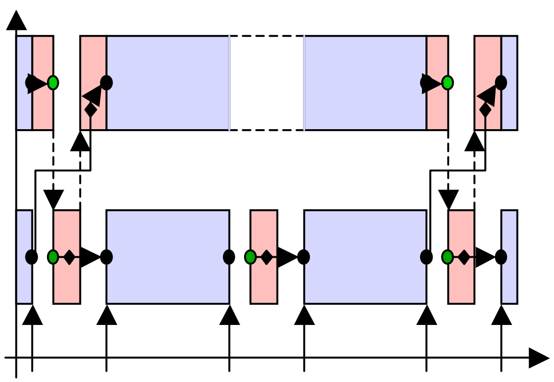
nav_type NavOutput := c_nav_init ;

driver sensing (GPS) output (gps_type gps)
{ c_gps_pre_processing (GPS, gps) }

task Navigation (gps_type gps) output (NavOutput)
{ c_matlab_navigation_code (gps, NavOutput) }

...

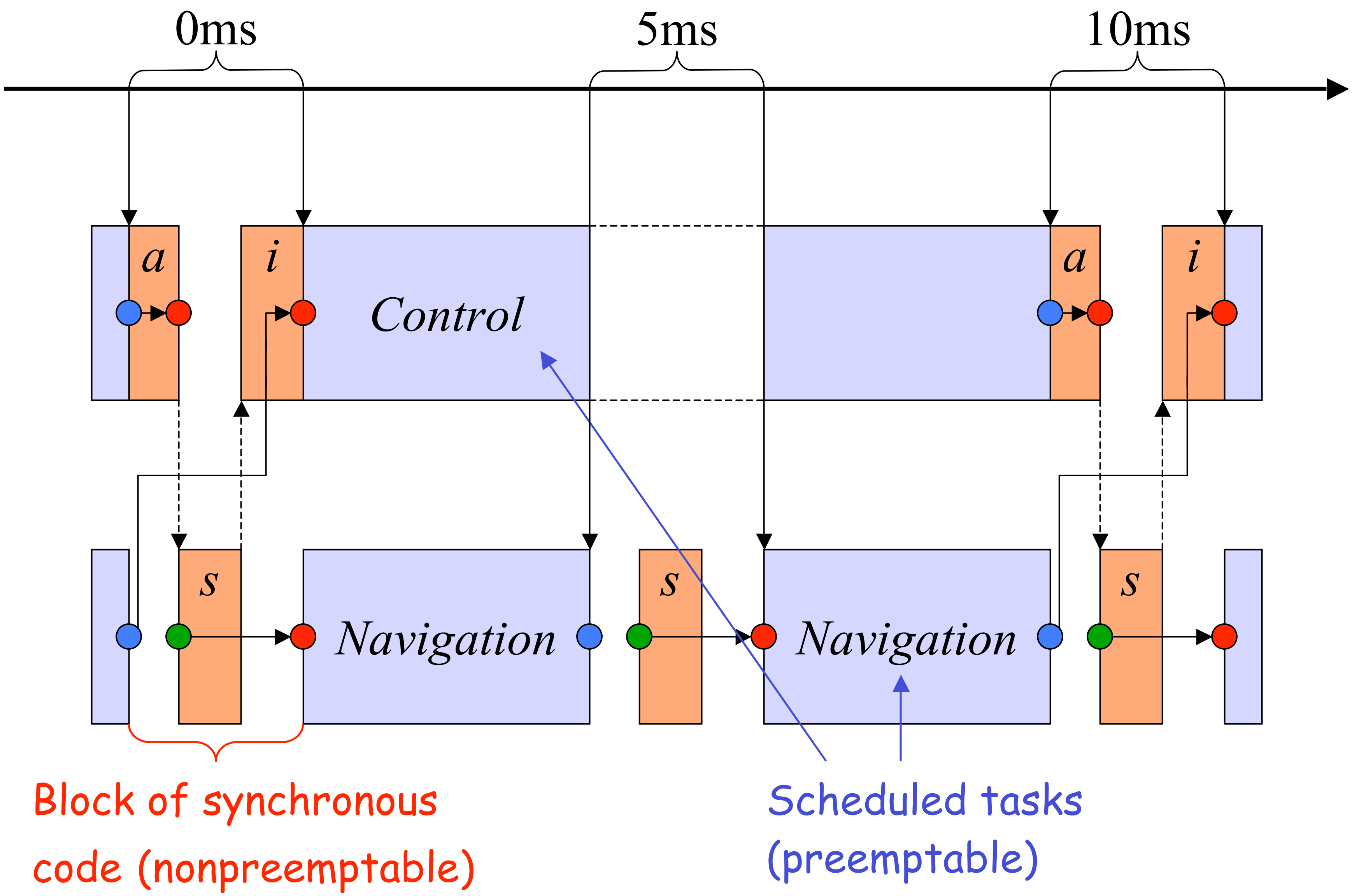
Giotto Syntax (Timing)



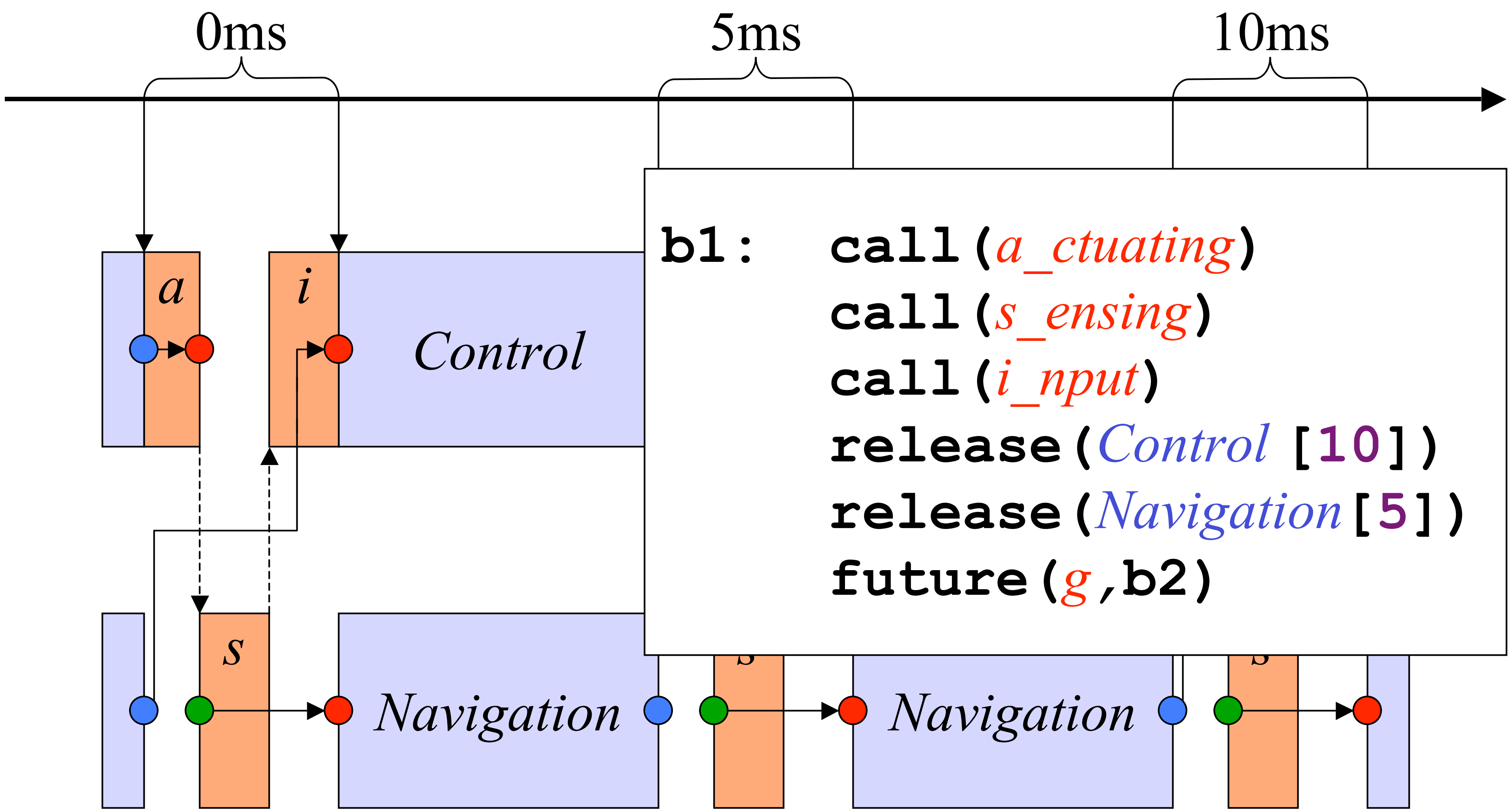
```
…
mode Flight ( ) period 10ms
  {
      actfreq 1 do Servo ( actuating ) ;

      taskfreq 1 do Control ( input ) ;
      taskfreq 2 do Navigation ( sensing ) ;
  }
…
```

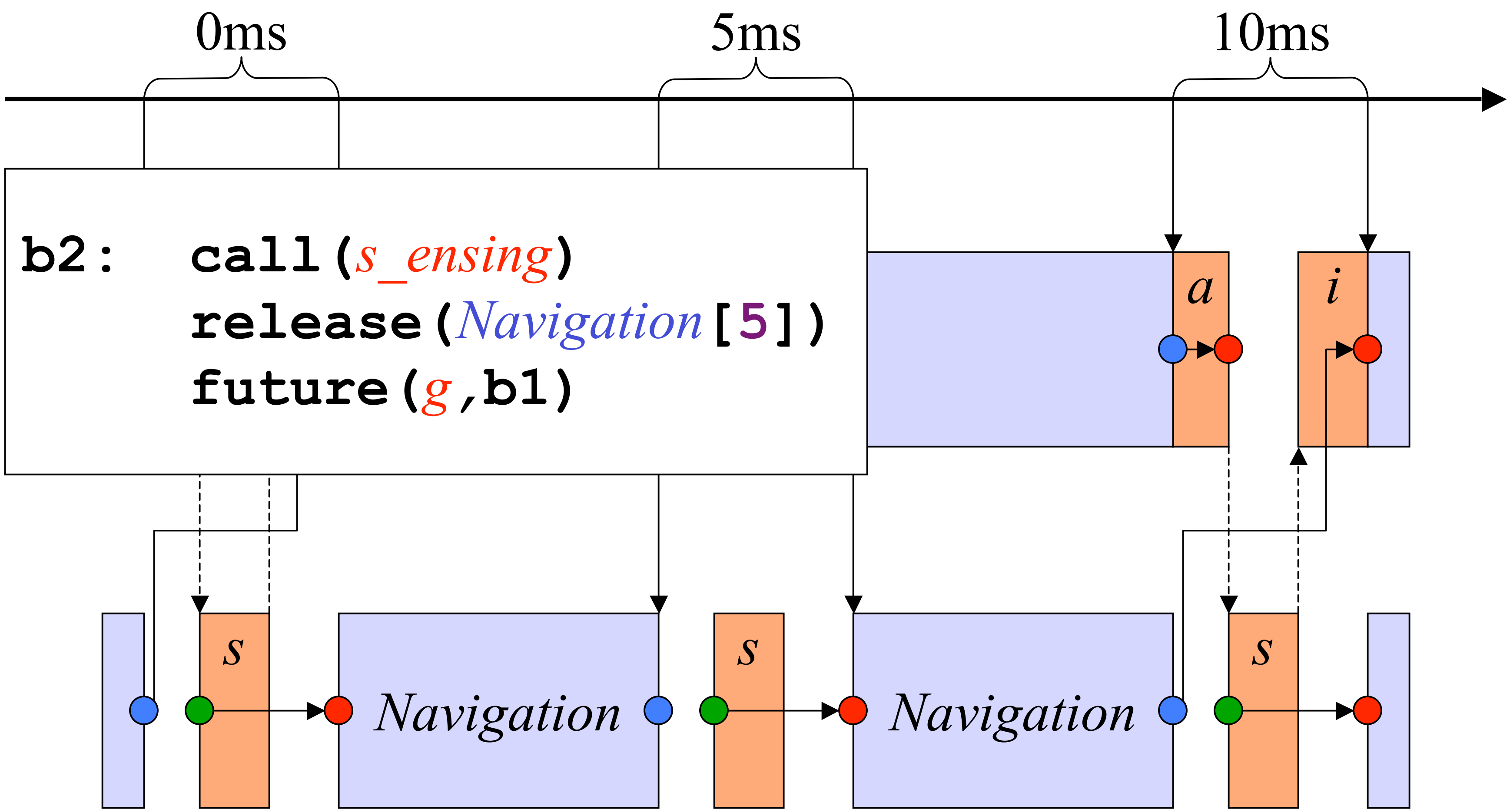
Environment Timeline



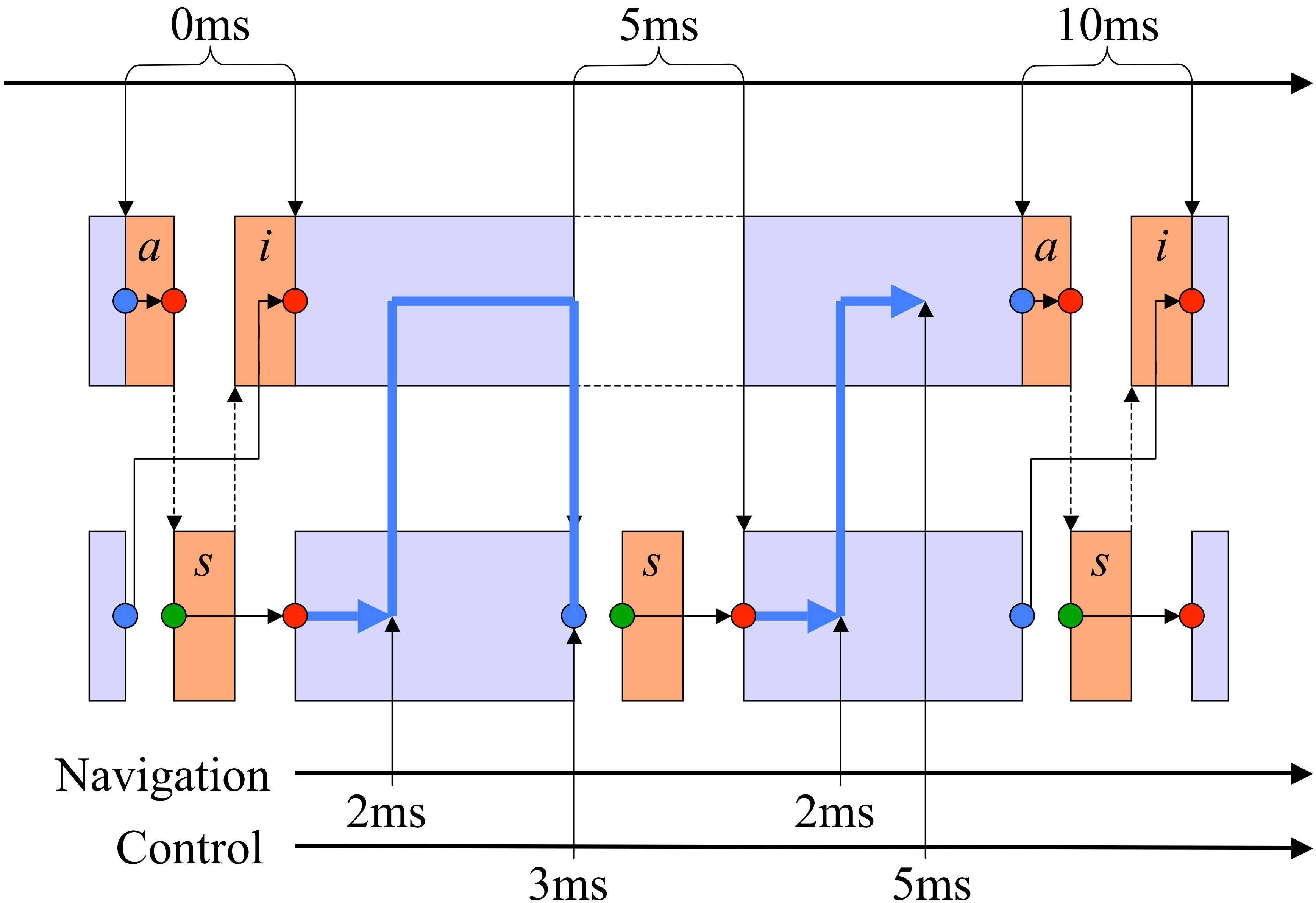
E Code


0ms
5ms
10ms
a
i
Control
s
Navigation
Navigation
b1: call(a_ctuating)
call(s_ensing)
call(i_nput)
release(Control [10])
release(Navigation[5])
future(g,b2)

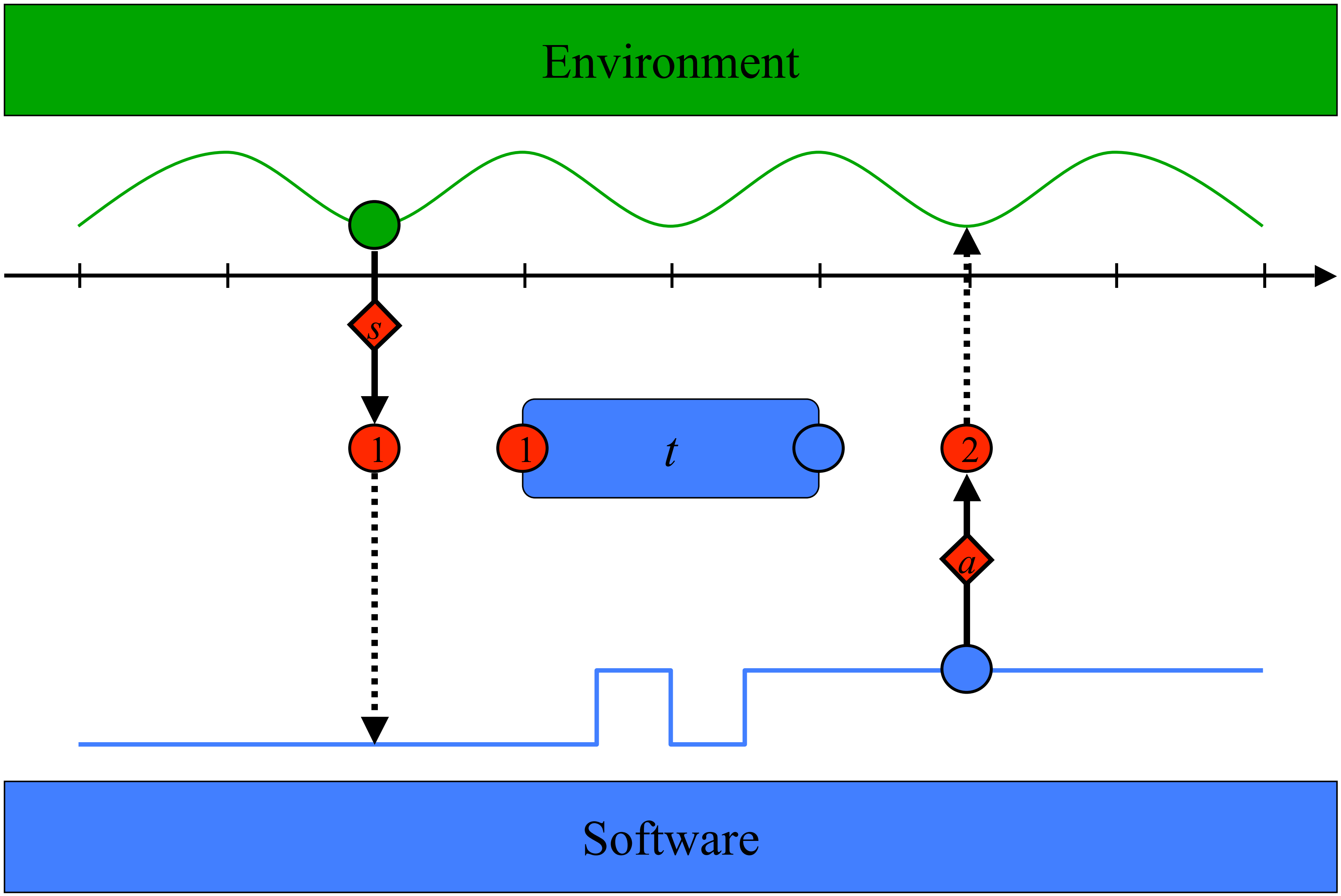
E Code


0ms
5ms
10ms
b2: call(s_ensing)
release(Navigation[5])
future(g,b1)
a
i
s
Navigation
s
Navigation
s

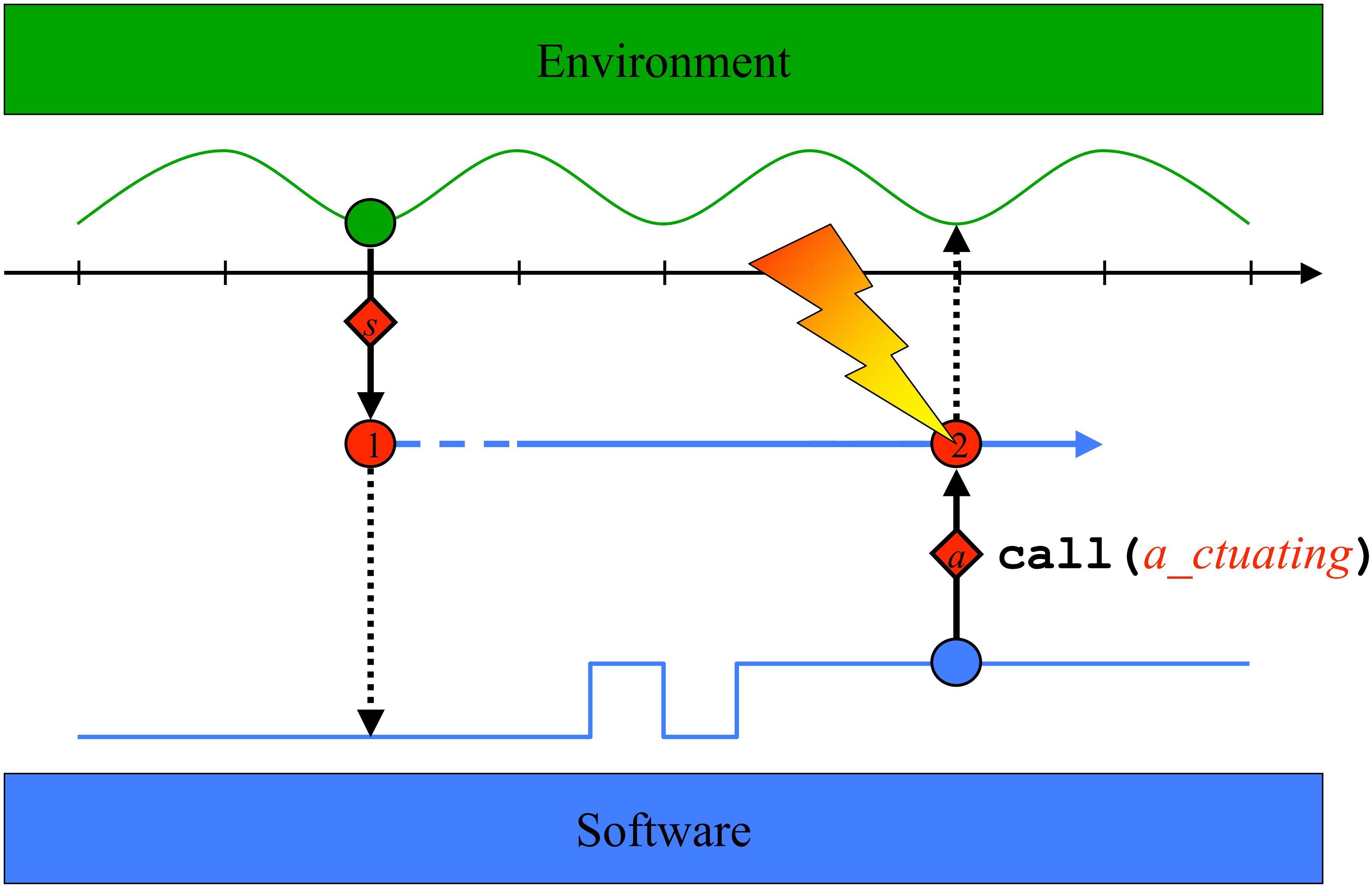
Platform Timeline: EDF



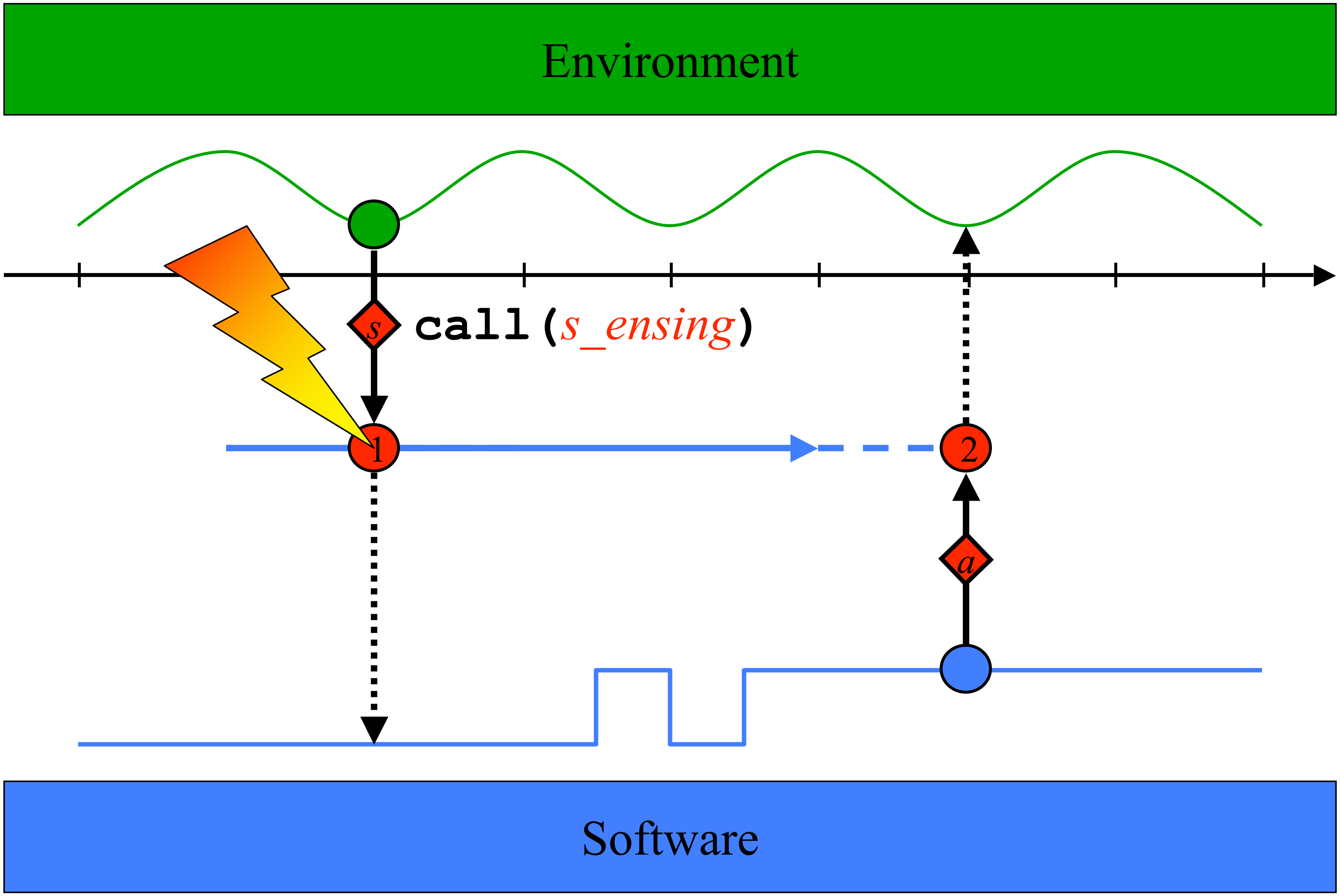
Time Safety



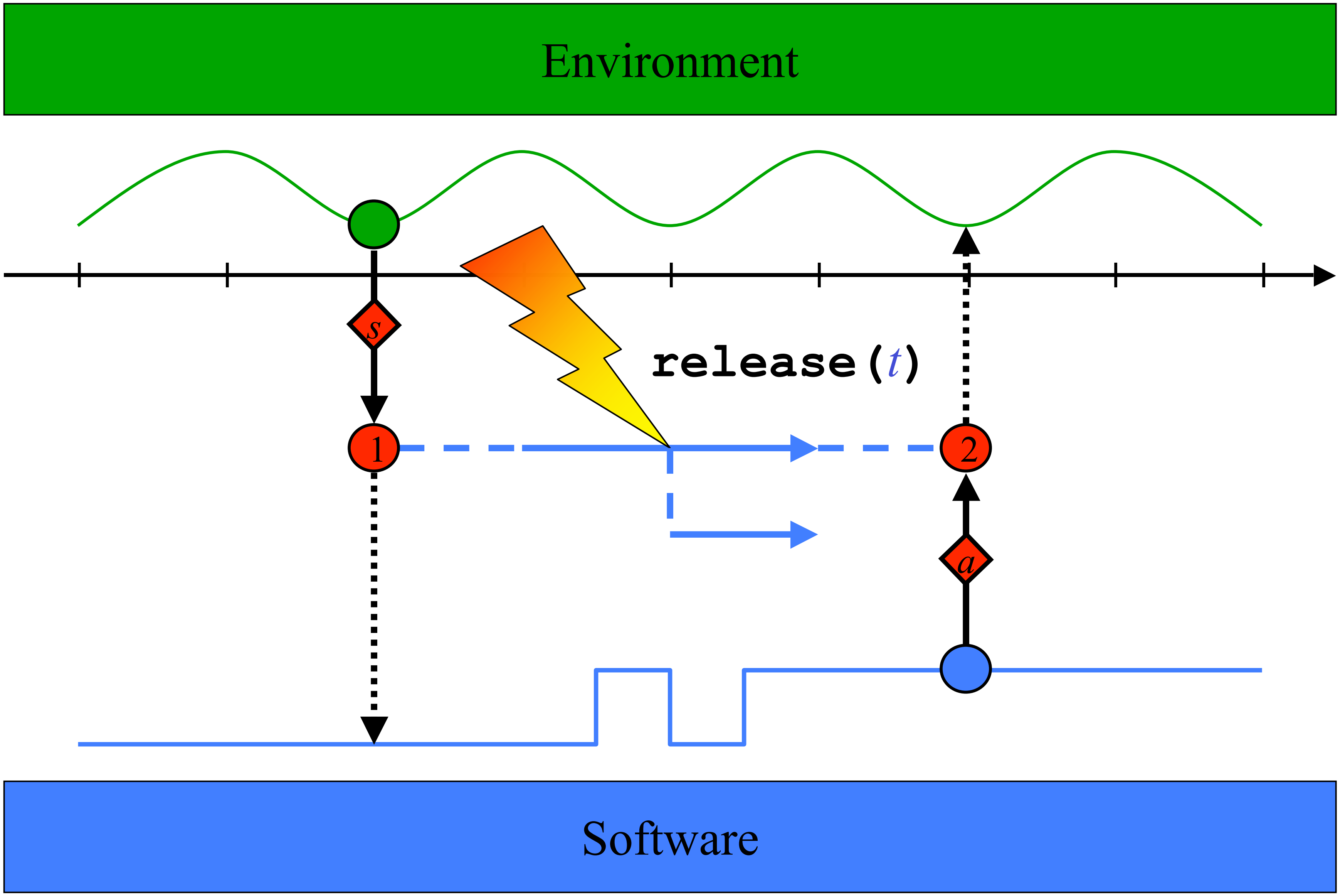
Runtime Exceptions I



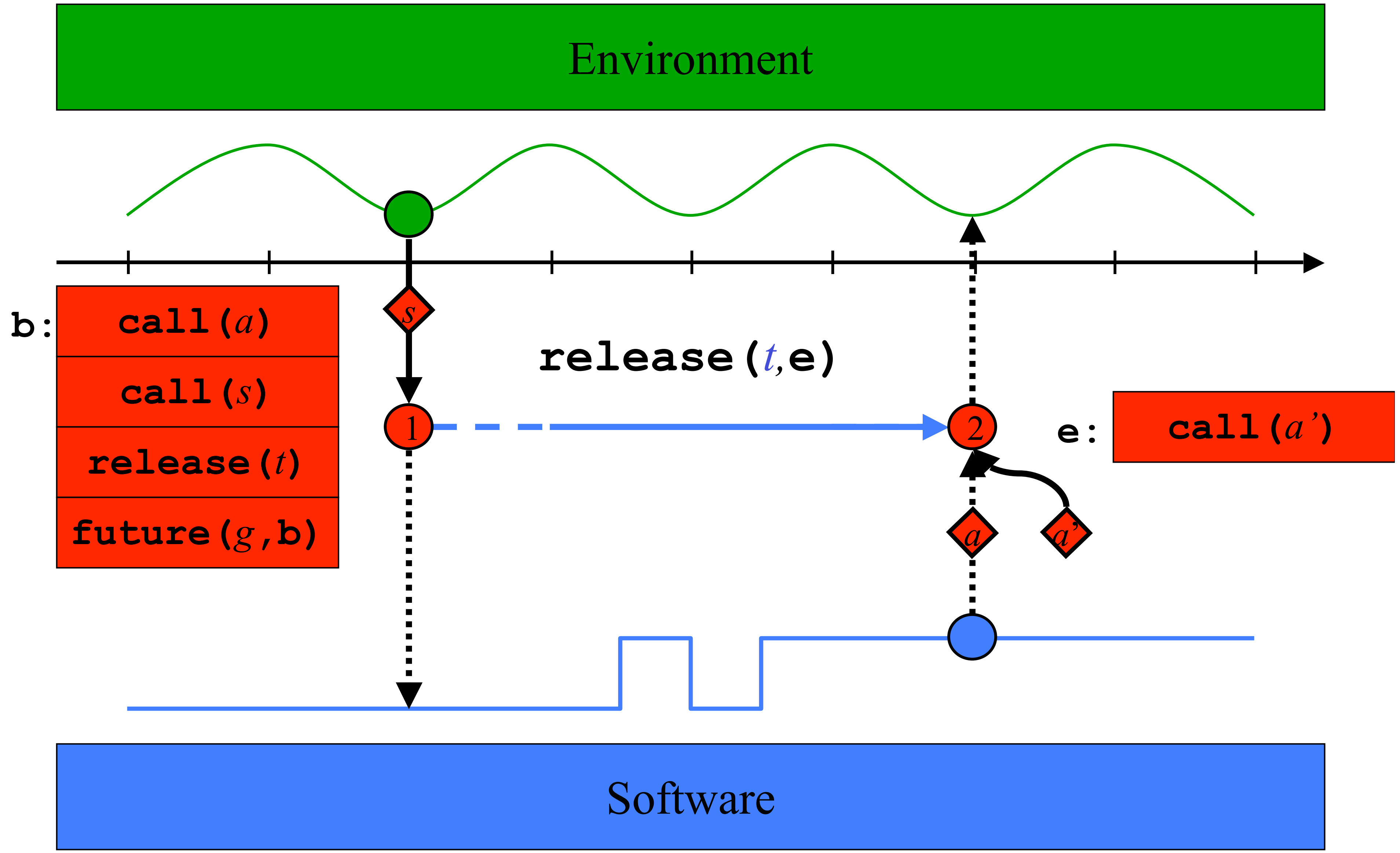
Runtime Exceptions II



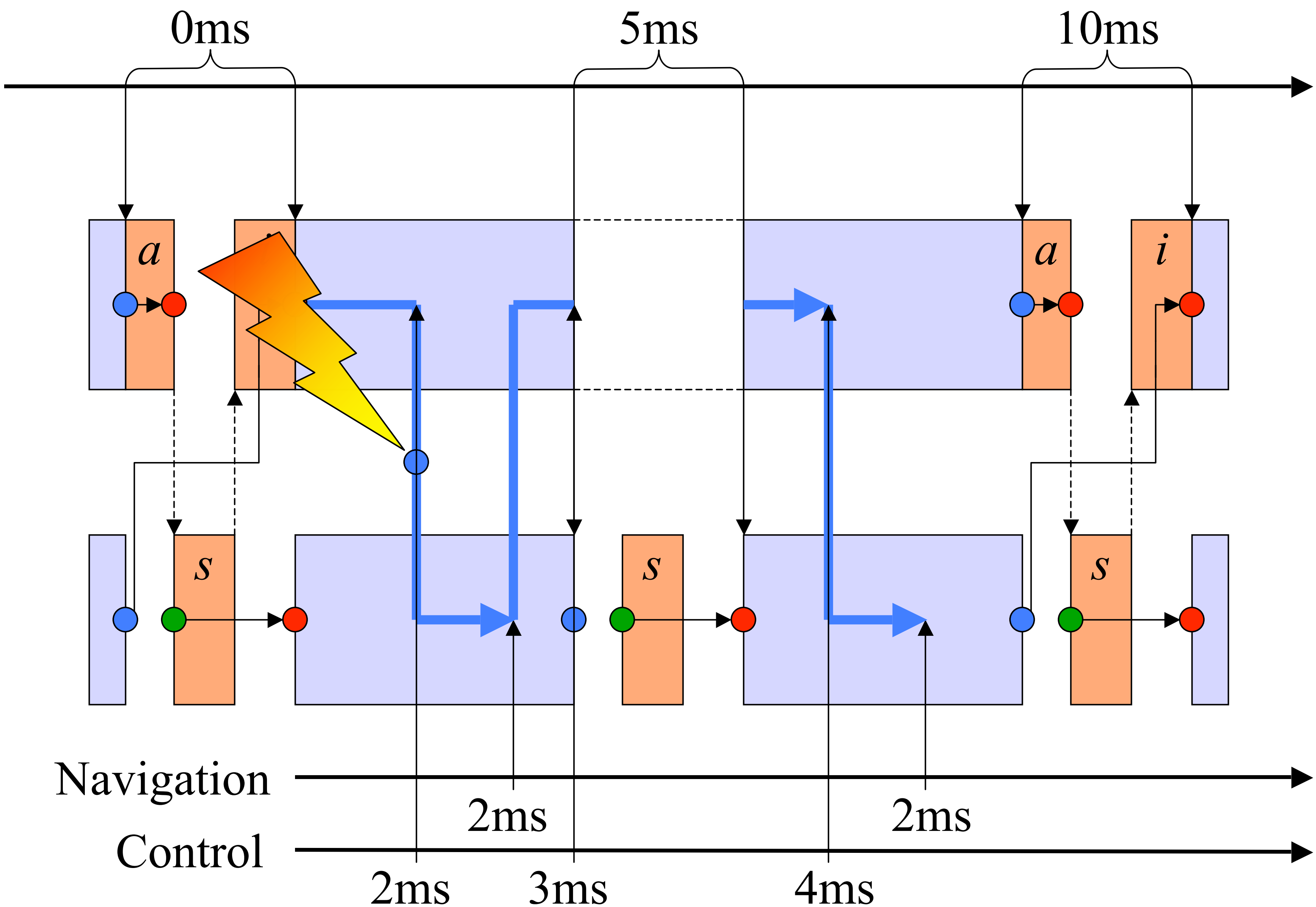
Runtime Exceptions III



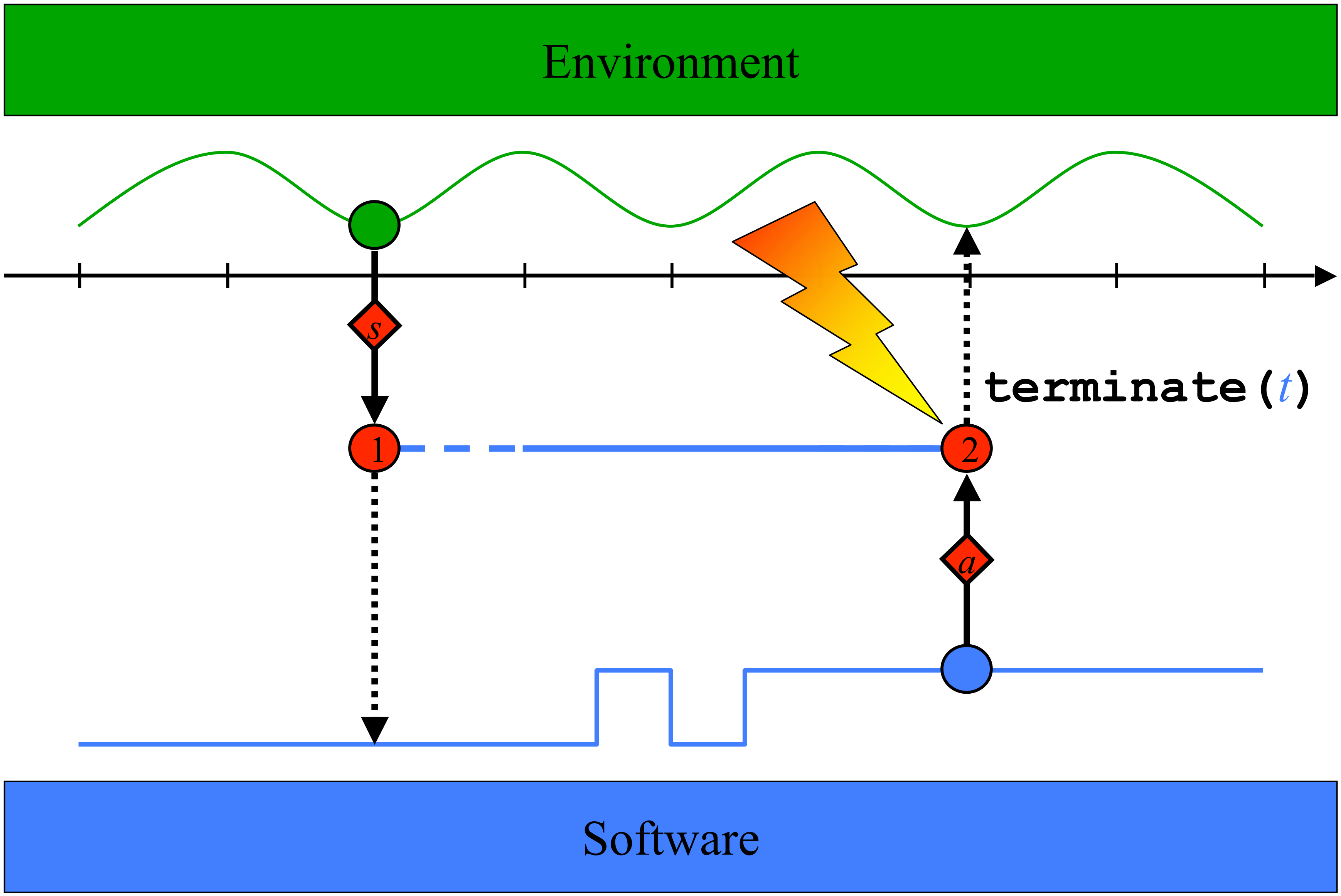
An Exception Handler e



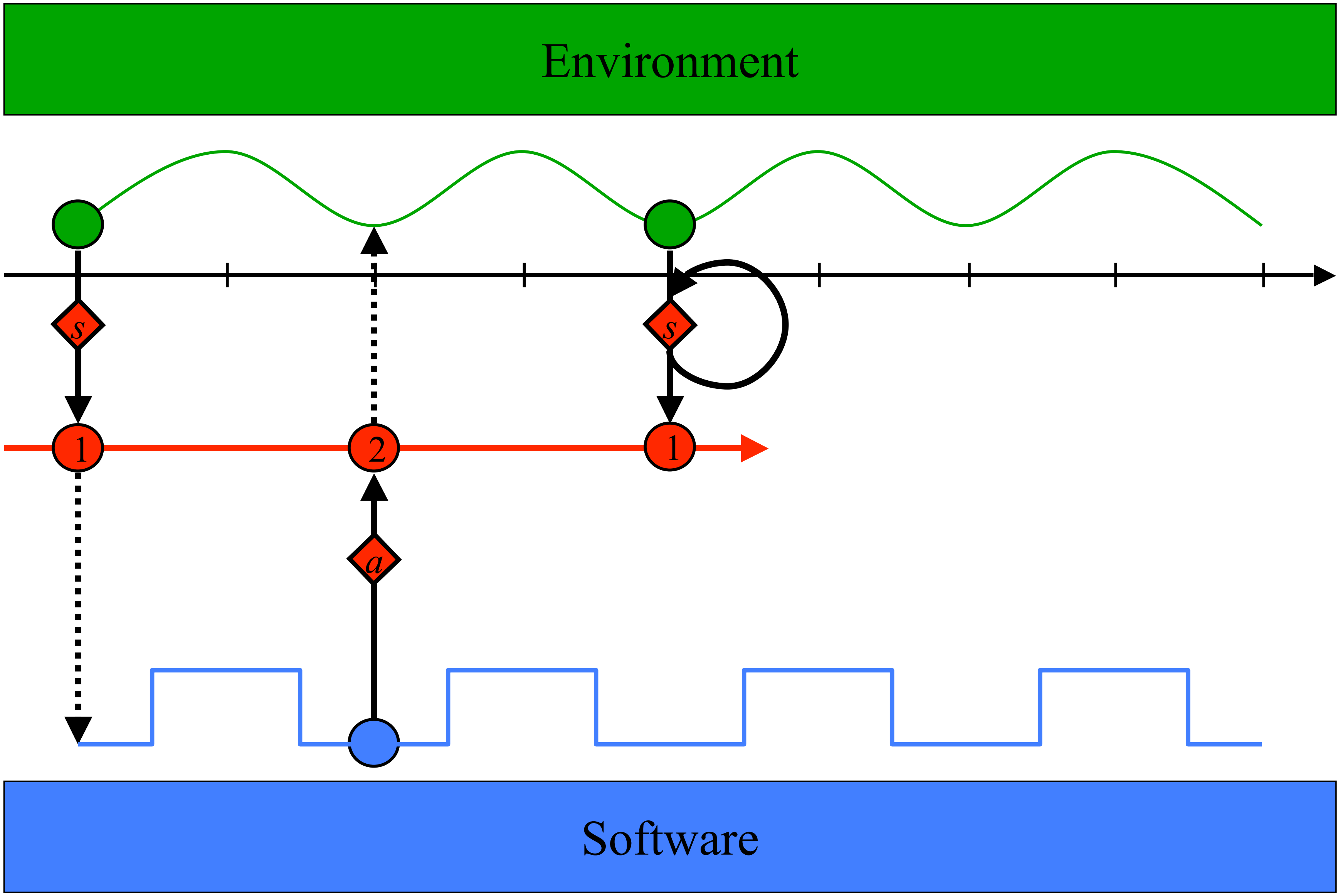
How to Loose Determinism: Task Synchronization



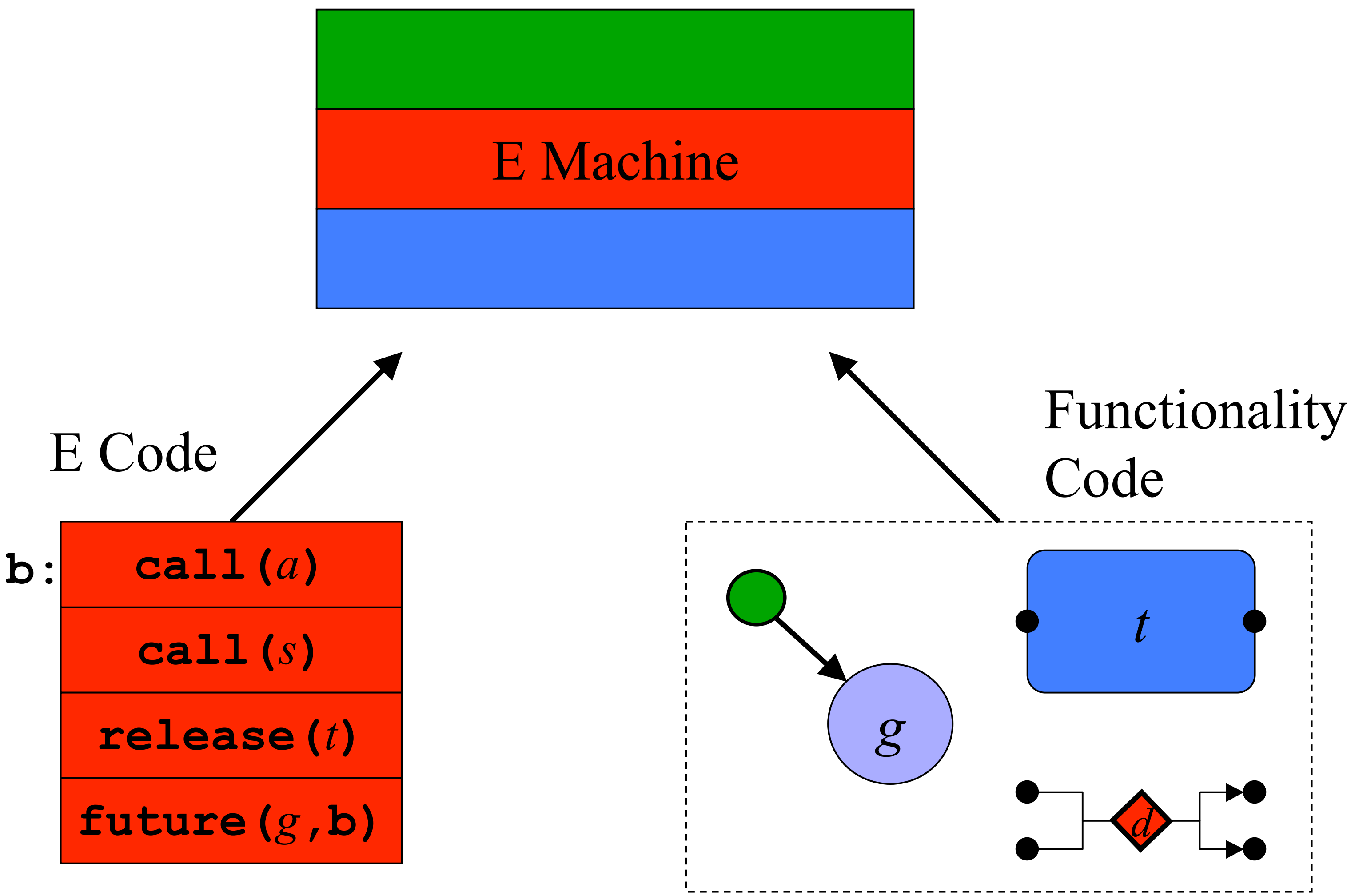
How to Loose Determinism: Termination



Time Liveness: Infinite Traces

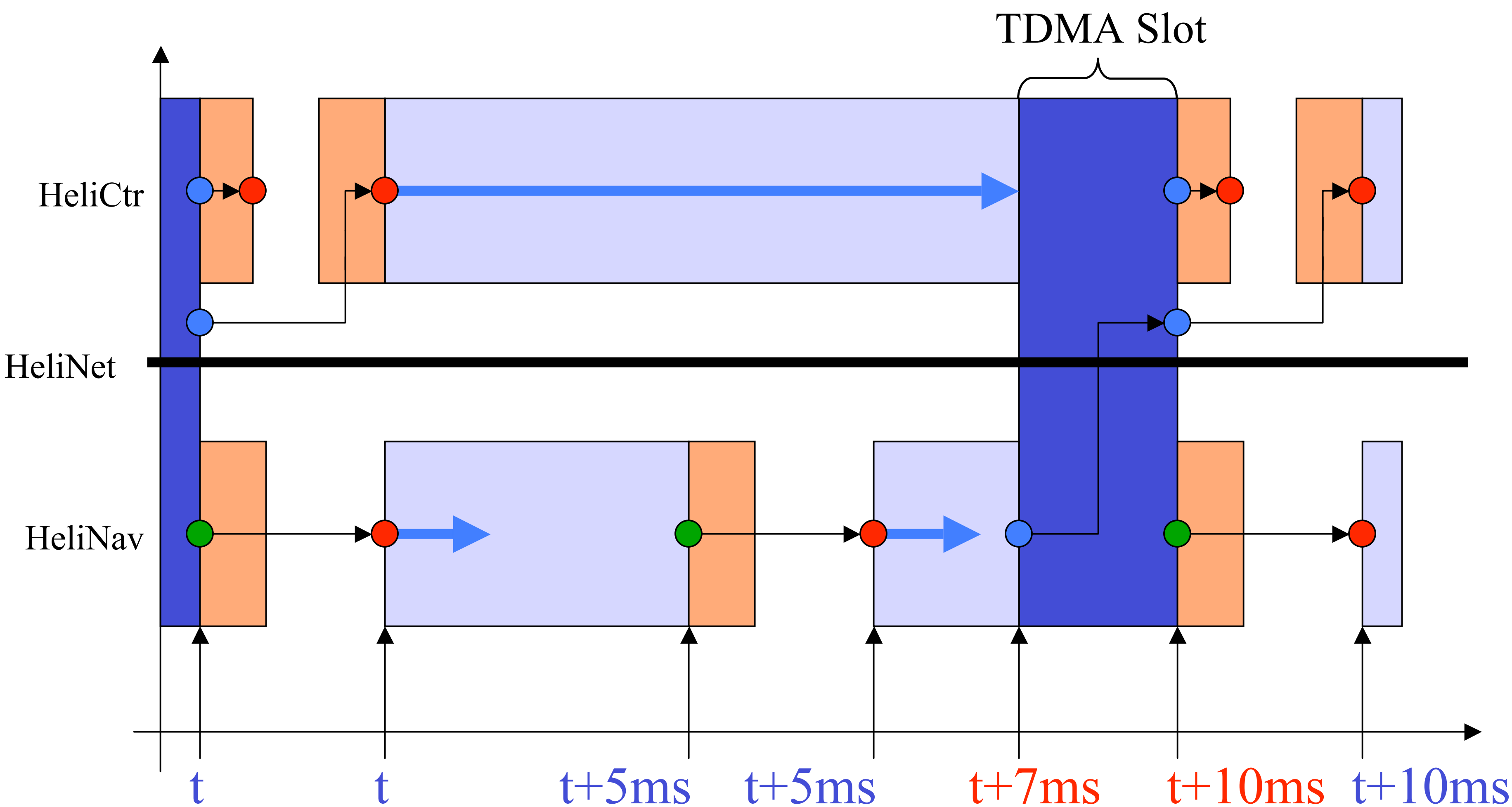


Dynamic Linking



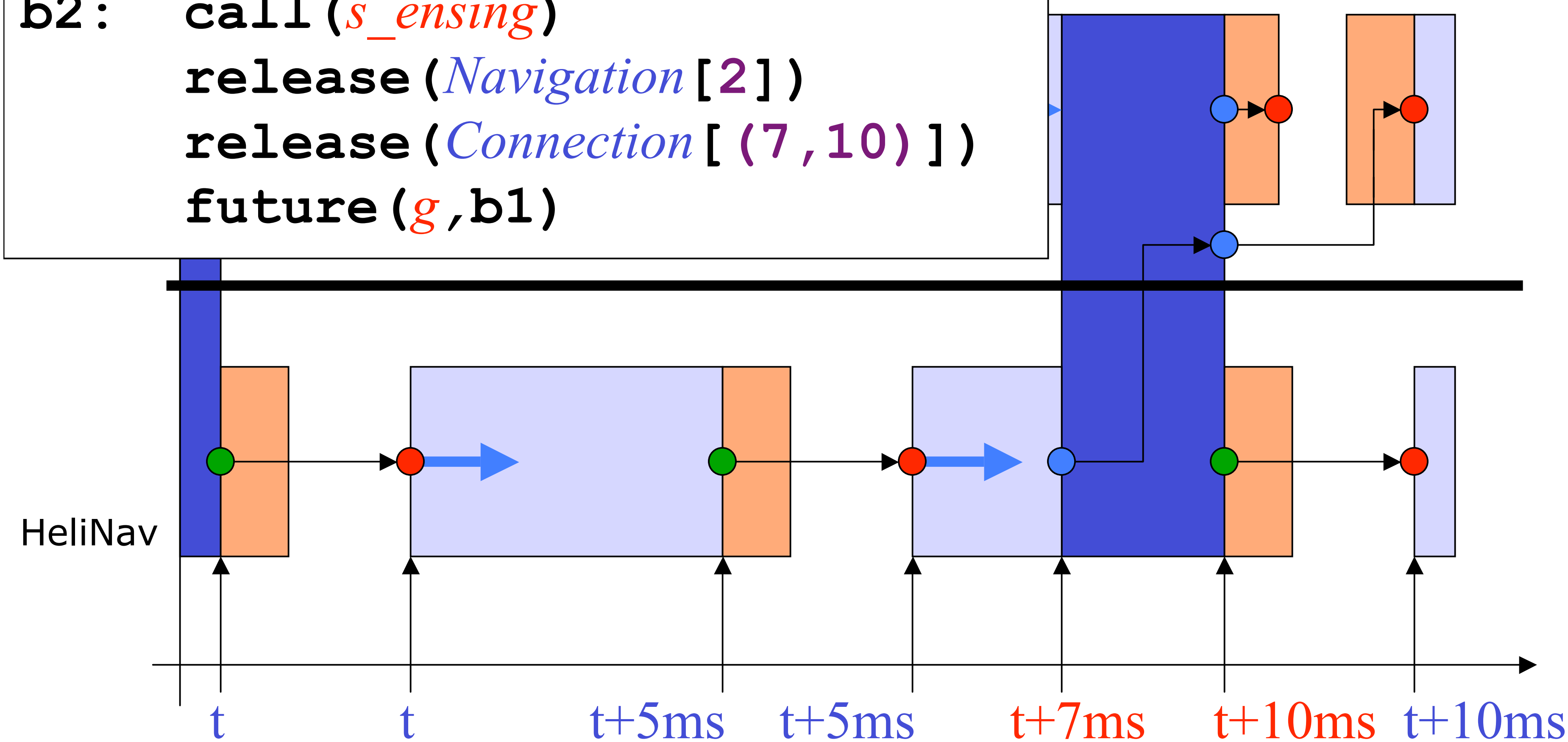
The Berkeley Helicopter

Platform Timeline: Time-triggered Communication



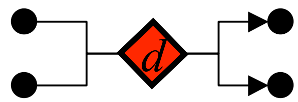
Code Generation for HeliNav

```
b2:   call(s_ensing)
      release(Navigation[2])
      release(Connection[(7,10)])
      future(g,b1)
```



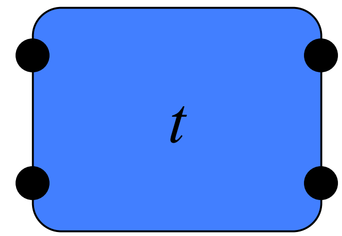
Instructions

Synchronous Driver:



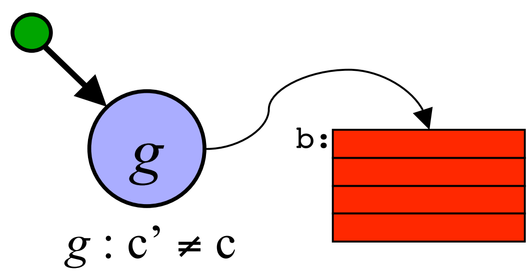
call(d**)**

Scheduled Task:



release(t**)**

Triggering:



future(g**,b)**