An Overview of Céu

A synchronous language inspired by Esterel

Francisco Sant'Anna

francisco@ime.uerj.br

www.ceu-lang.org

“Hello world!” in Céu

“Hello world!” in Céu

- Blinking a LED
    1. *on ↔ off every 500ms*

“Hello world!” in Céu

- Blinking a LED
    1. *on ↔ off every 500ms*

```
loop do
    await 500ms;
    _led_toggle();
end
```

“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*

```
loop do
   await 500ms;
   _led_toggle();
end
```

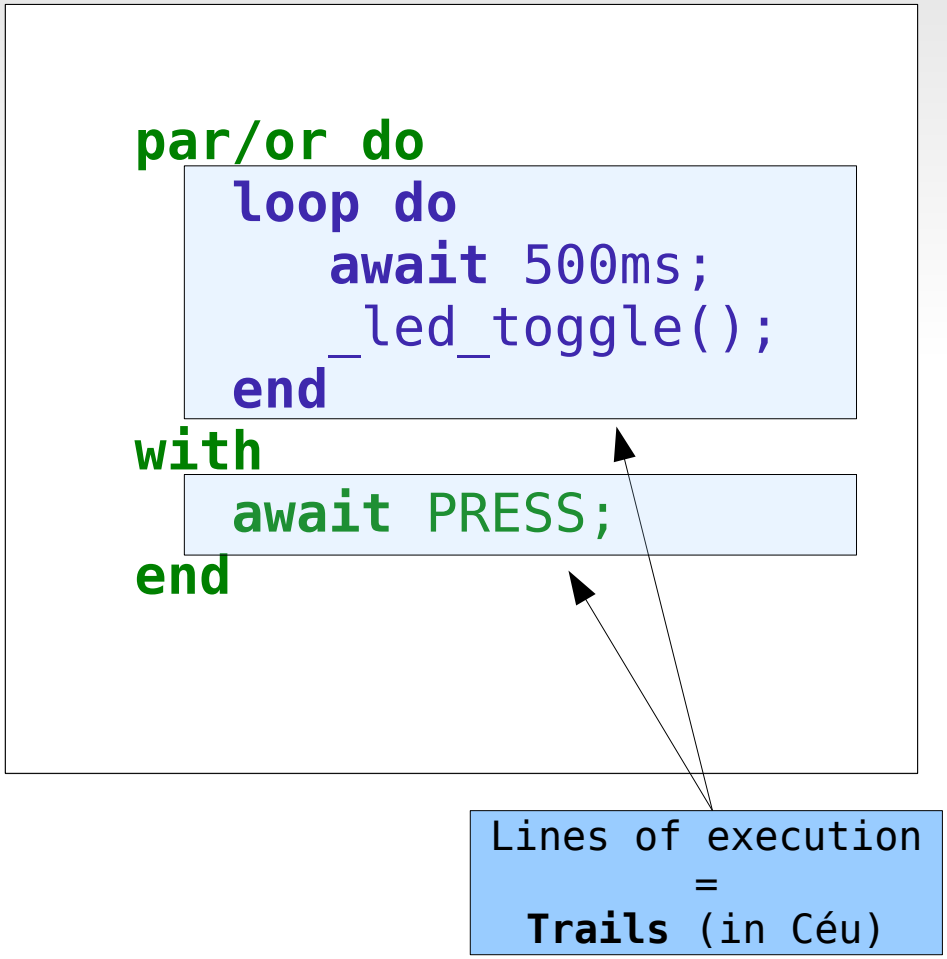
“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*

```
par/or do
    loop do
        await 500ms;
        _led_toggle();
    end
with
    await PRESS;
end
```

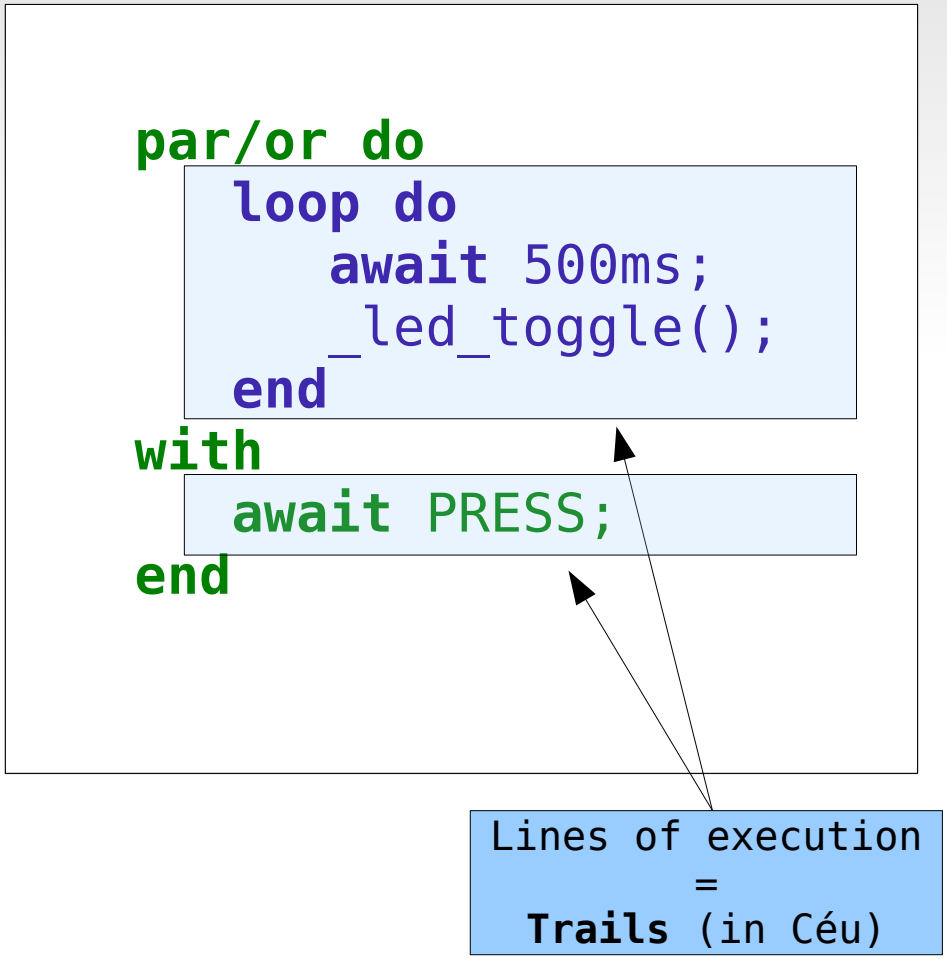
“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*



“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*
  3. *restart after 2s*



“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*
  3. *restart after 2s*

```
loop do
    par/or do
        loop do
            await 500ms;
            _led_toggle();
        end
    with
        await PRESS;
    end
    await 2s;
end
```

Lines of execution
=
Trails (in Céu)

“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*
  3. *restart after 2s*

- Compositions
  - seq, loop, par *(trails)*
    - At any level of depth

```
loop do
   par/or do
      loop do
         await 500ms;
         _led_toggle();
      end
   with
      await PRESS;
   end
   await 2s;
end
```

Lines of execution
=
Trails (in Céu)

“Hello world!” in Céu

- Blinking a LED
  1. *on ↔ off every 500ms*
  2. *stop after “press”*
  3. *restart after 2s*

- Compositions
  - seq, loop, par *(trails)*
    - At any level of depth
  - ~~state variables / communication~~

```
loop do
   par/or do
      loop do
         await 500ms;
         _led_toggle();
      end
   with
      await PRESS;
   end
   await 2s;
end
```

Lines of execution = **Trails** (in Céu)

Céu is heavily inspired by Esterel

Céu Peculiarities

Céu Peculiarities

1. External events

Céu Peculiarities

1. External events
    - notion of time ~ queue of unique events, mutual exclusion
2. Internal events

Céu Peculiarities

1. External events
   - notion of time ~ queue of unique events, mutual exclusion
2. Internal events
   - intra reactions, stack based, rich control mechanisms
3. Concurrency: internal determinism + static checks

Céu Peculiarities

1. External events
   - notion of time ~ queue of unique events, mutual exclusion
2. Internal events
   - intra reactions, stack based, rich control mechanisms
3. Concurrency: internal determinism + static checks
   - simple, concurrent assignments/system calls
4. Safe integration with C

Céu Peculiarities

1. External events
   - notion of time ~ queue of unique events, mutual exclusion
2. Internal events
   - intra reactions, stack based, rich control mechanisms
3. Concurrency: internal determinism + static checks
   - simple, concurrent assignments/system calls
4. Safe integration with C
   - finalization for local/external resources
5. First-class timers

Céu Peculiarities

1. External events
   - notion of time ~ queue of unique events, mutual exclusion
2. Internal events
   - intra reactions, stack based, rich control mechanisms
3. Concurrency: internal determinism + static checks
   - simple, concurrent assignments/system calls
4. Safe integration with C
   - finalization for local/external resources
5. First-class timers
   - dedicated syntax, automatic readjustment
6. Dynamic execution

Céu Peculiarities

1. External events
   - notion of time ~ queue of unique events, mutual exclusion
2. Internal events
   - intra reactions, stack based, rich control mechanisms
3. Concurrency: internal determinism + static checks
   - simple, concurrent assignments/system calls
4. Safe integration with C
   - finalization for local/external resources
5. First-class timers
   - dedicated syntax, automatic readjustment
6. Dynamic execution
   - pool allocation, static/lexical memory management

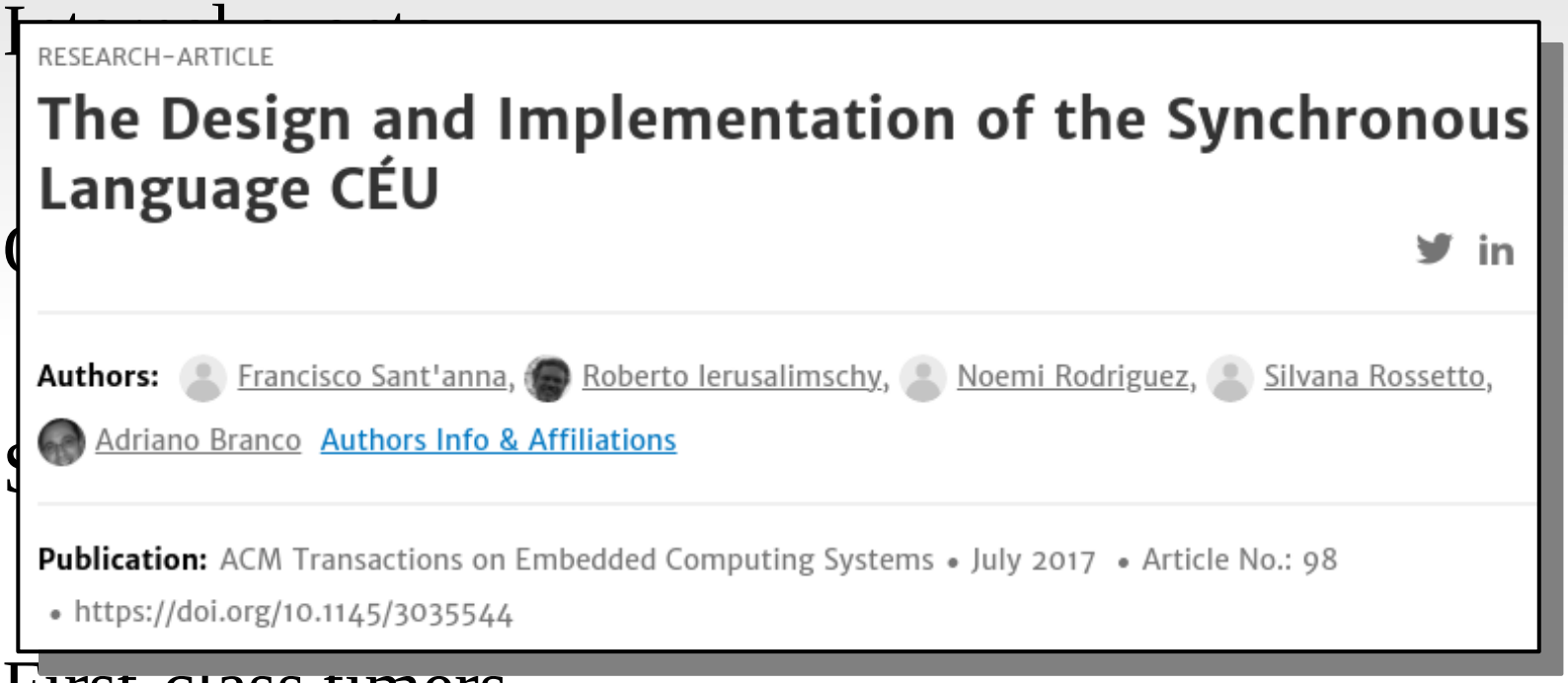
Céu Peculiarities

1. External events
   - notion of time ~ queue of unique events, mutual exclusion
2. Internal events
   - intra reactions, stack based, rich control mechanisms
3. Concurrency: internal determinism + static checks
   - simple, concurrent assignments/system calls
4. Safe integration with C
   - finalization for local/external resources
5. First-class timers
   - dedicated syntax, automatic readjustment
6. Dynamic execution
   - pool allocation, static/lexical memory management

15 min video at **`ceu-lang.org`**

Céu Peculiarities

1. External events
    - notion of time ~ queue of unique events, mutual exclusion
2. I
3. C
4. S
5. First-class timers
    - dedicated syntax, automatic readjustment
6. Dynamic execution
    - pool allocation, static/lexical memory management



RESEARCH-ARTICLE

The Design and Implementation of the Synchronous Language CÉU

Authors: Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, Silvana Rossetto, Adriano Branco Authors Info & Affiliations

Publication: ACM Transactions on Embedded Computing Systems • July 2017 • Article No.: 98 • https://doi.org/10.1145/3035544

15 min video at **`ceu-lang.org`**

1. External Events

1. External Events

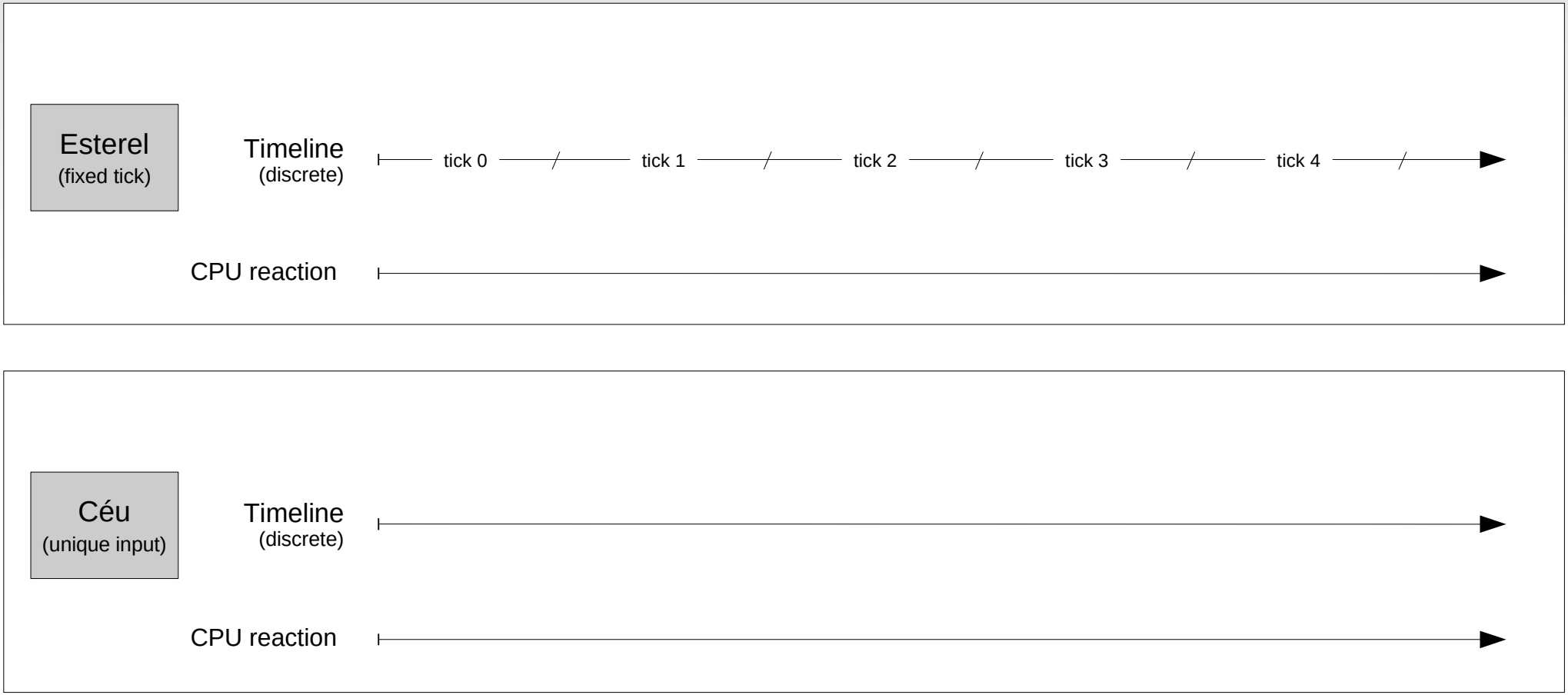
Esterel
(fixed tick)

Céu
(unique input)

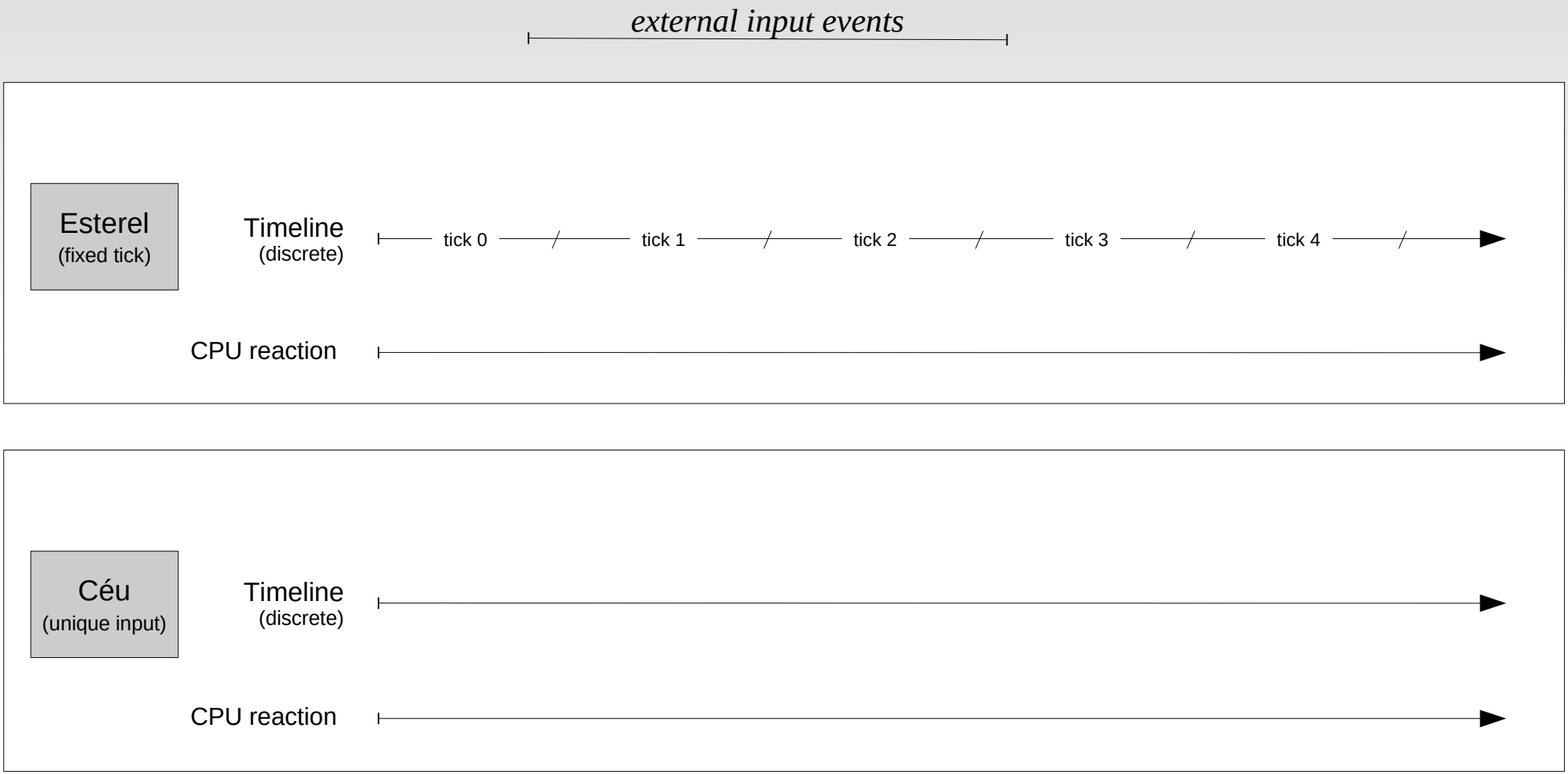
1. External Events

1. External Events

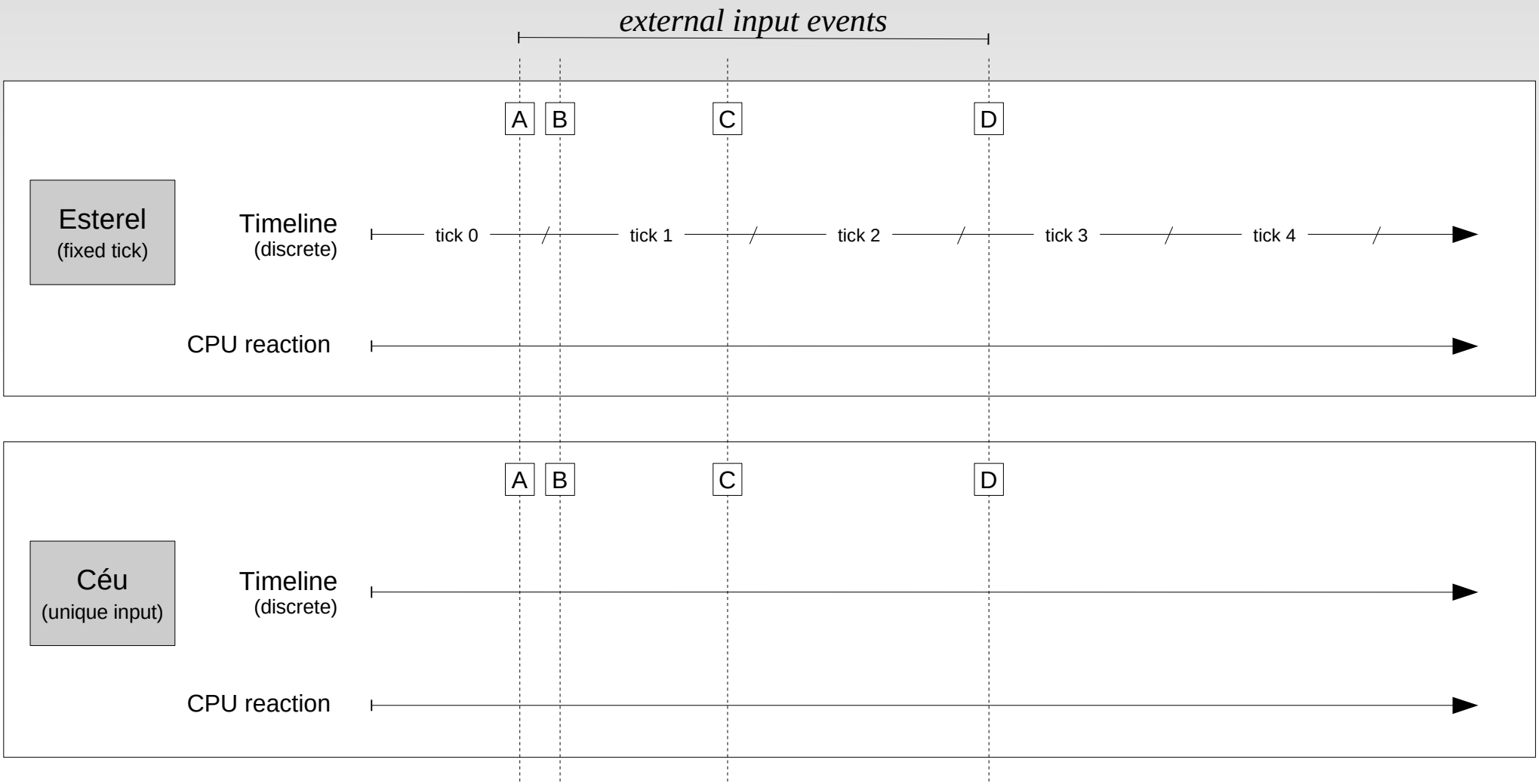
1. External Events



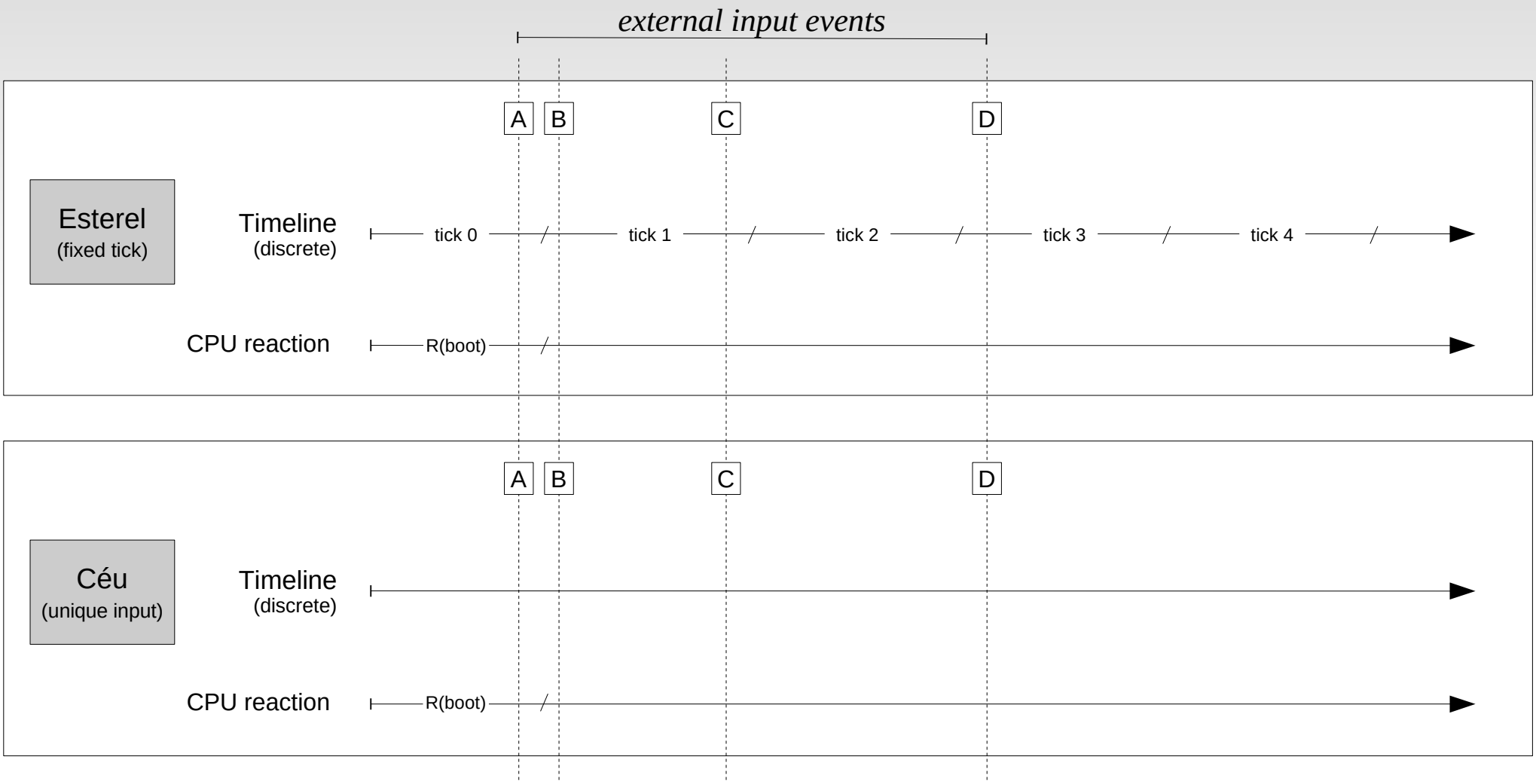
1. External Events



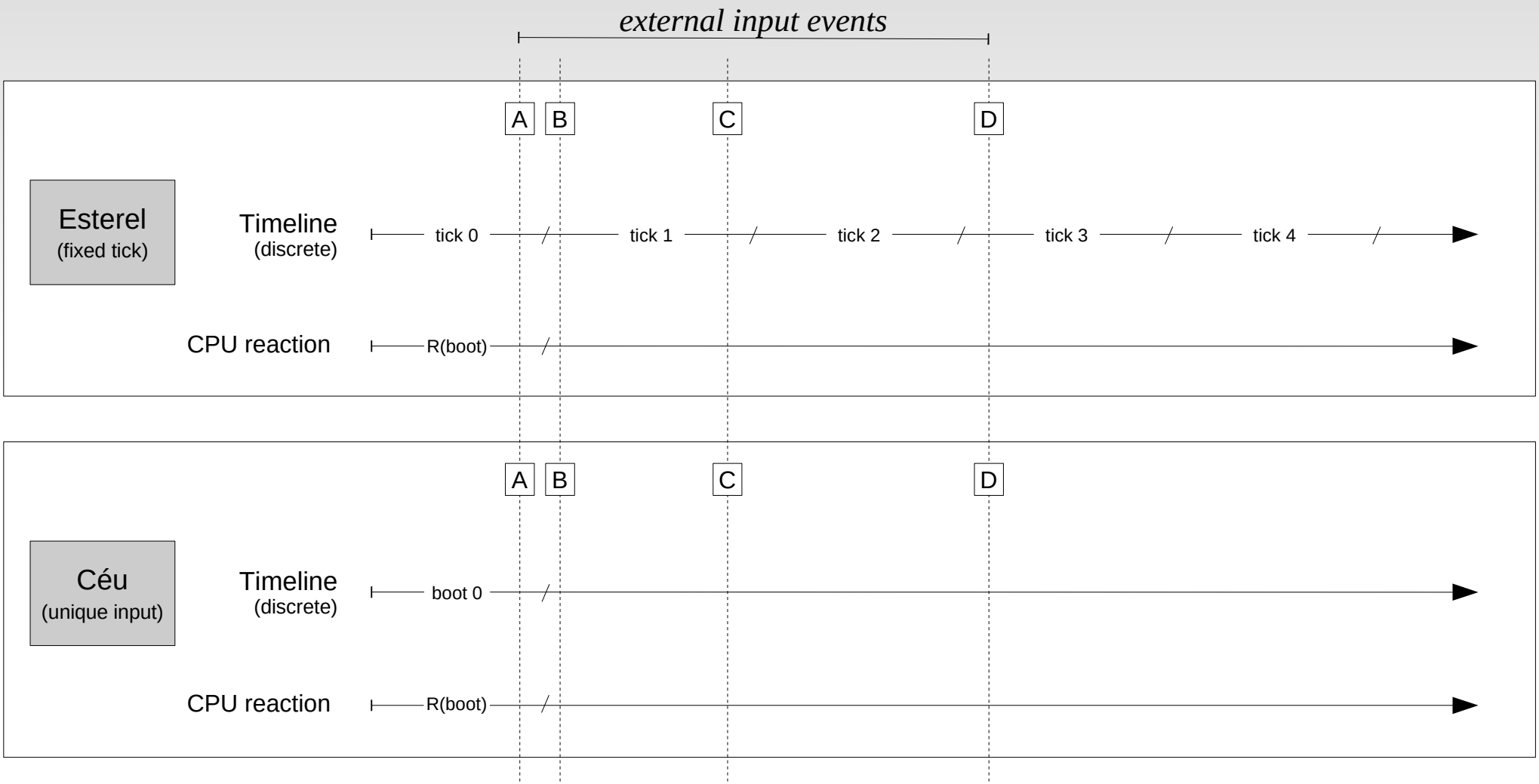
1. External Events



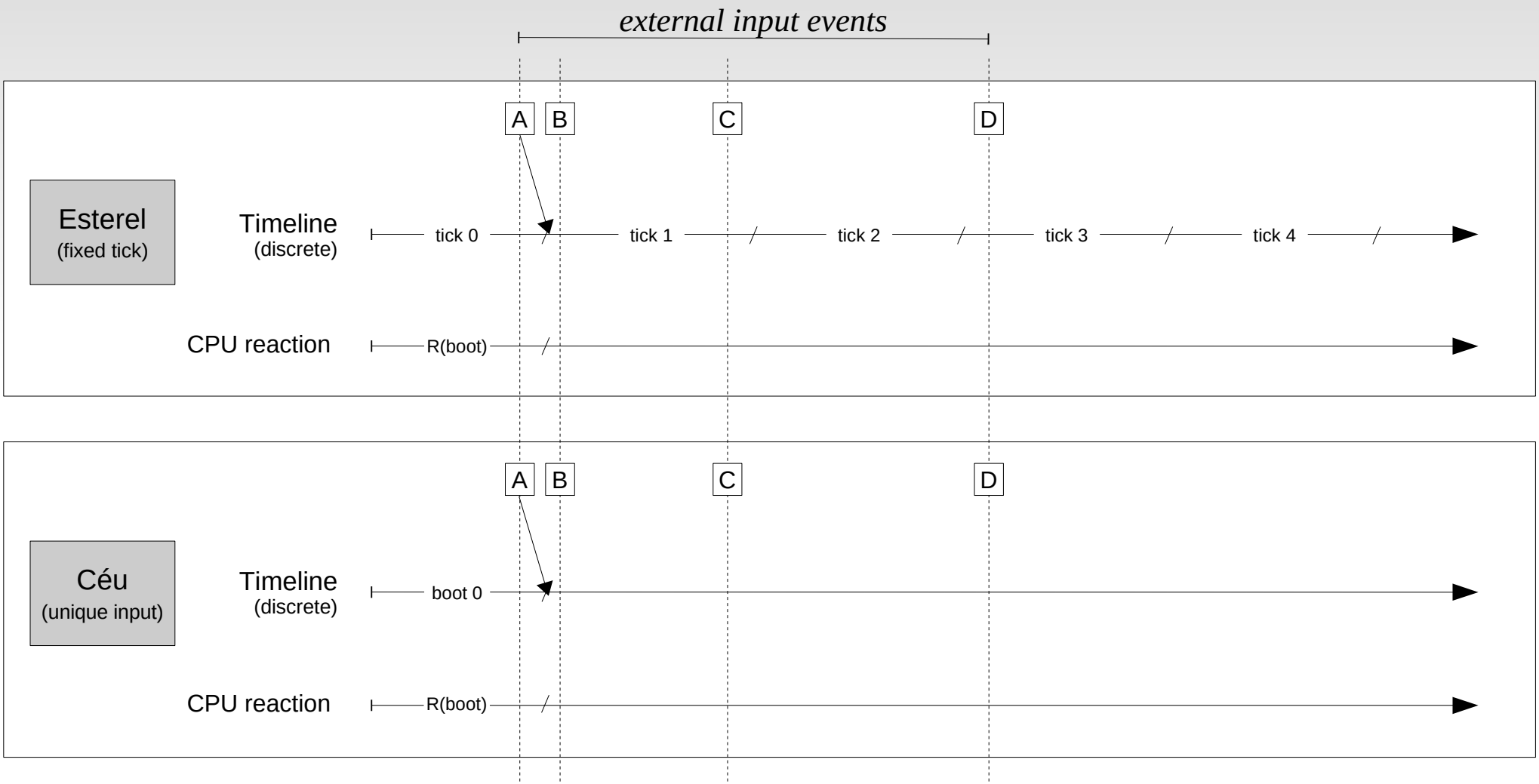
1. External Events



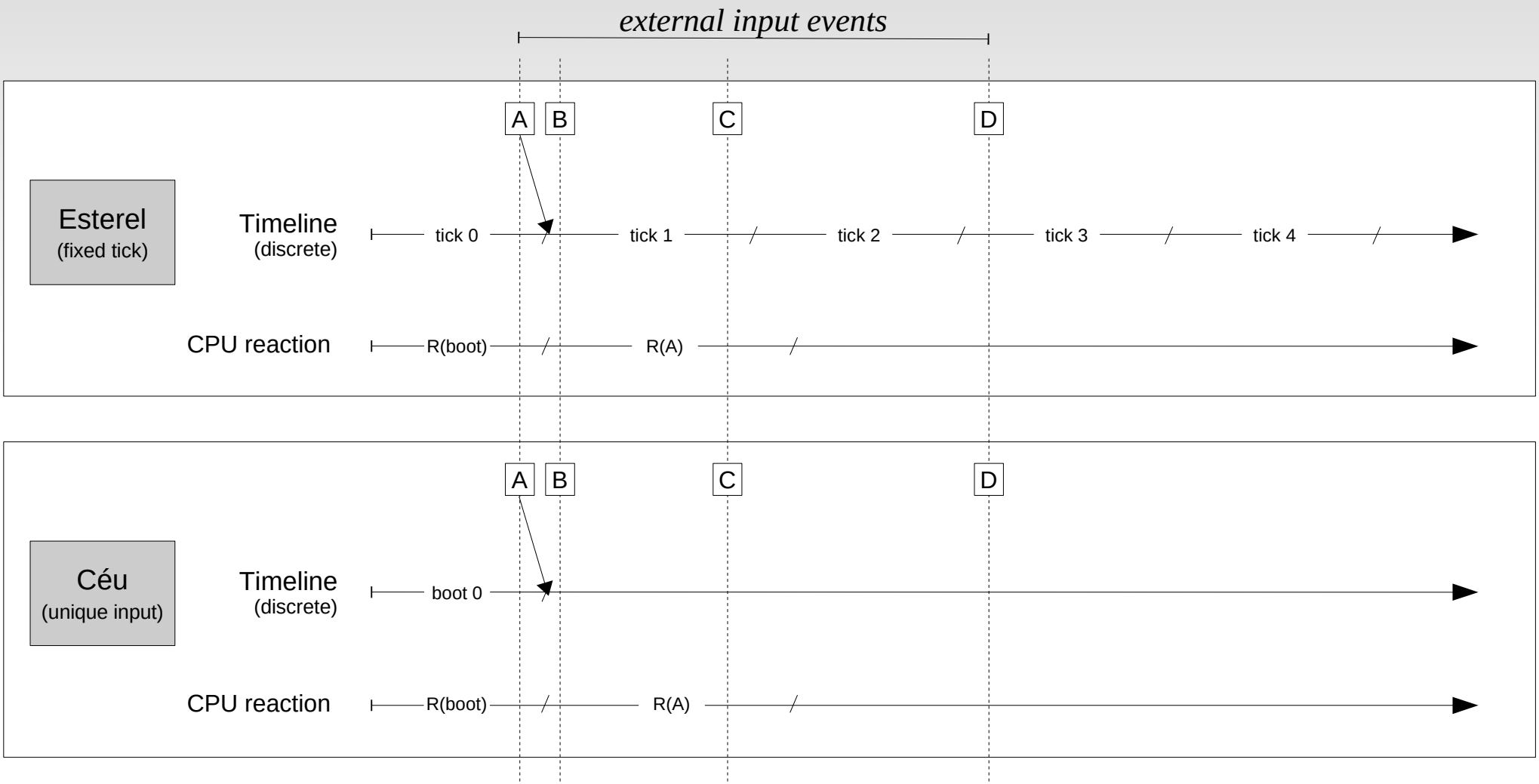
1. External Events



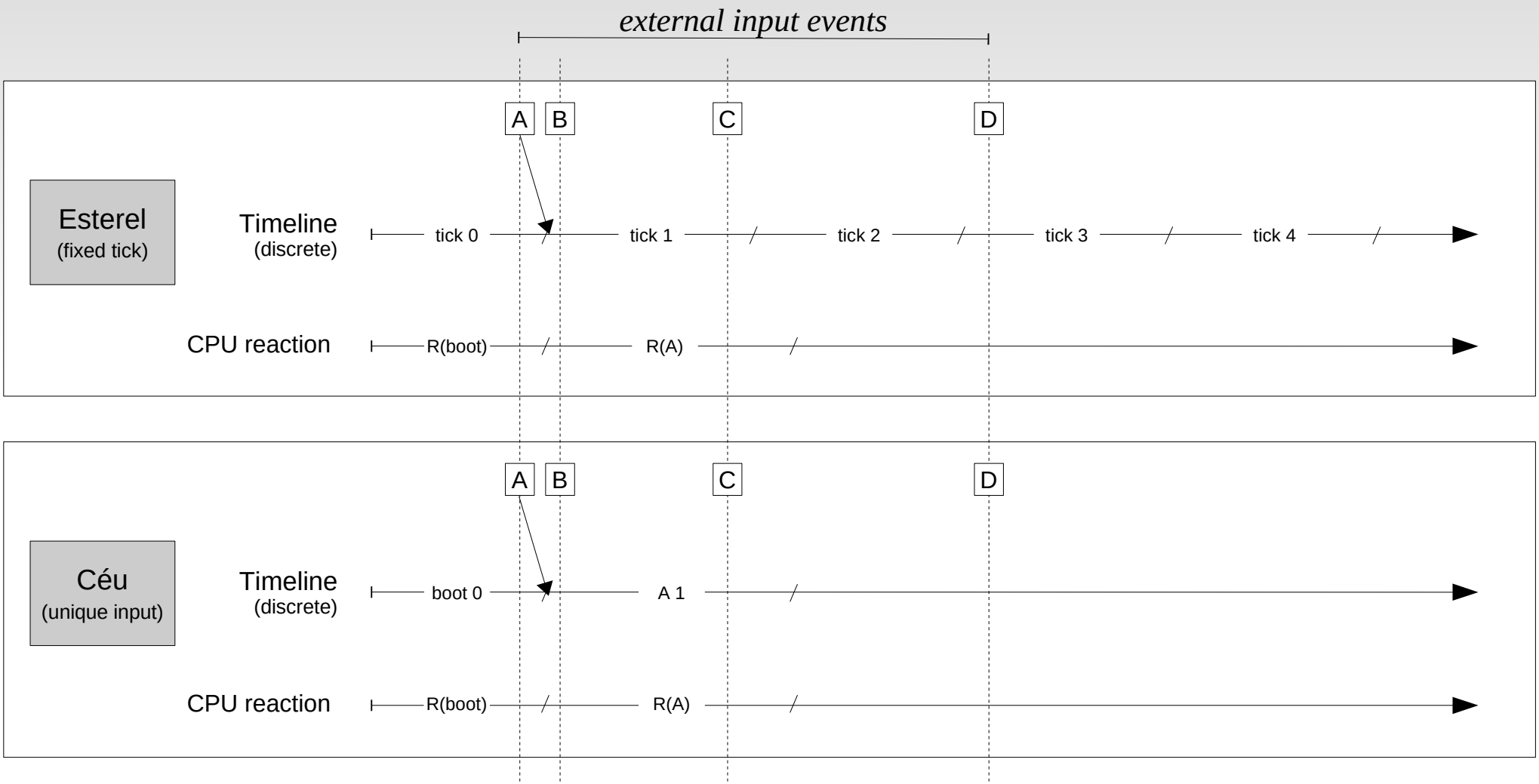
1. External Events



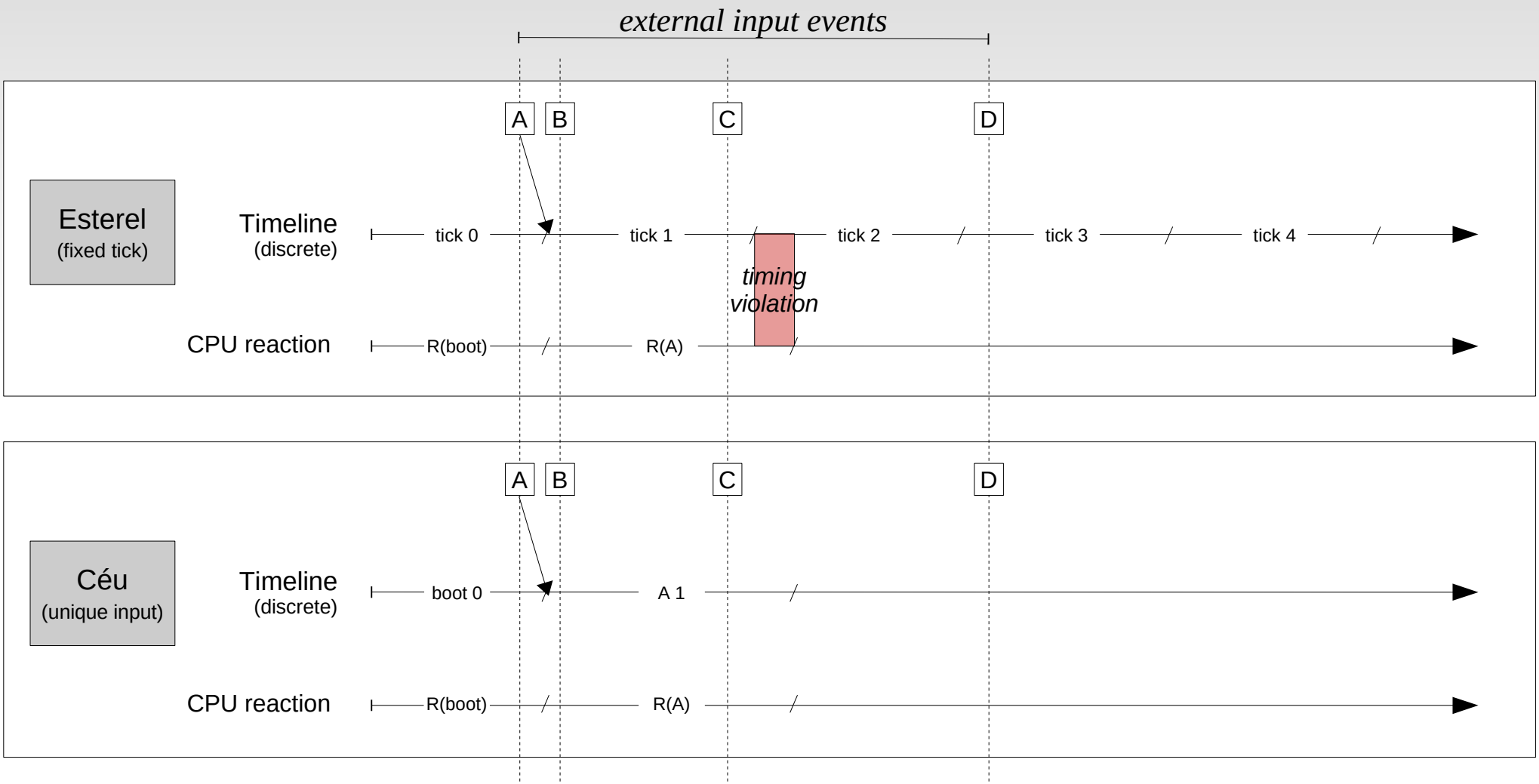
1. External Events



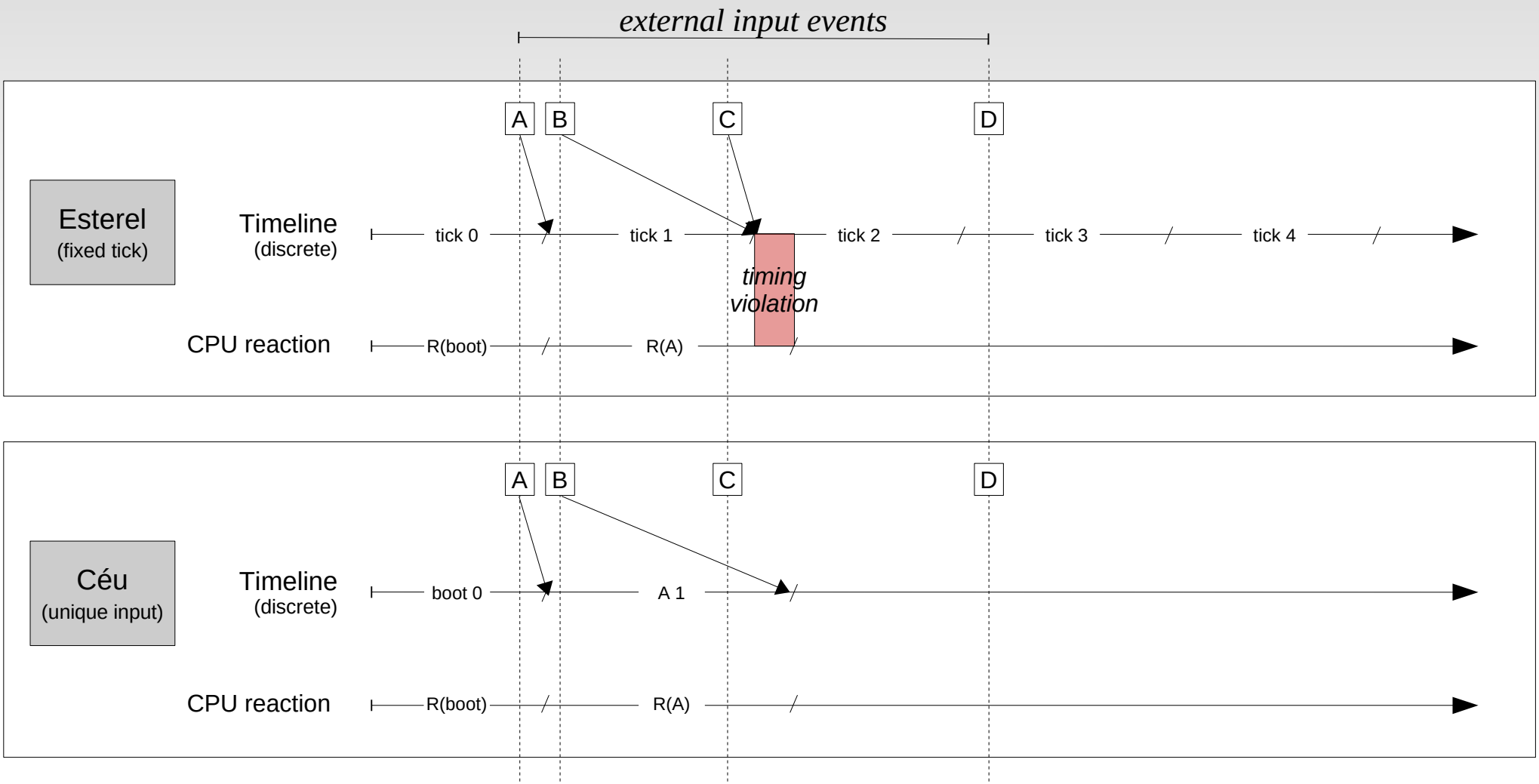
1. External Events



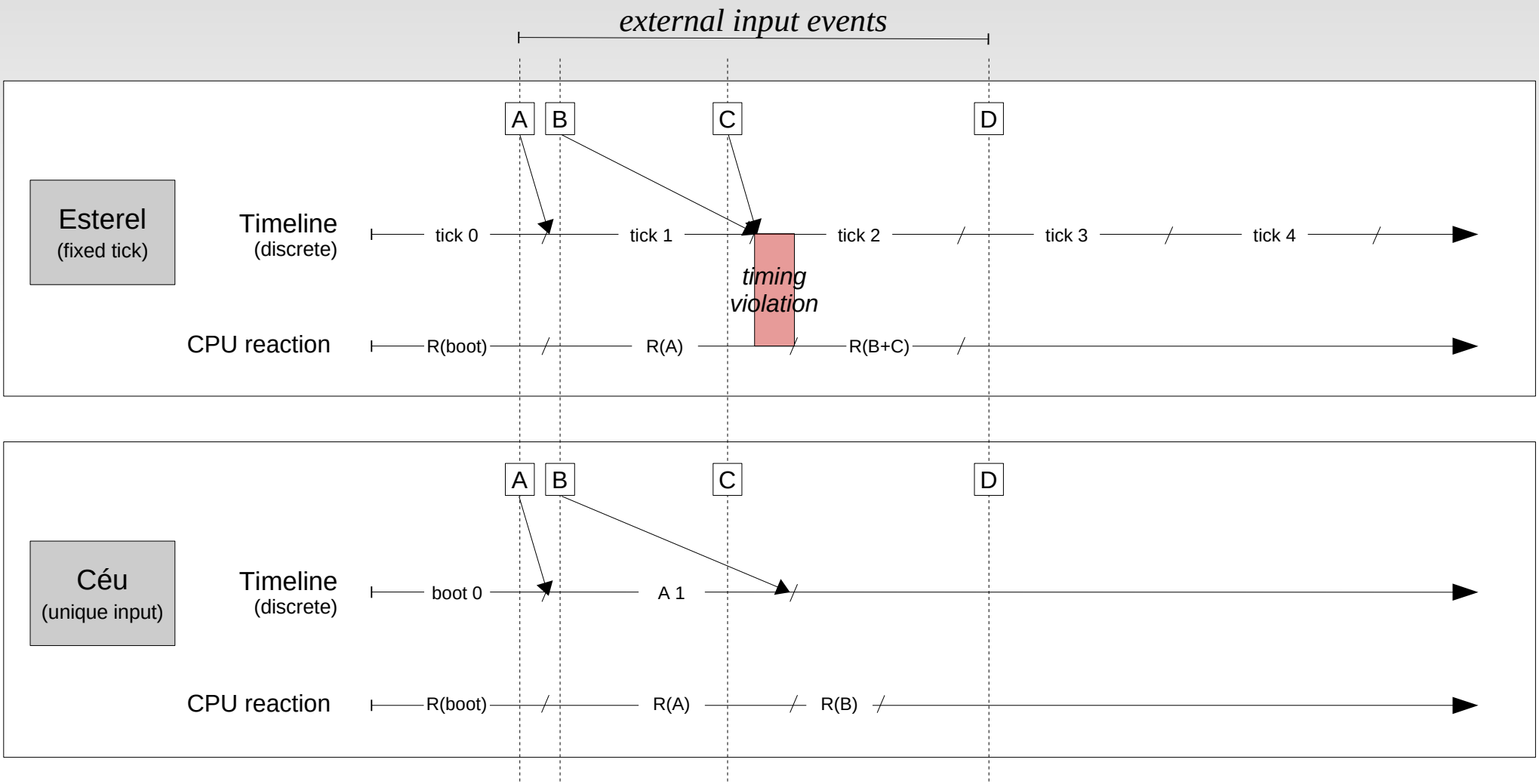
1. External Events



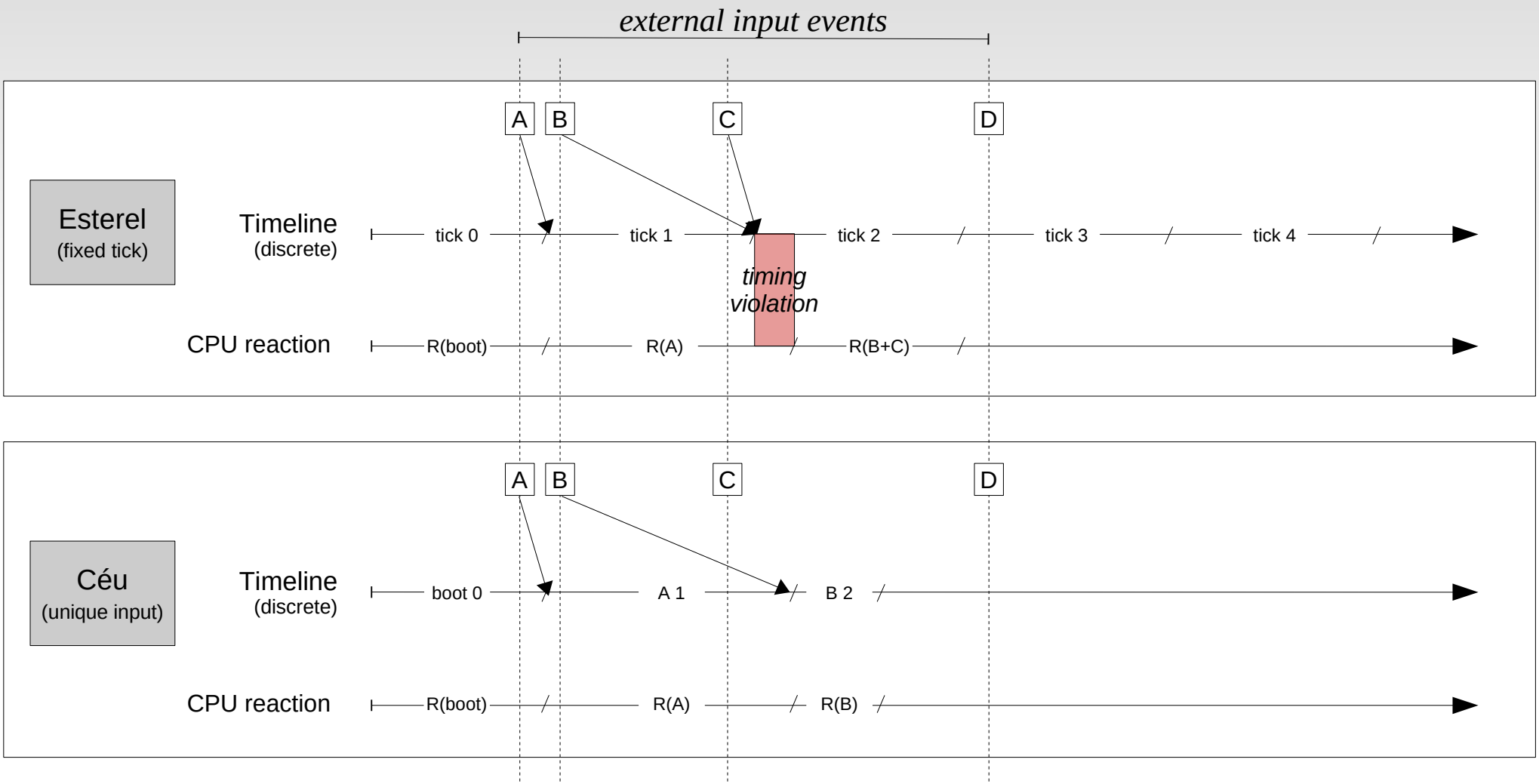
1. External Events



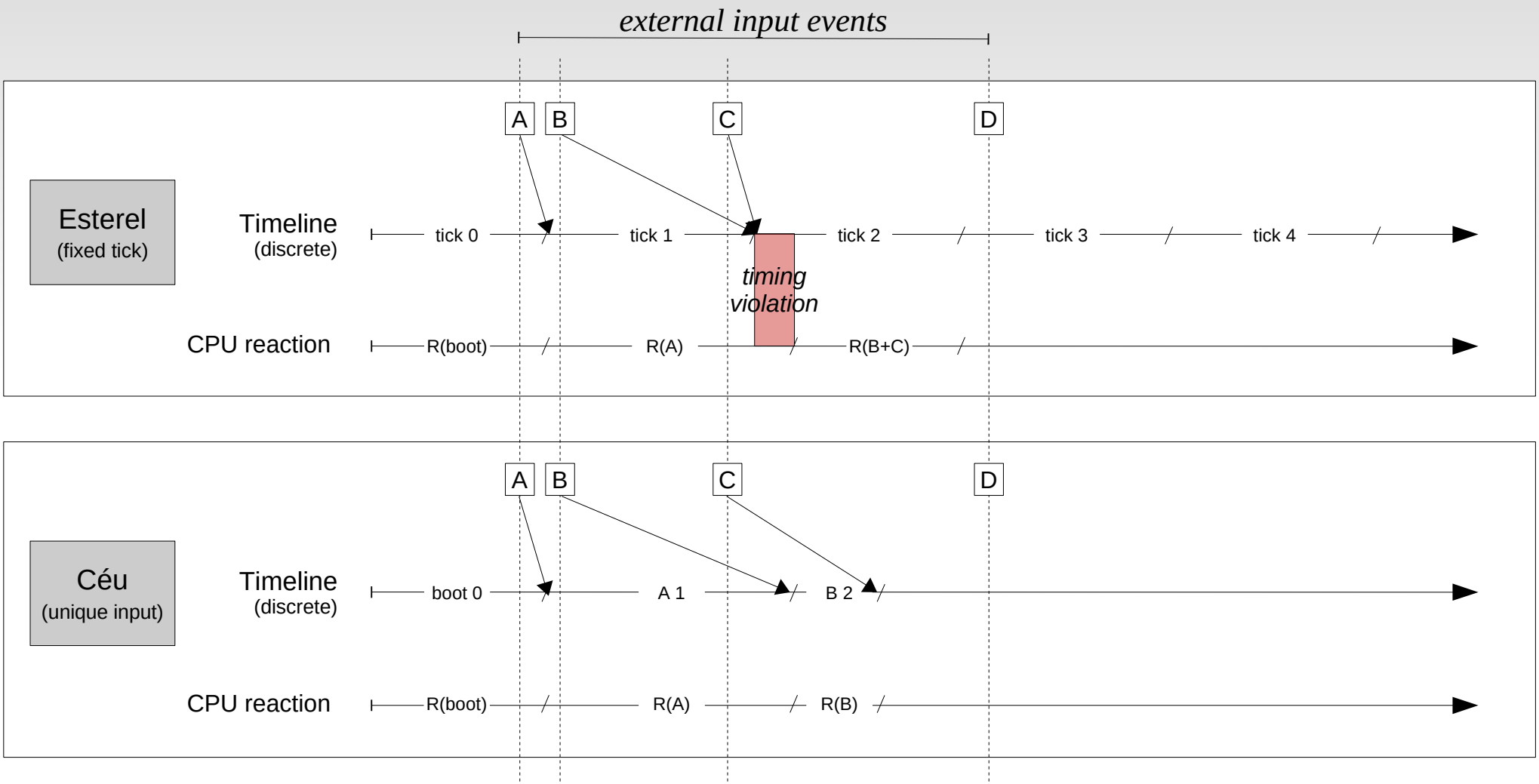
1. External Events



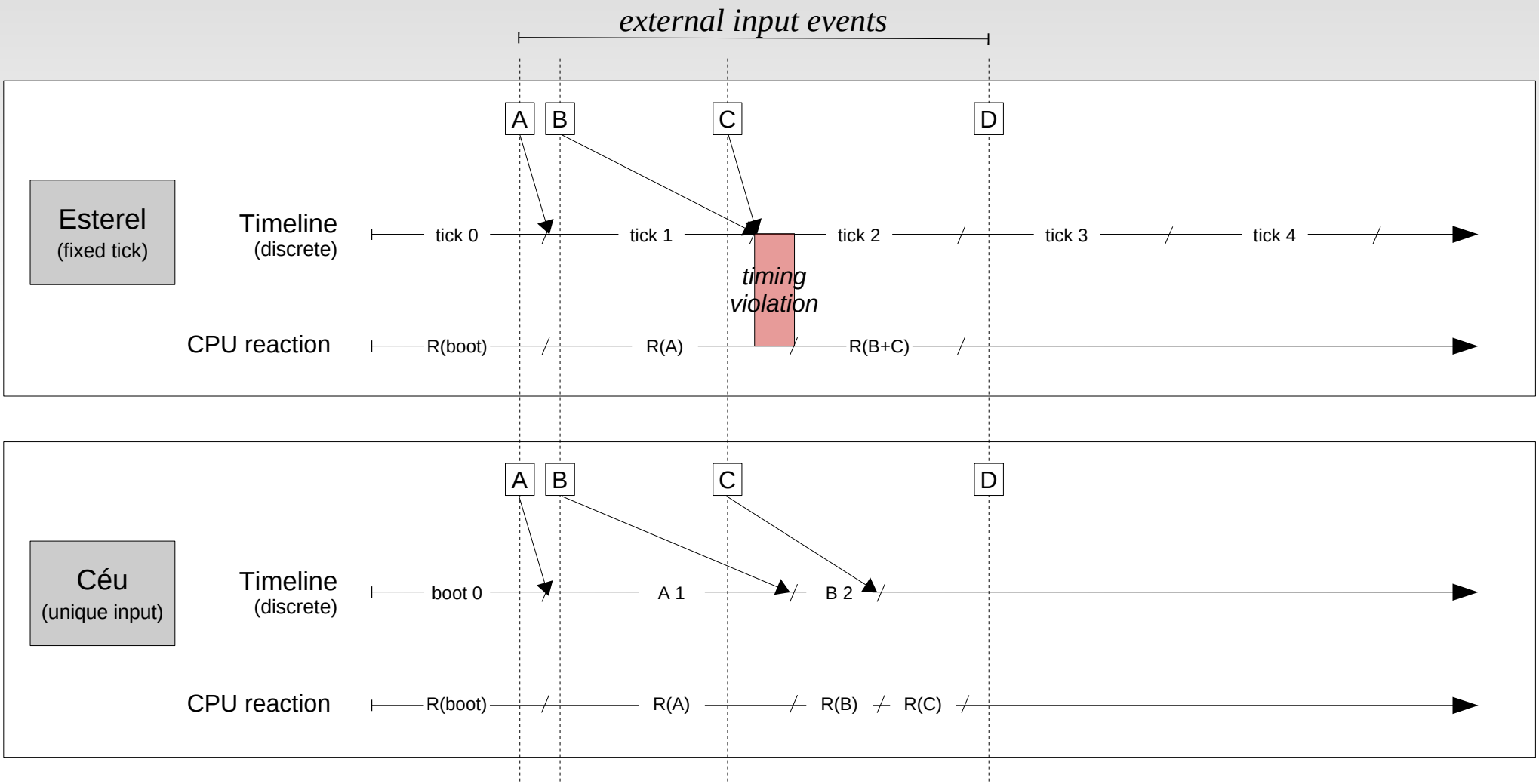
1. External Events



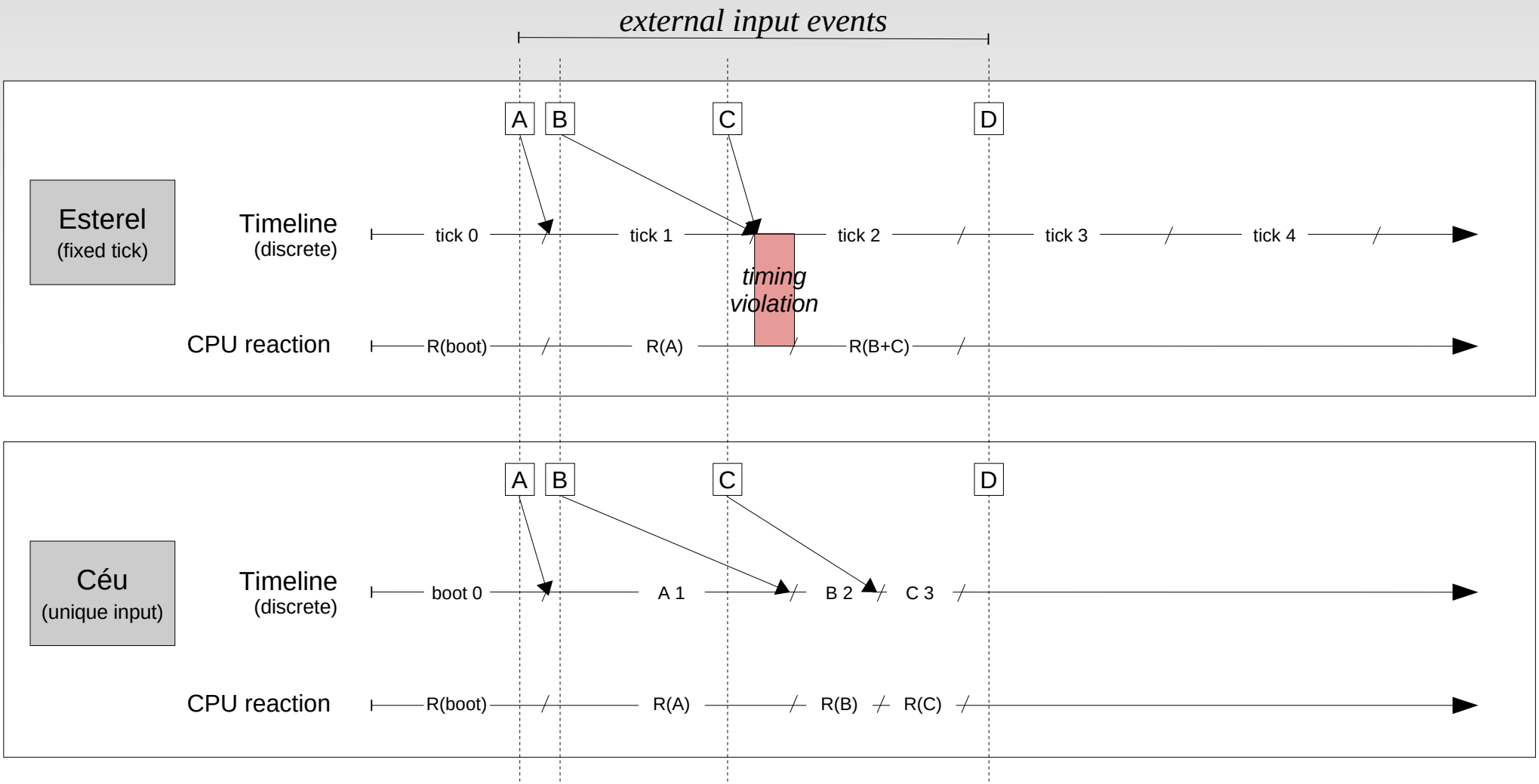
1. External Events



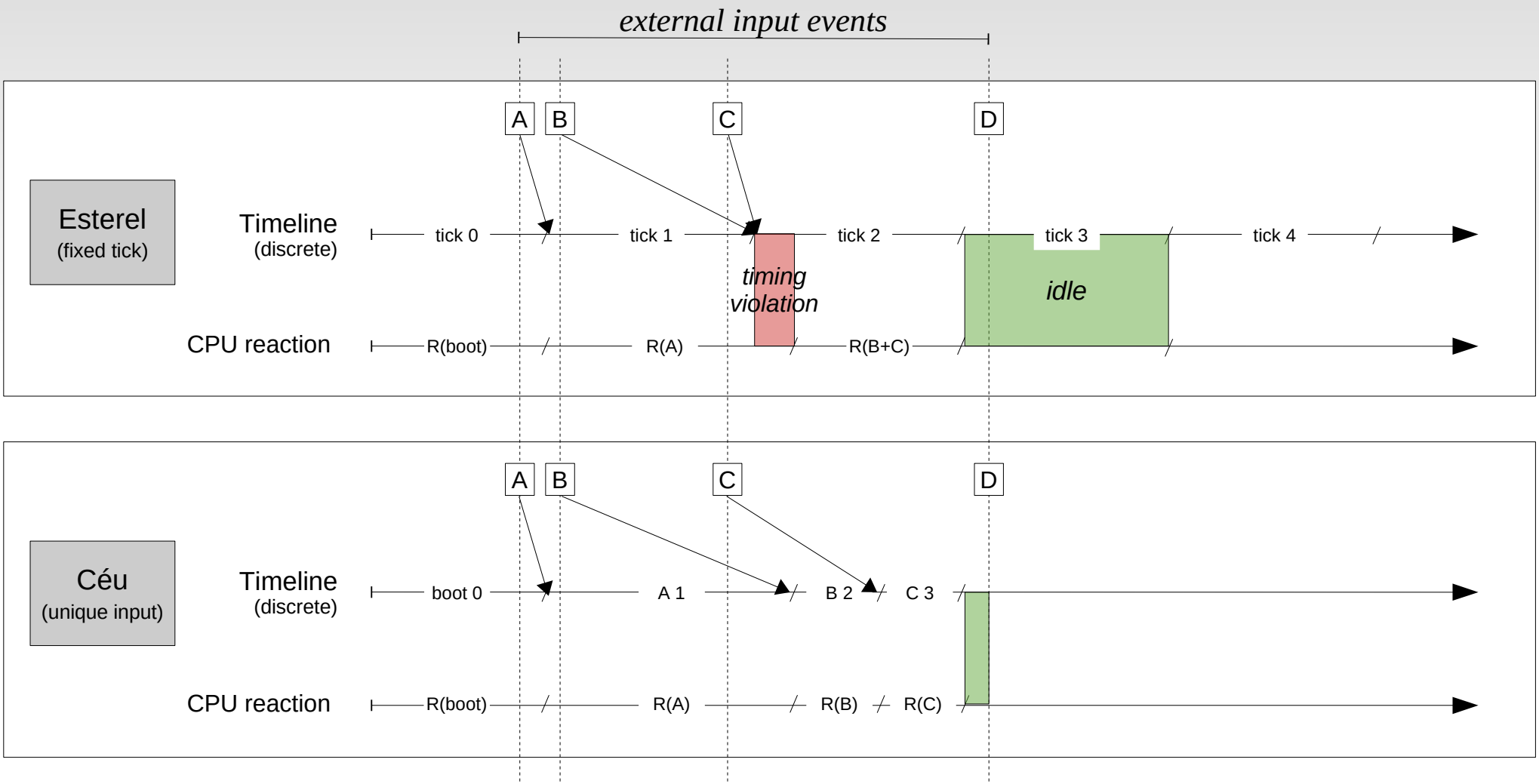
1. External Events



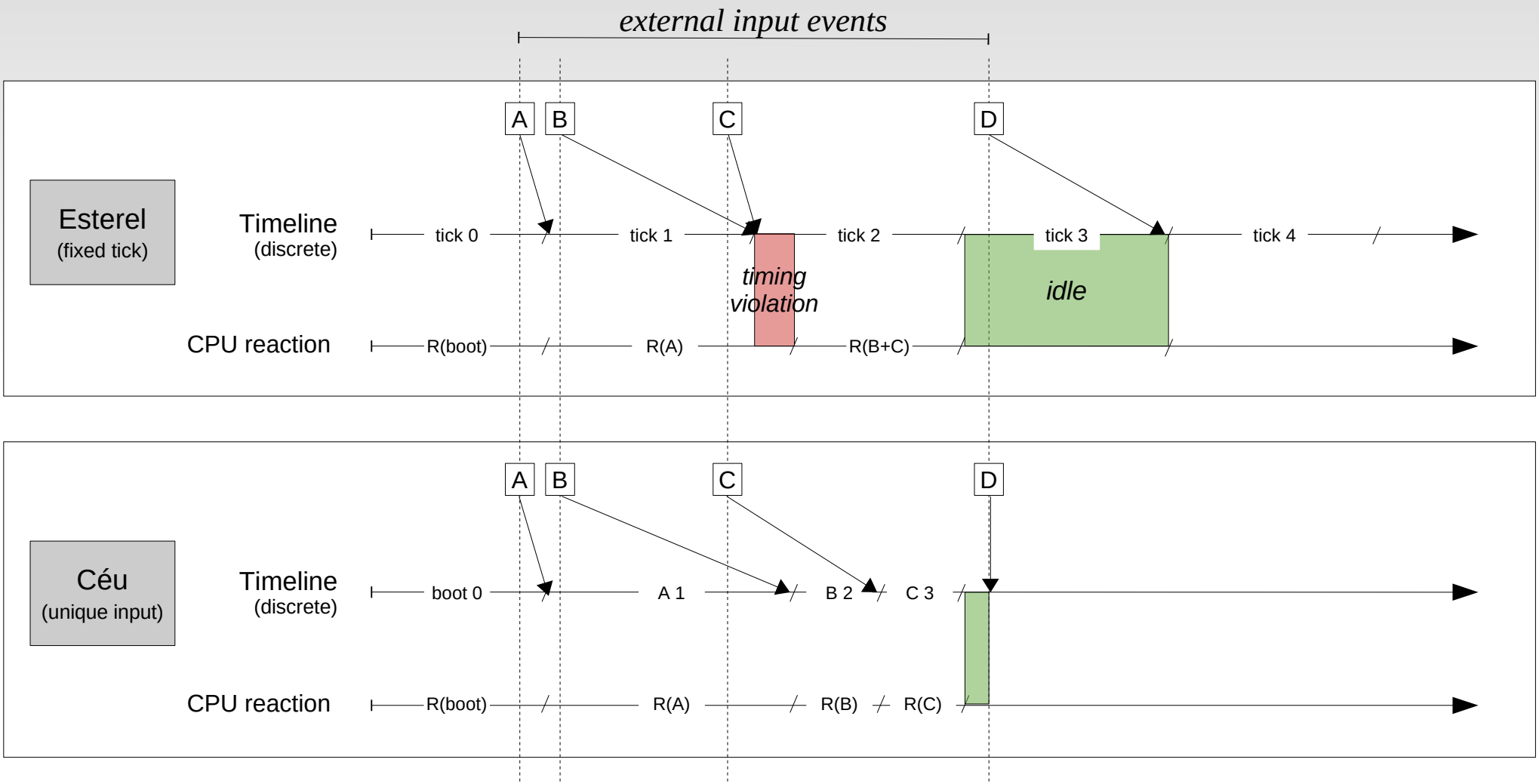
1. External Events



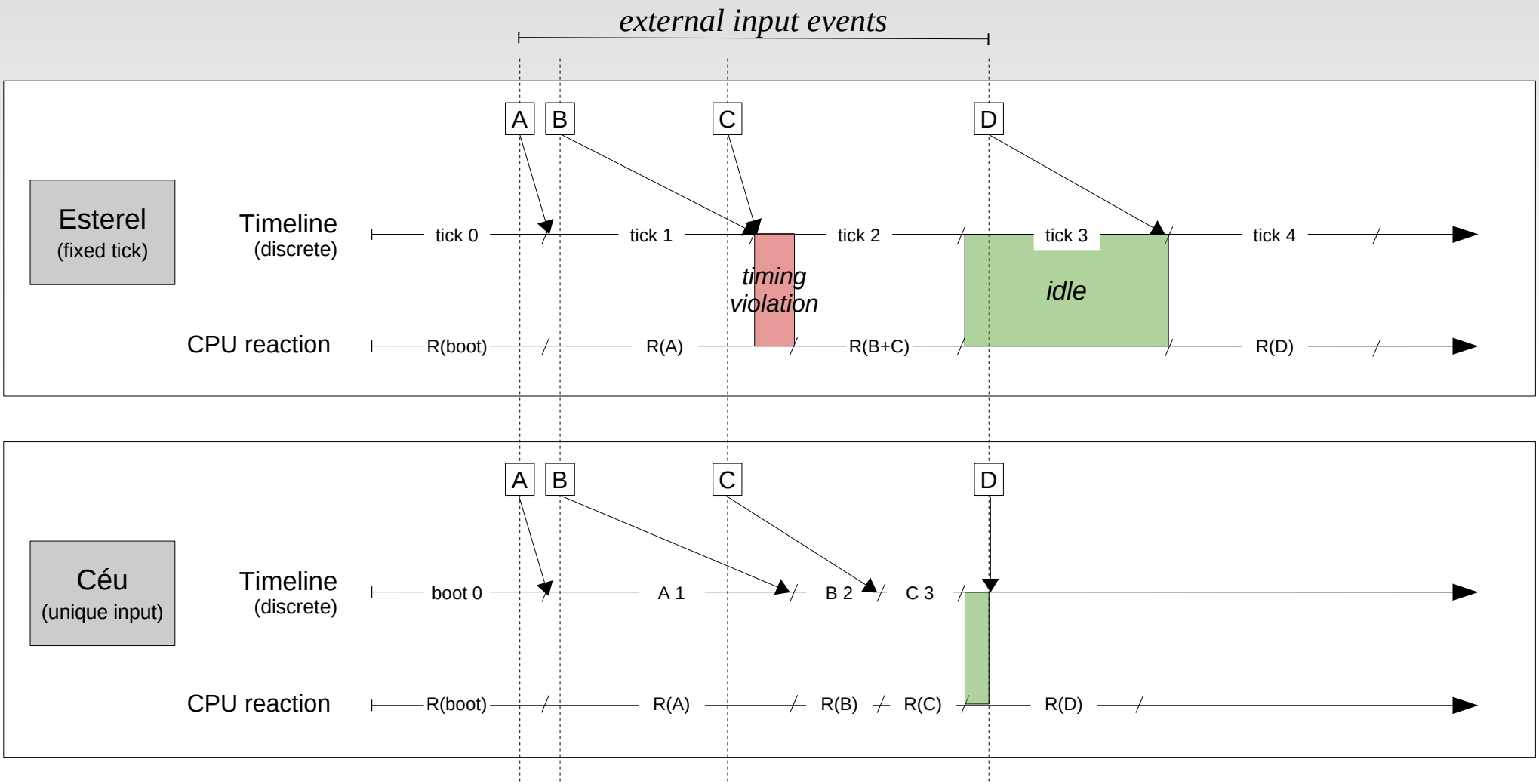
1. External Events



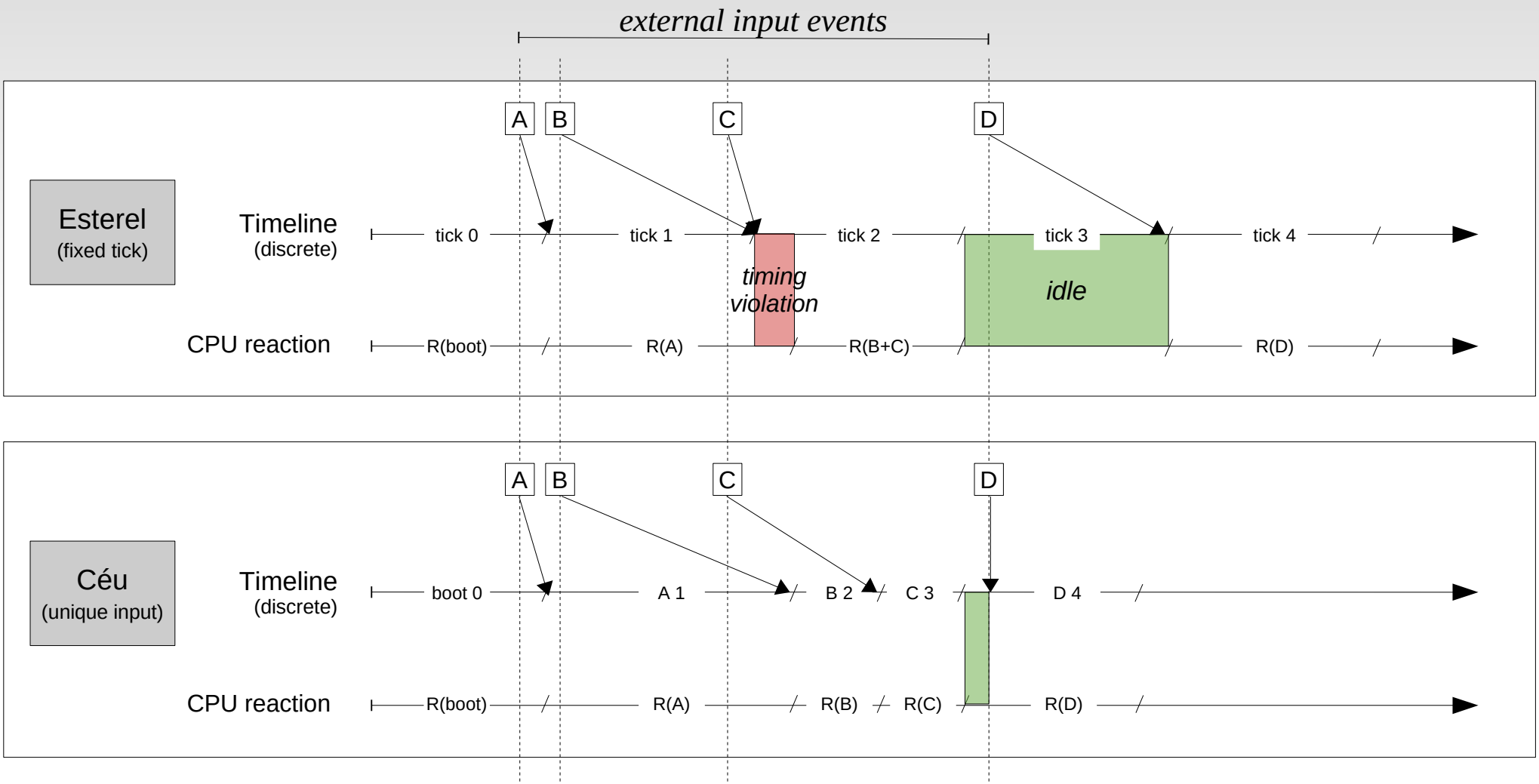
1. External Events



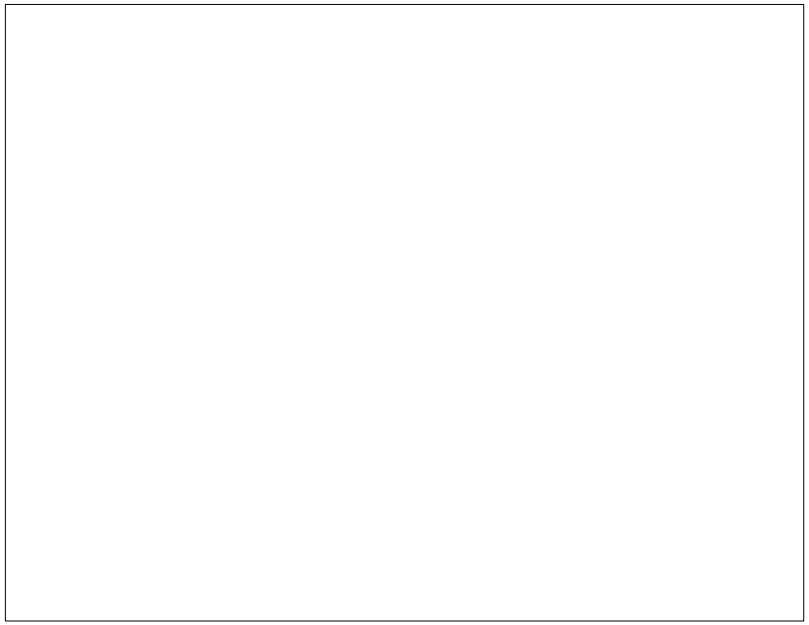
1. External Events



1. External Events

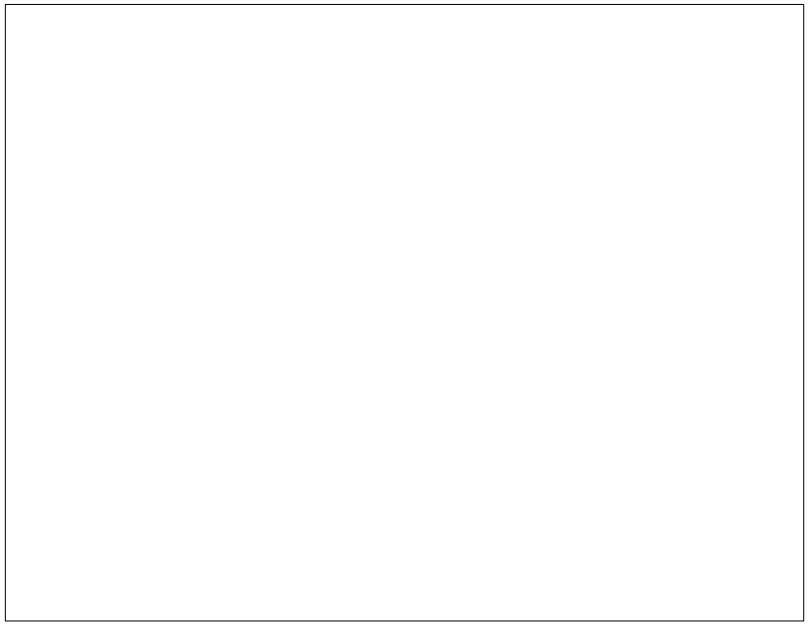


2. Internal Events



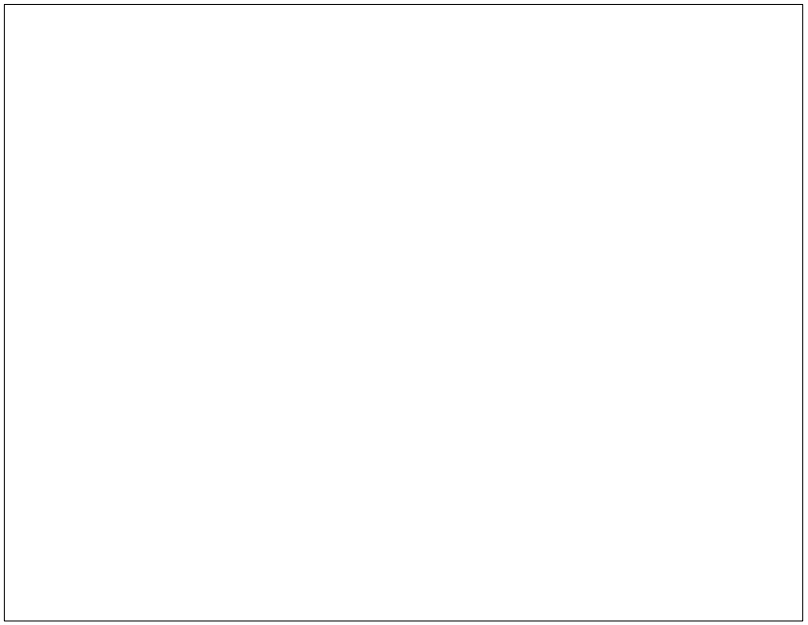
2. Internal Events

- Stack-based execution



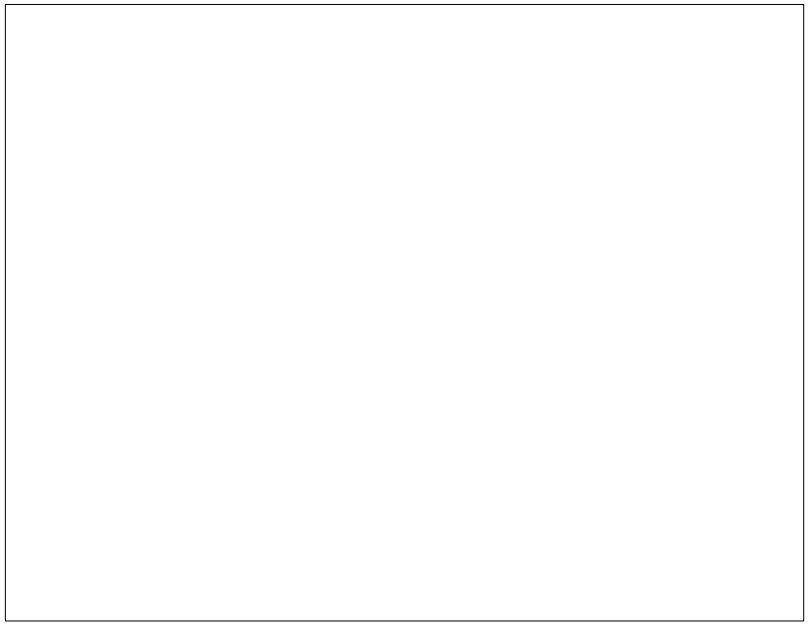
2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*



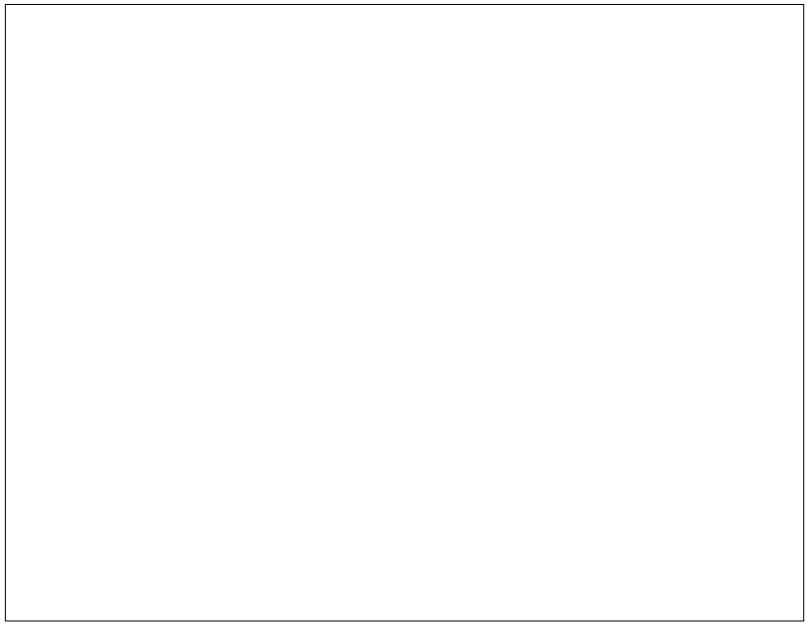
2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)



2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls



2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc;
      *p = *p + 1;
   end
with
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc;
      *p = *p + 1;
   end
with
   var int v = 1;
   <...>
   emit inc => &v;
   assert(v==2);
end
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc; 1
      *p = *p + 1;
   end
with
   var int v = 1;
   <...>
   emit inc => &v;
   assert(v==2);
end
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc; 1
      *p = *p + 1;
   end
with
   var int v = 1;
   <...>
   emit inc => &v; 2
   assert(v==2);
end
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc; (1)
      *p = *p + 1;
   end
with
   var int v = 1;
   <...>
   emit inc => &v; (2)
   assert(v==2);
end
```

stacked

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc; (1)
      *p = *p + 1; (3)
   end
with
   var int v = 1;
   <...>
   emit inc => &v; (2)
   assert(v==2);
end
```

stacked

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc;   (1) (4)
      *p = *p + 1;              (3)
   end
with
   var int v = 1;
   <...>
   emit inc => &v;              (2)
   assert(v==2);                    stacked
end
```

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls

```
event int* inc;
par/or do
   loop do
      var int* p = await inc; (1) (4)
      *p = *p + 1; (3)
   end
with
   var int v = 1;
   <...>
   emit inc => &v; (2)
   assert(v==2); (5)
end
```

stacked

2. Internal Events

- Stack-based execution
  - an **`emit`** stacks next statement → awakes awaiting trails in an *intra reaction*
  - emits can nest (hence a stack)
- Like function calls
  - but richer: coroutines, resumable exceptions, reactive variables

```
event int* inc;
par/or do
   loop do
      var int* p = await inc; (1) (4)
      *p = *p + 1; (3)
   end
with
   var int v = 1;
   <...>
   emit inc => &v; (2)
   assert(v==2); (5)
end
```

stacked

3. Internal Determinism

3. Internal Determinism

- Esterel:

3. Internal Determinism

- Esterel:
  - *“if there is no control dependency, as in (call f1() || call f2()) , the order is unspecified and it would be an error to rely on it”*

3. Internal Determinism

- Esterel:
  - *“if there is no control dependency, as in (call f1() || call f2()) , the order is unspecified and it would be an error to rely on it”*
  - *“if a variable is written by some thread, then it can neither be read nor be written by concurrent threads”*

3. Internal Determinism

- Esterel:
  - *“if there is no control dependency, as in (call f1() || call f2()) , the order is unspecified and it would be an error to rely on it”*
  - *“if a variable is written by some thread, then it can neither be read nor be written by concurrent threads”*
- Céu:

3. Internal Determinism

- Esterel:
  - *“if there is no control dependency, as in (call f1() || call f2()) , the order is unspecified and it would be an error to rely on it”*
  - *“if a variable is written by some thread, then it can neither be read nor be written by concurrent threads”*
- Céu:
  - *“when multiple trails are active during the same reaction, they are scheduled in lexical order”*

3. Internal Determinism

- Esterel:
  - *"if there is no control dependency, as in (call f1() || call f2()) [1], the order is unspecified and it would be an error to rely on it"*
  - *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"*
- Céu:
  - *"when multiple trails are active during the same reaction, they are scheduled in lexical order"*

3. Internal Determinism

- Esterel:
  - *“if there is no control dependency, as in (call f1() (1) || call f2() (2)) , the order is unspecified and it would be an error to rely on it”*
  - *“if a variable is written by some thread, then it can neither be read nor be written by concurrent threads”*
- Céu:
  - *“when multiple trails are active during the same reaction, they are scheduled in lexical order”*

3. Internal Determinism

- Esterel:
  - *"if there is no control dependency, as in (call f1() || call f2()) , the order is unspecified and it would be an error to rely on it"*
  - *"if a variable is written by some thread, then it can neither be read nor be written by concurrent threads"*
- Céu:
  - *"when multiple trails are active during the same reaction, they are scheduled in lexical order"*
  - pragmatic (e.g., `printf`, `redraw`), but fragile

3. Simple Static Checks

3. Simple Static Checks

```
input void A, B;
var int x = 1;
```

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
```

3. Simple Static Checks

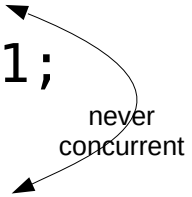
```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
```

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```

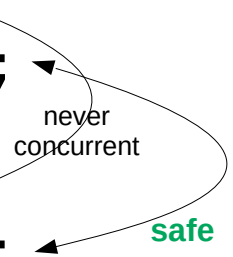
3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```



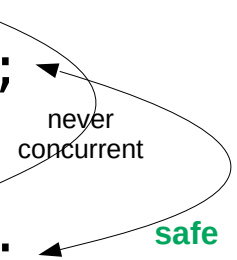
3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```



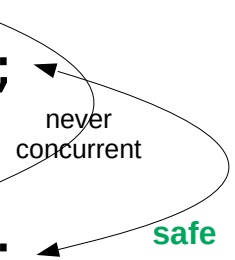
3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```



3. Simple Static Checks

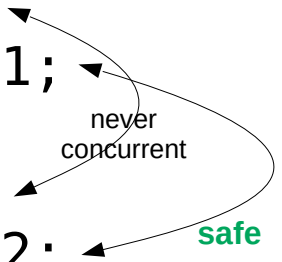
```
input void A, B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
```




```
input void A;
var int y = 1;
```

3. Simple Static Checks

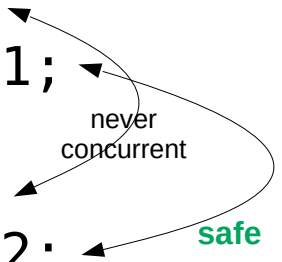
```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```




```
input void A;
var int y = 1;
par/and do
```

3. Simple Static Checks

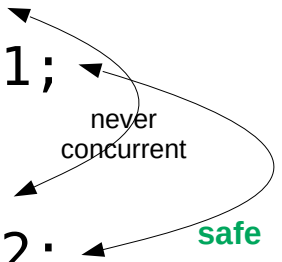
```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```




```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
```

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```




```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```

never concurrent

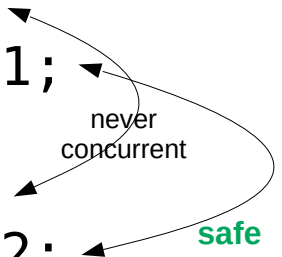
safe

```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```

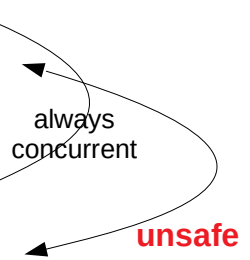
always concurrent

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```

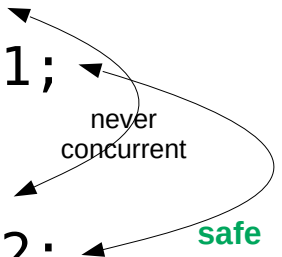



```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```

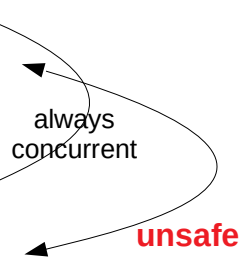


3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```



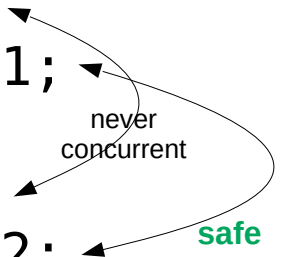

```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```



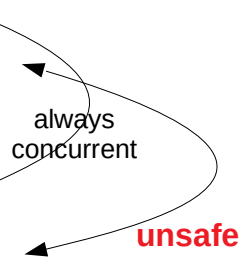
- Static checks

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```



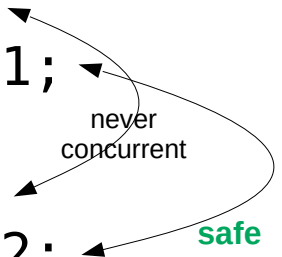

```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```



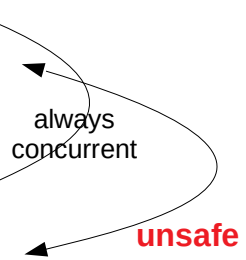
- Static checks
  - Level 0: both are refused

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```



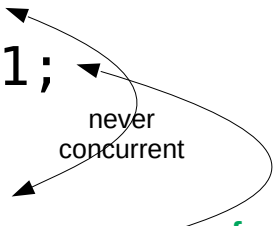

```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```



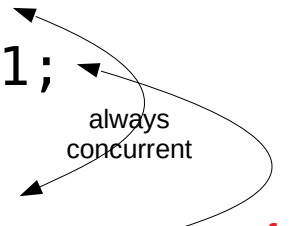
- Static checks
  - Level 0: both are refused
  - Level 1: unsafe is refused

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```



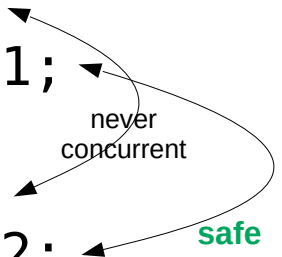

```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```



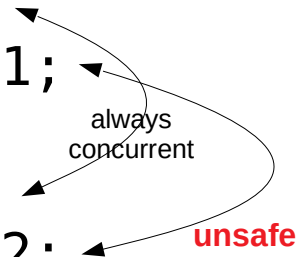
- Static checks
  - Level 0: both are refused
  - Level 1: unsafe is refused
  - Level 2: both are accepted

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```



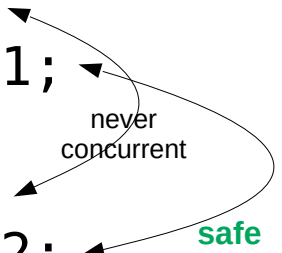

```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```



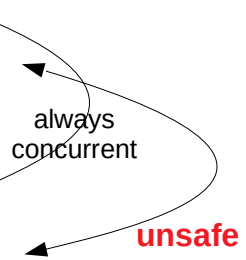
- Static checks
  - Level 0: both are refused
  - Level 1: unsafe is refused
  - Level 2: both are accepted
- Possible because of uniqueness of inputs

3. Simple Static Checks

```
input void A, B;
var int x = 1;
par/and do
   await A;
   x = x + 1;
with
   await B;
   x = x * 2;
end
```

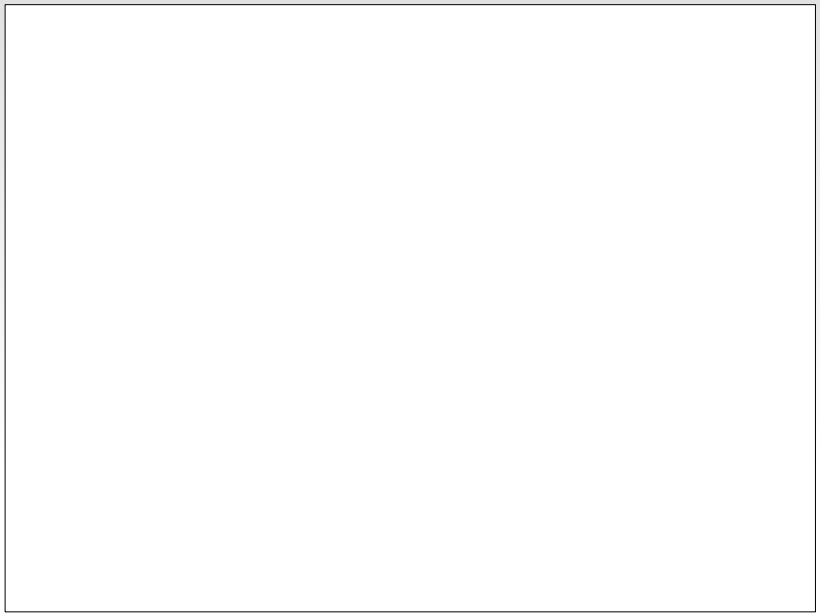



```
input void A;
var int y = 1;
par/and do
   await A;
   y = y + 1;
with
   await A;
   y = y * 2;
end
```



- Static checks
  - Level 0: both are refused
  - Level 1: unsafe is refused
  - Level 2: both are accepted
- Possible because of uniqueness of inputs
- Do not affect the semantics

4. Safe Integration with C



4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end
```

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

- Trackable identifiers `_´ (*C hat*)

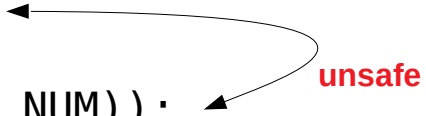
4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```



- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

unsafe

```
native @const _NUM;
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

unsafe

```
native @const _NUM;
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

unsafe

```
native @const _NUM;
native @pure  _id();
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

unsafe

```
native @const _NUM;
native @pure  _id();
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C

```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

unsafe

```
native @const _NUM;
native @pure  _id();
native @safe  _f() with _g();
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C

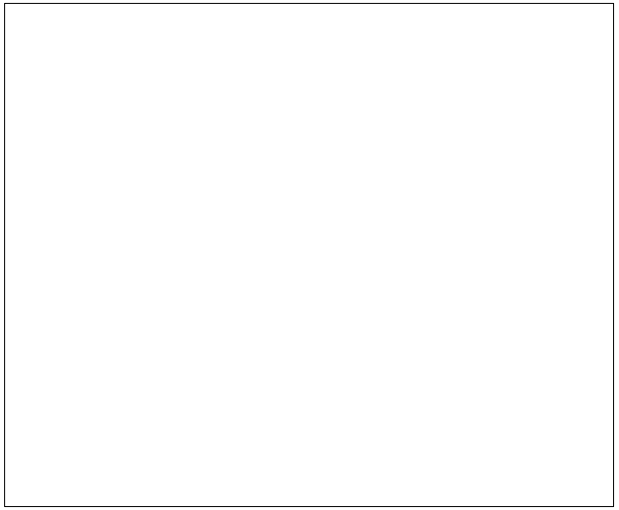
```
native do
   #define NUM 10
   void f  (void)  { <...> }
   void g  (int v) { <...> }
   int  id (int v) { <...> }
end

par/and do
   _f();
with
   _g(_id(_NUM));
end
```

unsafe

```
native @const _NUM;
native @pure  _id();
native @safe  _f() with _g();
```

- Trackable identifiers `_´ (*C hat*)
- Assumes all identifiers are conflicting
- Annotations to eliminate conflicts

4. Safe Integration with C



4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

```
par/or do
   var _FILE* f;
   f = _fopen(...);
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
   _fclose(f);
with
   <...>
end
```

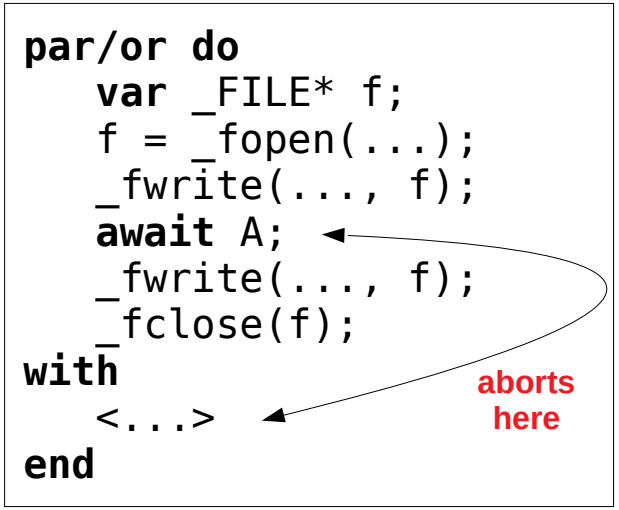
4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe

```
par/or do
   var _FILE* f;
   f = _fopen(...);
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
   _fclose(f);
with
   <...>
end
```

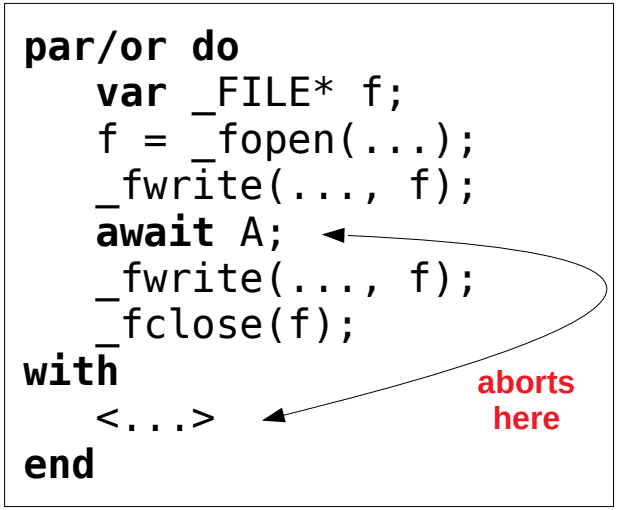
4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism



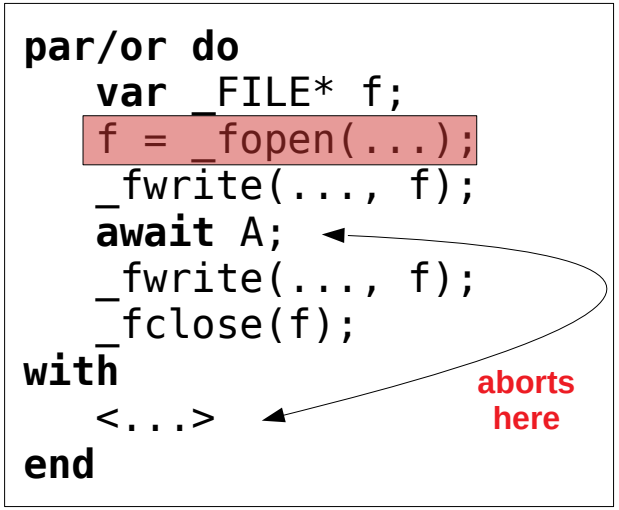
4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized



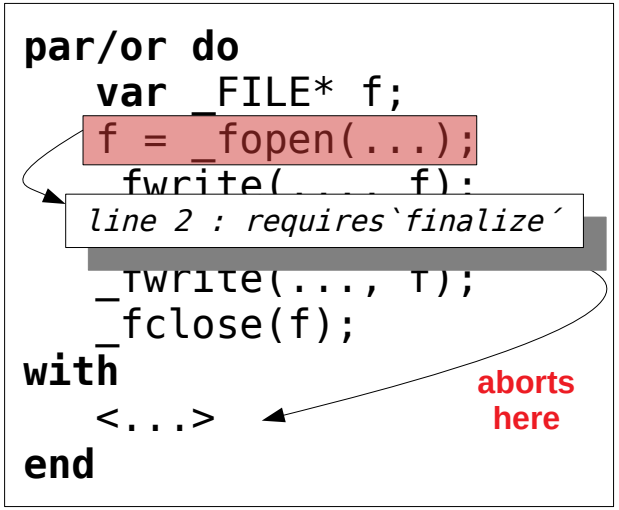
4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized



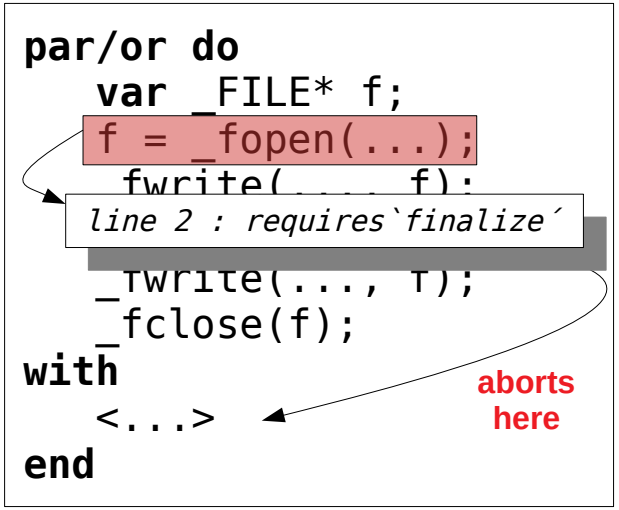
4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized



4. Safe Integration with C

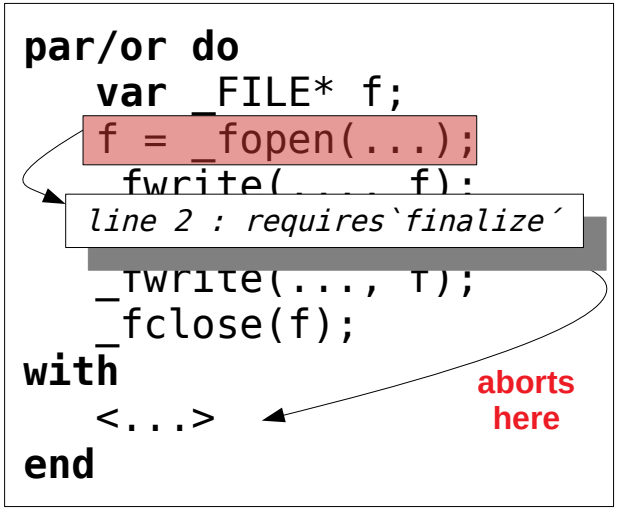
- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized




```
par/or do
   var _FILE* f;
```

4. Safe Integration with C

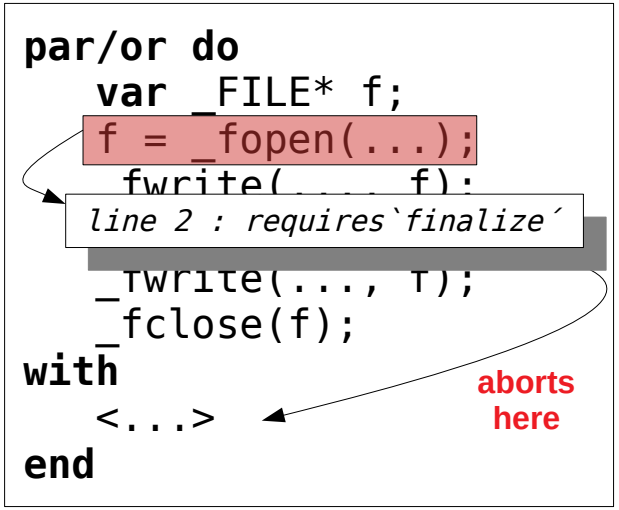
- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized




```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

4. Safe Integration with C

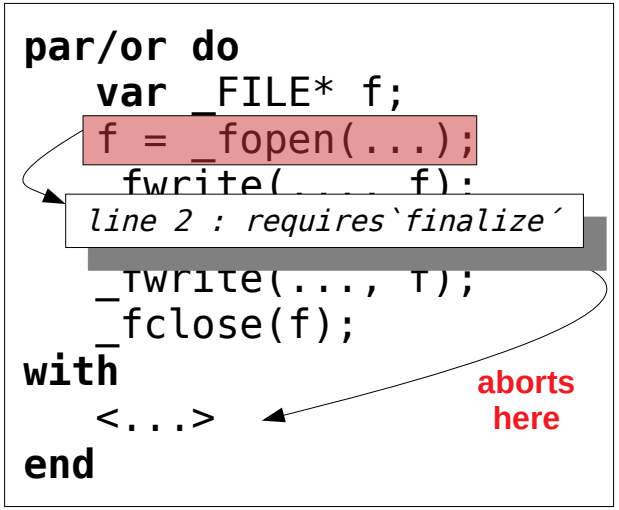
- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)




```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

4. Safe Integration with C

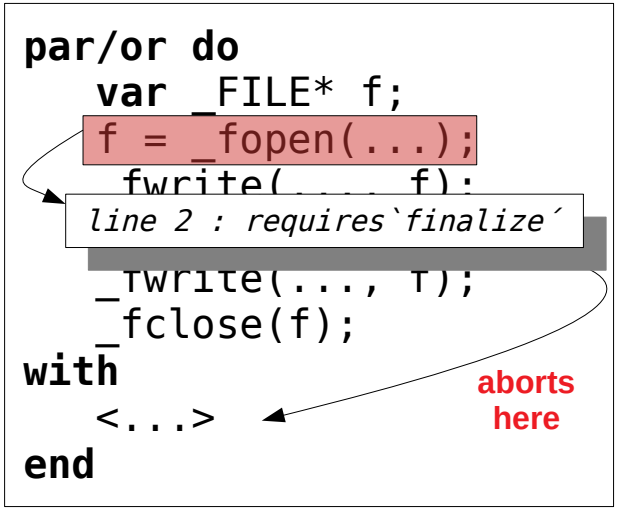
- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)




```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

4. Safe Integration with C

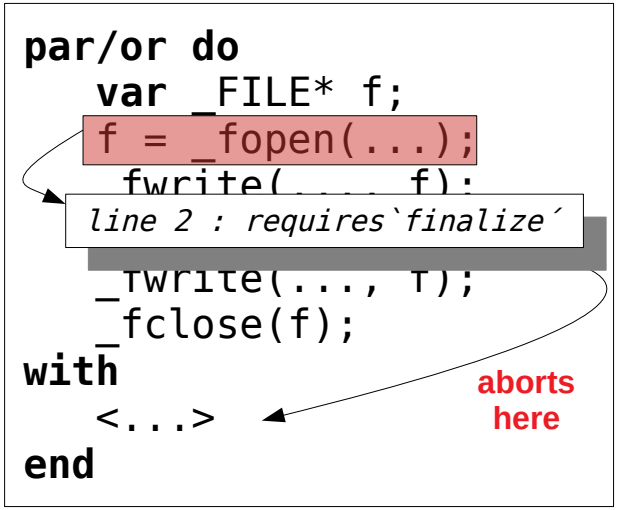
- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)




```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)

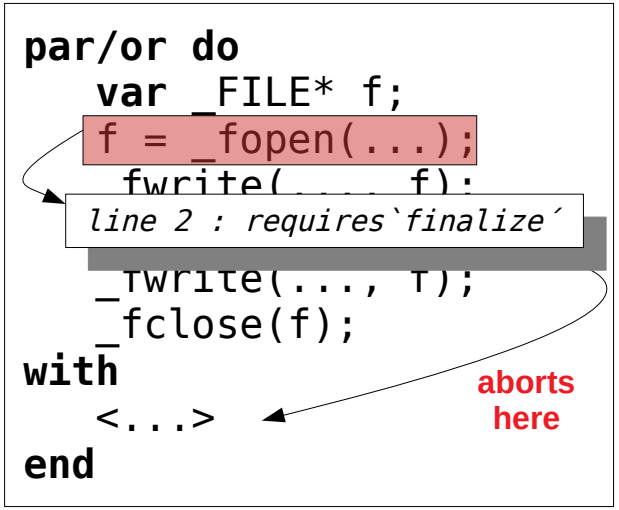



```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

```
par/or do
```

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)




```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

```
par/or do
   var _buffer_t msg;
```

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)

```
par/or do
   var _FILE* f;
   f = _fopen(...);
   _fwrite(..., f);
   _fclose(f);
with
   <...>
end
```

line 2 : requires`finalize´

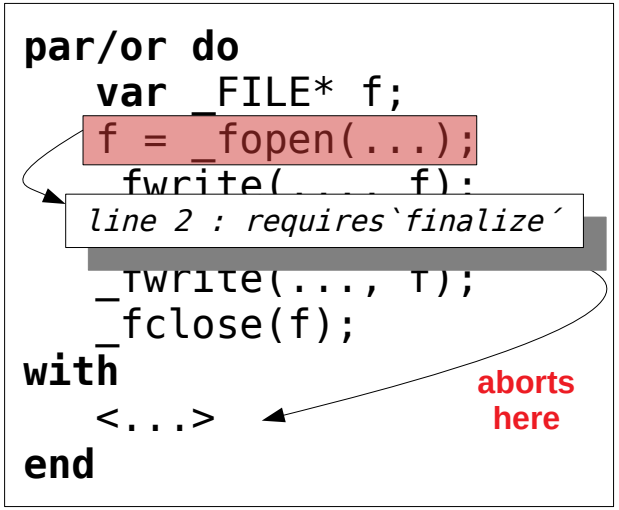
aborts here

```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

```
par/or do
   var _buffer_t msg;
   <...> // prepare msg
```

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)




```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

```
par/or do
   var _buffer_t msg;
   <...> // prepare msg
   finalize
      _send_request(&msg);
   with
      _send_cancel(&msg);
   end
```

4. Safe Integration with C

- Abortion of trails dealing with resources is unsafe
- Finalization mechanism
  - Pointer assignment must be finalized
  - External resource: pointer from C to Céu (*memory leak*)
  - Local resource: pointer from Céu to C (*dangling pointer*)

```
par/or do
   var _FILE* f;
   f = _fopen(...);
   _fwrite(..., f);
   _fclose(f);
with
   <...>
end
```

line 2 : requires `finalize´

aborts here

```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await A;
   _fwrite(..., f);
with
   <...>
end
```

```
par/or do
   var _buffer_t msg;
   <...> // prepare msg
   finalize
      _send_request(&msg);
   with
      _send_cancel(&msg);
   end
   await SEND_ACK;
with
   <...>
end
```

5. First-Class Timers

5. First-Class Timers

- Timers, watchdogs, sampling all very common

5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax

5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax

```
var int v;
await 10ms;
v = 1;
await 1ms;
v = 2;
```

5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)

```
var int v;
await 10ms;
v = 1;
await 1ms;
v = 2;
```

5. First-Class Timers

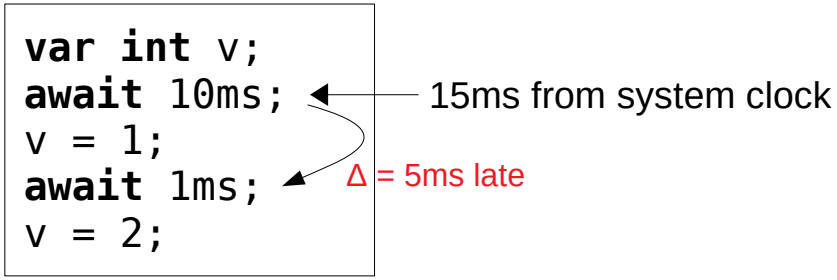
- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)

```
var int v;
await 10ms;
v = 1;
await 1ms;
v = 2;
```

15ms from system clock

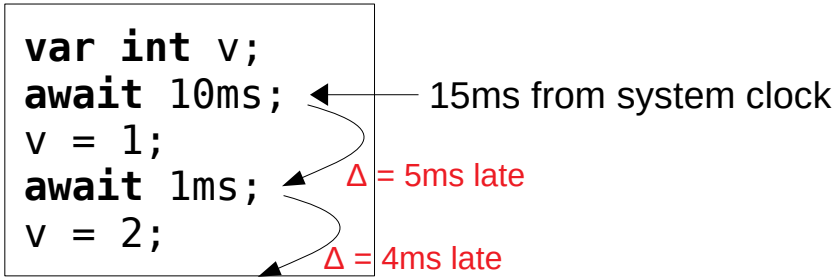
5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)



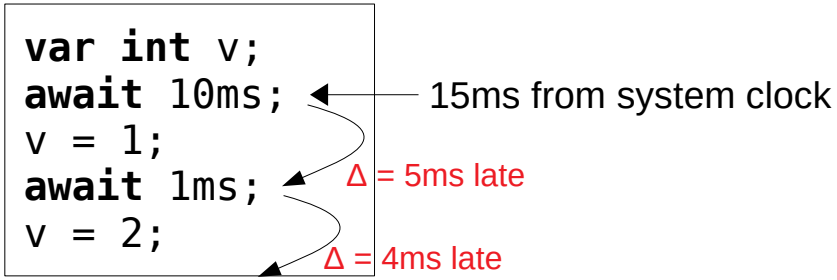
5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)



5. First-Class Timers

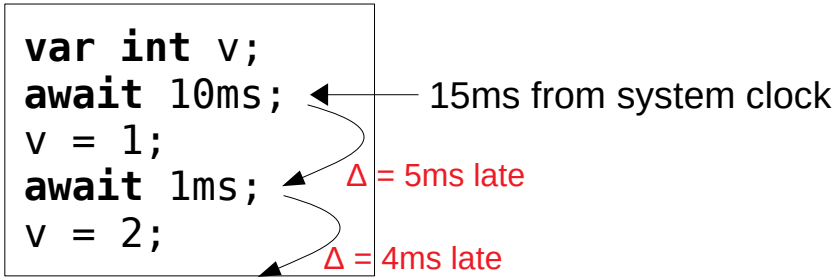
- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)




```
var int v;
par/or do
   await 10ms;
   await 1ms;
   v = 1;
with
   await 12ms;
   v = 2;
end
```

5. First-Class Timers

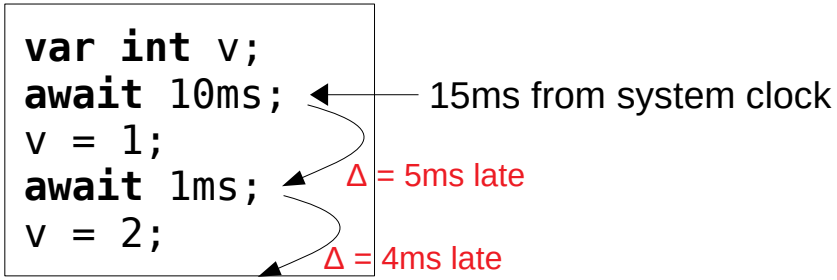
- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)




```
var int v;
par/or do
   await 10ms;
   await 1ms;
1  v = 1;
with
   await 12ms;
   v = 2;
end
```

5. First-Class Timers

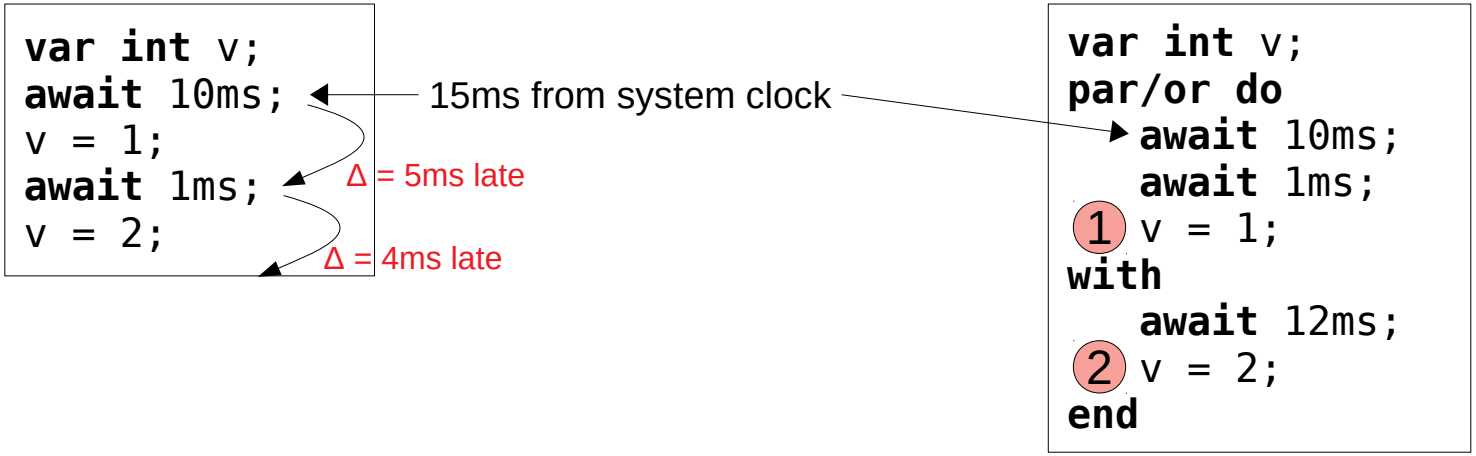
- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)




```
var int v;
par/or do
    await 10ms;
    await 1ms;
1   v = 1;
with
    await 12ms;
2   v = 2;
end
```

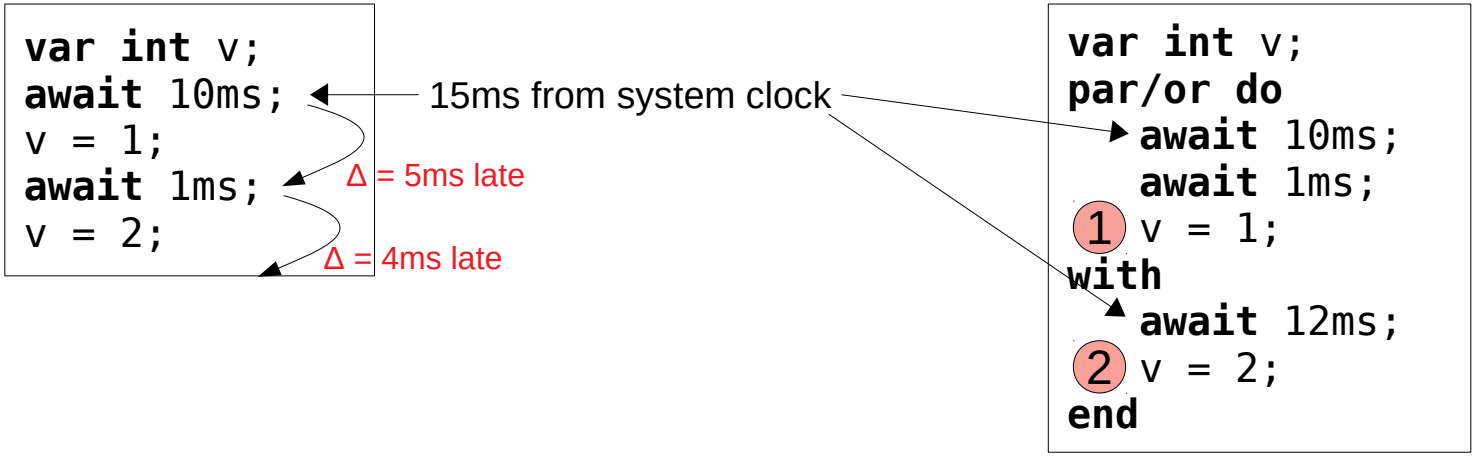
5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)



5. First-Class Timers

- Timers, watchdogs, sampling all very common
  1. Dedicated syntax
  2. Delta compensation (system vs program mismatch)



Applications / Other Work

Applications / Other Work

- ~10-year effort (first commit in 2011)

Applications / Other Work

- ~10-year effort (first commit in 2011)

- `[games]` *Structured Synchronous Reactive Programming for Game Development Case Study: On Rewriting Pingus from C++ to Céu*, SBGames, 2018

Applications / Other Work

- ~10-year effort (first commit in 2011)

- `[games]` *Structured Synchronous Reactive Programming for Game Development Case Study: On Rewriting Pingus from C++ to Céu*, SBGames, 2018
  - **10k/40k reactive code rewritten**
- `[embed]` *Transparent Standby for Low-Power, Resource-Constrained Embedded Systems: A Programming Language-Based Approach*, LCTES, 2018

Applications / Other Work

- ~10-year effort (first commit in 2011)

- `[games]` *Structured Synchronous Reactive Programming for Game Development Case Study: On Rewriting Pingus from C++ to Céu*, SBGames, 2018
  - **10k/40k reactive code rewritten**
- `[embed]` *Transparent Standby for Low-Power, Resource-Constrained Embedded Systems: A Programming Language-Based Approach*, LCTES, 2018
  - **interrupt-service routines, automatic standby**
- `[media]` *Céu-Media: Local Inter-Media Synchronization Using Céu*, WebMedia, 2016

Applications / Other Work

- ~10-year effort (first commit in 2011)

- `[games]` *Structured Synchronous Reactive Programming for Game Development Case Study: On Rewriting Pingus from C++ to Céu*, SBGames, 2018
  - **10k/40k reactive code rewritten**
- `[embed]` *Transparent Standby for Low-Power, Resource-Constrained Embedded Systems: A Programming Language-Based Approach*, LCTES, 2018
  - **interrupt-service routines, automatic standby**
- `[media]` *Céu-Media: Local Inter-Media Synchronization Using Céu*, WebMedia, 2016
  - **multimedia applications (videos, slideshows)**
- `[wsns]` *Terra: Flexibility and Safety in Wireless Sensor Networks*, TOSN, 2015

Applications / Other Work

- ~10-year effort (first commit in 2011)

- `[games]` *Structured Synchronous Reactive Programming for Game Development Case Study: On Rewriting Pingus from C++ to Céu*, SBGames, 2018
  - **10k/40k reactive code rewritten**
- `[embed]` *Transparent Standby for Low-Power, Resource-Constrained Embedded Systems: A Programming Language-Based Approach*, LCTES, 2018
  - **interrupt-service routines, automatic standby**
- `[media]` *Céu-Media: Local Inter-Media Synchronization Using Céu*, WebMedia, 2016
  - **multimedia applications (videos, slideshows)**
- `[wsns]` *Terra: Flexibility and Safety in Wireless Sensor Networks*, TOSN, 2015
  - **remote reprogramming**

Céu Peculiarities

1. External events
   - time is a queue of unique external events
2. Internal events
   - intra reactions, stack based
3. Concurrency: internal determinism + static checks
   - simple, concurrent assignments/system calls
4. Safe integration with C
   - finalization for local/external resources
5. First-class timers
   - dedicated syntax, automatic synchronization

An Overview of Céu

A synchronous language inspired by Esterel

Francisco Sant'Anna

francisco@ime.uerj.br

www.ceu-lang.org

1. External Events

1. External Events

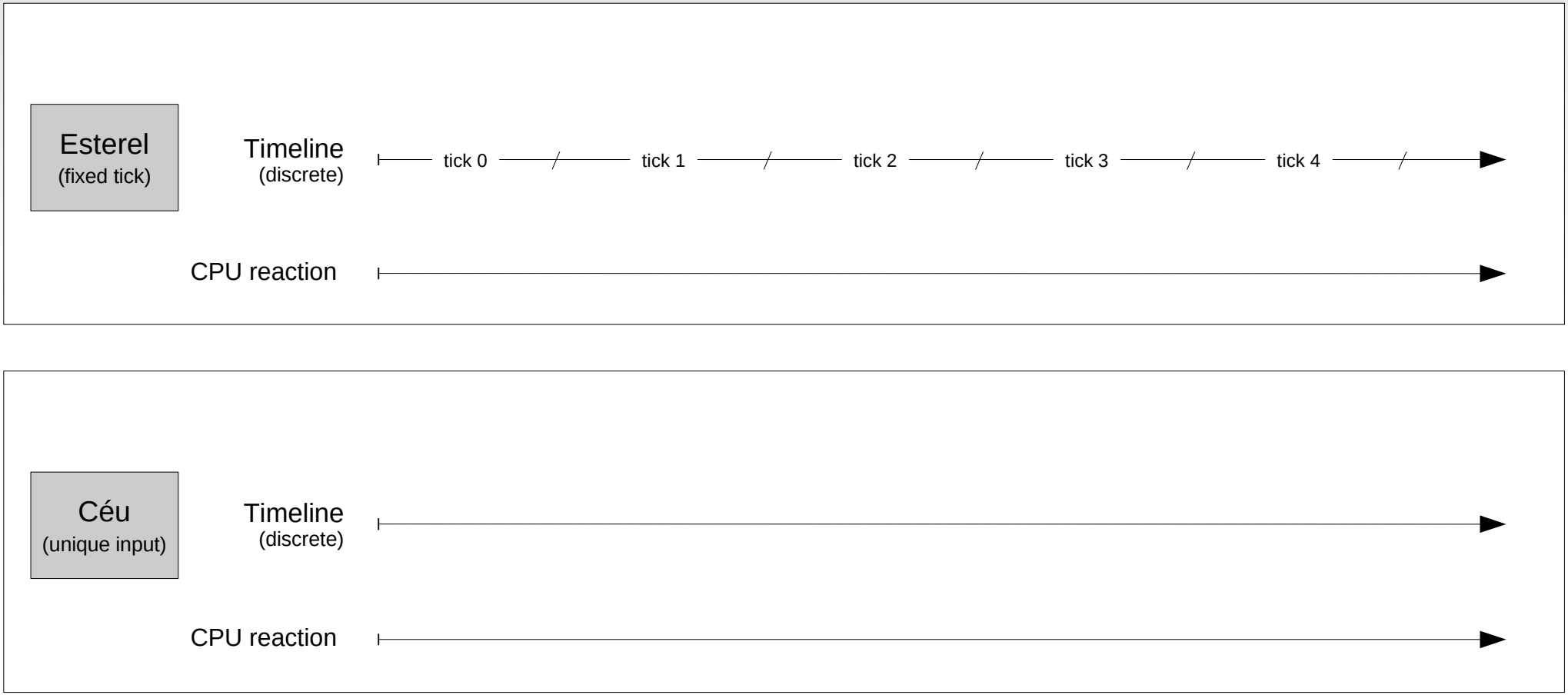
Esterel
(fixed tick)

Céu
(unique input)

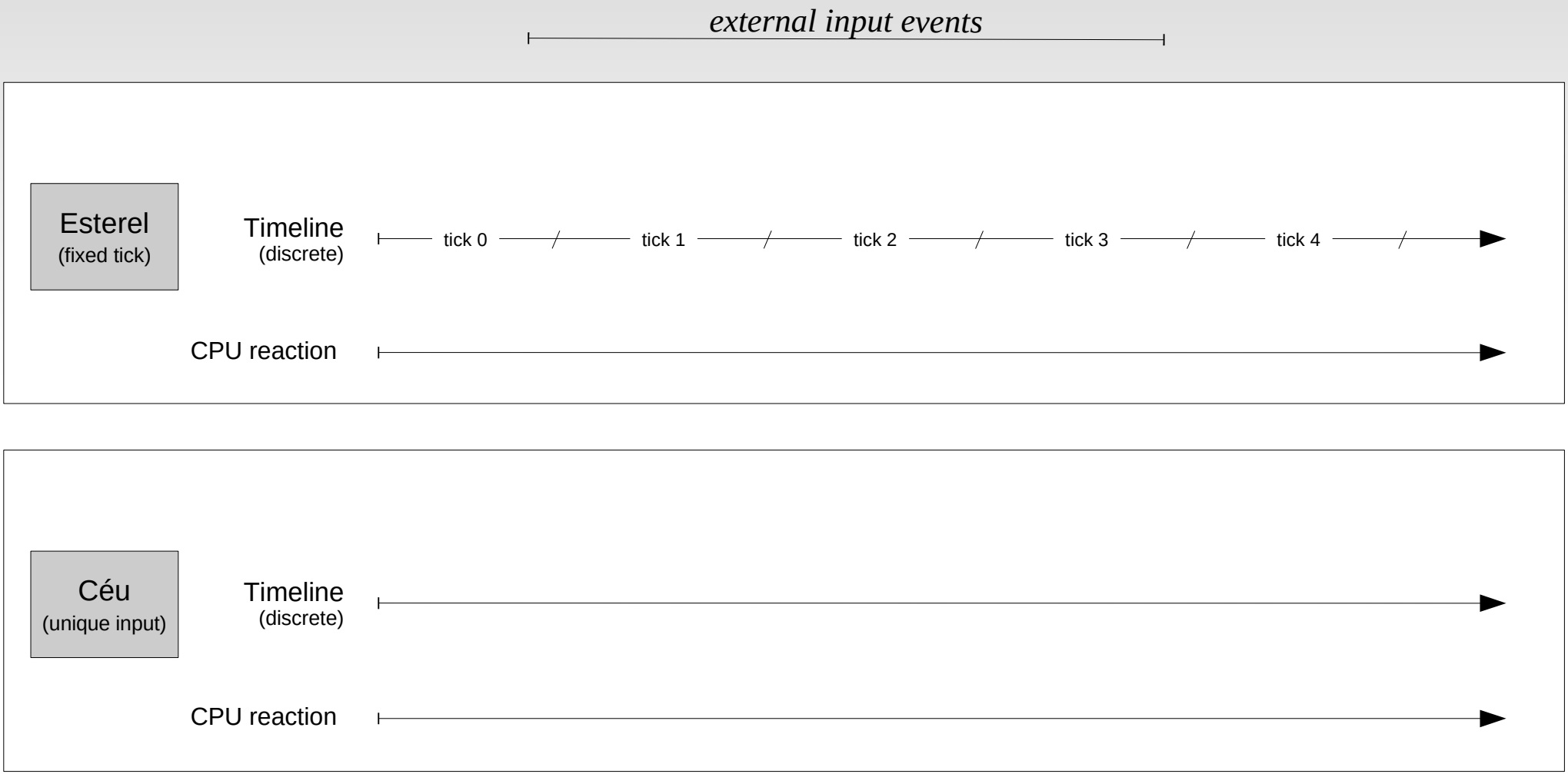
1. External Events

1. External Events

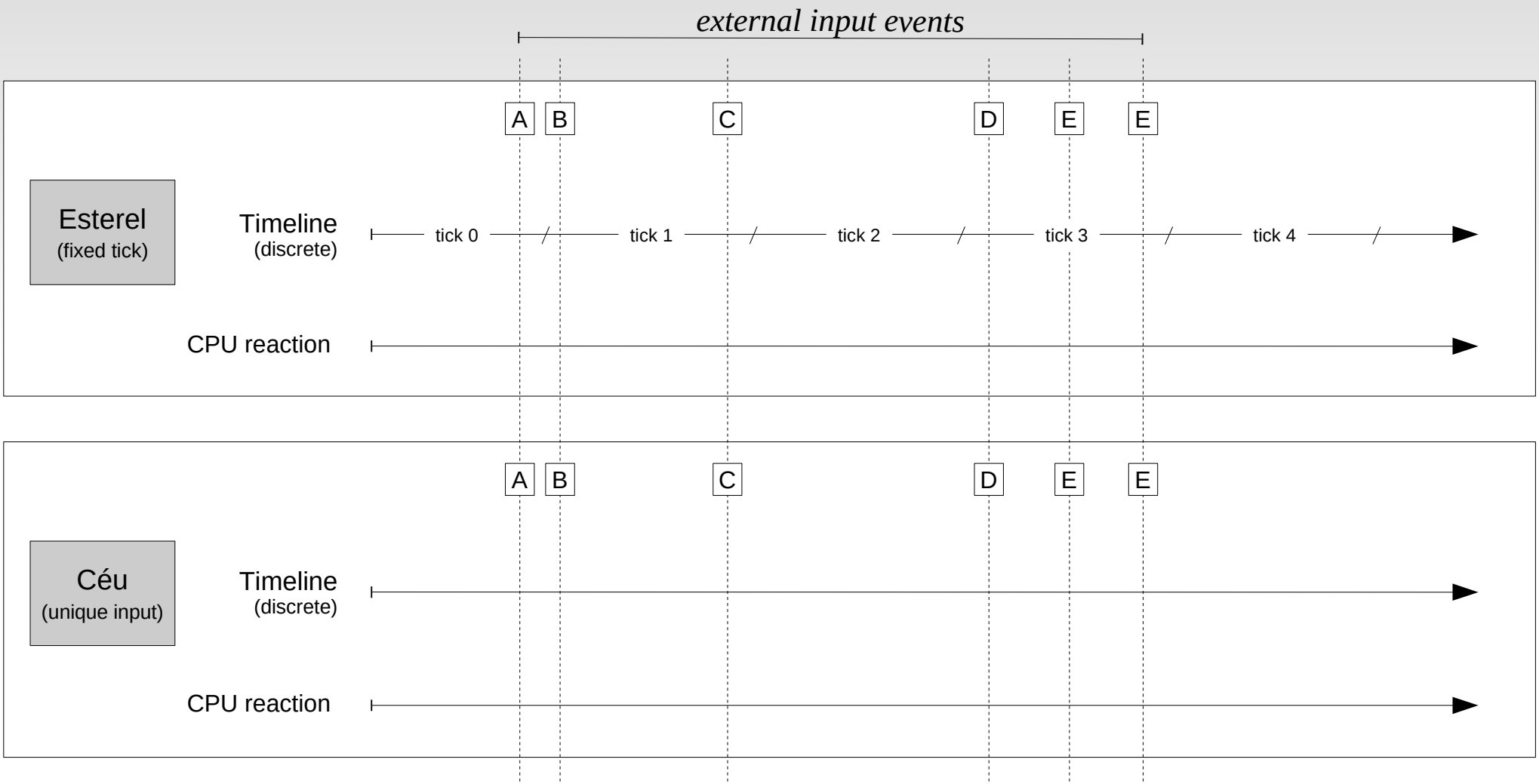
1. External Events



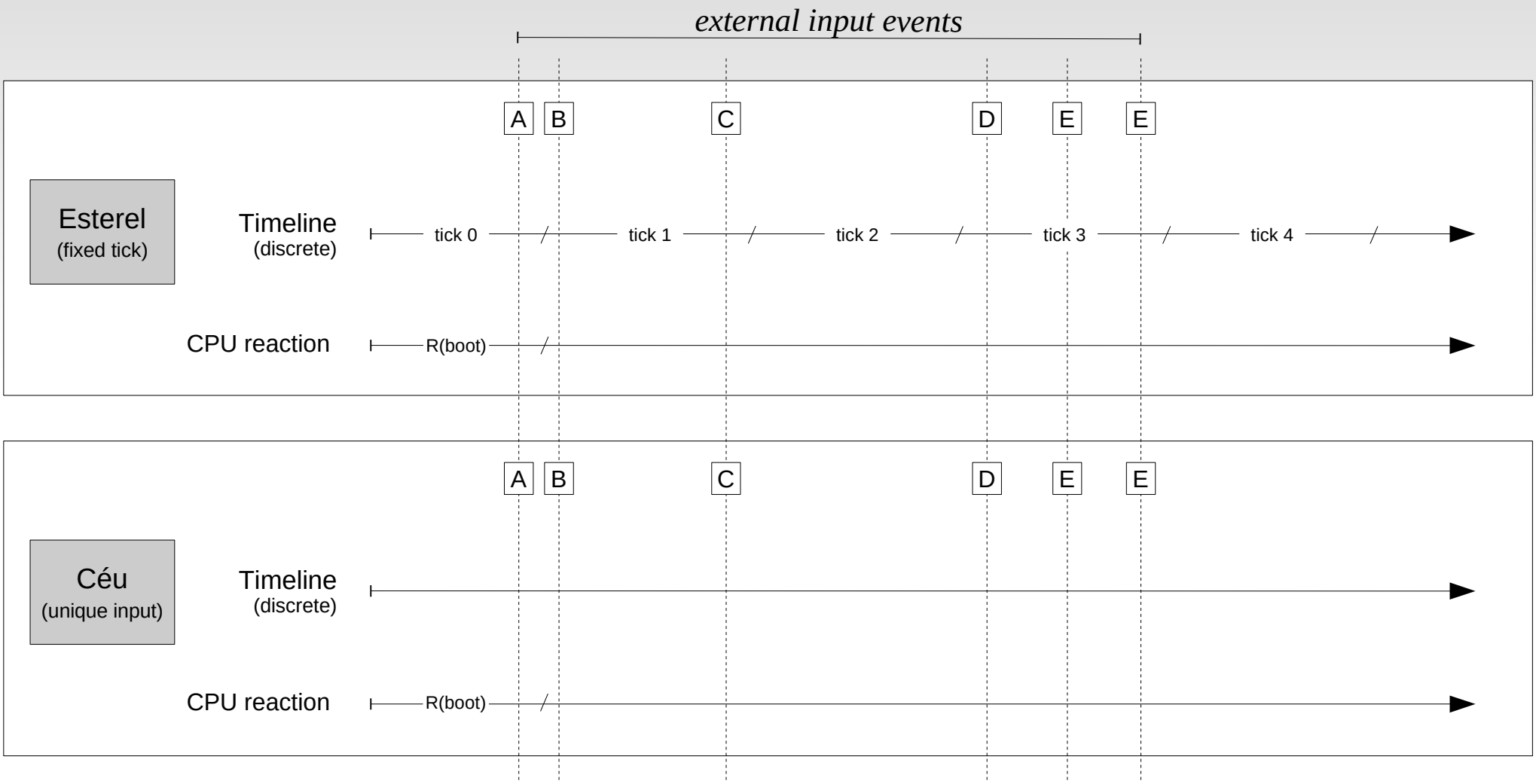
1. External Events



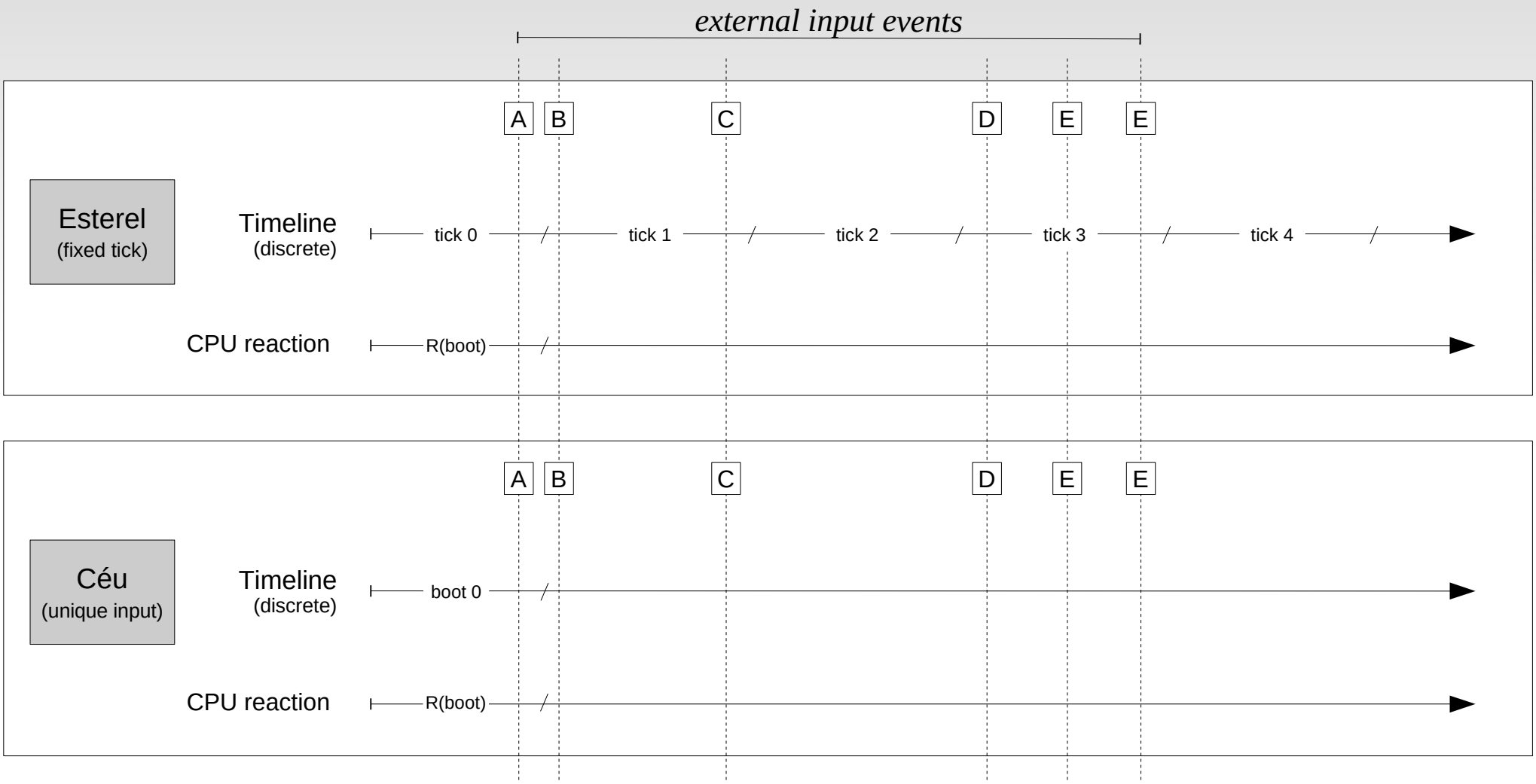
1. External Events



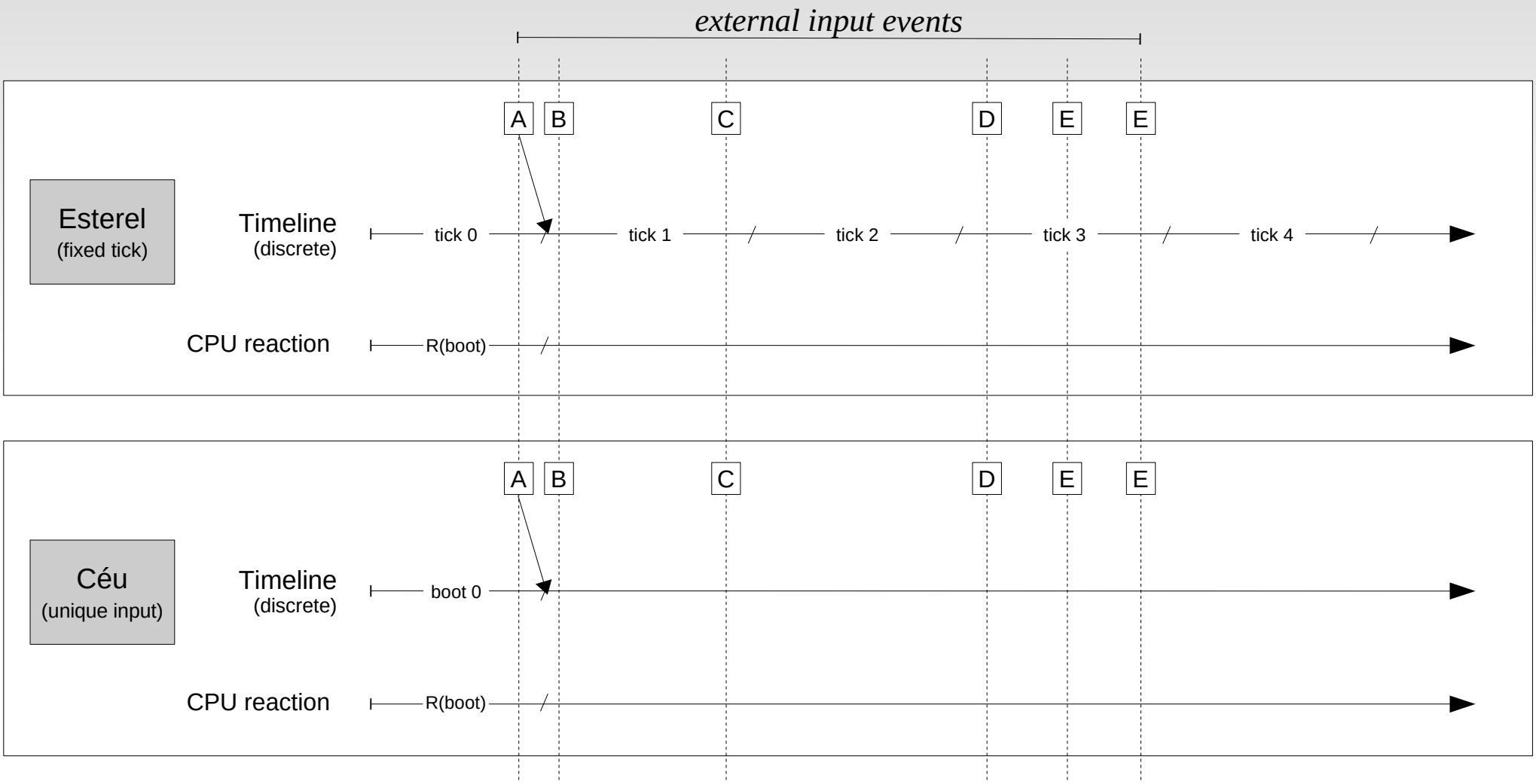
1. External Events



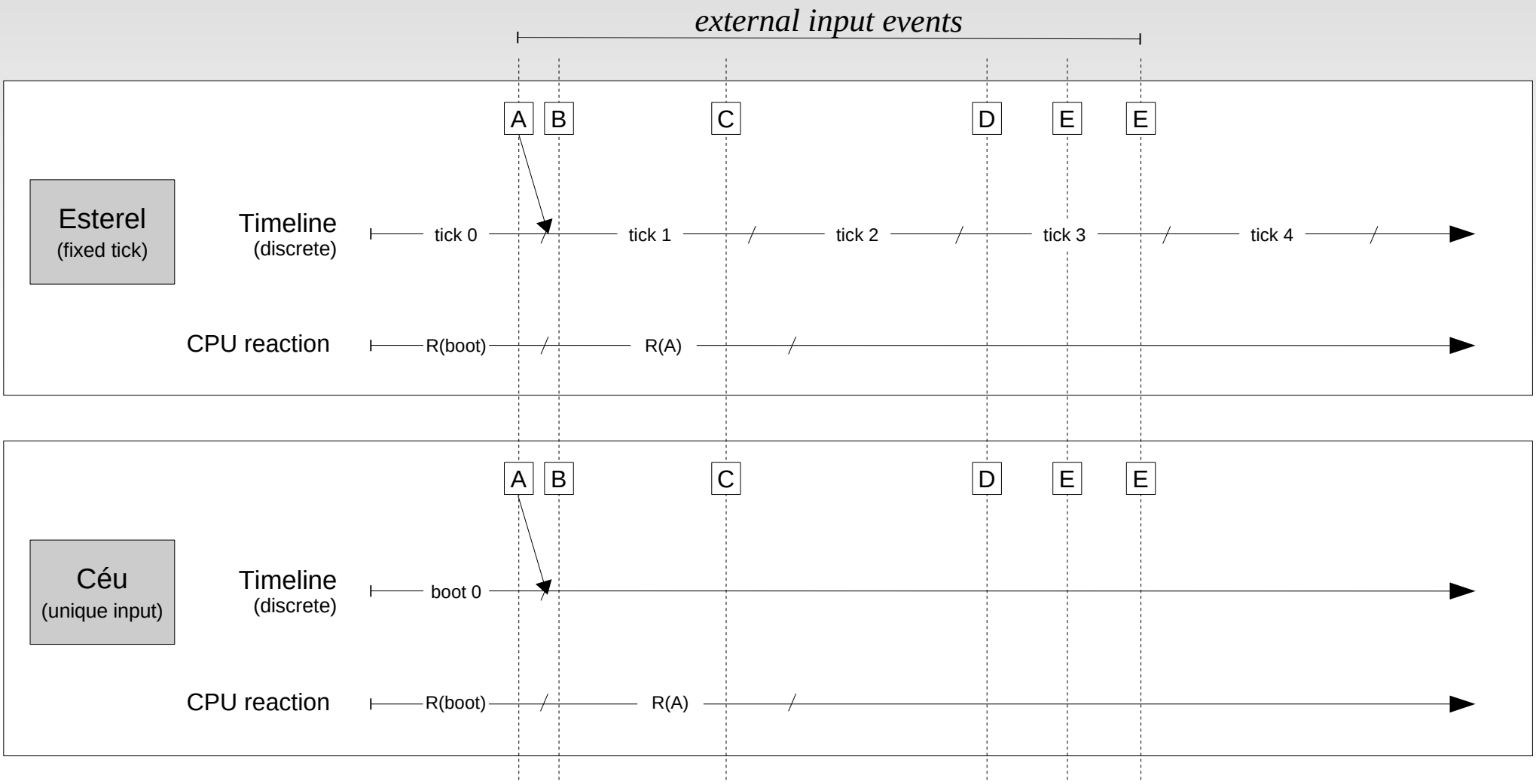
1. External Events



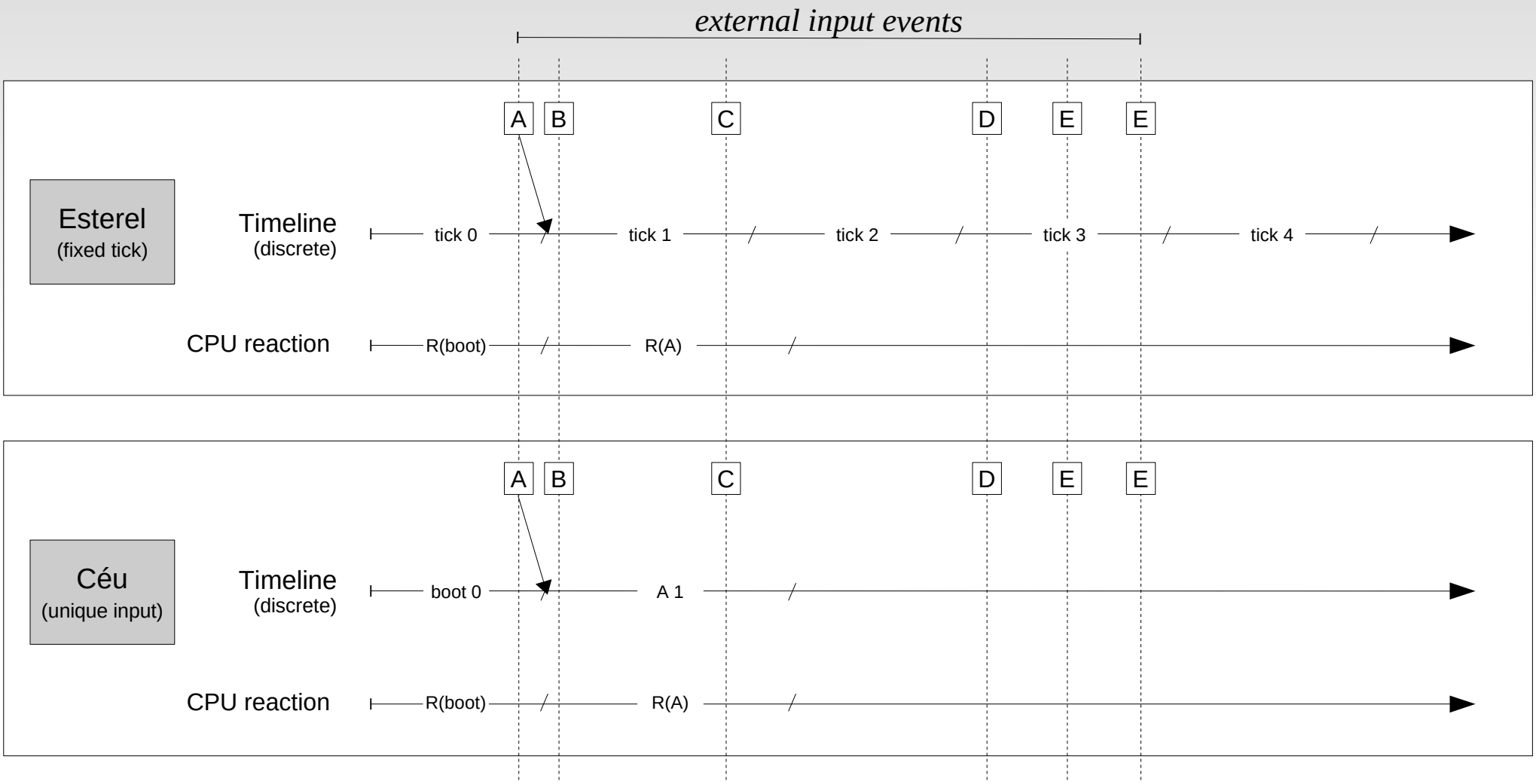
1. External Events



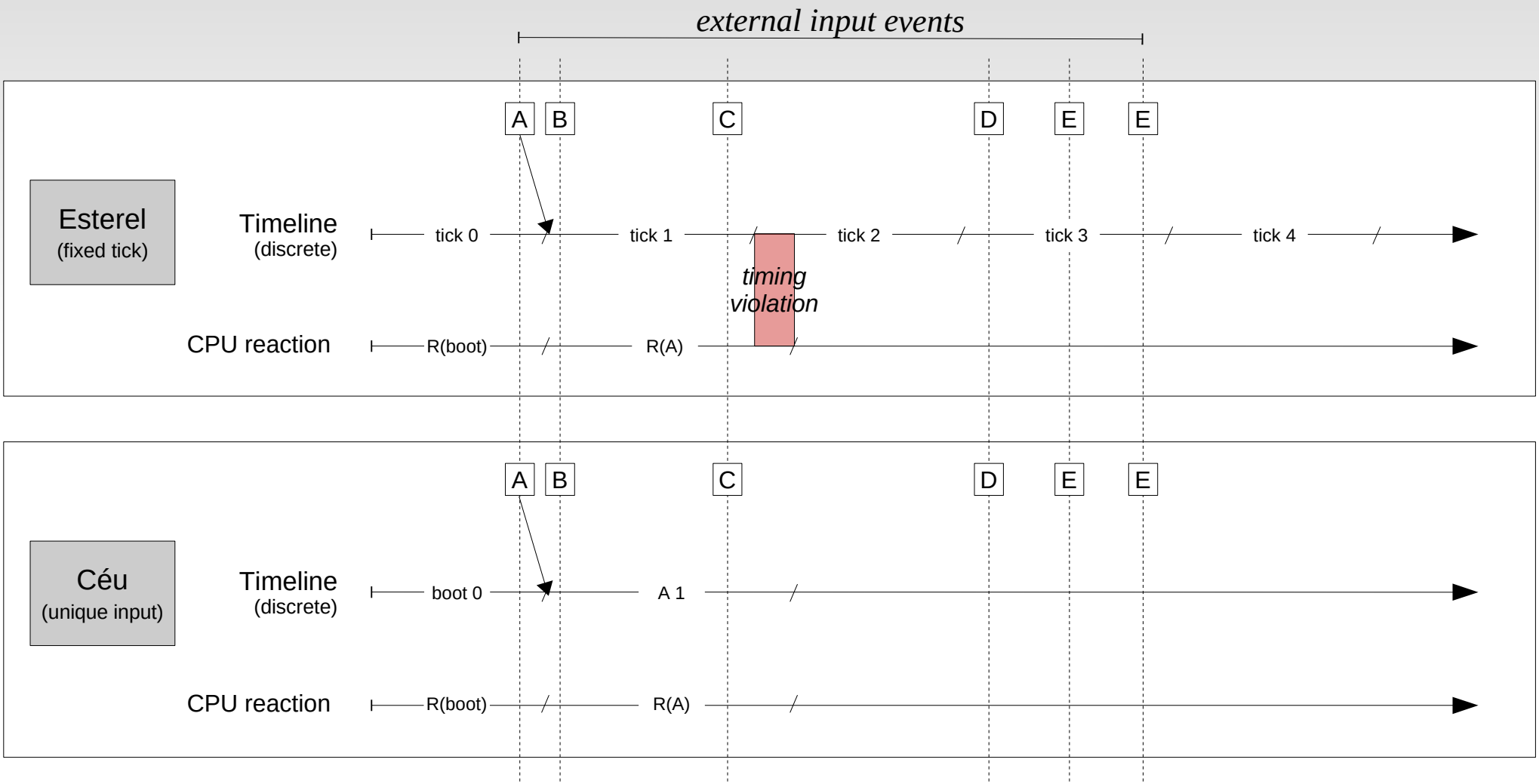
1. External Events



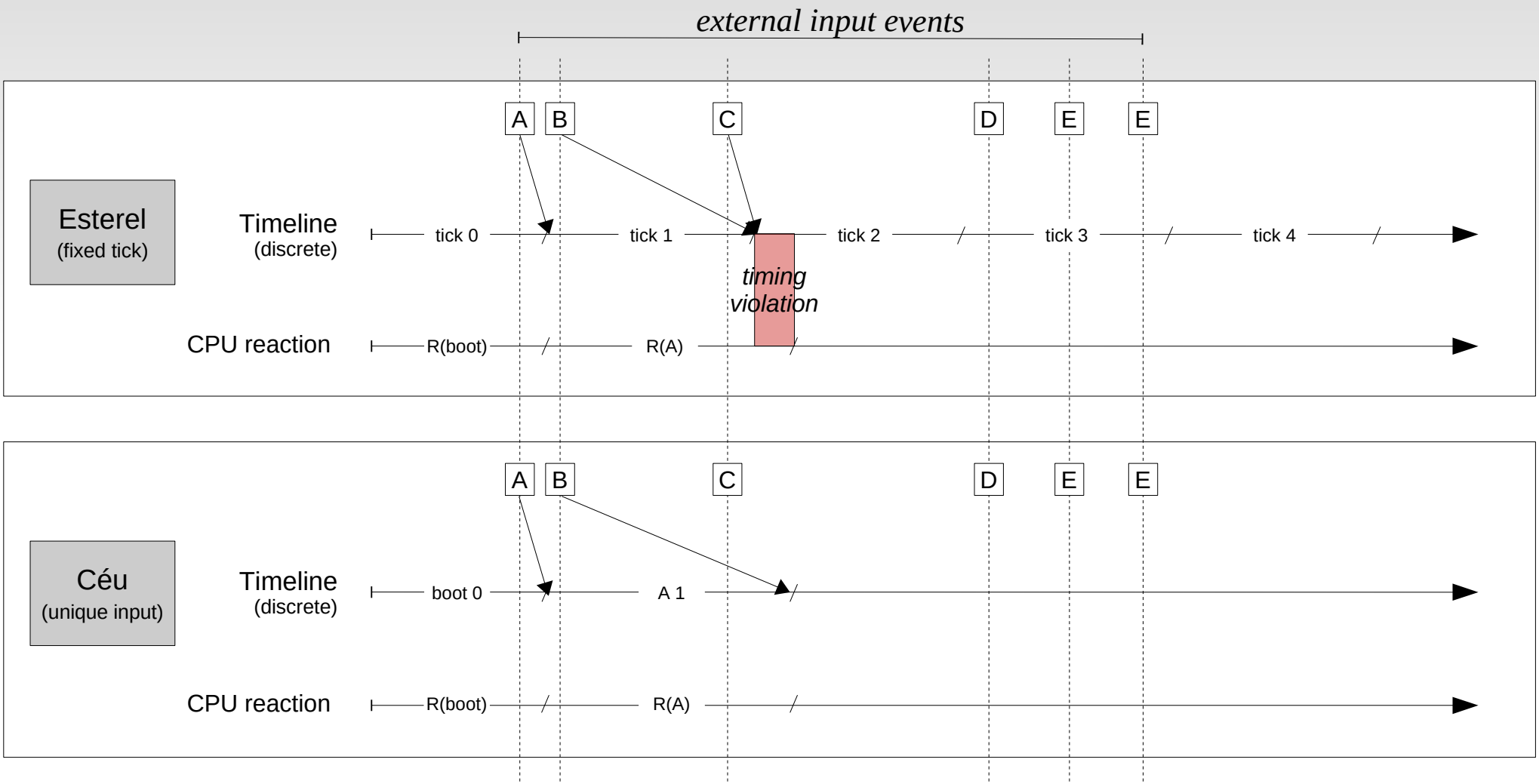
1. External Events



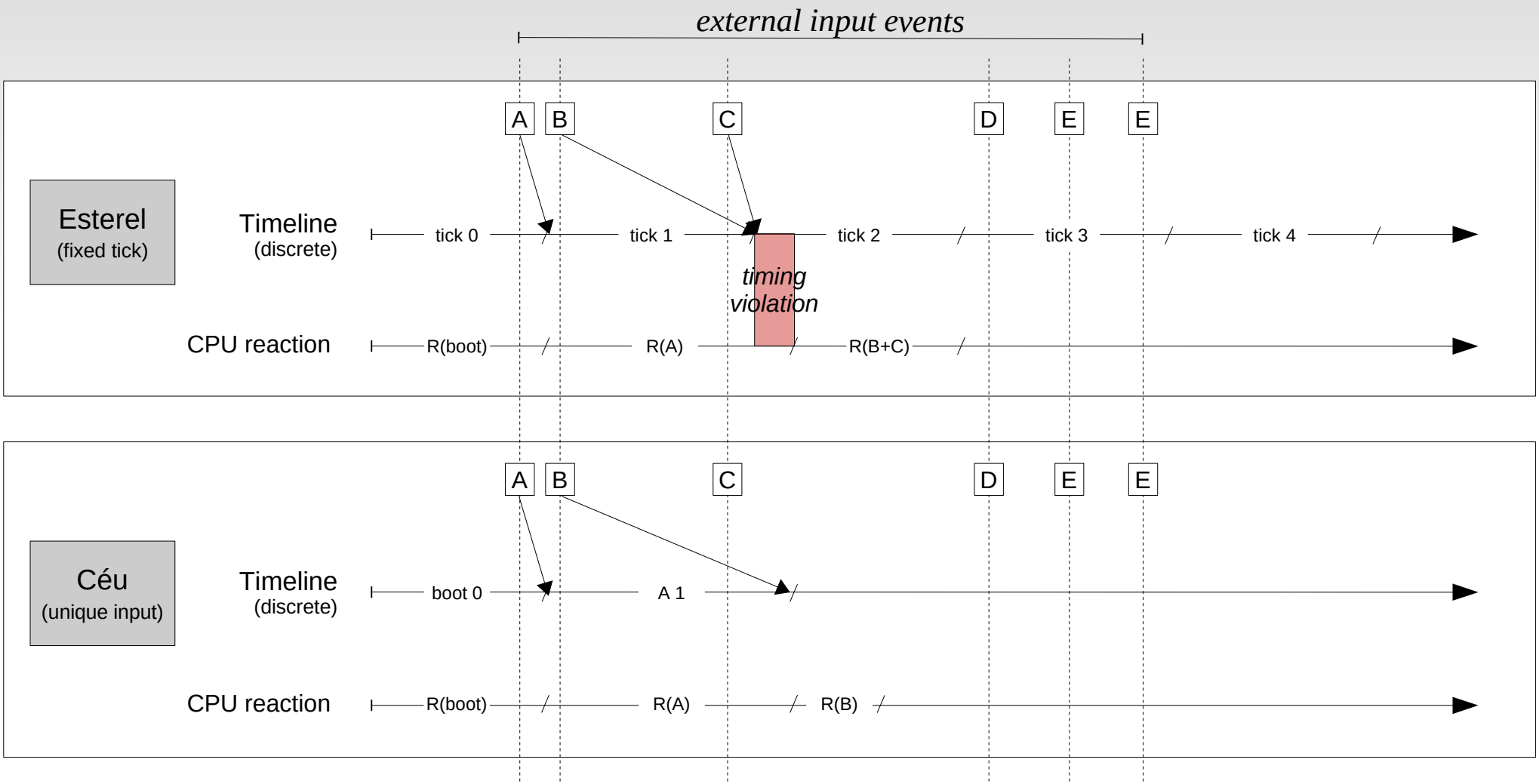
1. External Events



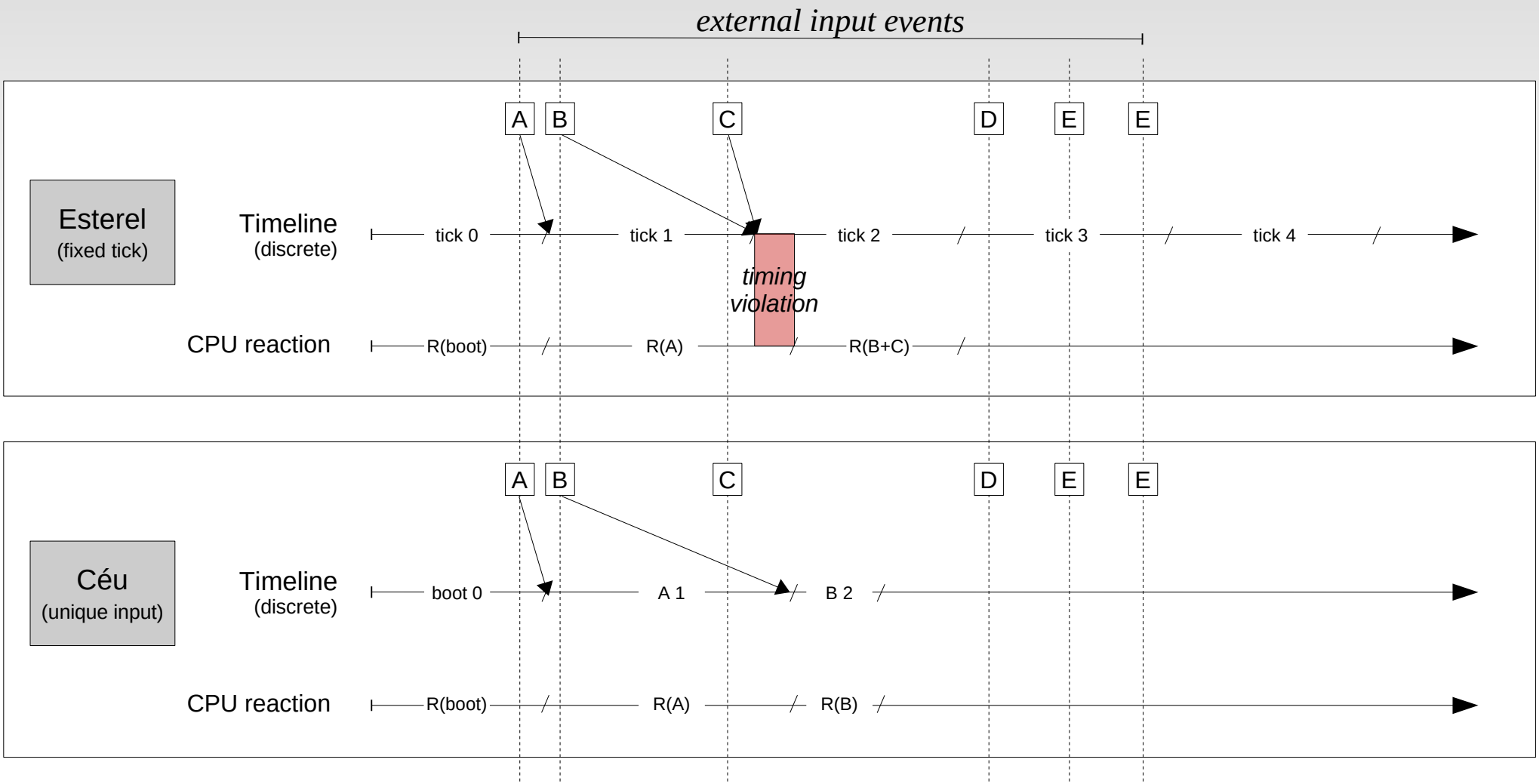
1. External Events



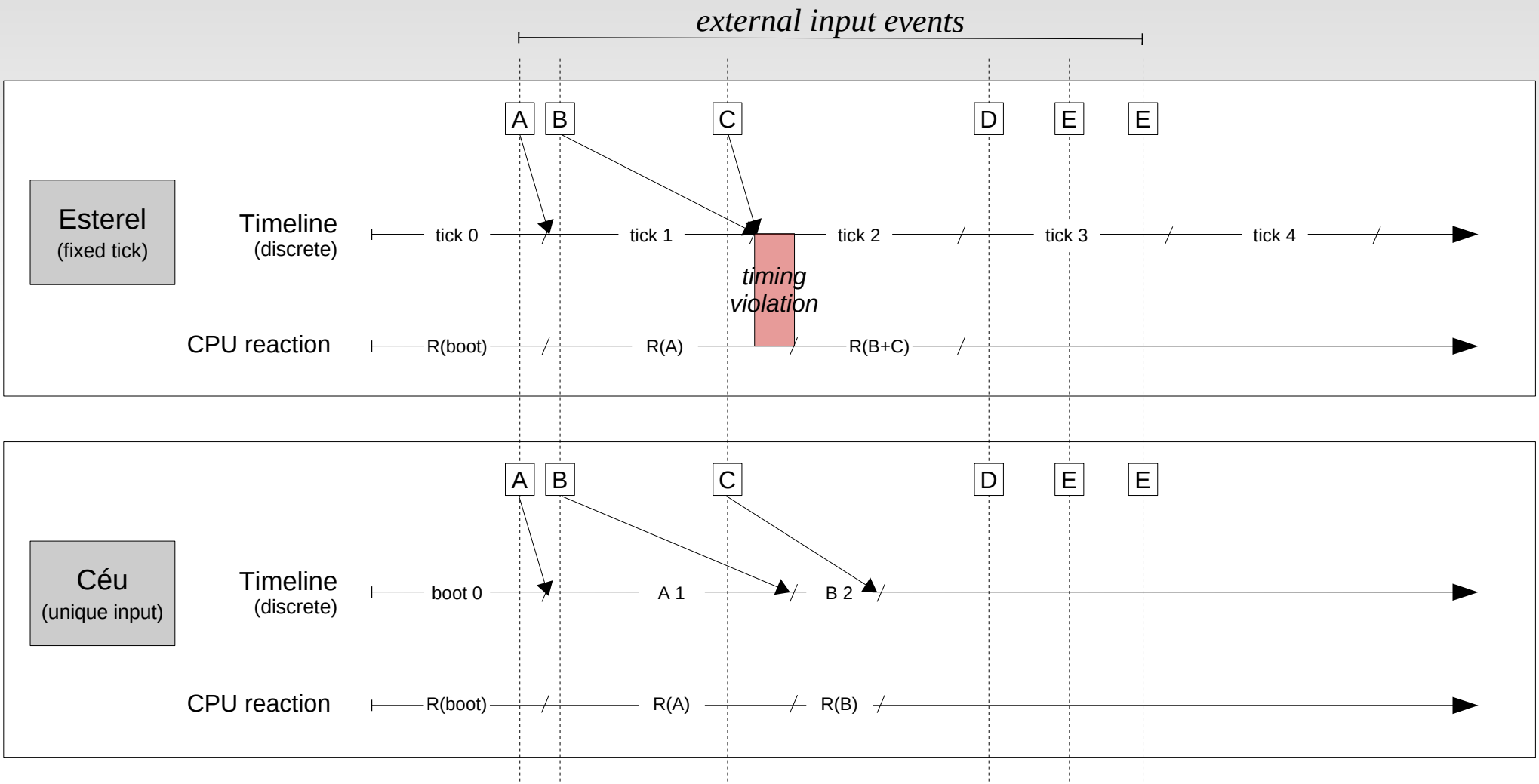
1. External Events



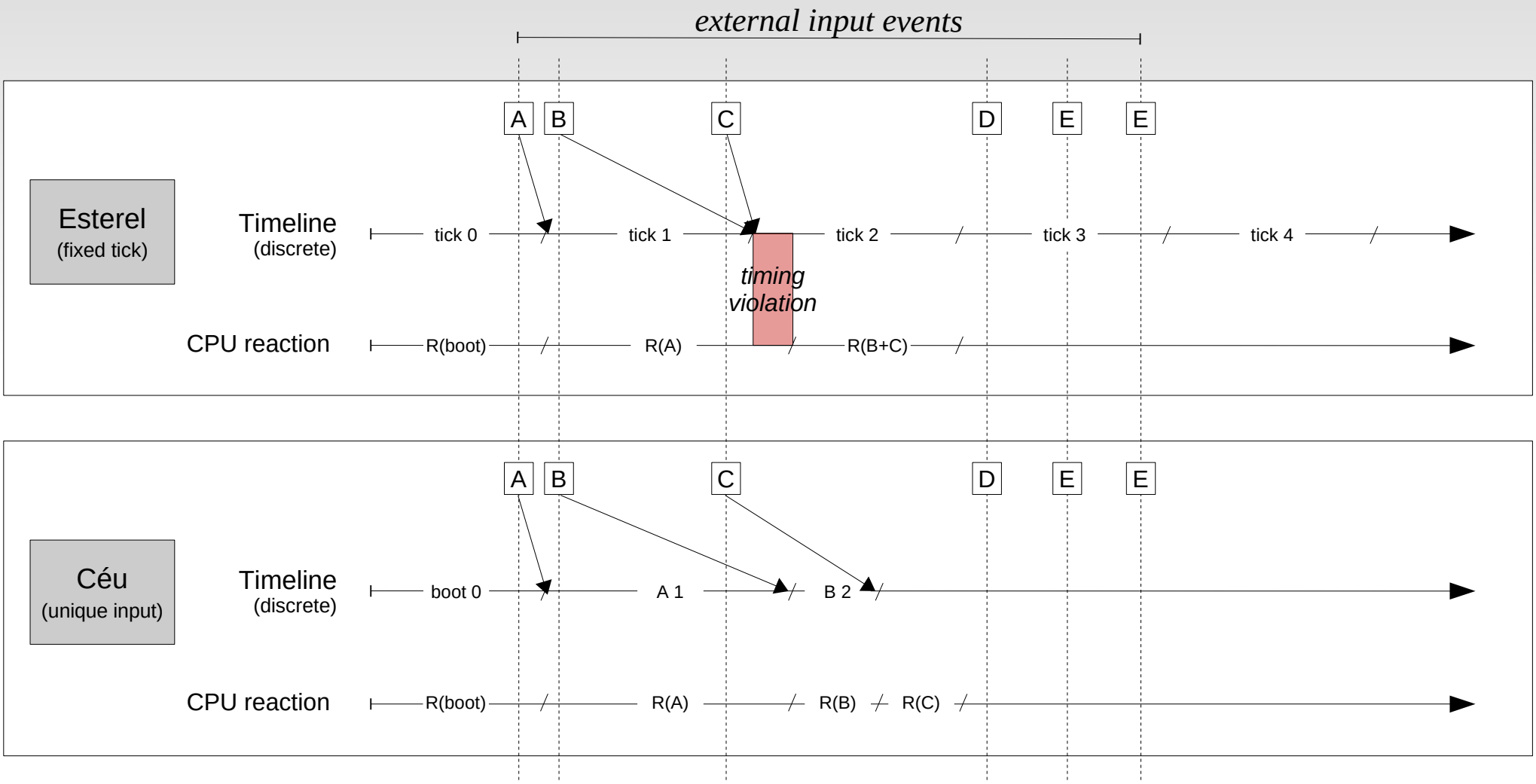
1. External Events



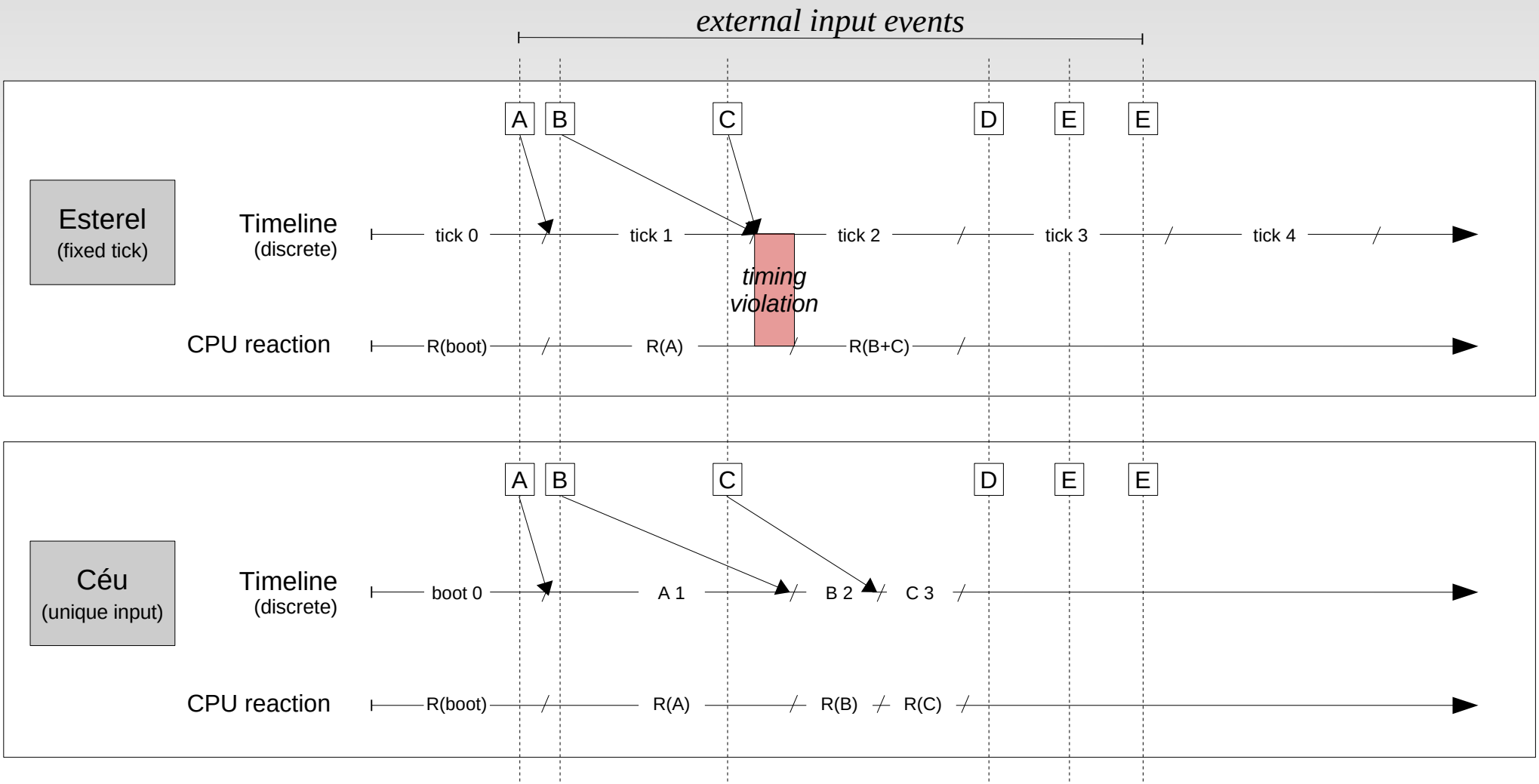
1. External Events



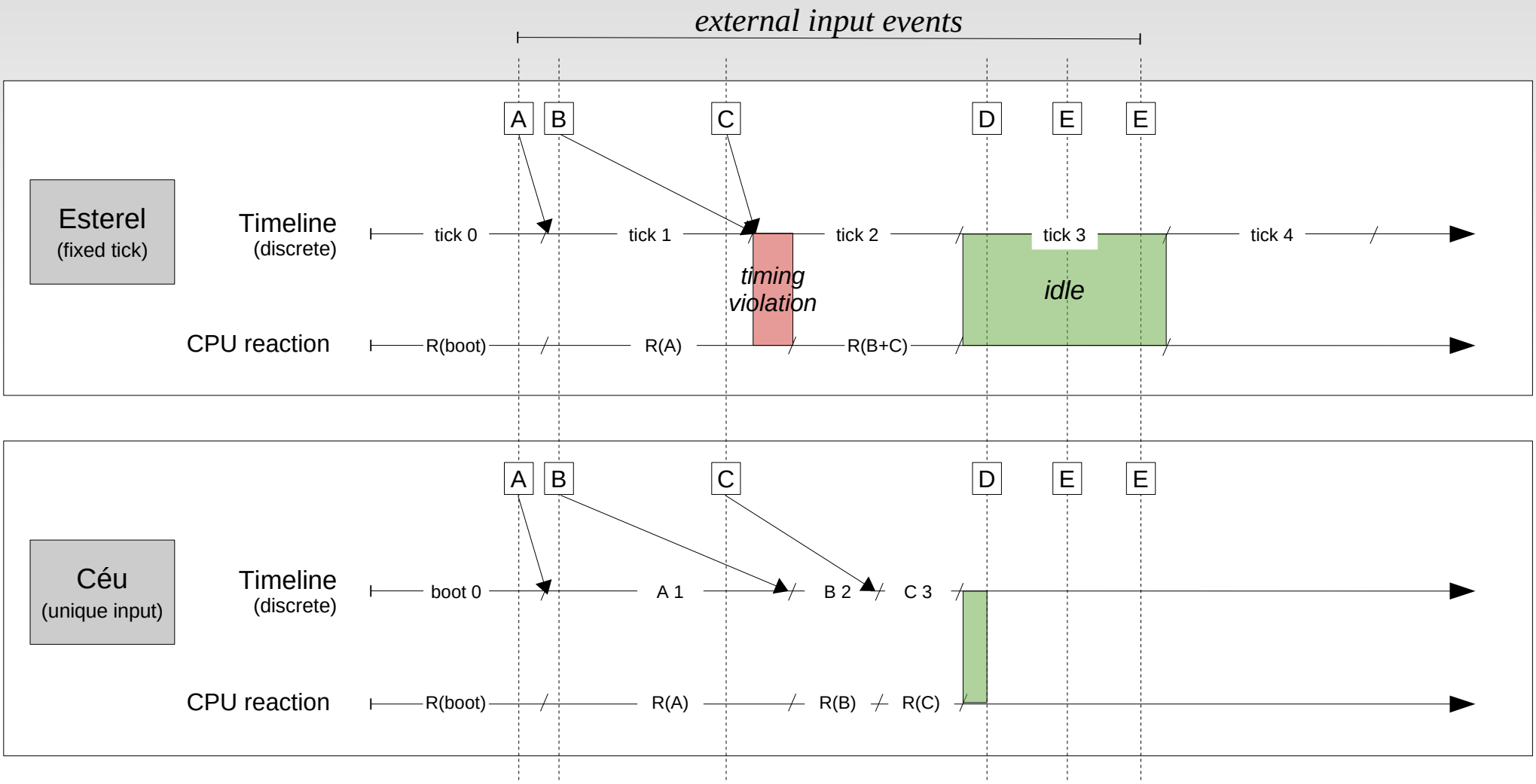
1. External Events



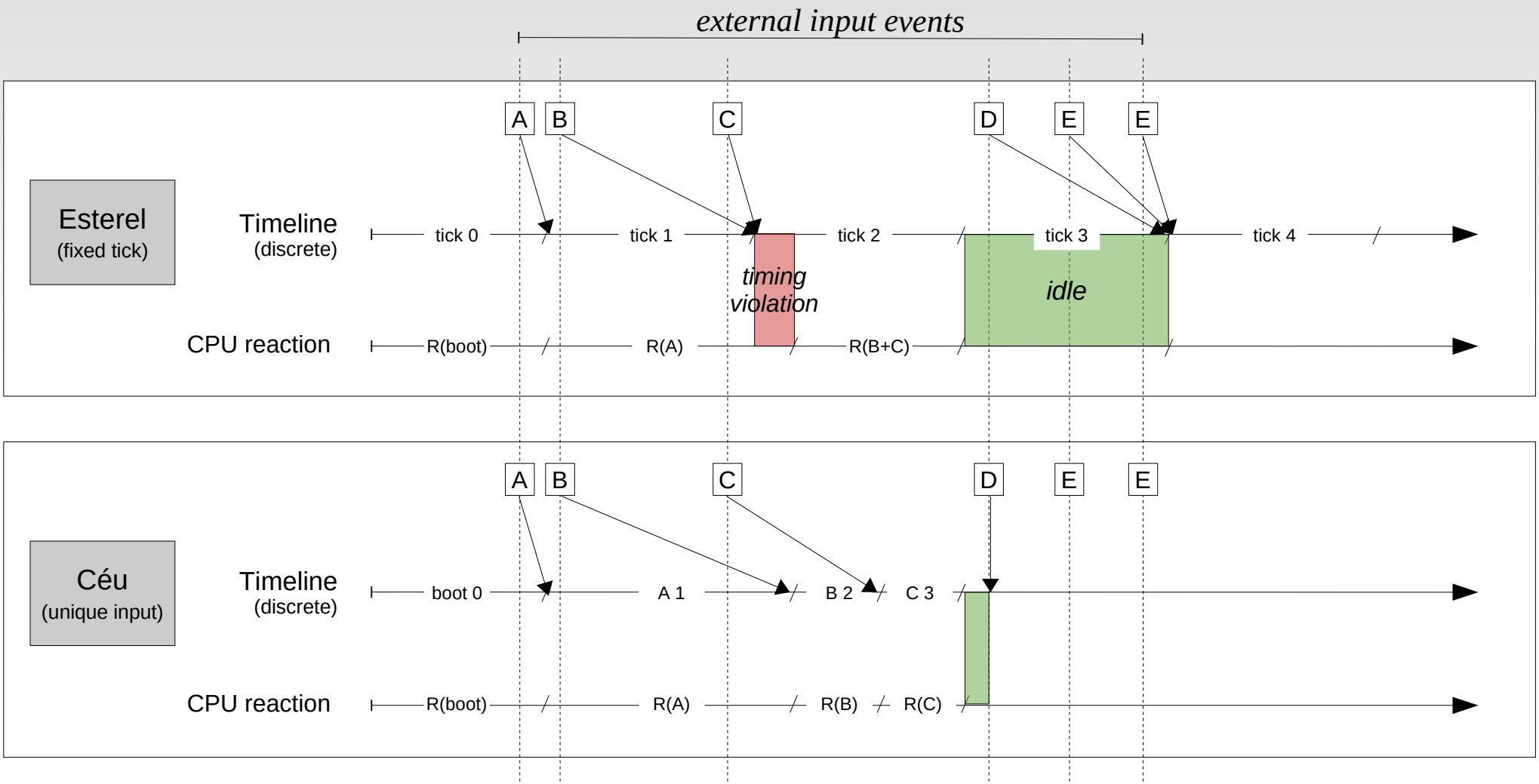
1. External Events



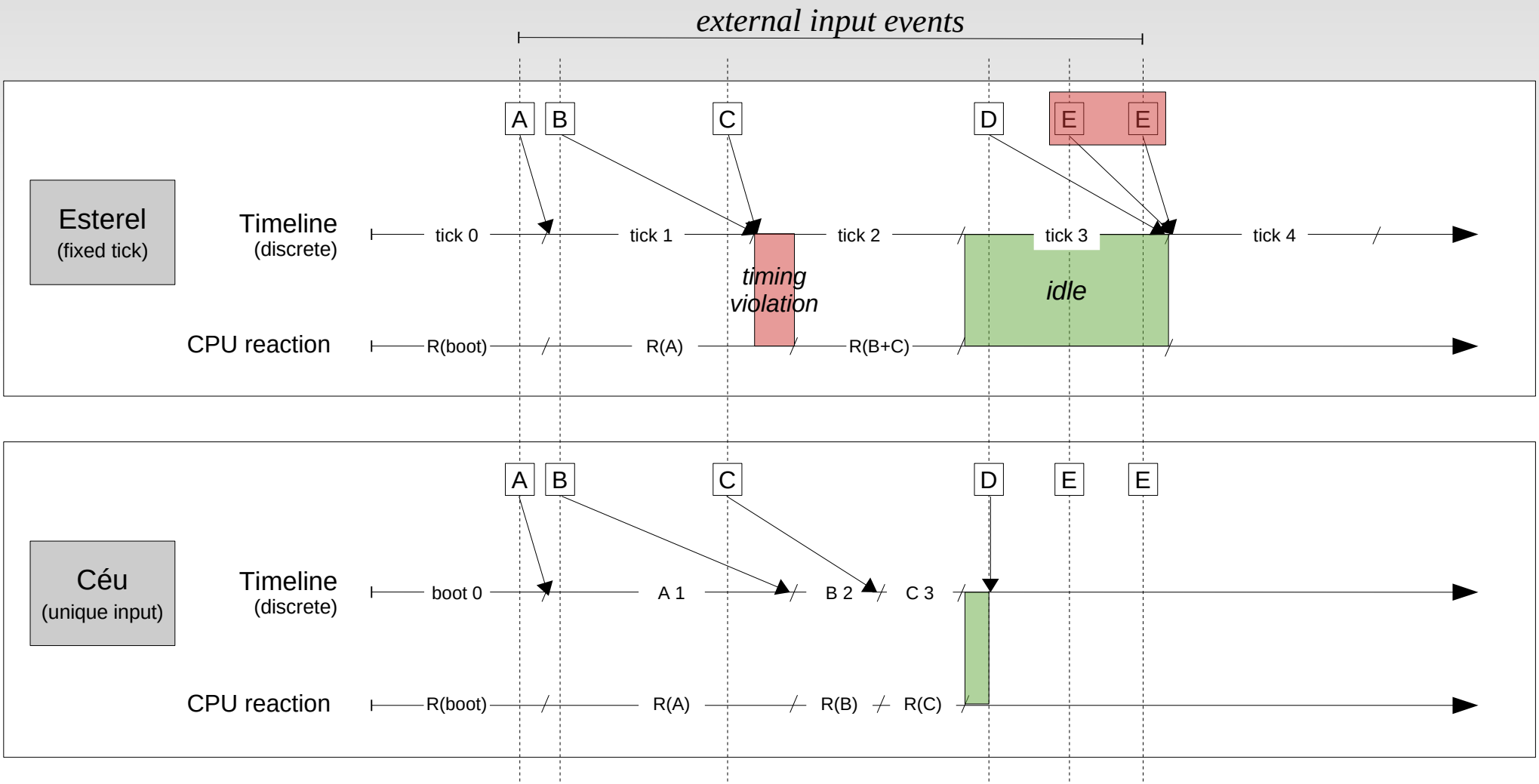
1. External Events



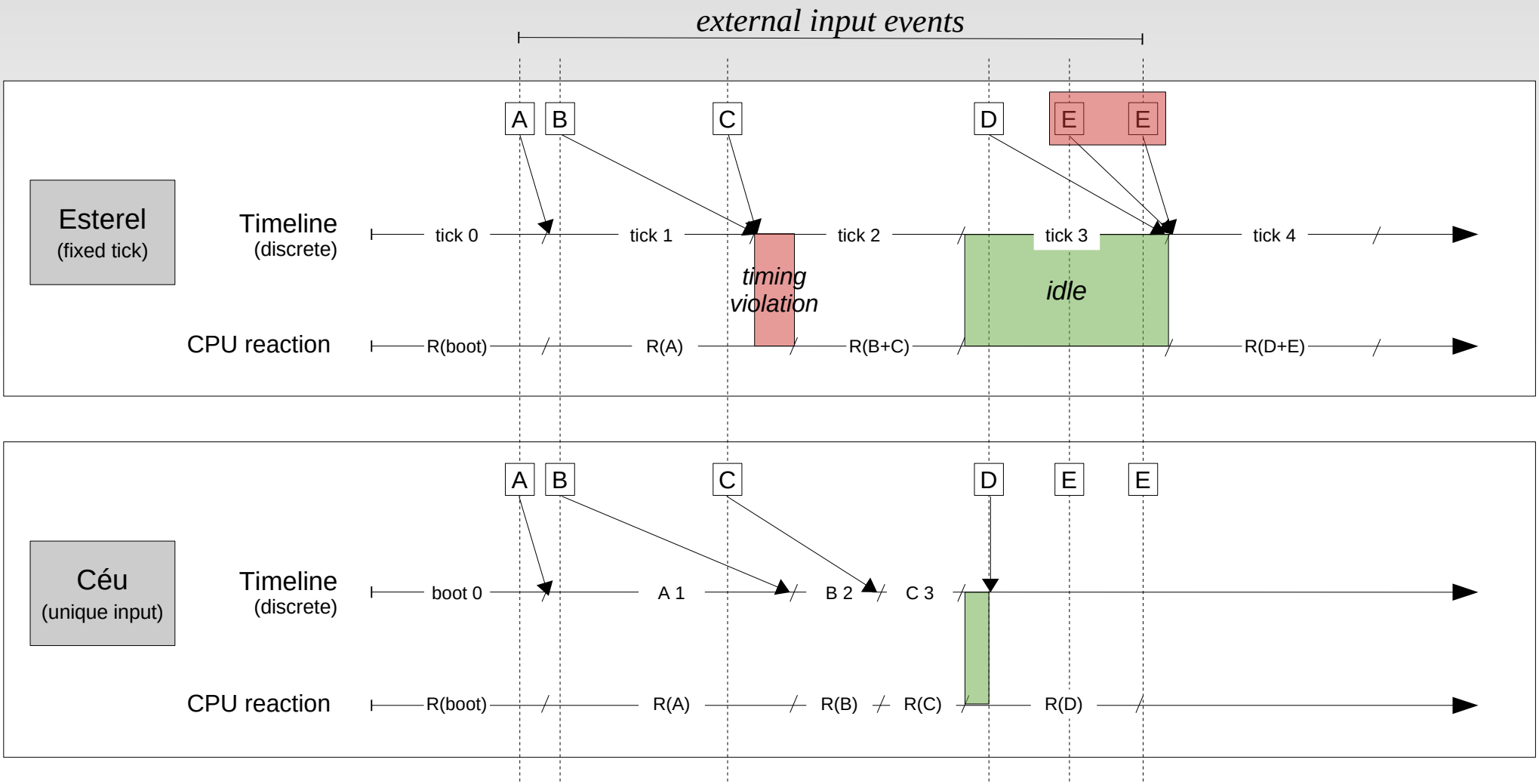
1. External Events



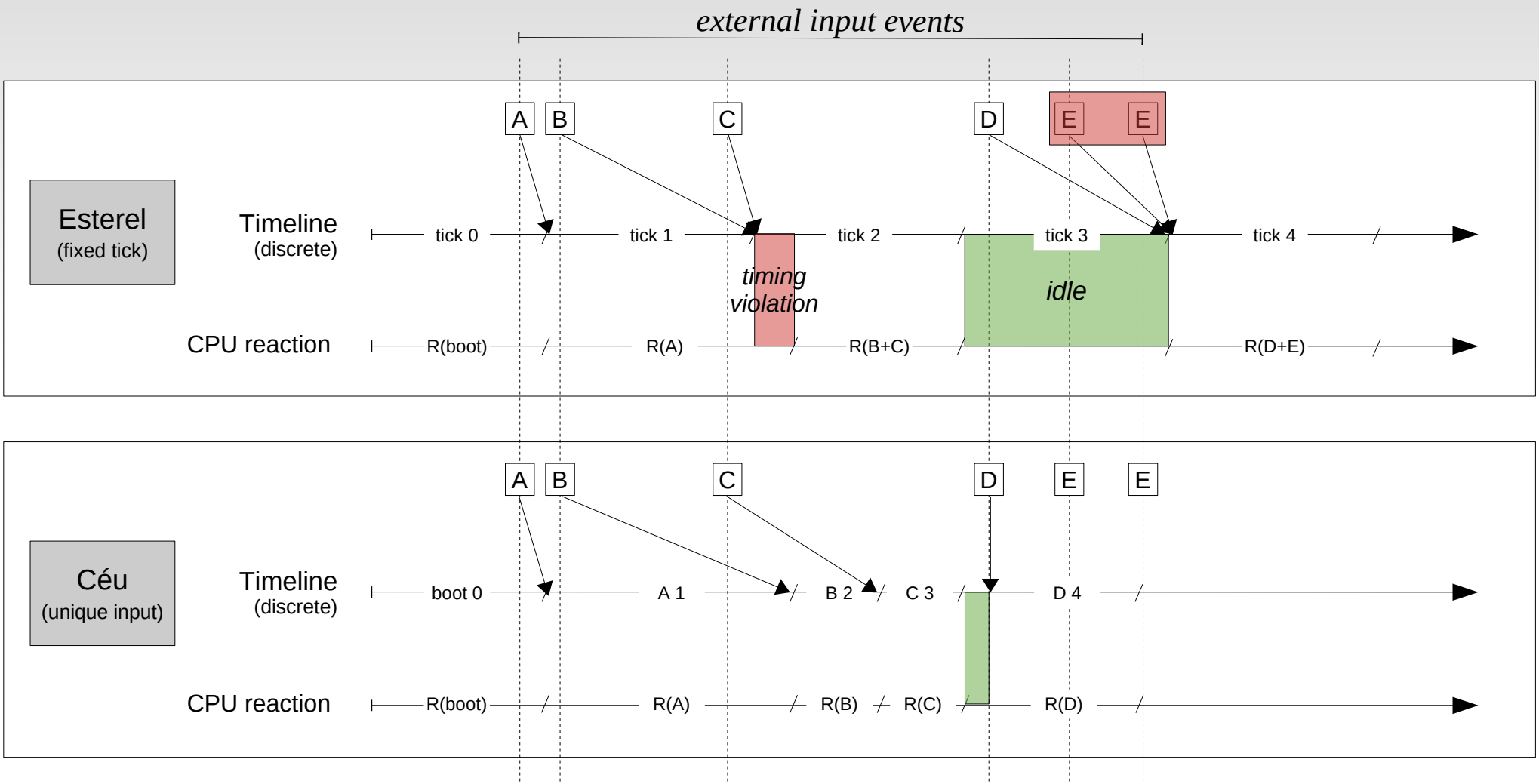
1. External Events



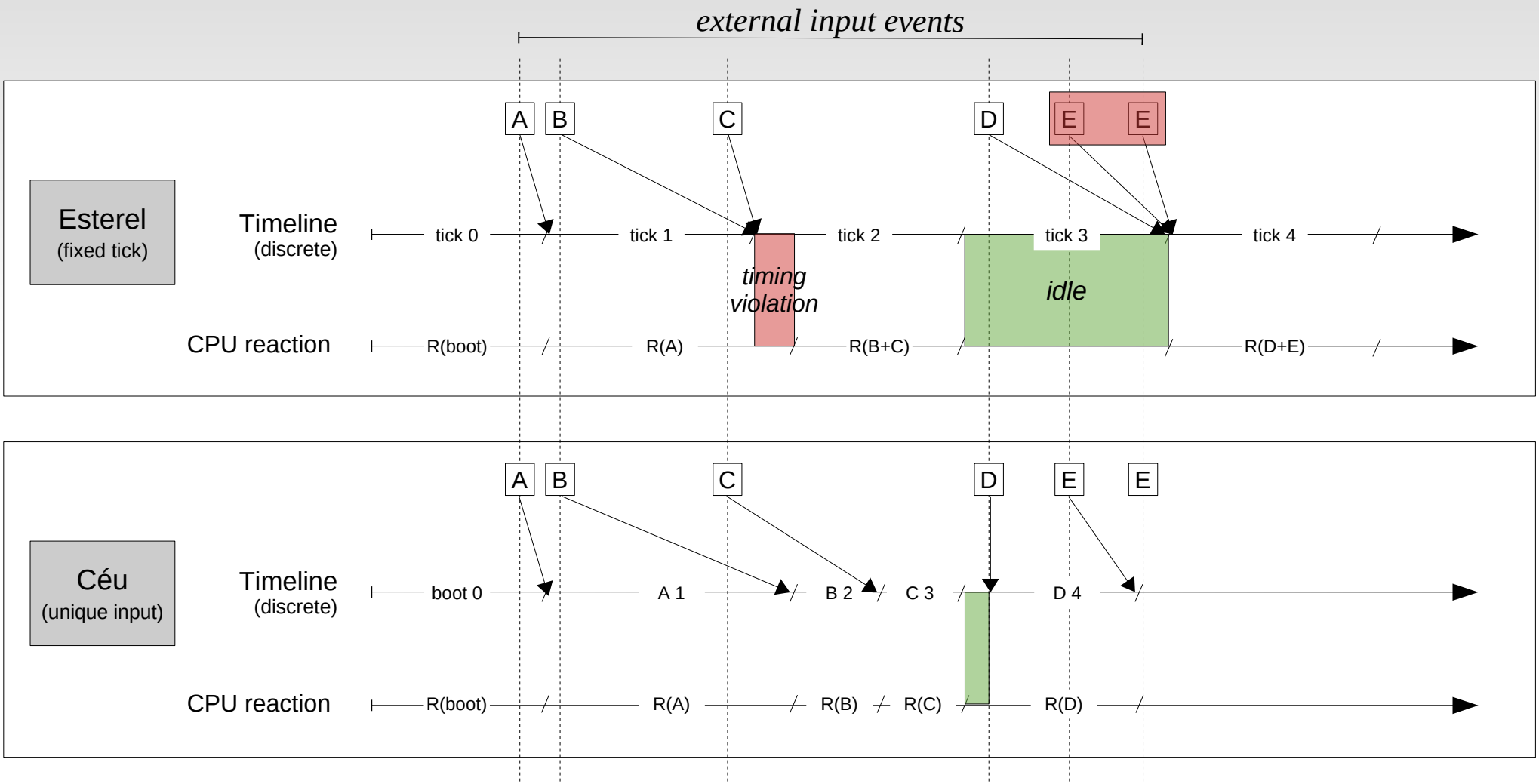
1. External Events



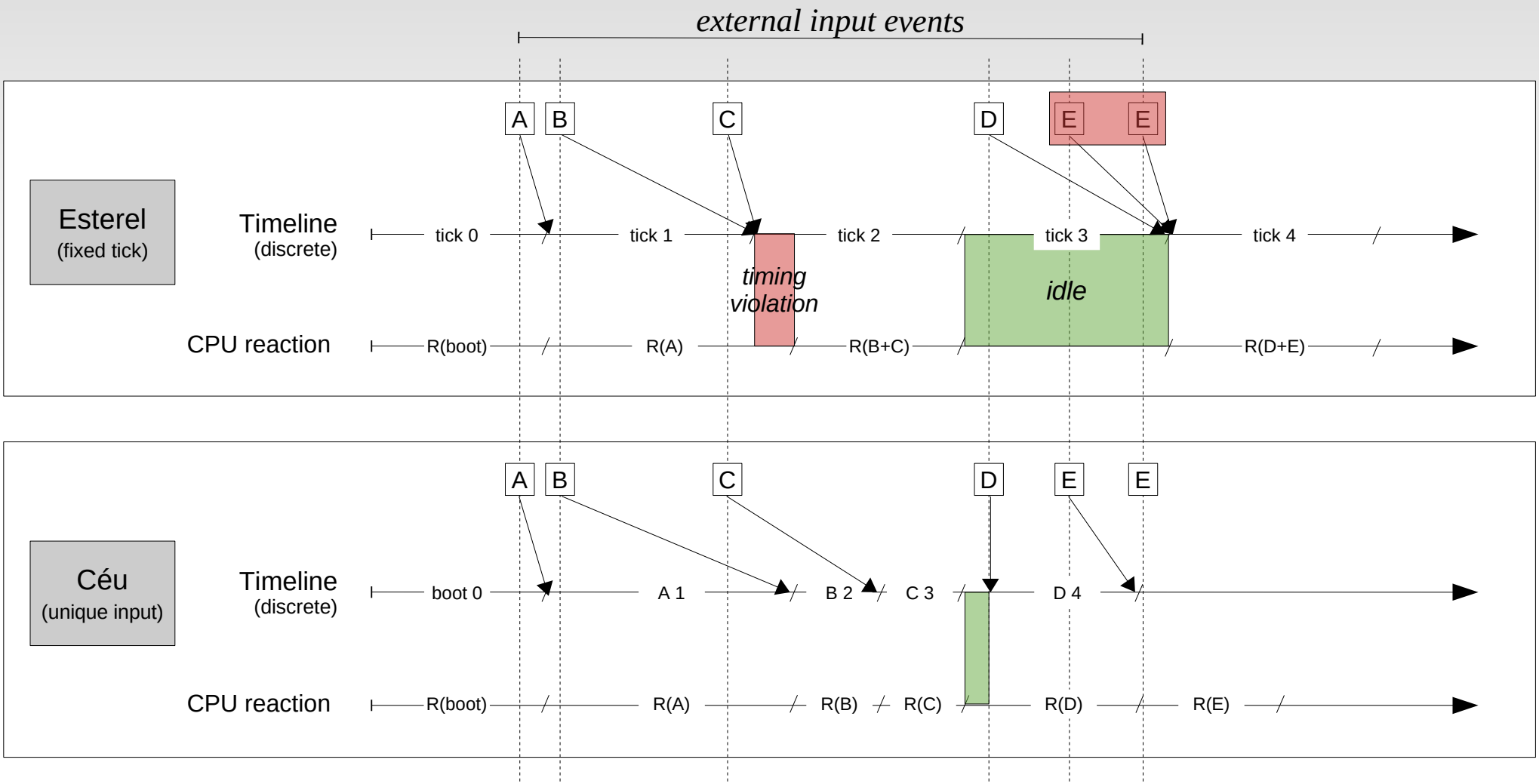
1. External Events



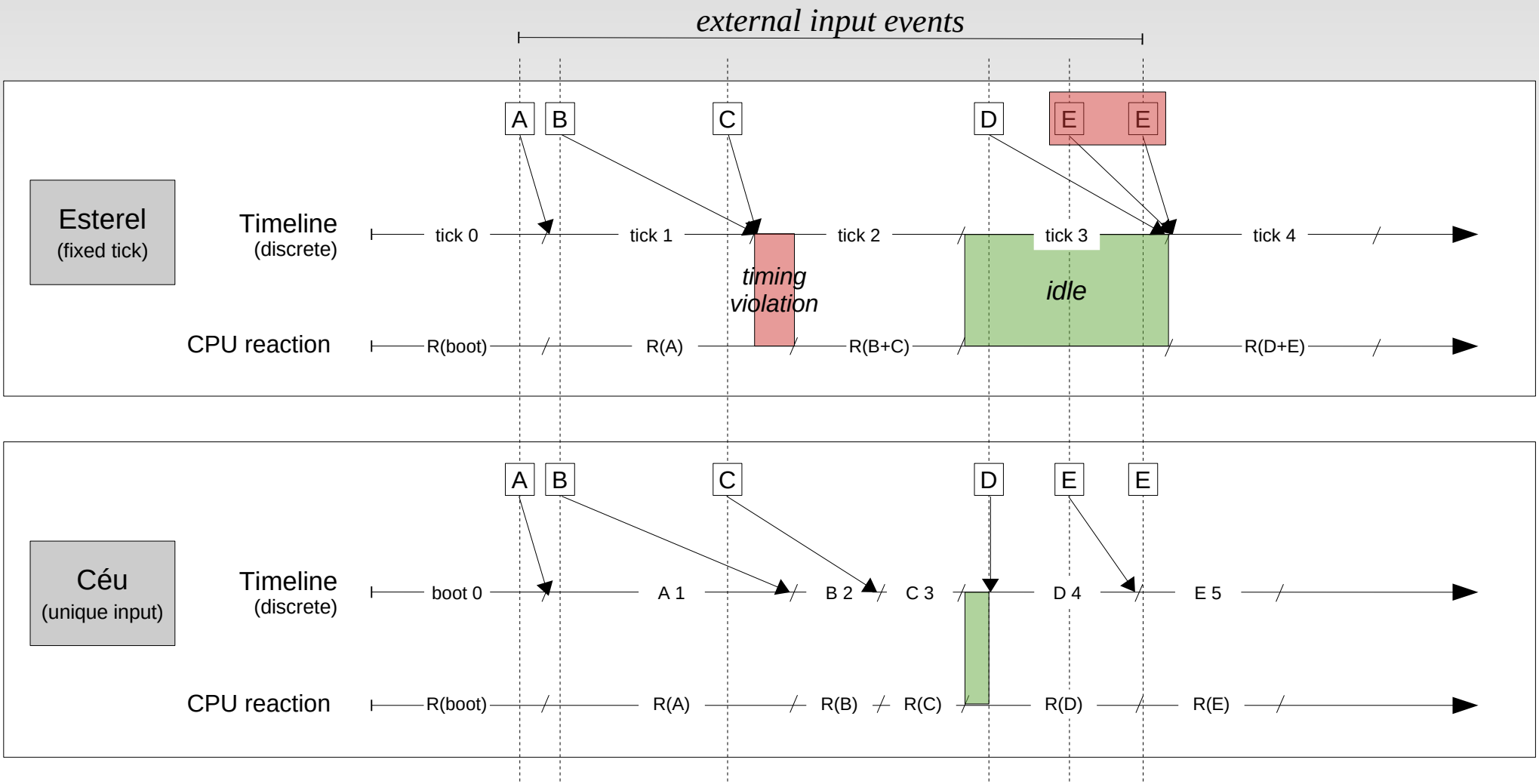
1. External Events



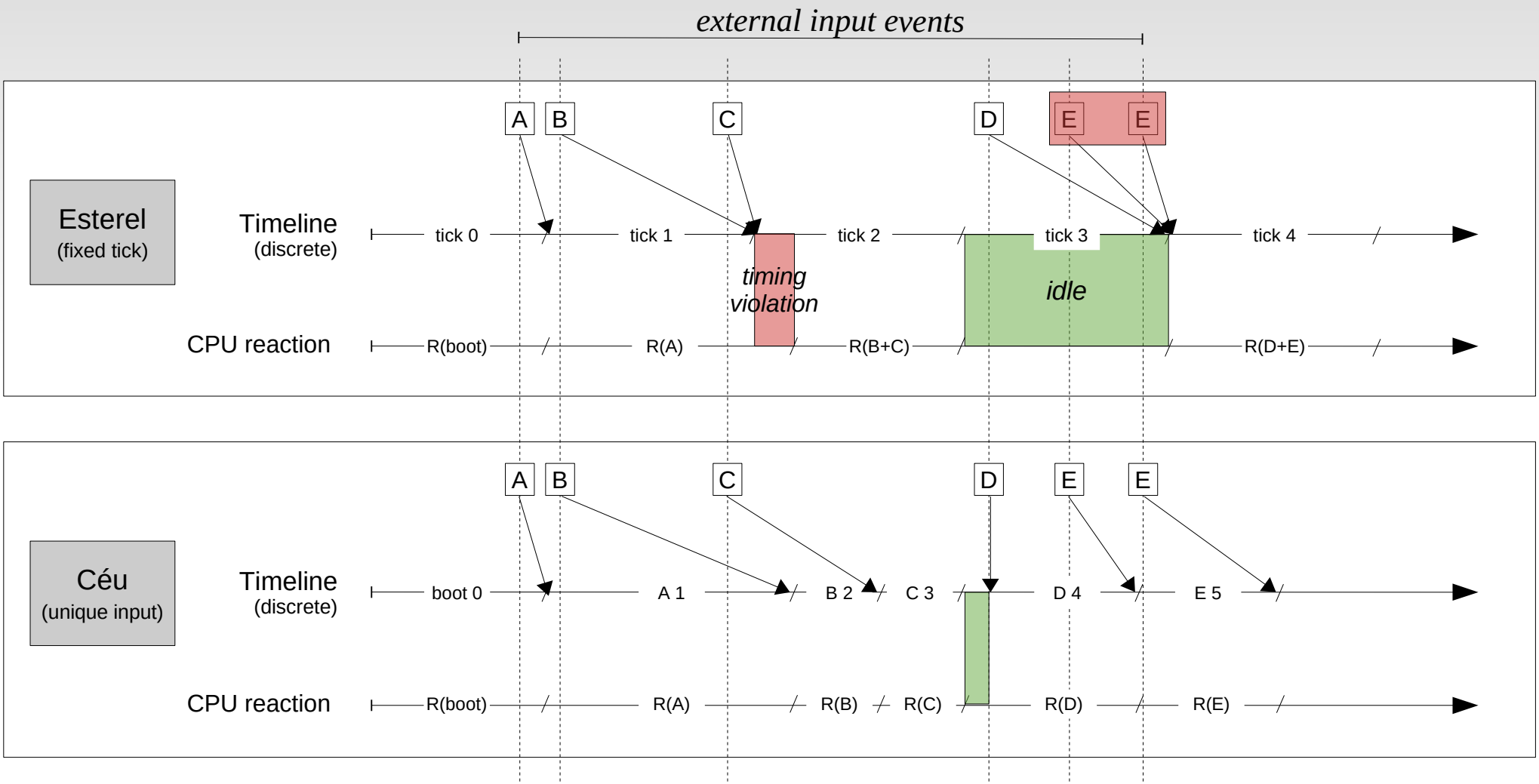
1. External Events



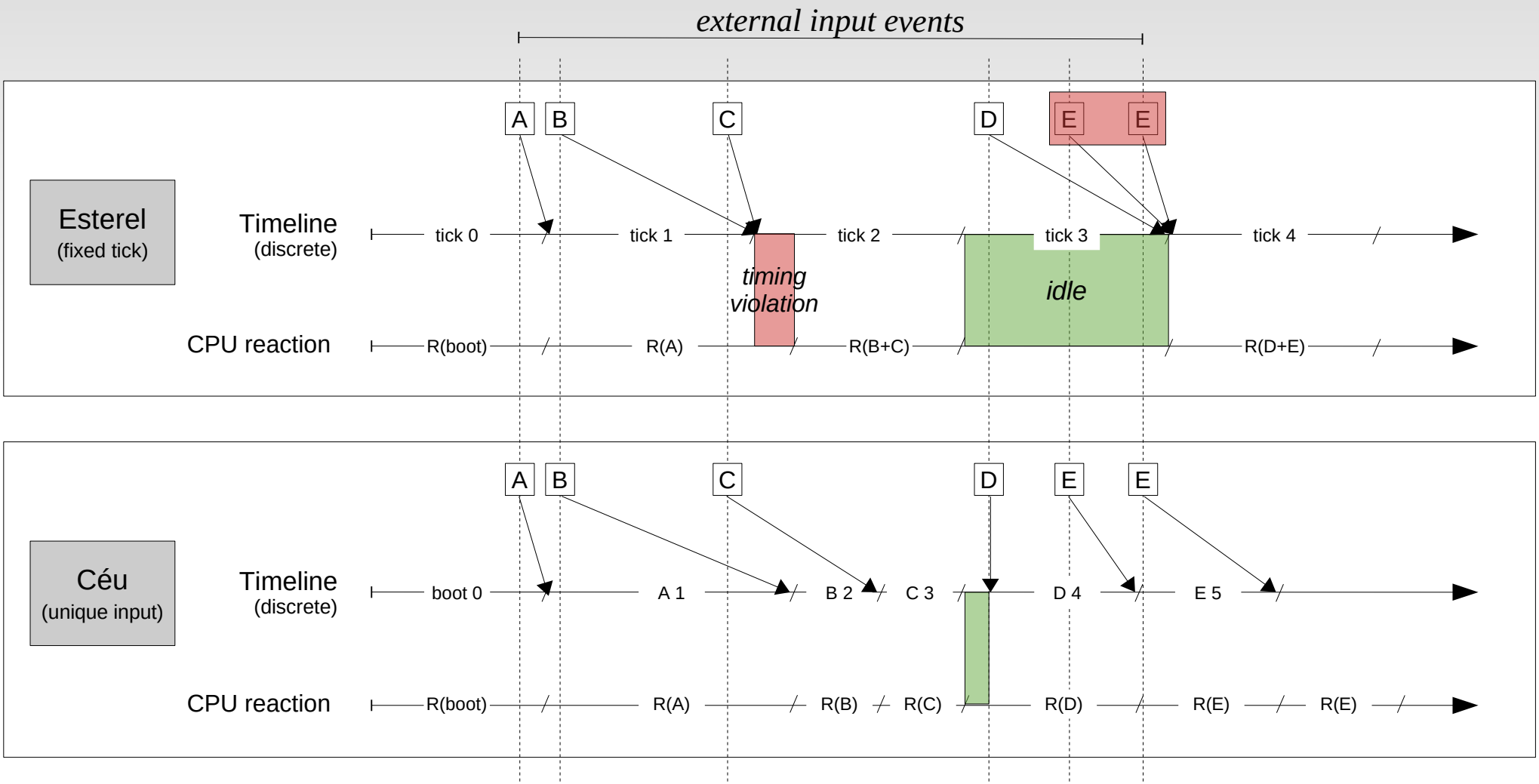
1. External Events



1. External Events



1. External Events



1. External Events