Dynamic Organisms in Céu

(a reactive abstraction mechanism)

www.ceu-lang.org

LabLua – PUC-Rio
www.lua.inf.puc-rio.br

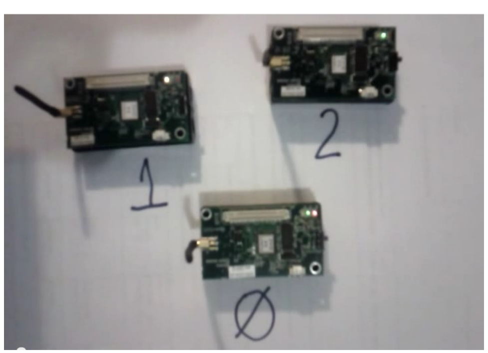
Authors
Francisco Sant'Anna
Noemi Rodriguez
Roberto Ierusalimschy

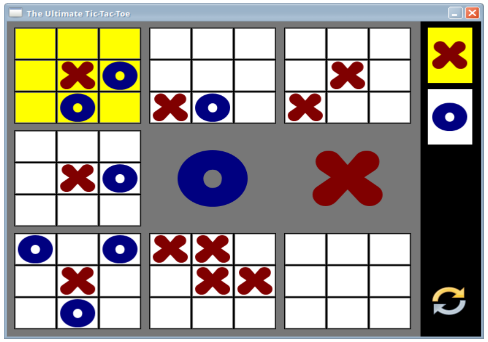
Céu goals

- Front-end applications
  - GUIs, Games, interactive apps
  - Desktops → Arduino *(4K RAM)*
- Ruled by the environment
  - Immediate/real-time feedback
  - Global consensus
- Logical reasoning
  - Concurrency w/ Determinism

Céu non-goals

- Back-end infrastructure
  - High-performance servers
  - Clusters, Cloud computing
- Independent sessions/actors
  - Latency in communication
  - Distribution
- The C10K problem
  - Concurrency w/ Parallelism





"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;
         _leds_toggle();
      end
   with
      await PRESS;
   end
   await 2s;
end
```

Lines of execution
=
Trails (in Céu)

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

From “Structured Programming (SP)” To “Structured Reactive Programming (SRP)”

- Control Structures
  - Sequences, Loops, Conditionals
- Blocks, Scopes, Locals
  - Automatic memory management
- Subroutines
  - Abstraction mechanism
- What about reactivity?
  - Environment event → Short-lived callback
    - No more loops, scopes, etc.
    - Breaks structured programming
      - “Callbacks as our Generations' `goto`” [Miguel de Icaza]

- The `await` statement
  - *Imperative-reactive* nature
- Compositions
  - Control structures + parallels
- Synchronous execution model
  - Time ~ Sequence of events
- Esterel did this back in the *'80s*
- What about abstractions?

Graphical

Demonstrations

*(using the **SDL** C library for graphics and events)*

[sdl-01]
(one rectangle)

```
input void SDL_REDRAW;  // input event from the environment

<...>  // initialize graphical library, renderer, etc.

var _SDL_Rect r = { x=100,y=100,w=20,h=20 };

par do
    every 20ms do
        r.x = r.x + 1;
    end
with
    every SDL_REDRAW do
        _SDL_SetDrawColor(0xFF,0xFF,0xFF,0);
        _SDL_FillRect(&r);
    end
end
```

```
loop do
    await 20ms;
    r.x = r.x + 1;
end
```

[sdl-02]
(two rectangles)

```
var _SDL_Rect r1 = { 100,100,20,20 };
var _SDL_Rect r2 = { 100,300,20,20 };

par do
    every 20ms do
        r1.x = r1.x + 1;
    end
with
with
    end
with
    every SDL_REDRAW do
        _SDL_SetDrawColor(0xFF,0xFF,0xFF,0);
        _SDL_FillRect(&r2);
    end
end
```

r1

The need for abstractions!

The O.O. way

- Use the observer pattern
    1. Create a Rect class
    2. Create two instances
    3. Add instances as listeners for timers and redrawing events
    - Explicit references
    - No loops and parallel compositions (short-lived methods)
- Memory (heap) management:
    - Memory leaks
    - Dangling pointers
    - Garbage collection

We want something as simple as locals with automatic management!

Céu Organisms

- ***Organism*** *~ (Object + Trails)*

```
class <ID> with
    <interface>
      // properties
      // methods
do
    <body>
      // Céu code
end
```

```
class Rect with
   var _SDL_Rect r;
   var int       dt;
do
   par do
      every (dt)ms do
         r.x = r.x + 1;
      end
   with
      every SDL_REDRAW do
         _SDL_SetDrawColor(<...>);
         _SDL_FillRect(&r);
      end
   end
end
```

Organisms react directly to the environment

```
class Rect with
    <interface>
do
    <body>
end

var Rect r1 with
    this.r.y = 100;
    this.dt  = 20;
end;

var Rect r2 with
    this.r.y = 300;
    this.dt  = 10;
end;

await FOREVER;
```

A normal variable declaration:

var <type> <ID>;

(but with a constructor and **body in parallel**)

```
class Bird with
    <...>
    var int speed;      // px/secs
do
    <...>
end

var Bird b1 with
    this.r.y   = 100;
    this.speed = 100;
end;

var Bird b2 with
    this.r.y   = 300;
    this.speed = 200;
end;

await FOREVER;
```

Reaction to the environment is abstracted

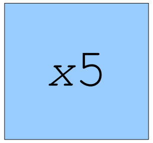
On instantiation, only the interface matters

[birds-02]
(vectors)

```
class Bird with
    <...>
do
    <...>
end

var int i = 1;
var Bird[5] birds with
    this.r.y    = 20  *  4*i;
    this.speed = 100 + 10*i;
    i = i + 1;
end;

await FOREVER;
```



```
class Bird with
    <...>
do
    <...>
end

loop do
    var int i = 1;
    var Bird[5] birds with
        this.r.y     =  20 *  4*i;
        this.speed = 100 + 10*i;
        i = i + 1;
    end;
    await SDL_MOUSEBUTTON;
end
```

organisms have
lexical scope

organism out of scope:
data reclaimed **and body aborted**

```
class Bird with
    <...>
do
    <...>
end

every 1s do
    spawn Bird with
        this.r.y   =  20 + _rand()%HEIGHT;
        this.speed = 100 + _rand()%100;
    end;
end
```

spawn dynamically allocates an **anonymous** organism

[birds-04]
(Bird implementation)

```
class Bird with
    <...>
do
    par do
        every SDL_FRAME do
            <animate>
        end
    with
        every SDL_REDRAW do
            <redraw>
        end
    end
end

every 1s do
    spawn Bird with
        this.r.y     =  20 + _rand()%HEIGHT;
        this.speed = 100 + _rand()%100;
    end;
end
```

animation trail

redrawing trail

[birds-05]
(animation break)

```
class Bird with
    <...>
do
    par/and do
        every SDL_FRAME do
            <animate>
            if r.x >= WIDTH-DX then
                break;
            end
        end
    with
        every SDL_REDRAW do
            <redraw>
        end
    end
end

every 1s do
    spawn Bird with
        this.r.y   =  20 + _rand()%HEIGHT;
        this.speed = 100 + _rand()%100;
    end;
end
```

only this trail terminates

animation trail

redrawing trail

```
class Bird with
    <...>
do
    par/or do
        every SDL_FRAME do
            <animate>
            if r.x >= WIDTH-DX then
                break;
            end
        end
    with
        every SDL_REDRAW do
            <redraw>
        end
    end
end

every 1s do
    spawn Bird with
        this.r.y     =  20 + _rand()%HEIGH
        this.speed = 100 + _rand()%100;
    end;
end
```

only this trail terminates

but this trail is aborted

spawned organisms are anonymous

dynamic organisms are automatically reclaimed on termination

[birds-07]

(pools - bounded)

```
class Bird with
    <...>
do
    <...>
end

pool Bird[2] birds;

every 1s do
    spawn Bird in birds with
        this.r.y    =  20 + _rand()%HEIGHT;
        this.speed = 100 + _rand()%100;
    end;
end
```

static memory

[birds-08]
(pools - unbounded)

heap memory

```
class Bird with
    <...>
do
    <...>
end

pool Bird[] birds;

every 1s do
    spawn Bird in birds with
        this.r.y   =  20 + _rand()%HEIGHT;
        this.speed = 100 + _rand()%100;
    end;
end
```

```
class Bird with
    <...>
do
    <...>
end

loop do
    par/or do
        pool Birds[] rs;
        every 1s do
            spawn Bird in birds with
                this.r.y    =  20 + _rand()%HEIGHT;
                this.speed = 100 + _rand()%100;
            end;
        end
    with
        await SDL_MOUSEBUTTON;
    end
end
```

pools of organisms also have lexical scope

static or heap memory

pool out of scope:
data and body of
all organisms reclaimed

Pointers

- Pointers to organisms are sometimes required:
  - **Static**: `&´ operator

```
var T t;
<use &t>
```

  - **Dynamic**: pool iterators

```
pool T ts;
loop (T*)t in ts do
    <use t>
end
```

- All pointers are temporary references

```
class Bird with
    <...>
    event void collided;
do
    par/or do
        <animate>
    with
        await this.collided;
    with
        every SDL_REDRAW do
            <redraw>
        end
    end
end

par do
    <bird creation>
with
    every SDL_FRAME do
        loop (Bird*)b1 in birds do
            loop (Bird*)b2 in birds do
                if <&b1:r vs &b2:r> then
                    emit b1:collided;
                    emit b2:collided;
                end
            end
        end
    end
end
```

Interfaces support internal events

Await to abort the body

For every frame

Iterate over the birds

Emits on collision detection

```
class Bird with
    <...>
    event void collided;
do
    par/or do
        <...>
    with
        await this.collided;
        every SDL_FRAME do
            <animate>
            if r.y >= HEIGHT-DY then
                break;
            end
        end
    with
        <...>
    end
End

<...>
```

[birds-12]
(blinking)

```
class Bird with
    <...>
do
    var bool visible = true;
    par/or do
        <...>
    with
        await this.collided;
        <...>
        par/or do
            await 1s;
        with
            every 100ms do
                visible = not visible;
            end
        end
    with
        every SDL_REDRAW do
            if visible then
                _SDL_RenderCopy(img);
            end
        end
    end
end

<...>
```

During 1 second, toggle the "visible" state every 100ms.

Temporary References

- Alive (valid) within a whole reaction
- Dead (invalid) across reactions

```
var T* ptr = ...;
ptr->x = 1;
await 1s;
_printf("x = %d\n", ptr->x);
```

possibly a dangling pointer

- Pointers can be tracked across reactions

```
var T* ptr = ...;
ptr->x = 1;
watching ptr do
    await 1s;
    _printf("x = %d\n", ptr->x);
end
```

```
par/or do
    await ptr._killed;
with
    <...>
end
```

```
class Bird with
    <...>
do
    <...>
end

par do
    <bird creation>
with
    loop do
        var _SDL_MouseEvent* mse = await SDL_MOUSEBUTTON;
        var Bird* ptr = null;
        loop (Bird*)b in birds do
            if <mse vs &b:r> then
                ptr = b;
                break;
            end
        end
        if ptr != null then
            watching ptr do
                every SDL_REDRAW do
                    _SDL_DrawLine(WIDTH/2,HEIGHT,
                                  ptr:r.x,ptr:r.y);
                end
            end
        end
    end
end
```

Iterator that checks if a bird was clicked

Watches the bird while drawing a line

More examples

- 10k organisms, 40k trails, >100fps
- A complete open source game for Android (available in the “Play Store”)
- Composing the demonstrations

```
// ALL.CEU

#define ALL

<initialize SDL>

var int ex = 1;
<...>
    if show == 1 then
        #define Bird Bird1
        #include "birds/birds-01.ceu"
    else/if show == 2 then
        #undef Bird
        #define Bird Bird2
        #include "birds/birds-02.ceu"
    else/if show == 3 then
        #undef Bird
        #define Bird Bird3
        #include "birds/birds-03.ceu"
    else/if <...> then
        <...>
    end
```

```
// APP.CEU

#ifndef ALL
<initialize SDL>
#else
_SDL_SetWindowTitle("Title");
#endif

<code-for-the-app>
```

Summary

- Organisms reconcile objects and (synchronous) threads in a single concept
- Static and dynamic instances have lexical scope with automatic memory management
- References are invalidated across `await` statements, but can be supervised with the `watching` statement

- Issues in dynamic allocation:
  - **Memory leaks**: lexical scope and body termination
  - **Dangling pointers**: controlled references
  - **Garbage collection**: lexical scope and body termination

Not covered

- Synchronous model
  - Bounded execution
  - Safe shared-memory concurrency
- External vs Internal events
  - Queue vs Stack-based execution
- C integration & Finalization
  - Abortion for global resources
- Timers
  - Delay compensation
- Asynchronous blocks
  - Long computations, input generation/simulation

Obrigado!

(Thank you!)

Francisco Sant'Anna

francisco.santanna@gmail.com

@fsantanna_puc

(I'll tweet the slides here!)

www.ceu-lang.org

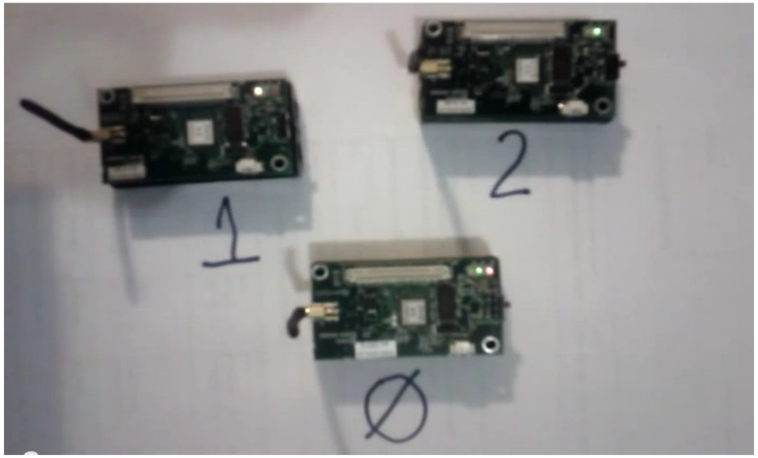
LabLua – PUC-Rio

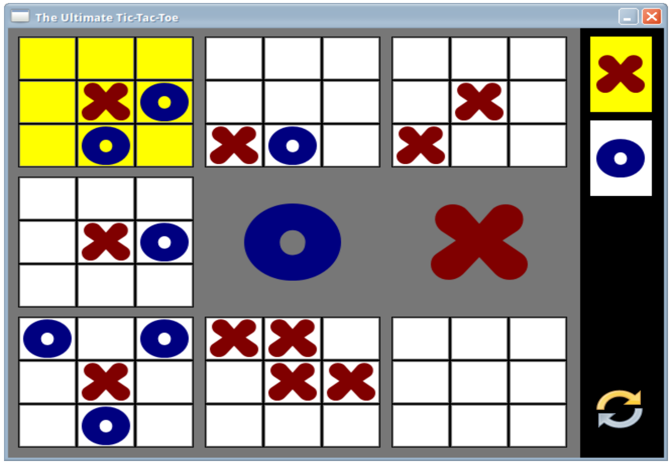
www.lua.inf.puc-rio.br

Extra

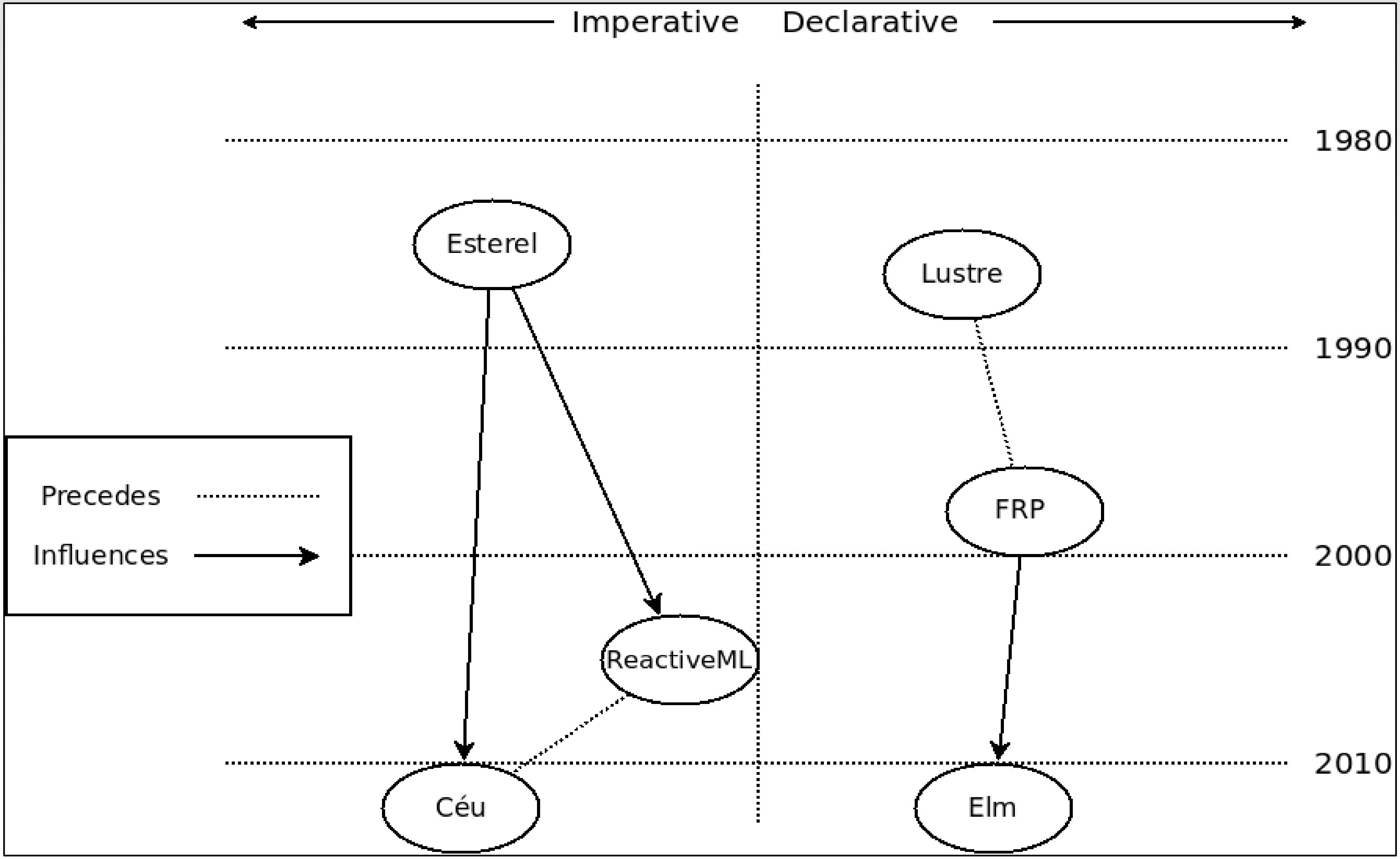
Slides

Reactive/RT applications

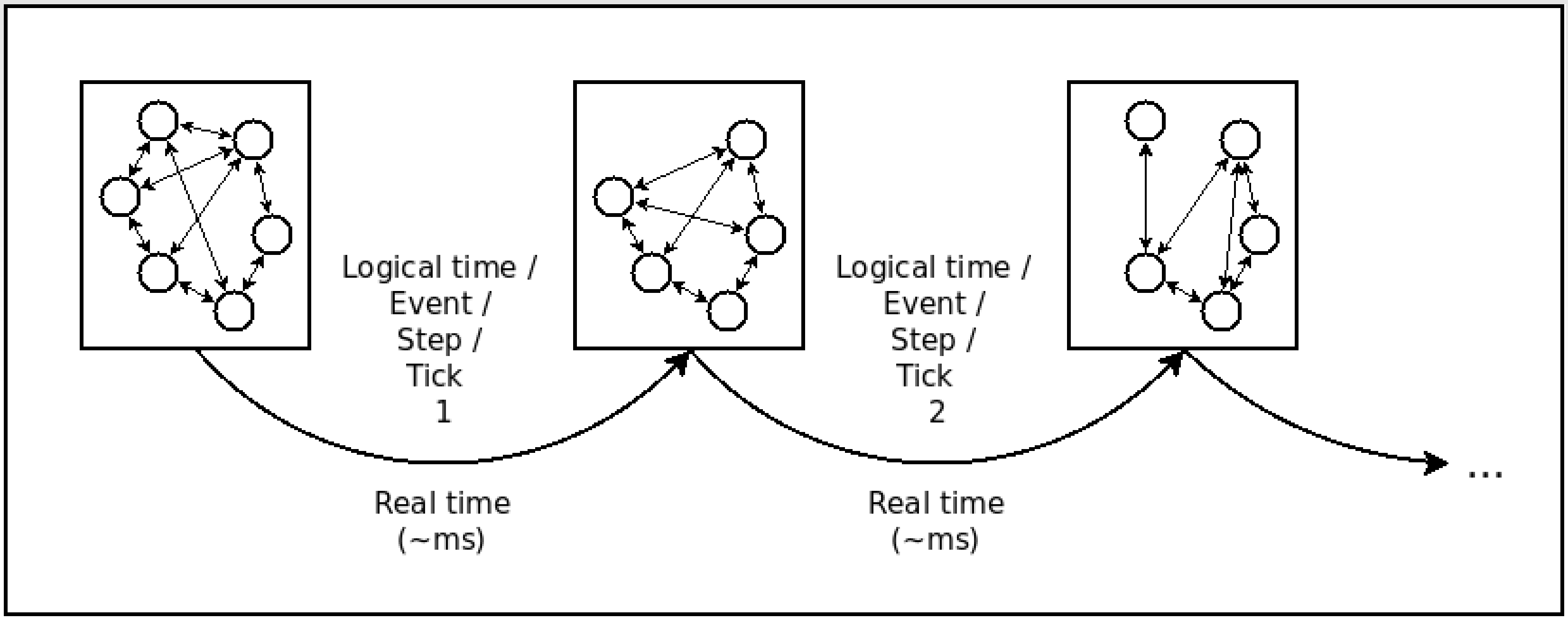




Reactive languages



Synchronous execution

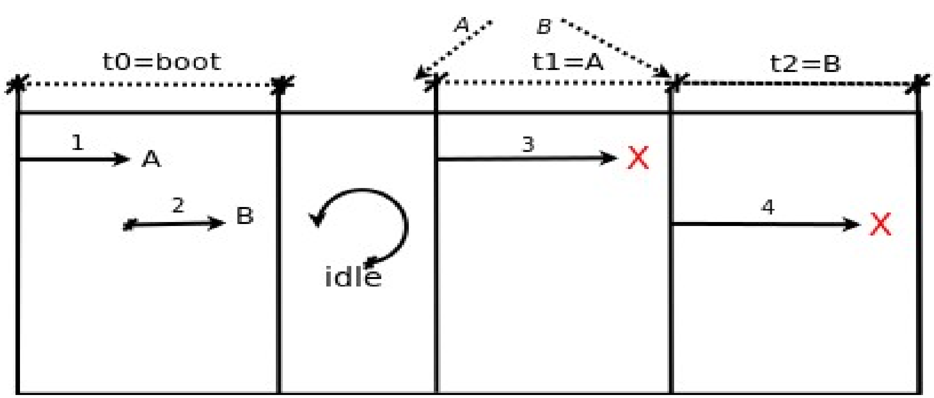


- Game Loop *(i.e., input → logic → redraw → *)*
- Arduino Loop *(i.e., input → logic → output → *)*
- Observer Pattern *(i.e., input → observers → output → *)*
- Digital Circuits *(i.e., input → propagation → output → *)*

Execution model

1. Programs starts in the “boot reaction” in a single trail.
2. Trails execute until they await or terminate. This step is known as “reaction chain” and always executes in bounded time.
3. An input event occurrence awakes **all** trails awaiting that event. Repeat “step 2”.

```
par/and do
   <...>        // 1
   await A;
   <...>        // 3
with
   <...>        // 2
   await B;
   <...>        // 4
end
```



<...> *are trail segments that do not await (e.g. atrbuições, chamadas de função)*

Execution model

- Synchronous hypothesis: “Reactions execute infinitely faster in comparison to the input rate.”
- *In theory:* a program does not take time on “step 2” and is always idle on “step 3”.
- *In practice:* if an event occurs while a reaction takes place, the event is enqueued for the next reaction.
- Reactions to events never overlap.
- When multiple trails are active (i.e., awoken from the same event), they execute in the order they appear in the source code.

Execution model

- Céu ensures bounded execution
  - All loops must `await` or `break`

```
loop do
  if <cond> then
    break;
  end
end
```

```
loop do
  if <cond> then
    break;
  else
    await A;
  end
end
```

- *Limitation: long computations*

Shared memory

```
var int x=1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end
_printf("%d\n", x);
```

```
var int x=1;
par/and do
    await A;
    x = x + 1;
with
    await A;
    x = x * 2;
end
_printf("%d\n", x);
```

- Static analysis
  - `ceu --safety 0    # allows both`
  - `ceu --safety 1    # allows LEFT, refuses RIGHT`
  - `ceu --safety 2    # refuses both`

C integration

- Well-marked syntax ("_")

```
native do
  #include <assert.h>
  int I = 0;
  int inc (int i) {
    return i+1;
  }
end
_assert(_inc(_I));
```

- "C hat" (unsafe execution)
- No "bounded" analysis
- What about side effects?

C integration

- **pure** and **safe** annotations

```
pure _inc();
safe _f() with _g();

par do
   _f(_inc(10));
with
   _g();
end
```

Local scopes & Finalization

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

local memory
to
external pointer

May terminate

```
line 5 : call to "_send_request"
         requires `finalize´
```

external memory
to
local pointer

```
par/or do
   var _FILE* f = _fopen(...);
   _fwrite(..., f);
   await 1s;
   _fwrite(..., f);
   await 1s;
   _fclose(f);
with
   <...>
end
```

May terminate

```
line 2 : attribution requires
         `finalize´
```

Local scopes & Finalization

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

```
par/or do
   var _FILE* f;
   finalize
      f = _fopen(...);
   with
      _fclose(f);
   end
   _fwrite(..., f);
   await 1s;
   _fwrite(..., f);
   await 1s;
   _fclose(f);
with
   <...>
end
```

1st class Timers

- Very common in reactive applications
  - *sampling, timeouts*
- **`await`** supports time (i.e., *ms*, *min*)
  - … and compensates system delays

```
await 2ms;
v = 1;
await 1ms;
v = 2;
```

- 5ms elapse
- late = 3ms
- late = 2ms

```
par/or do
    await 10ms;
    <...>   // no awaits
    await 1ms;
    v = 1;
with
    await 12ms;
    v = 2;
end
```

(Related) Simula

- Similar syntactic structure
  - Interface + Body
- Different execution models
  - Cooperative vs Synchronous/Reactive/Independent
- Compositions
  - Single body vs Parallel compositions
- Memory management (reasoning)
  - Heap vs Lexical scope