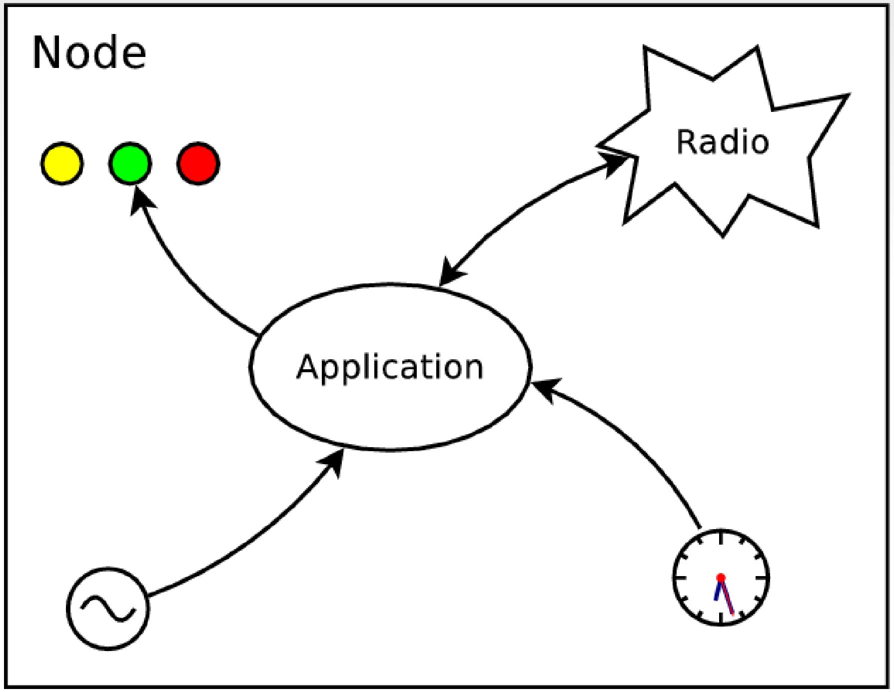
Safe System-level Concurrency on Resource-Constrained Nodes with Céu

Autor:
Francisco Sant'Anna

Orientadores:
Roberto Ierusalimschy
Noemi Rodriguez

Wireless Sensor Networks

- Reactive
  - guided by the environment
- Concurrent
  - safety aspects
- Constrained
  - 32K ROM
  - 4K RAM



Hello world!

- Blinking LEDs
  1. *on ↔ off every 500ms*
  2. *stop after 5s*
  3. *repeat after 2s*

- Compositions
  - par, seq, loop
  - avoid state vars
  - static inference

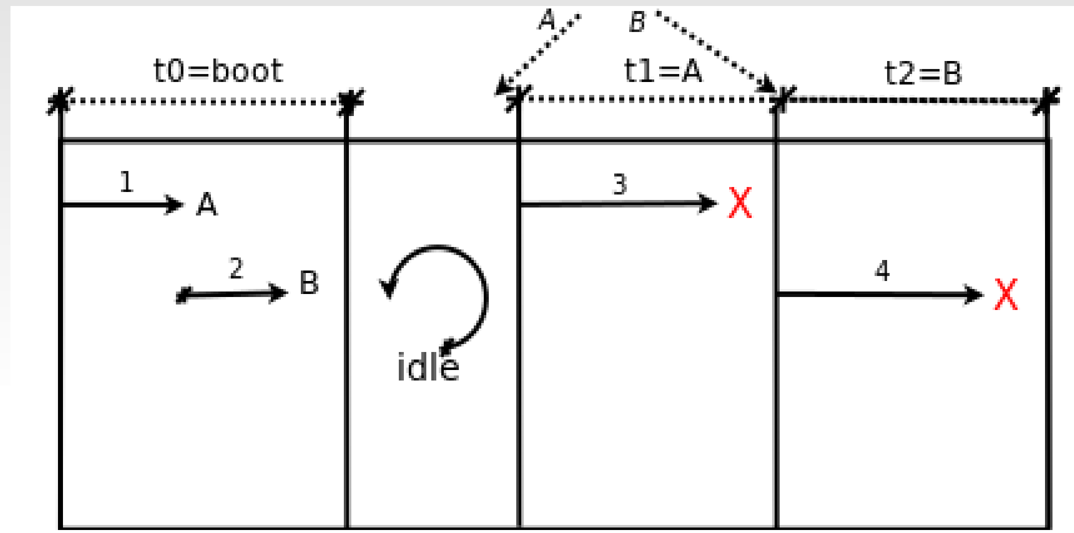
```
loop do
   par/or do
      loop do
         await 500ms;
         _leds_toggle();
      end
   with
      await 5s;
   end
   await 2s;
end
```

The design of Céu

1. Reactive/Synchronous execution
   - based on Esterel
2. Shared-memory concurrency
3. Internal events
4. Integration with C
5. Local scopes & Finalization
6. First-class timers

1. Synchronous execution

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



<...> *are trail segments that do not await (e.g. assignments, system calls)*

- Reactions to external events never overlap
- The synchronous hypothesis: *“reactions run infinitely faster in comparison to the rate of events”*

2. Shared-memory concurrency

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

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

3. Internal events

- Emitted by the program
  - *(vs environment)*

- Multiple can be active at the same time
  - (vs single)

- Stack-based execution policy
  - (vs queue)

3. Internal events

- Enable advanced control mechanisms (e.g. subroutines, exceptions)

```
event int* inc;
par do
    // define subroutine
    loop do
        var int* p = await inc;
        *p = *p + 1;
    end
with
    // use subroutine
    <...>
    var int v = 1;
    emit inc => &v;    // stack this continuation
    _assert(v == 2);
end
```

- Use bounded memory and execution time

4. Integration with C

- Well-marked syntax (“_”)

```
native _assert(), _inc(), _I;
_assert(_inc(_I));

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

- “C hat” (unsafe execution)
- no bounded-execution analysis
- what about side effects in parallel trails?

4. Integration with C

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

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

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

5. Local scopes & Finalization

```
loop do
   await 10ms;
   var _message_t msg;
   <...> // prepare msg
   _send_request(&msg);
   await SEND_ACK;
end
```

5. Local scopes & Finalization

```
par/or do
   loop do
      await 10ms;
      var _message_t msg;
      <...> // prepare msg
      _send_request(&msg);
      await SEND_ACK;
   end
with
   await STOP;
end
var int x = 1;
```

local pointer

5. Local scopes & Finalization

```
par/or do
   loop do
      await 10ms;
      var _message_t msg;
      <...> // prepare msg
      finalize
         _send_request(&msg);
      with
         _send_cancel(&msg);
      end
      await SEND_ACK;
   end
with
   await STOP;
end
var int x = 1;
```

6. First-class timers

- Very common in WSNs
    - sampling, timeouts
- **await** supports time (i.e. *ms*, *min*)
    - *it also 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
```

Formalization

- Small-step operational semantics

- Control aspects of the language
  - parallel compositions,
    stack-based events, finalization

- Mapping: formal → concrete

$$\frac{val(id,n) \neq 0}{\langle S, (mem(id)\ ?\ p\ :\ q)\rangle \xrightarrow[n]{} \langle S, p\rangle} \qquad \textbf{(if-true)}$$

$$\frac{val(id,n) = 0}{\langle S, (mem(id)\ ?\ p\ :\ q)\rangle \xrightarrow[n]{} \langle S, q\rangle} \qquad \textbf{(if-false)}$$

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, (p\ ;\ q)\rangle \xrightarrow[n]{} \langle S', (p'\ ;\ q)\rangle} \qquad \textbf{(seq-adv)}$$

$$\langle S, (mem(id)\ ;\ q)\rangle \xrightarrow[n]{} \langle S, q\rangle \qquad \textbf{(seq-nop)}$$

$$\langle S, (break\ ;\ q)\rangle \xrightarrow[n]{} \langle S, break\rangle \qquad \textbf{(seq-brk)}$$

$$\langle S, (loop\ p)\rangle \xrightarrow[n]{} \langle S, (p\ @\ loop\ p)\rangle \qquad \textbf{(loop-expd)}$$

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, (p\ @\ loop\ q)\rangle \xrightarrow[n]{} \langle S', (p'\ @\ loop\ q)\rangle} \qquad \textbf{(loop-adv)}$$

$$\langle S, (mem(id)\ @\ loop\ p)\rangle \xrightarrow[n]{} \langle S, loop\ p\rangle \qquad \textbf{(loop-nop)}$$

$$\langle S, (break\ @\ loop\ p)\rangle \xrightarrow[n]{} \langle S, nop\rangle \qquad \textbf{(loop-brk)}$$

$$\frac{\langle S, p\rangle \xrightarrow[n]{} \langle S', p'\rangle}{\langle S, (p\ par\ q)\rangle \xrightarrow[n]{} \langle S', (p'\ par\ q)\rangle} \qquad \textbf{(par-adv1)}$$

$$\frac{isBlocked(n, S, p)\ ,\ \ \langle S, q\rangle \xrightarrow[n]{} \langle S', q'\rangle}{\langle S, (p\ par\ q)\rangle \xrightarrow[n]{} \langle S', (p\ par\ q')\rangle} \qquad \textbf{(par-adv2)}$$

$$\langle S, (break\ par\ q)\rangle \xrightarrow[n]{} \langle S, break\rangle \qquad \textbf{(par-brk1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ par\ break)\rangle \xrightarrow[n]{} \langle S, break\rangle} \qquad \textbf{(par-brk2)}$$

$$\langle S, (mem(id)\ and\ q)\rangle \xrightarrow[n]{} \langle S, q\rangle \qquad \textbf{(and-nop1)}$$

$$\langle S, (p\ and\ mem(id))\rangle \xrightarrow[n]{} \langle S, p\rangle \qquad \textbf{(and-nop2)}$$

$$\langle S, (mem(id)\ or\ q)\rangle \xrightarrow[n]{} \langle S, nop\rangle \qquad \textbf{(or-nop1)}$$

$$\frac{isBlocked(n, S, p)}{\langle S, (p\ or\ mem(id))\rangle \xrightarrow[n]{} \langle S, nop\rangle} \qquad \textbf{(or-nop2)}$$

Evaluation

- Source code size
  - number of tokens, data/state variables
- Memory usage
  - ROM, RAM
- Responsiveness
  - time-consuming C calls

- Comparison to *nesC*
  - WSNs protocols, radio driver

Code size & Memory usage

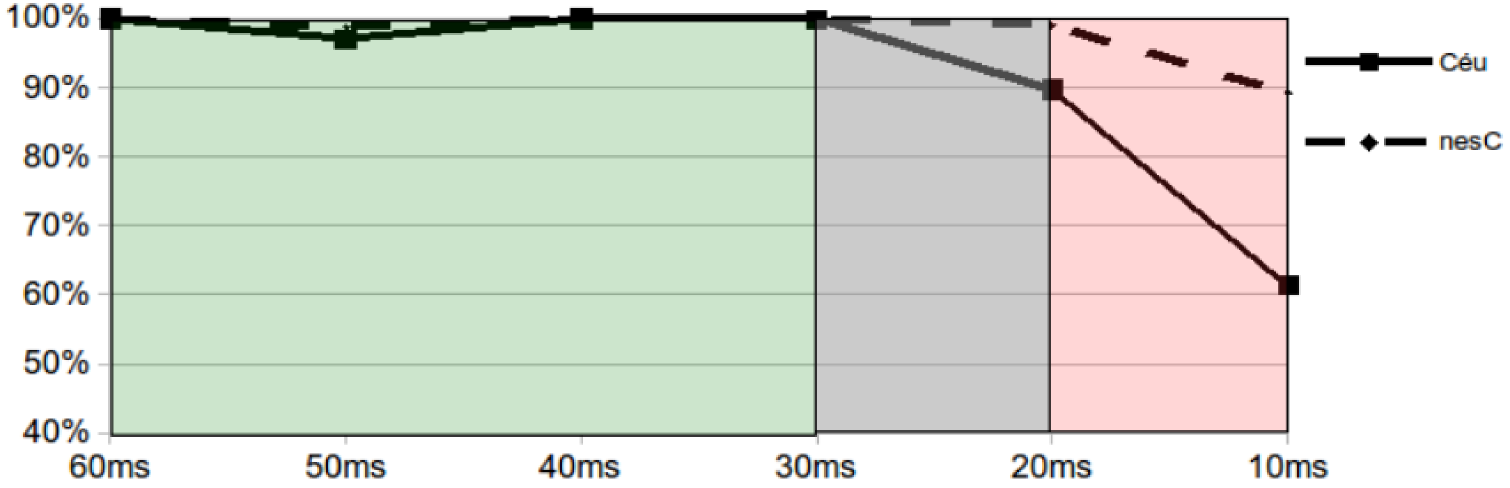
		Code size				Memory usage			
				globals					
Application	Language	tokens	*Céu vs nesC*	state	data	ROM	*Céu vs nesC*	RAM	*Céu vs nesC*
CTP	nesC	383	-23%	4	5	18896	9%	1295	2%
	Céu	295		-	2	20542		1319	
SRP	nesC	418	-30%	2	8	12266	5%	1252	-3%
	Céu	291		-	4	12836		1215	
DRIP	nesC	342	-25%	2	1	12708	8%	393	4%
	Céu	258		-	-	13726		407	
CC2420	nesC	519	-27%	1	2	10546	2%	283	3%
	Céu	380		-	-	10782		291	

no control globals

globals → locals

Responsiveness

- 10 sending nodes
  - 20-byte msgs, 600-100ms/msg
- 1 receiving node
  - 60-10ms/msg
  - 8ms operation in sequence w/ every msg



Conclusion

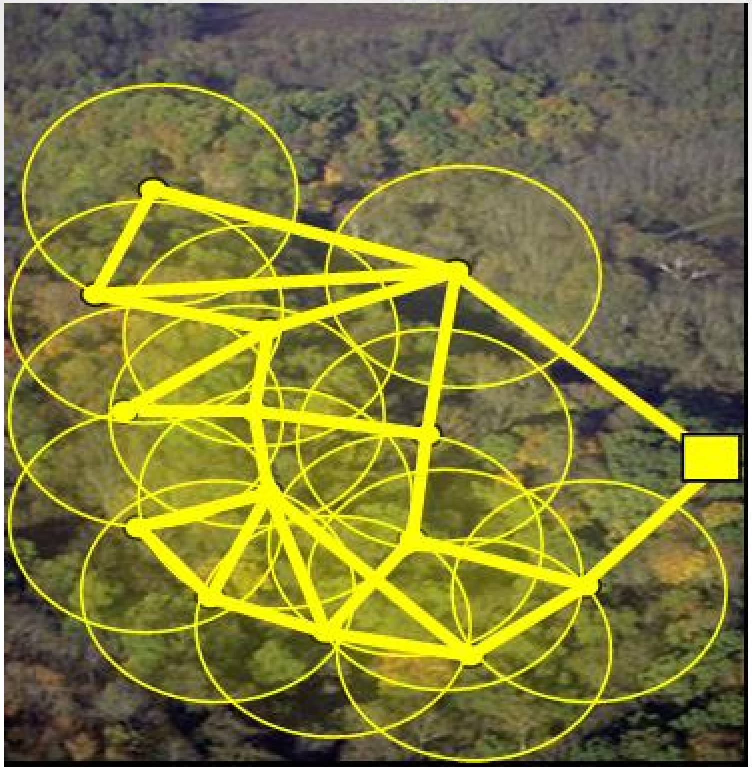
- Main contributions:
  - a comprehensive and resource-efficient design
  - stack-based internal events
  - a set of compile-time guarantees

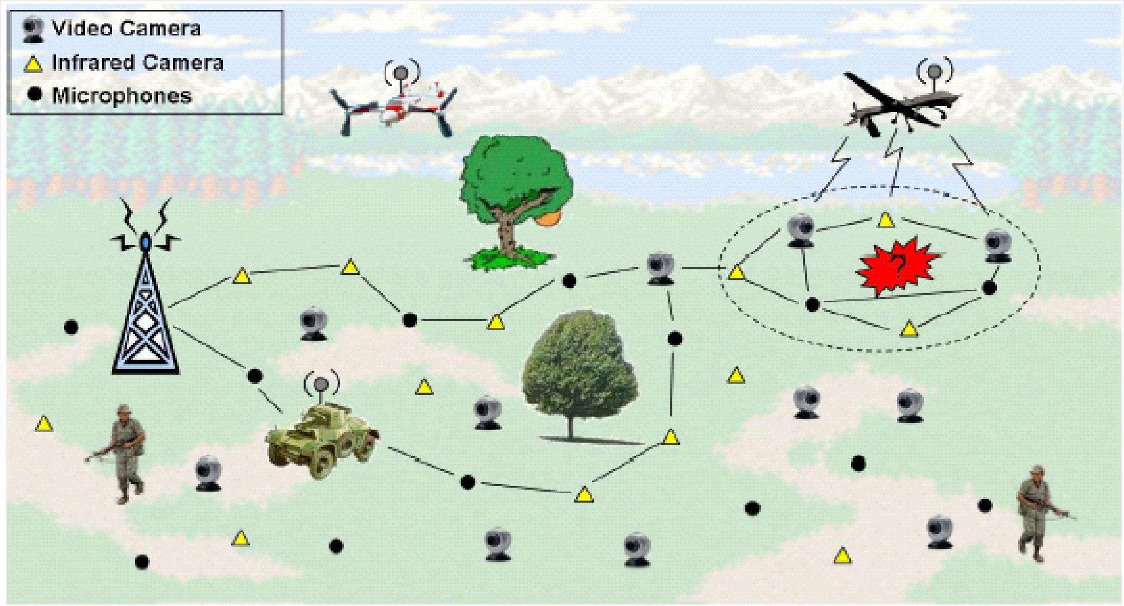
Safe System-level Concurrency on Resource-Constrained Nodes with Céu

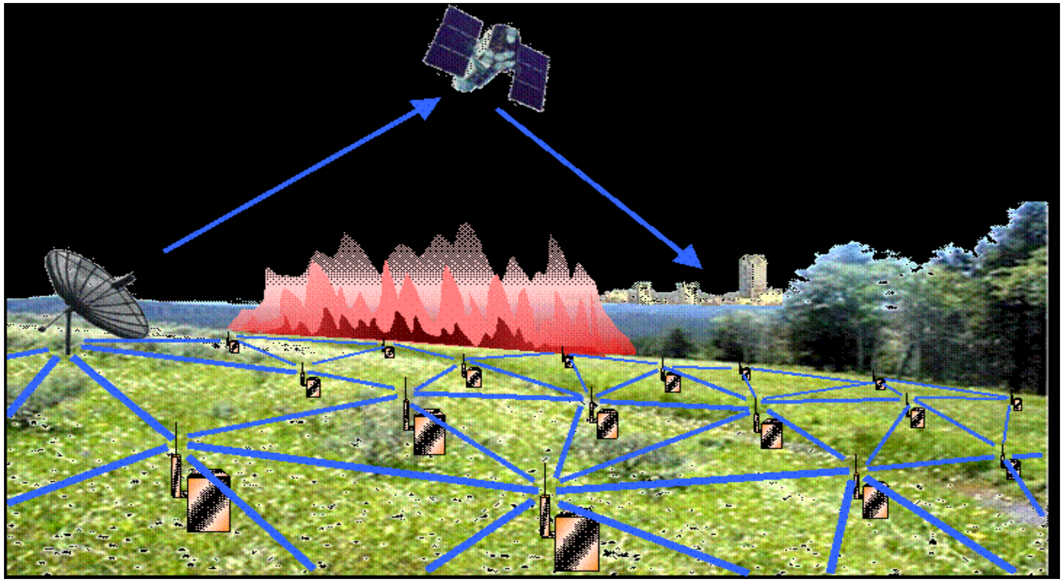
Autor:
Francisco Sant'Anna

Orientadores:
Roberto Ierusalimschy
Noemi Rodriguez

Wireless Sensor Networks



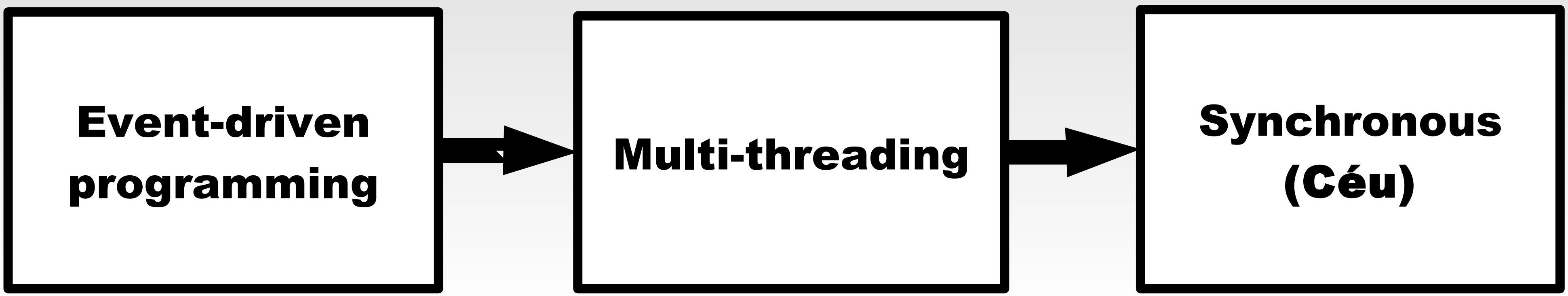




Programming models in WSNs

- Event-driven programming
  - ***TinyOS/nesC***, ***Contiki/C***
- Multi-threading
  - ***Protothreads***, *TinyThreads*, *OCRAM*
- Synchronous languages
  - *Sol*, *OSM*, *Céu*

Programming models in WSNs



- unstructred code
- manual memory management

- multiple threads
- unrestricted shared memory

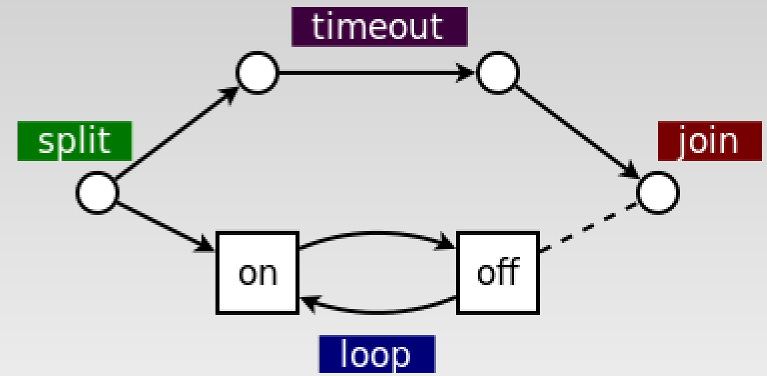
- composable threads
- safety analysis

low level → **high level**

Overview of Céu

- Reactive
  - environment in control: *events*
- Imperative
  - sequences, loops, assignments
- Concurrent
  - multiple lines of execution: *trails*
- Synchronous
  - trails synchronize at each external event
- Deterministic
  - trails execute in a specific order

- Blinking a LED
  - *sequential*: on=2s, off=1s
  - *parallel*: 1-minute timeout




```
// nesC: event-driven

event void Boot.booted () {
  call T1.start(0);
  call T2.start(60000);
}
event void T1.fired() {
  static int on = 0;
  if (on) {
    call Leds.led0Off();
    call T1.start(1000);
  } else {
    call Leds.led0On();
    call T1.start(2000);
  }
  on = !on
}
event void T2.fired() {
  call T1.cancel();
  call Leds.led0Off();
  <...> // continue
}
```

```
// Protothreads: multi-threaded

int main() {
  PT_INIT(&blink);
  timer_set(&timeout,60000);
  while (
    PT_SCHEDULE(blink()) &&
    !timer_expired(timeout)
  );
  leds_off(LEDS_RED);
  <...> // continue
}
PT_THREAD blink() {
  while (1) {
    leds_on(LEDS_RED);
    timer_set(&timer,2000);
    PT_WAIT(expired(&timer));
    leds_off(LEDS_RED);
    timer_set(&timer,1000);
    PT_WAIT(expired(&timer));
  }
}
```

```
// Céu: synchronous

par/or do
  loop do
    _Leds_led0On();
    await 2s;
    _Leds_led0Off();
    await 1s;
  end
with
  await 1min;
end
_Leds_led0Off();
<...> // continue
```

Synchronous execution

1. **Program is idle.**
2. On any external event, awaiting trails awake.
3. Active trails execute, until they await or terminate.
4. Goto step 1.

- Reactions to external events never overlap
- The synchronous hypothesis:

  "reactions run infinitely faster in comparison to the rate of events"

Synchronous execution

- Parallel compositions

```
loop do
  par/and do
    <...>
  with
    await 100ms;
  end
end
```

```
loop do
  par/or do
    <...>
  with
    await 100ms;
  end
end
```

- *Sampling and Timeout patterns*

Synchronous execution

- Céu enforces bounded execution

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

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

- *Limitation: time-consuming operations*

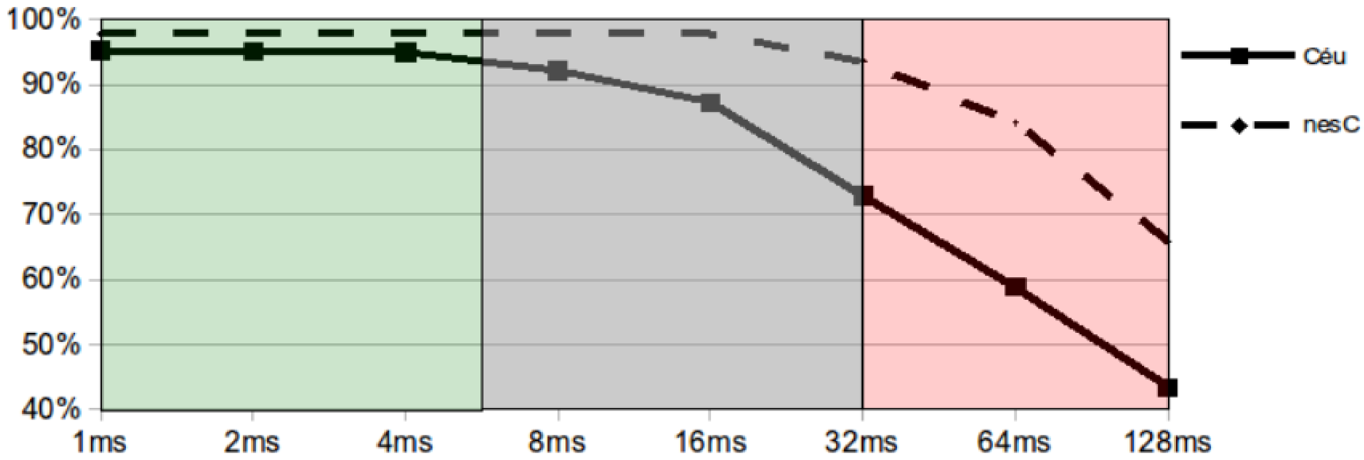
Local scopes

```
par/and do
  var int a;
  <...>
with
  var int b;
  <...>
end
var int c;
<...>
```

- blocks in parallel: sum memory
- blocks in sequence: reusue memory

Responsiveness

- 10 sending nodes
  - 20-bytes msgs, 200ms/msg
- 1 receiving node
  - 50msg/s
  - 1-128ms operation (every 150ms)



Operation	Duration
Block cypher [26, 18]	1ms
MD5 hash [18]	3ms
Wavelet decomposition [41]	6ms
SHA-1 hash [18]	8ms
RLE compression [38]	70ms
BWT compression [38]	300ms
Image processing [37]	50–1000ms

Safety

- Time-bounded reactions
- No concurrency in variables
- No concurrency in C calls
- Finalization for blocks going out of scope
- Auto-adjustment for timers in sequence
- Synchronization for timers in parallel

Related work

Language		Complexity				Safety			
name	year	1: sequential execution	2: local variables	3: parallel compositions	4: internal events	5: deterministic execution	6: bounded execution	7: safe shared memory	8: finalization blocks
Preemptive	many	✔	✔		✔		*rt*		
nesC	2003					✔	*async*	✔	
OSM	2005		✔	✔	✔				
Protothreads	2006	✔				✔			
TinyThreads	2006	✔	✔			✔			
Sol	2007	✔	✔	✔		✔	✔		
FlowTalk	2011	✔	✔						
Ocram	2013	✔	✔			✔			
Céu		✔	✔	✔	✔	✔	✔	✔	✔

- Demo applications
    - explore the programming style of Céu
- Semantics of Céu
    - control aspects
        - determinism, stacked internal events
- Implementation of Céu
    - parsing, temporal analysis, code generation

Safe System-level Concurrency on Resource-Constrained Nodes

Authors:

Francisco Sant'Anna
Roberto Ierusalimschy
Noemi Rodriguez
PUC-Rio, Brazil

Olaf Landsiedel
Philippas Tsigas
Chalmers, Sweden

Conference on Embedded Networked Sensor Systems

ACM SenSys'13 – Rome