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

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
“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

```plaintext
loop do
  par/or do
    loop do
      await 500ms;
      _leds_toggle();
    end
    with
      await 5s;
    end
  end
  await 2s;
end
```
The design of Céu

- Synchronous execution model:
  - Reactions do not overlap (based on *Esterel*)
    - Pros: safety, resource efficiency
    - Cons: heavy (async) computations

- Contributions (*safety aspects*):
  1. Shared-memory concurrency
  2. Internal events
  3. Integration with C
  4. Local scopes & Finalization
  5. First-class timers
1. Shared-memory concurrency

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

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

 Compile-time race detection
3. Integration with C

- Well-marked syntax ("_")

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

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

- **pure** and **safe** annotations
4. Local scopes & Finalization

```java
loop do
    await 1s;
    var _message_t msg;
    <!-- prepare msg -->
    _send_request(&msg);
    await SEND_ACK;
end
```
4. Local scopes & Finalization

```plaintext
par/or do
  loop do
    await 1s;
    var _message_t msg;
    <!-- prepare msg
    _send_request(&msg);
    await SEND_ACK;
    end
with
  await STOP;
end
var int x = 1;
```

Compile-time error
4. 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;
```
5. First-class timers

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

```plaintext
await 2ms;  # 3ms elapse
v = 1;
```

```plaintext
await 1ms;  # late = 1ms
v = 2;
```

```plaintext
with
await 12ms;
end
```

```
11 < 12  (always!)
```
Evaluation

- Source code size
  - number of tokens, data/state variables
- Memory usage
  - ROM, RAM
- Responsiveness
  - time-consuming C calls (e.g. encryption)

- Comparison to nesC
  - WSNs protocols, radio driver
# Code size & Memory usage

## Table

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th>Code size</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>tokens</td>
<td>ROM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Céu vs nesC</td>
<td>state</td>
</tr>
<tr>
<td>CTP</td>
<td>nesC</td>
<td>383</td>
<td>-23%</td>
</tr>
<tr>
<td></td>
<td>Céu</td>
<td>295</td>
<td></td>
</tr>
<tr>
<td>SRP</td>
<td>nesC</td>
<td>418</td>
<td>-30%</td>
</tr>
<tr>
<td></td>
<td>Céu</td>
<td>291</td>
<td></td>
</tr>
<tr>
<td>DRIP</td>
<td>nesC</td>
<td>342</td>
<td>-25%</td>
</tr>
<tr>
<td></td>
<td>Céu</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>CC2420</td>
<td>nesC</td>
<td>519</td>
<td>-27%</td>
</tr>
<tr>
<td></td>
<td>Céu</td>
<td>380</td>
<td></td>
</tr>
</tbody>
</table>

- **no control globals**
- **globals → locals**
Responsiveness

- 10 sending nodes → 1 receiving node
- 60-10 ms / msg
- 8ms operation in sequence w/ every msg
Conclusion

- A comprehensive and resource-efficient design
- A set of compile-time guarantees
  1. time/memory bounded reactions
  2. race-free shared variables
  3. race-free native calls
  4. finalization for locals
  5. auto-adjustment for timers in sequence
  6. synchronization for timers in parallel
Safe System-level Concurrency on Resource-Constrained Nodes
Wireless Sensor Networks
Wireless Sensor Networks

- Reactive
  - guided by the environment

- Concurrent
  - safety aspects

- Constrained
  - 32K ROM
  - 4K RAM
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

- Event-driven programming
  - unstructured code
  - manual memory management

- Multi-threading
  - multiple threads
  - unrestricted shared memory

- Synchronous (Céu)
  - composable threads
  - safety analysis
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

```c
// 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.

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

```plaintext
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"
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
2. Internal events

(vs external events)

- Emitted by the program
  - (vs environment)

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

- Stack-based execution policy
  - (vs queue)
2. Internal events

- Stack-based execution policy
  - (vs queue)
- Advanced control mechanisms
  - (e.g. subroutines, exceptions)
- Bounded memory & execution
  - no recursion

```
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;
  _assert(v == 2);
end
```
3. Integration with C

- Well-marked syntax ("_")

```c
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?
Local scopes

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

- blocks in parallel: sum memory
- blocks in sequence: reuse memory
Formalization

- Small-step operational semantics
- Control aspects of the language
  - parallel compositions,
  - stack-based events, finalization
- Mapping: formal $\rightarrow$ concrete
Responsiveness

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

<table>
<thead>
<tr>
<th>Language</th>
<th>year</th>
<th>Complexity</th>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preemptive</td>
<td>many</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> rt</td>
</tr>
<tr>
<td>nesC</td>
<td>2003</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> async <img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>OSM</td>
<td>2005</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>Protothreads</td>
<td>2006</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>TinyThreads</td>
<td>2006</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>Sol</td>
<td>2007</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>FlowTalk</td>
<td>2011</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>Ocram</td>
<td>2013</td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
<tr>
<td>Céu</td>
<td></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
<td><img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /> <img src="https://gitlab.com/checked.png" alt="check" /></td>
</tr>
</tbody>
</table>
- 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