Design, Simulate, Execute Embedded Systems

CPAL: High-Level Abstractions for Safe Embedded Systems

Nicolas NAVET, University of Luxembourg
Loïc FEJOZ, RealTime-at-Work

October 30, 2016 – DSM Workshop, Amsterdam
Software has become the key to innovation.

Amount of software is growing exponentially – what about productivity gains in software development?

Innovation increasingly relies on software.

Model-Driven Development is a powerful enabler but..

Programming environments still lack:

- the high-level concepts: embedded system specific language abstractions
- automation features ("state the what, not the how") that would make them more productive

CPAL: high-level programming model for embedded systems

- Allow to express non-functional requirements, timing for now
- Synthesis step ensures requirements are met

Inspired from posts at http://www.theenterprisearchitect.eu/
5-steps of MDD

- **Code only**: “What’s a model?“
- **Code visualization**: “The code is the model“
- **Runtime Environment**: „Manage code and model“
- **Model-centric**: „The model is the code“
- **Model only**: „Let’s talk models!“

**Inspired from interpreter-based interlocking systems e.g.: RATP, SNCF [4], Westinghouse**

**Figure from [2] and [3]**
CPAL is a real-time embedded systems specific language

A Model and program
functional and non-functional concerns

B Simulate
possibly embedded within external tools such as RTaW-Pegase™ and Matlab/Simulink™

C Execute
bare metal or hosted by an OS - prototypes or real systems

A joint project of RealTime-at-Work and University of Luxembourg since 2012
CPAL: views created out of the code

Functional view

Finite State Machine describing the logic of a process

Code

Activation of the processes over time

Available from http://designcps.com
CPAL language design objectives

1. Facilitating the writing of **correct embedded code**

2. Speeding up the development through **domain-specific abstractions** for:
   - Periodic activities and real-time scheduling
   - Time measurements and manipulation
   - Finite state machines
   - High-level interfaces to I/Os
   - etc

3. “Write once, Run Anywhere” with **equally acceptable timing behaviour** on different platforms
Facilitating the writing of **correct code/system**

- Designed with simplicity in mind - small and readable language
- Strongly typed language: conversions must be explicit
- No dynamic memory & no pointers
- **Built-in loop over** construct to prevent “off-by-one” errors when iterating over collections
- Testing the equality of floating-point numbers is forbidden
- All processes are known before run-time - workload is bounded
- **Built-in** code execution time monitoring support
- Can run on bare hardware without OS
- Utilities: schedulability analysis, code formatter and naming convention verifier
Domain-specific constructs

```c
struct Fruit {  APPLE,  BANANA,  ORANGE
};

struct Item {  uint32: quantity  Fruit: f;
```
Hello, World

process def Hello_World()
{
    state Main {
        IO.println("Hello, world");
    }
}

process Hello_World: a_task[100ms]();
processdef Servo_Tester(
  in bool: change_mode_cmd,
  in uint16: manual_position,
  out int32: position
){
  state Manual {
    position = int32(as(manual_position) + int32.FIRST);
  }
  on (change_mode_cmd) to Neutral;
  state Neutral {
    position = 0;
  }
  on (change_mode_cmd) to Auto_Min;
  state Auto_Min {
    position = int32.FIRST;
  }
  on (change_mode_cmd) to Manual;
  after (1s) to Auto_Max;
  state Auto_Max {
    position = int32.LAST;
  }
  on (change_mode_cmd) to Manual;
  after (1s) to Auto_Min;
}

var bool: pin_gpio_c10_in;
var uint16: pin_adc16_0_1;
var int32: pwm_c_0_0;

process Servo_Tester: main_task[100ms](pin_gpio_c10_in, pin_adc16_0_1, pwm_c_0_0);

---

**FSM in processes**

- Transition first semantics
- Code in transitions and states
Working with time

```javascript
const time64: sleep_time = 3ms;

processdef Manipulating_Time() {
  /* Internal granularity of time is picosecond (ps) */
  var time64: a_duration = 5s + 150ms + 3ns + 1ps;
  var time64: same_duration = 5s150ms3ns1ps;
  var time64: another_duration = 2 * a_duration - 1ps;
  var time64: t0 = time64.time();
  var time64: t1;

  state A {
    IO.println("Value of a_duration is \%t", a_duration);
    assert(ls == 1000ms);
    assert(lms == 1000us);
    assert(lus == 1000ns);
    assert(lns == 1000ps);
    sleep(sleep_time);
    t1 = time64.time();
    assert(t1 - t0 >= sleep_time);
  }
}

process Manipulating_Time: p1[100ms]()
process Manipulating_Time: p2[0.1Hz]()
```

time64 type to measure and manipulate time – granularity is picosecond
Units: s, ms, ns, us, ps and Hz
Designer’s objective: model behaves as the real-system

“digital mockups”
“digital twins”

Inject delays in simulation mode so as to reproduce the time it takes to execute the code on a specific platform
Simulating execution time

- Annotations for real-time scheduling and activation patterns others than periodic
- Delays can be obtained from runtime monitoring
Co-simulation in Matlab/Simulink® [10]

Driving scenarios

CPAL controller

Ongoing work: characterize HW resources required for timing correctness and ensure them at run-time
Interacting with hardware

```plaintext
process define LED_Control in bool: button, out bool: led
{
    /* IO.sync() implicitly called upon each activation of task */
    state Main {
        /* Some bit banging */
        if (button) {
            led = true;
            IO.sync(); /* Explicitly synchronizes I/Os */
            sleep(250ms);
            led = false;
        }
    }
    /* IO.sync() implicitly called at the end of execution of the process*/
}
process LED_Control: blinker[500ms](pin2_in, pin0_out);
```

IOs are synced upon the activation and exit of the process, and calls to `IO.sync()`
Introspection features

Eases portability and self-adaptive behaviour

```plaintext
processdef Self_Adapting()
{
    var time64: jitter_threshold = self.period * 3/2;
    common{
        /* Query process pid and activation offset at startup */
        IO.println("pid: %u offset: %", self.pid, self.offset);
        /* A strictly periodic process would start to execute every period */
        if (self.current_activation - self.previous_activation > jitter_threshold){
            /* Warning: start-of-execution jitter is currently very high, possible
            counter-measures that can be taken at run-time include adapting
            1) the control algorithm (e.g. mode change),
            2) the process activation pattern (e.g. increase period),
            3) the scheduling parameters (e.g. increase process priority*/
        }
    }
    /* Body of the process */
    state A {
        /* ... */
    }
    process Self_Adapting: p1[100ms]();
}
```
Use-Case
Developing CPS: a smart parachute for UAV

UAVs autopilots cannot be trusted – minimal safety through a remote termination component
Partnership with Alérion company

Termination upon loss of connection or pilot’s decision
Software architecture

Communication

On-board module

UI

HW control

- `uplink`
- `rcp_xbeeTask [50ms]`
- `rcp_emergencyCommand`
- `downlink`
- `rcp_modeTask [50ms]`
- `rcp_emergencyActivated`
- `rcp_uiTask [200ms]`
- `rcp_powerLED`
- `rcp_inEmergencyLED`
- `rcp_hwTask [20ms]`
- `rcp_powerSwitch`
- `rep_servo`
- `rep_ip`
Actual max. latency depends on the ground speed target, the minimum acceptable altitude, the weight of the UAS and the characteristics of the parachute (opening time, lift, etc).
Model-based fault-injection

Time for the parachute to deploy (in seconds) and satisfaction of requirement R4 versus network quality ratio [11]
Ongoing & future work

- Upcoming releases: HW annotations, multi-core & power mode support
- Code generation and/or hook to native code for higher performances
- CPAL: MDD for IoT
- **Medium term:**
  - timing equivalence between simulation and execution
  - “State the what, not the how” for energy & safety
  - SILx qualification for the execution engine

*CPAL is free to use*
Thank you for your attention!

Want to give it a try? Binaries, code examples and playground at [https://designcps.com](https://designcps.com)
References


