CPAL: High-Level Abstractions for Safe Embedded Systems

Nicolas Navet
University of Luxembourg / FSTC
6 rue Richard Coudenhove-Kalergi
Luxembourg, 1359, Luxembourg
nicolas.navet@uni.lu

Loïc Fejoz
RealTime-at-Work (RTaW)
4 rue Piroux, Centre Atrium
Nancy, 54000, France
loic.fejoz@realtimeatwork.com

Abstract

Innovation in the field of embedded systems, and more broadly in cyber-physical systems, increasingly relies on software. The productivity gain in software development can hardly keep up with the demand for software despite the increasing adoption of Model-Driven Development (MDD). In this context, we believe that major productivity and quality improvements are still ahead of us through better programming languages and environments. CPAL, the Cyber-Physical Action Language, is a contribution in that direction with the objective to speed-up the development of embedded systems with dependability constraints. The objective of this paper is to present and illustrate the use-cases of the high-level abstractions offered to the developer in CPAL with respect to real-time scheduling, introspection mechanisms, native support of Finite State Machines (FSMs), abstracting the hardware and decoupling functional concerns from non-functional concerns.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures

Keywords Cyber-Physical Systems, Embedded Systems, Model-Driven Development, Control Applications, Dependability

1. Introduction and Related Work

Context of the Work. Innovation crucially relies on software today and software is actually disrupting complete fields of the economy and the industry. Embedded systems, and Cyber-Physical Systems at large, are no exception whatever their application domains: vehicles, medical devices, home appliances, factory automation, power distribution, Internet of Things (IoT), etc. If the amount of software is growing fast, the productivity gains in software development are much slower. In this landscape, Model-Driven Development is certainly a key technology but, in our view, programming languages still lack the high-level abstractions and automation features ("*state the what, not the how*") that would make them more productive.

The objective of CPAL is to offer a solution to fill this gap for the domain of embedded systems. CPAL is a high-level interpreted language supporting a model-driven development flow for the development of timing-predictable embedded systems. CPAL can be used in a simulation mode early in the early design phases, or, in real-time mode, to execute programs on top of an OS or on a bare-metal platform. CPAL serves to describe both the functional behavior of the functions, that is their code, as well as the functional architecture of the system (i.e., the set of functions, how they are activated, and the data flows among the functions). CPAL is however not a full-fledged Architecture Description Language able to describe complex system architectures, like AADL [8] or Chariot [14] are, but a programming language that can possibly be used to develop software components from systems described in an ADL. As discussed in this paper, in addition to the functional description, the developer can also express timing-related non-functional concerns.

CPAL has been used in a number of academic and industrial case-studies, such as the development of a smart parachute add-on component for UAVs [7] and an AUTOSAR compliant engine function [17], proposing an answer to the Thales FMTV challenge [3] and developing a simulation model of an automotive middleware on top of Ethernet [15]. CPAL has also been used since 2013 as the supporting language to teach Model-Driven Engineering for embedded systems at the University of Luxembourg. The CPAL documentation [11], graphical editor and the execution engine for Windows, Linux and Raspberry Pi platforms are freely available from `http://www.designcps.com`. To experiment with CPAL without any installation, a "playground" is available on the web site along with code examples illustrating the language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

DSM'16, October 30, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4894-2/16/10...$15.00
http://dx.doi.org/10.1145/3023147.3023153

Why a New Domain-Specific Programming Language? The design and development of embedded systems with dependability constraints necessitates the adoption of many good practices throughout the entire development cycle. The need to develop a new language originates from our analysis that none of the programming languages we were aware of possessed all the features we think needed to combine productivity and code correctness:

- General-purpose programming languages do not offer all the right abstractions for today's real-time embedded systems: scheduling periodic activities, time as a first class citizen, safe inter-process communication, native support for finite-state machines, high-level interfaces to I/Os, support for timing and formal verification, etc. As an illustration, just consider how cumbersome it is to program in *C* a periodic activity on a real-time OS, let alone programming a set of tasks with real-time constraints sharing resources and synchronizing constraints. We note that an alternative to developing a new language is to adapt, by domain-specific extensions and restrictions, an existing language. In the field of embedded systems, this is what has been with the *mbeddr* framework [19] which is based on *C*. Such an approach has limits about what can be done since the underlying language dictates many things, but possesses advantages for instance in terms of tool-set availability and ease of adoption by the programmers.
- Synchronous languages (Lustre [6], Signal[4], etc) are meant for critical applications but they impose many constraints and a programming style that is very specific and does not suit everyone. The initial learning curve is steep too. In addition, many of the most appealing synchronous languages, such as Prelude [13] or Giotto [10], are actually only Architecture Description Languages (ADL) which means that the algorithms have to be coded in an other language. An important benefit of synchronous languages is that formal proof support, both in the time domain and value domain, is usually available, whereas CPAL currently only offers schedulability analysis to check for possible deadline misses (see [1]) and simulation.
- Correctness starts with simplicity: simplicity of the language constructs, but also of the runtime environment. The CPAL language is Turing complete but relatively small in the number of constructs and abstractions it defines, which facilitates the learning and teaching of CPAL. Although code-generation is currently being investigated, CPAL is for the time being an interpreted language whose execution engine is in the range of 10,000 lines of code (see §2.2). Especially when the execution engine runs directly on the hardware, without an OS, the amount of code involved is several order of magnitude less than with compiled code which is of course makes it easier to verify the code correctness.

The reader is referred to [12] and [2] for a more thorough comparison with other programming languages and, more generally, programming paradigms, especially those belonging to the realm of synchronous approaches.

2. High-Level Abstractions for the Domain of Critical Embedded Systems

The main objective of CPAL is to provide high-level abstractions suited to express domain-specific patterns of behaviors and enforce the good programming practices of the domain of embedded systems. In this Section, after an introduction to CPAL and its development environment, we focus on features related to time, introspection and interactions with the hardware. Another requirement in the design of CPAL has been to facilitate the writing of correct code. For instance, there is no untyped data, no pointers, no dynamic memory after initialization, only simple control-flow constructs, a `loop over` construct ensuring a bounded number of loop iterations, communication channels with LIFO or FIFO semantics, etc. Some statements which are know to be error-prone are also forbidden in CPAL, such as testing the equality of two floating-point variables.

2.1 Basics of CPAL: Processes and Automata

We here first introduce the basic features of CPAL through the example of a servo tester whose code is shown in Figure 1. At the core of CPAL is the concept of `process` which is a built-in type for a recurrent activity of the system. Processes can be seen as functions that are activated periodically, or when certain activation conditions are met. The process is first defined (line 1 to 28 in in Figure 1), like a class is in object-oriented languages, then it is instantiated with certain activation conditions and global variables - possibly mapped to I/O ports as in the servo tester example - as input and output parameters (line 34 to 36 in Figure 1). Processes can have arguments in input and output; here the servo tester takes two input variables and outputs the servo signal. Embedded in each process is a Finite-State Machine (FSM), possibly reduced to a single state, that helps to describe the logic of the process in a correct and non-ambiguous manner.

A process has a memory in the sense that, when it is activated, it resumes in the state in which it was at the end of the previous activation but the code of the state is not executed at this point. A transition out of this state is first taken if any can be, and only then the code of the current state of the FSM is executed. A specific situation is the very first execution of the process where it is assumed that a dummy transition doing nothing leads to the process initial state. After the execution of the state code, the execution of the process is then finished and the CPU becomes available for another process that is pending execution, the choice of which by the execution engine depends on the scheduling

```
processdef ServoTester(
  in bool: changeModeCmd,
  in uint16: manualPosition,
  out int32: position)
{

  state Manual {
    position = int32.as(manualPosition) + int32.FIRST;
  }
  on (changeModeCmd) to Neutral;

  state Neutral {
    position = 0;
  }
  on (changeModeCmd) to AutoMin;

  state AutoMin {
    position = int32.FIRST;
  }
  on (changeModeCmd) to Manual;
  after (1s) to AutoMax;

  state AutoMax {
    position = int32.LAST;
  }
  on (changeModeCmd) to Manual;
  after (1s) to AutoMin;
}

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

process ServoTester: mainTask[100ms](pin_gpio_c10_in,
                                     pin_adc16_0_1,
                                     pwm_c_0_0);
```

Figure 1. CPAL code implementing a servo tester as a periodic process. The logic is described as a FSM whose transitions between states are triggered by a Boolean condition that evaluates to true, or after a certain time spent in a continuous manner in a state. The access to hardware is abstracted through the use of global variables mapped onto the I/O ports.

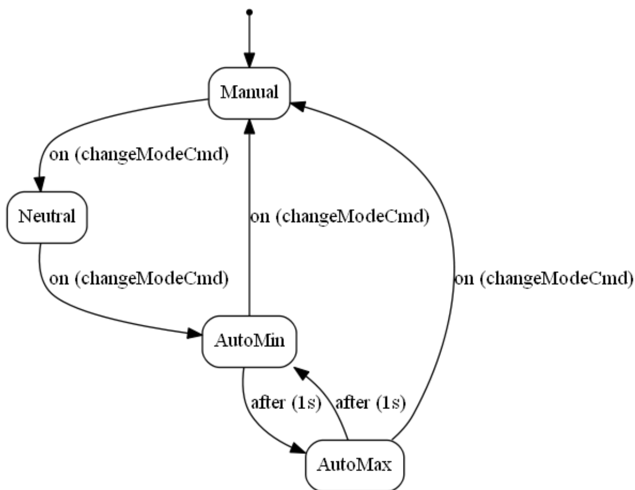

Figure 2. Graphical view of the FSM describing the logic of the servo tester (screenshot from the CPAL-Editor).

parameters (see §2.4). The transition-first semantics allows for faster response to external events and the transitions can execute code in CPAL.

The CPAL language is fully textual but views are produced out of the code to inform the developers, and possibly other stakeholders, about certain facets or behaviors of the program. Currently, the CPAL-Editor building on Graphviz is able to generate views of the FSMs in the processes, as in Figure 2 for the servo tester example, a Gantt chart of the process executions over time (see Figure 8) and a view of the functional architecture, that is the set of processes making up an application and the flows of data among them. In a first phase of the project, we developed the prototype of a graphical editor to specify the FSMs but decided to re-allocate the engineering efforts to other areas more critical at this stage of the project such as the timing correctness of the interpretation engine. If a graphical editor could certainly be helpful to some users, we believe that for users with prior programming experience, the loss of productivity of writing textual code is limited; the important thing in our view being that the FSMs can be visualized.

2.2 Execution Environments and Execution Modes

During the development, the interpreter is executed from within the CPAL-Editor. As soon as a change in the source files is detected, the CPAL-Editor automatically parses, analyzes and executes the code in background to generate the views out of it. CPAL can also be executed in a co-simulation environment. For the development of control systems, the execution in Matlab/Simulink as an S-function is supported (see [16]). CPAL is also the simulation language to program high-level protocol layers in the RTaW-Pegase embedded network simulator from RTaW. The CPAL code that is deployed on a target is either launched from a shell or uploaded in memory and executed immediately after the startup on bare-metal platforms.

There are two execution modes in CPAL: the *simulation mode* and the *real-time mode*. In the latter mode, the timing characteristics of the processes are respected (e.g., periods) and the programs can access I/O ports. Depending on the platform, the execution engine runs on top of an OS (on Linux, Windows, Mac OS X and Raspbian) or on the bare hardware (Freescale FRDM-K64F). In simulation mode, the code is executed as fast as possible, thus not in real-time, and do not have access to the I/O ports. Through timing and scheduling annotations (see Section 3), it is however possible to enforce a more timing-realistic behavior by simulating execution times and possibly execution jitters. An interesting feature of CPAL in terms of productivity is that the simulation code can be re-used without changes in real-time mode on a target, for rapid-prototyping or even for the production system. All platforms possess both simulation and real-time mode, except the FRDM-K64F board, which does not support the simulation mode. This platform however offers the best real-time predictability in real-time mode since it does

```
1  const time64: sleep_time = 3ms;
2
3  processdef Manipulating_Time() {
4    /* Internal granularity of time is picosecond (ps) */
5    var time64: a_duration = 5s + 150ms + 3ns + 1ps;
6    var time64: same_duration = 5s150ms3ns1ps;
7    var time64: another_duration = 2 * a_duration - 1ps;
8    var time64: t0 = time64.time();
9    var time64: t1;
10
11   state A {
12     IO.println("Value of a_duration is %t", a_duration);
13     assert(1s  == 1000ms);
14     assert(1ms == 1000us);
15     assert(1us == 1000ns);
16     assert(1ns == 1000ps);
17     sleep(sleep_time);
18     t1 = time64.time();
19     assert(t1 - t0 >= sleep_time);
20   }
21 }
22
23 process Manipulating_Time: p1[100ms]();
24 process Manipulating_Time: p2[0.1Hz]();
```

Figure 3. Expressing, measuring and manipulating time quantities in CPAL. Function `sleep()` suspends the execution of a process during the specified duration and `time64.time()` measures the time since the startup of the interpreter with a picosecond accuracy.

not have the interferences from an OS. The reader is referred to [11] for more information about the execution modes and platforms capabilities.

2.3 Working with Time Durations

Performing actions at certain time points in time and being able to express time quantities is a common need in embedded programming. To measure and manipulate time quantities, CPAL possesses the native `time64` data type. The time units available are `s`, `ms`, `us`, `ns`, `ps` and `Hz` (Hertz). The latter unit is a natural unit in many applications to express the rate of activation of activities. It can however only be used for assignment and defining the periods of the processes, and not for doing arithmetic on time quantities like the other time units since the meaning of such operations would be unclear. CPAL provides addition, subtraction and modulo operators between `time64` values and multiplication and division between `time64` and `uint64`. The use of `time64` quantities, and the `sleep()` and `time64.time()` functions with the usual semantics, is illustrated in Figure 3.

2.4 Introspection Mechanisms and Dynamic Reconfiguration

The ability to implement adaptive behaviors requires that the activities of an application can learn about themselves during the execution. For this purpose, CPAL offers introspection features enabling to query at run-time the pid, period, offset, priority, deadline and activation time of the current and previous instance of any process. The example in Figure 4 shows a process detecting at run-time that its start-

```
1  processdef Self_Adapting()
2  {
3    var time64: jitter_threshold = self.period * 3/2;
4    common{
5
6      /* Query process pid and activation offset at startup */
7      IO.println("pid: %u offset: %t", self.pid, self.offset);
8
9      /* A strictly periodic process would start to execute every period */
10     if (self.current_activation - self.previous_activation > jitter_threshold){
11       /* Warning: start-of-execution jitter is currently very high, possible
12       counter-measures that can be taken at run-time include adapting
13       1) the control algorithm (e.g. mode change),
14       2) the process activation pattern (e.g. increase period),
15       3) the scheduling parameters (e.g. increase process priority*/
16     }
17   }
18   /* Body of the process */
19   state A {
20     /* ... */
21   }
22 }
23
24 process Self_Adapting: p1[100ms]();
```

Figure 4. Snippet of code showing how a process can detect that its start-of-execution jitter is abnormally high. The code in the `common` section is shared among all the states of a process and will be executed before the state-specific code.

of-execution jitter is very high, which is detrimental to the quality of control in many systems. The threshold condition depends here on the actual rate of execution of the process, which eases the portability and genericity of the code.

Once an abnormal timing behavior has been detected, appropriate measures can be taken. First, a change in the functioning mode of the application can be triggered, and the FSMs embedded in the processes are well suited to describe applications with different modes and the transitions between the modes. Another possibility is to adapt the characteristics of the tasks and the quality of service offered by the run-time environment. In CPAL, some characteristics of the system can be reconfigured at run-time using (non-functional) annotations to the code (see Section 3). For instance, it is possible to change the rate of activation of a process, increase its priority if Non-Preemptive Fixed-Priority scheduling (NPFP) is used, or reduce its deadline if Non-Preemptive Earliest-Deadline First (NPEDF) is used.

Another useful design pattern to detect unwanted conditions occurring at run-time is to have a process dedicated to the monitoring of the other processes or a subset of them. Whenever needed, this supervision process can take appropriate measures, such as initiating an error recovery strategy or a mode change.

2.5 Interacting with the Hardware

A domain-specific language for embedded systems should offer a good support to the programmer for sensing and actuating operations. In CPAL, global variables can be mapped to I/O ports to abstract the hardware from the programmer point of view. By default, these I/O mapped variables used by a process are updated in reading when the process starts to execute, and in writing at the end of its execution. Sometimes, as in the example of Figure 5, it is however needed to update I/O mapped variables during the execution of the process, this can be done by an explicit call to `IO.sync()`.

```
processdef 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);

```

Figure 5. Update of the I/O ports through the use of the `IO.sync()` function. I/O ports are transparently updated when the process arguments are evaluated and upon the completion of the process execution.

The synchronous languages we are aware of do not provide such a possibility and impose a single write operation at the end of execution.

The mapping between the global variables and the I/O ports is in the version of CPAL available at the time of writing (version 1.15) achieved through a naming convention (e.g., `pin_gpio_c10_in` in Figure 1). To ease the portability of CPAL applications to new platforms and the use of new I/Os, we have started extending the annotations mechanisms in CPAL to hardware annotations with a new `@cpal:hardware` annotation with is for instance used to configure communication over UDP. In the spirit, this annotation scheme is similar to the one in SenseDSL [5].

3. A Clear Separation Between Functional and Non-Functional Concerns

The correctness of a real-time application is twofold. First, the results of the computation must be correct which is called the correctness in the *value domain*. Second, the timing behavior of the application must be correct too, this is the correctness in the *time domain*. Typically, the latter involves that deadlines must be met and jitters occurring at run-time (*i.e.*, process start-of-execution and end-of-execution jitters) must be kept within acceptable bounds. Timing is an important non-functional concerns but there are other dimensions of importance in CPS such as power consumption, security and safety. We discuss hereafter the timing concerns which have been the focus of CPAL until now.

In CPAL, functional and non-functional concerns are decoupled. The non-functional concerns are expressed in annotations, placed inside or outside the code, as the programmer decides it, but anyway well identified as not belonging to the code itself. In simulation mode (see §2.2), the annotations are used for instance to experiment scheduling strategies or reflect the processing power of different execution platforms. The same code can be re-used in real-time execution mode at a later stage, but then, many annotations will be ignored. Typically this is the case for the annotations describing the execution time of the code, that are used in simulation to obtain a more timing-realistic behavior as shown in the Ganttt diagram of Figure 8.

```
processdef Varying_Execution_Time()
{
  state State1 {
    @cpal:time {
      State1.execution_time = 15ms;
    }
  }
  on (true) to State2;

  state State2 {
    a_named_block: {
      @cpal:time {
        block.execution_time = 35ms;
      }
    }
  }
  on (true) to State1;
}

processdef Conditional_Execution_Time()
{
  state Main {
    @cpal:time {
      if (uint16.rand_uniform(0,2)==0) {
        Main.execution_time = 1ms;
      } else {
        Main.execution_time = 15ms;
      }
    }
  }
}

process Varying_Execution_Time: p1[70ms]();
process Conditional_Execution_Time: p2[200ms]();
```

Figure 6. Timing annotations used in simulation to capture the execution times observed at run-time. The Matlab/Simulink port of CPAL [16] enables to re-inject these delays in the control loops for the design of timing-sensitive controllers.

3.1 Simulating Execution Times

In simulation mode, it is possible to take into account the time it takes to execute the code of a process through a `@cpal:time` annotation. The execution time can be expressed at the level of a state (see lines 4-6 in Figure 6), the level of a block (called *named block* in CPAL, see lines 11-15 in Figure 6) or globally. In the later case, the annotation is defined at the global scope for each instance of a process and holds whatever the state in which this instance of the process is. The execution time can be static or dynamic, it can depend on the value of a condition (as on line 24 in Figure 6) or it can be the result of a computation. Annotations are implemented as blocks of code that are evaluated immediately before the activation of the processes.

State-level and block-level annotations obey the same logic but cannot be used interchangeably as they may result in different timing behavior. With state-level annotations, all code statements belonging to the state are executed at the time of the entry in the state; the time being only incremented by the execution time upon the exit of the state. Whereas with block-level annotations, the time is incremented upon the exit of the block, which means that the

remaining instructions coming afterward will be executed at the new value of time. Block-level annotations can be used in transitions too and permits a fine-grained timing characterization of the code. In practice, the execution times in the processes can be derived by monitoring the execution on the target using the `--stats` option of the interpreter, or, for more accuracy, with an oscilloscope. The later solution must be used on a bare-metal platform. A set of execution time annotations can be defined for each execution platform.

3.2 Simulating Release Jitters

In many systems, due for instance to limited time accuracy or runtime overhead, tasks will not be released and ready to execute exactly when they are scheduled to be. A certain delay, called release jitter, may happen and vary from a task instance to the next. This can be modeled in CPAL using a `@cpal:time` annotation as illustrated in the code snippet below.

Although formally the release jitter is something different from the start-of-execution jitter seen in §4 that is mainly caused by the execution of higher priority processes, the release jitter annotation can be used to aggregate all the latencies created by the other processes at early stages of the development process when the other processes are not precisely known.

```
1  const time64: min_jitter = 0ms;
2  const time64: max_jitter = 10ms;
3
4  processdef Process_With_Release_Jitter() {
5   /* .. */
6  }
7
8  process Process_With_Release_Jitter: p1[100ms] ();
9
10 @cpal:time:p1{
11   p1.jitter = time64.rand_uniform(min_jitter, max_jitter);
12 }
```

Figure 7. Process release jitters can be reproduced in simulation through an annotation. The value of the jitters is set individually for each process instance. A timing annotation at the global scope suffixed by the name of a process like here will be executed before each activation of the process while, without the suffix, it will only be executed once at program startup.

3.3 Specifying Scheduling Policies

The default CPAL scheduling policy is FIFO: processes are executed in the order of their release. Although FIFO performs rather poorly in terms of meeting deadlines (see [1]), especially when there are tasks with large execution times, it possesses a unique property among all non-idling policies: the execution order of the processes is the same whatever the execution time of the code, and thus whatever the execution platform. This property called *event-order determinism* ensures that simulated code will behave identically to the deployed code with respect to execution order of the "significant" events of the program (*i.e.*, sensing and actuating points). CPAL provides also the Non-Preemptive Earliest Deadline First (NPEDF) and Non-Preemptive Fixed Priority scheduling policies (NPFP). Both policies require to set one parameter per process, respectively the deadline and the priority. The scheduling policy and its parameters are defined with a `@cpal:time` annotation, and they can be changed at run-time depending on the functioning mode of the application.

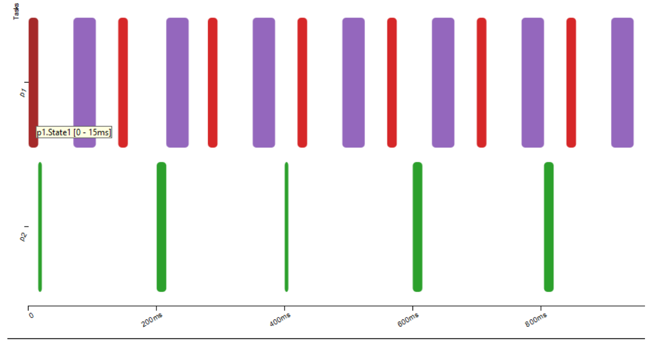

Figure 8. Gantt diagram of the execution of the processes defined in the code of Figure 6 (screenshot from the CPAL-Editor).

4. Ongoing Work

CPAL has been designed with two objectives related to non-functional concerns which are driving the ongoing developments:

1. *"State the what, not the how"*: the idea is that the developer should state the permissible behavior of the program, and a system synthesis step involving both analysis and optimization generates executable artifacts that guarantee the requirements to be met. The feasibility of this idea with respect to scheduling has been assessed in [2, 18] but other dimensions such as power consumption and dependability remains to be studied. A question of particular interest in the targeted application domain of CPAL is how an application can be made sufficiently robust to respect a given safety level (e.g., SIL2) by introducing at the synthesis step fault-tolerant mechanisms such as for instance redundant executions or watchdog mechanisms.

2. *"Timing-equivalent behavior between simulation and execution"*: a program is executed and verified in simulation mode on a workstation and the same program later deployed on an embedded board executes with a timing-equivalent run-time behavior. Timing-equivalence will have different meanings depending on the applications but this property would significant ease the design of latency-sensitive applications such as control systems. How to achieve such a property is a question currently

investigated in a PhD thesis in our group at the University of Luxembourg.

CPAL is currently being extended to multicore and distributed systems. The approach taken is to have one interpreter per computational resource with synchronization points between the interpreters when needed to ensure that all interpreters have a consistent view of the system. This approach has been validated on the simulation of relatively simple multicore systems and networks but its efficiency on embedded targets and more complex simulation models remains to be ascertained. Although model interpretation brings benefits [17] and is suited for many applications, it does not cover all use-cases mainly because of the slowdown with respect to compiled code. We are considering several approaches ranging from allowing "opaque" binary functions to be called from CPAL code, like in the MAUDE language [9], to partial or complete code generation. Finally, we are currently extending CPAL in the domain of communication to better support IoT applications.

Acknowledgments

CPAL has been a team effort since its development started in 2012. The authors wish to thank all contributors to CPAL, especially Lionel Havet and Julien Rische from RTaW who have been the main developers of the interpretation engine.

References

[1] S. Altmeyer, S. Manikandan Sundharam, and N. Navet. The case for FIFO real-time scheduling. Technical report, University of Luxembourg, 2016.

[2] S. Altmeyer and N. Navet. Towards a declarative modeling and execution framework for real-time systems. *SIGBED Rev.*, 13(2):30–33, April 2016.

[3] S. Altmeyer, N. Navet, and L. Fejoz. Using CPAL to model and validate the timing behaviour of embedded systems. In *Proc. 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Lund, Sweden, July 2015.

[4] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103 – 149, 1991.

[5] C. Berger. SenseDSL: Automating the integration of sensors for mcu-based robots and cyber-physical systems. In *Proc.14th Workshop on Domain-Specific Modeling*, DSM '14, pages 41–46, New York, NY, USA, 2014. ACM.

[6] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for real-time programming. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[7] L. Ciarletta, L. Fejoz, A. Guenard, and N Navet. Development of a safe CPS component: the hybrid parachute, a remote termination add-on improving safety of UAS. In *Proc. Embedded Real-Time Software and Systems (ERTSS'16)*, January 2016.

[8] P. H. Feiler, B. A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. In *Proc. 2006 IEEE Conference on Computer Aided Control System Design*, pages 1206–1211, Oct 2006.

[9] N. Gobillot, C. Lesire, and D. Doose. *A Modeling Framework for Software Architecture Specification and Validation*, pages 303–314. Springer, 2014. Proc. SIMPAR 2014.

[10] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[11] N Navet and L. Fejoz. *The CPAL Programming Language*, September 2016. version 1.06, Available at `https://www.designcps.com/wp-content/uploads/cpal-intro.pdf`.

[12] N. Navet, L. Fejoz, L. Havet, and S. Altmeyer. Lean model-driven development through model-interpretation: the CPAL design flow. In *Proc. Embedded Real-Time Software and Systems (ERTSS '16)*, January 2016.

[13] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[14] S. Pradhan, A. Dubey, A. Gokhale, and M. Lehofer. CHARIOT: A domain specific language for extensible cyber-physical systems. In *Proc. Workshop on Domain-Specific Modeling*, DSM 2015, pages 9–16, 2015.

[15] J. R. Seyler, T. Streichert, M. Glaß, N. Navet, and J. Teich. Formal analysis of the startup delay of some/ip service discovery. In *Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 49–54, 2015.

[16] S. M. Sundharam, S. Altmeyer, L. Havet, and N. Navet. A model-based development environment for rapid-prototyping of latency-sensitive control software. In *Proc. 2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, Patna, India, December 2016.

[17] S. M. Sundharam, S. Altmeyer, and N. Navet. Model interpretation for an AUTOSAR compliant engine control function. In *Proc. 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Toulouse, France, July 2016.

[18] S. M. Sundharam, S. Altmeyer, and N. Navet. Poster abstract: An optimizing framework for real-time scheduling. In *Proc. 2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[19] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. Mbeddr: An extensible C-based programming language and ide for embedded systems. In *Proc. 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140, 2012.