Lean Model-Driven Development through Model-Interpretation: the CPAL design flow

Nicolas Navet, University of Luxembourg
Loïc Fejoz, RealTime-at-Work, France
Lionel Havet, RealTime-at-Work, France
Sebastian Altmeyer, University of Luxembourg

Abstract: We introduce a novel Model-Driven Development (MDD) flow which aims at more simplicity, more intuitive programming, quicker turnaround time and real-time predictability by leveraging the use of model-interpretation and providing the language abstractions needed to argue about the timing correctness on a high-level. The MDD flow is built around a language called Cyber-Physical Action Language (CPAL). CPAL serves to describe both the functional behaviour of activities (i.e., the code of the function itself) 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 meant to support two use-cases. Firstly, CPAL is a *development and design space exploration* environment for CPS with main features being the formal description, the editing, graphical representation and simulation of CPS models. Secondly, CPAL is a real-time execution platform. The vision behind CPAL is that a model is executed and verified in simulation mode on a workstation and the same model can be later run on an embedded board with a timing-equivalent run-time time behaviour.

Keywords: Model-based design, programming language, model interpretation, design space exploration, simulation, timing predictability.

1 Model Driven Development with model as the code

Existing commercial MBD flows such as Matlab/Simulink® and Scade® successfully capture most of the aspects of model based design: requirements traceability, model design, simulation, code generation, test-cases generation, etc. Even though they are very powerful and successfully used in a wide range of industrial applications, these design flows do not cover all existing needs, be it only because they are complex and expensive.

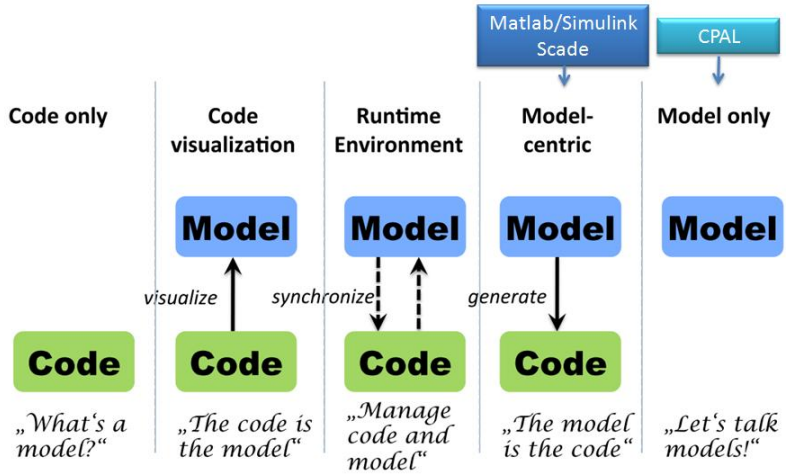


Figure 1: Spectrum of model-based design approaches (core of the figure from [Br04] and [Tr09]).

CPAL has been initially inspired by the success of three interpretation-based runtime environments, successfully certified at the highest criticality levels and deployed at large scale in railway interlocking systems over the last 20 years at SNCF (see [An12]) and RATP in France, and in UK and other countries through the Westlock interlocking system from Westingshouse. These technologies have proven to be technically successful in the sense that 100s of millions of people rely on them on a daily basis. They are however undisclosed proprietary technologies, specific to interlocking systems and have not been designed to meet the needs of most of today's and tomorrow's applications (Cyber-Physical Systems at large) and execution platforms (SOC, multicore, manycore).

Except for these applications and some implementations in industrial automata (PLCs), surprisingly, *model interpretation* has to the best of our knowledge not been widely explored yet for developing critical systems (see [Hu03] for one of the few notable works in that direction), albeit it possesses a number of key advantages:

- **Simplicity** for the end-user and quicker turnaround time: once designed and simulated the model can be uploaded to the target (e.g., by drag & drop).
- **Verifiability**: there is no discrepancy between the model and the code, and there are no software layers causing deviations from the expected temporal behavior at run-time.
- **Less error prone software**: because the total software size is greatly reduced both off-line and on the target (no operating systems, no compiler, no linker, etc). In particular, the interpreter is a thin software layer of a few thousand lines that can be more easily tested and verified than software made up of hundreds of thousands lines of code or more. In addition, as argued in [An12], the logic of the application is easier to verify since it is fully decoupled from the runtime services and written in a high-level language.
- **Cost-efficiency**: made possible thanks to the simplicity of the design flow and run-time environment.
- **Hardware-independence**: thanks to the ability of the interpretation layer to hide the complexity of the hardware platform from the programmer. A higher level of abstraction is important in light of the ongoing trend towards multi/manycore platforms and more heterogeneous execution platforms (SOC).

CPAL supports two types of model interpretation: the direct interpretation of the design models on an interpretation engine running on top of the hardware, called "bare-machine model interpretation" (BMMI), and the interpretation on top of an OS. The latter is less predictable from a timing point of view but more convenient for development and experimentations. In addition, the interpreter can re-use the interfaces to the I/O provided by the OS. Whatever the type of interpretation, there is a slowdown due to model interpretation. However, we believe that for a significant share of embedded systems, the simplified and accelerated model development (reduced time-to-market) will outweigh the overhead due to model interpretation on the target architecture. In addition, dedicated hardware support in FPGA or ASIC may offset part of the performance loss.

CPAL and associated tools are jointly developed by our research group at the University of Luxembourg and the company RTaW since 2011. The CPAL documentation, graphical editor and execution engine for Windows, Linux, embedded Linux and RaspberryPI are freely available for all uses at http://www.designcps.com. A BMMI port of CPAL is available for Freescale FRDM-K64F boards. A commercial version for embedded targets will be introduced progressively.

2 CPAL: providing high-level abstractions for embedded systems

The main requirement when designing CPAL was to natively provide the high-level abstractions familiar in the domain of embedded systems needed to express in an unambiguous and concise manner domain specific patterns of functional behaviors as well as non-functional properties. The concept of *process* is the core language entity to implement a recurrent activity having its own dynamic. A process is automatically activated at a specified rate, with the optional requirement that a specific logical condition is fulfilled to execute (this is called *guarded executions*). CPAL processes are classically referred to as tasks, runnables or threads in other contexts.

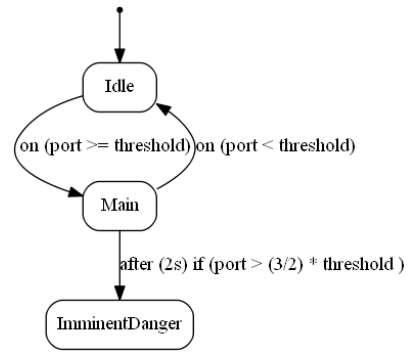
CPAL provides the programmers with high-level abstractions well suited for the domain of CPS such as

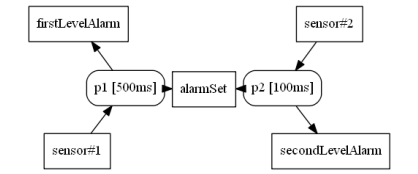
- *Real-time scheduling mechanisms*: processes are activated with a user-defined period, possibly with offset relationship with each other and additional execution conditions such as the occurrence of some external events.
- *Finite State Machines* (FSM): the logic of a process can be defined as a Finite State Machine (FSM) based on Mode-Automata [Ma03] where code can be executed in the states, or upon the firing of transitions. The semantics that is implemented in CPAL is to first execute a transition if possible and then execute the current state's code which enables the control program to react faster on external events,
- *Communication channels* to support data flow exchanges between processes, and reading/writing to hardware ports. The semantics attached to a channel can be chosen to be FIFO or LIFO buffering, or data overwriting,
- *Introspection mechanisms* that enable processes to query at run-time their execution characteristics such as their activation rate and activation jitters. This feature is typically used to implement

control algorithms that must adapt to their frequency of execution or their execution jitters by compensating for them.

The abstract and concrete syntax of CPAL has been inspired by a number of diverse languages such as Eiffel, MISRA C and Erlang, model-based design products such as Matlab/Simulink® and Scade®, verification frameworks such as Promela/Spin and more generally what is usually referred to as the synchronous programming approach, such as Giotto [He03]. However, CPAL has been designed with the requirement to remain a small, simple and unambiguous language, easy to start with for the C or Java programmer, and less demanding than the synchronous programming models.

The example CPAL program below defines a monitoring process which, when a threshold on the measured quantity is exceeded, signals an abnormal behavior and, after a certain time above another threshold, sets an alarm. When this happens, another process starts then being executed at a higher rate to confirm with measurements from another sensor the alarm condition. This can then be used by a supervision process to take the appropriate measures (e.g., error recovery or error mitigation).






```
processdef MonitorProc(in uint8: port, out bool: abnormal, out bool: danger)
{
  const uint8: threshold = 30;
  state Idle{ /* ... */ }
  on (port >= threshold) to Main;

  state Main {
    abnormal = (port > threshold);
    /* ... */
  }
  after (2s) if (port > (3/2) * threshold ) {
    danger = true;
  } to ImminentDanger;
  on (port < threshold) to Idle;

  state ImminentDanger {/* ... */}
}

var uint8: sensor#1; /* Mapped to some I/O port and updated */
var uint8: sensor#2; /* upon activation of the processes */
var bool: alarmSet = false;
var bool: firstLevelAlarm = false;
var bool: secondLevelAlarm = false;

/* Instantiation of periodic monitoring processes*/
process MonitorProc: p1[500ms](sensor#1, alarmSet, firstLevelAlarm);
/* Guarded execution: second process is only executed when firstLevelAlarm is true */
process MonitorProc: p2[100ms][firstLevelAlarm](sensor#2, alarmSet, secondLevelAlarm);
```

Figure 3: Example CPAL program illustrating the concepts of input and output ports, native support for FSM, conditional and timed transitions and periodic process activation (with and without guard). The top-left graphic is the representation of the FSM embedded in the monitoring process, while the bottom-left graphic is the functional architecture with the flows of data, as both seen in the CPAL-Editor.

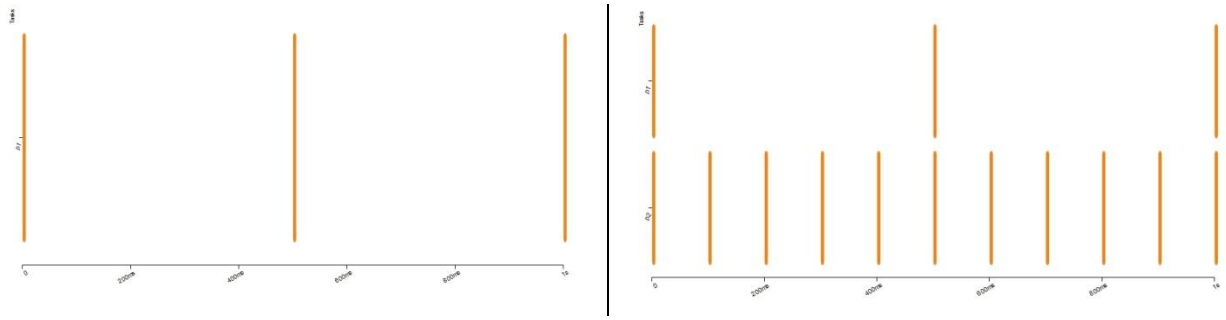


Figure 4: Gantt diagram of the activation of the processes as seen during execution. On the left, a single process is executed while, on the left, the second and more frequent process is being executed too because an alarm condition was signaled by the first one (screenshots from the CPAL-Editor).

3 CPAL to model, validate and execute embedded systems

3.1 CPAL use-cases

CPAL supports several use-cases discussed below.

3.1.1 High-level programming language for network simulation environments

CPAL can serve to describe the functional behavior of applications and high-level protocol layers. A CPAL model is for instance used in [Se15] to simulate the SOME/IP Service Discovery protocol in a Daimler Car's prototype network. Another CPAL program, available in [Fe15], implements the transmission of a video stream with segmented messages on an Ethernet network. The model hands over the frames once created to the simulation kernel of RTaW-Pegase, a communication architecture performance analysis tool from RTaW. Interestingly, the same CPAL simulation model can be executed with no changes on an embedded target or a workstation to experiment on a testbed later in the design process.

3.1.2 Modeling and simulation language for Design Space Exploration

CPAL is meant to support the formal description, the editing, graphical representation and simulation of cyber-physical systems. It can be used in its own development environment, like done for the FMTV Challenge [Al15], or within Matlab/Simulink to implement the controller, as done for the landing gear case-study [Bo14,Na14]. The simulation models can be executed in real-time (i.e., activation periods are respected) or as fast as possible in simulation mode. Simulation mode CPAL interpreters are available on Windows and Linux. This case-study is illustrated on the development of a smart parachute for UAVs in [Ci16] and further discussed in §3.2.

3.1.3 Real-time execution engine and overheads data

The intention of CPAL is to provide not only a modeling language, but also an interpreter which ensures equivalence between the simulated behavior of the model and the behavior on the execution platform. There are CPAL interpreters in real-time mode available for embedded Linux, Raspberry Pi and a BMMI port for the Freescale FRDM-K64F board which is based on a CortexM4 processor. On this latter inexpensive platform at 120MGhz (floating point enabled), the overheads we measured with a logic analyzer, or we calculated based on the code, are the following:

- the maximum activation jitter for periodic activation is 40us,
- the timer interrupts which occur periodically during the execution of processes takes less than 70 cycles, that is less than 0.6us,
- the time to decide the next process to execute and create future instances is 200 cycles + n * 560 cycles, that is 1.6us + n * 4.6us,
- in-between process overhead is 2us maximum.

CPAL models are interpreted at run-time which involves a significant performance loss with respect to compiled code, typically a slowdown factor larger than 3. For CPS requiring maximal performances, code generation from CPAL or hooks to call native object code from CPAL processes would be feasible options. This latter technique seems promising to us since it enables to keep most of the additional control and monitoring capabilities of interpretation while allowing the re-use of legacy code and a close to compiled-code execution speed.

3.1.4 CPAL for learning and teaching

CPAL has been used for teaching since 2012 at our University at the 3rd year Bachelor level. CPAL is used to teach model-based design (MBD) for embedded systems with practicals such as programming a capsule coffee machine, a simplified programmable floor robot and elevator control system, etc. Our experience has been positive in terms of how fast students have been able to work autonomously on the development of the system. Indeed, most students are to master the language within a few hours. In addition to the simplicity of the language, the free availability of the tools, the on-line examples and the CPAL-Playground facilitate the learning process. Improvements ahead of us include a better tool support for 1) methodological processes such the ability to link design artifacts with requirements and 2) verification tools that exist at a prototypical stage (WCET and response time analyzers, state-space exploration).

3.2 Solving the FMTV challenge 2015

The Formal Methods for Timing Verification (FMTV) Challenge 2015 is a schedulability analysis problem proposed by Thales that challenges current techniques and tools from the timing verification community. The challenge is built around an aerial video tracking system and consists of 4 sub-challenges. A number of solutions using different formalisms were presented at the WATERS Workshop 2015 (see [Wa15]). To solve the sub-challenges, we proposed solutions relying on CPAL for the modeling and the simulation along with manual schedulability analysis. The reader is referred to [Al15] for a more comprehensive description.

Our key takeaways from the FMTV2015 challenge are the following:

- The modeling efforts were limited and the complete models were written in CPAL within less than 3 hours, which was significantly faster than the development of the automata based models.
- The CPAL model, along with the associated graphical representations, reveals ambiguities in the description and thus forces the system designer to consider each important aspect of the modeled system.
- The simulation capacity of CPAL allows to explore the timing behavior at design-time and to validate or disprove assumptions about it. For instance, deriving the solution for the first challenge by hand proved to be error-prone, and the use of simulation was helpful to better understand the dynamics of the system, and specifically check whether the worst-case conditions we devised could actually happen.
- If worst-case behaviors are looked for by simulation, simulation should be biased to explore parts of the search space we know are likely to contain such behaviors. For instance, in order to increase the likelihood to meet unfavorable scheduling scenarios, we used a random number generator that gave higher probability to the bounds of the interval, instead of a uniform distribution. This simple strategy was effective in creating situations leading to the maximum interferences in our experiments on the first challenge. This is however clearly an open research problem.
- With the help of a simple utility it is possible to extract from the CPAL model the characteristics of the tasks and automate the schedulability analysis. However, we were unable and do not see how to answer in an automated manner complex questions like asked in sub-challenges 1A and 1B without resorting to ad-hoc analyses. Identifying the scope of what can be fully, or partially, automated is in our view a question that deserves future work.

```
struct Frame {
  uint32: id;
  uint32: emission_time;
};

processdef T1_PreProcessor(
  in channel<Frame>: input,
  out channel<Frame>: output)
{
  state Main {
    /* removes reflections
    normalizes intensity, etc.
    */
    assert(input.notEmpty());
    output.push(input.pop());
  }
}

processdef T2_Processor(...) { ... }
processdef T3_Filter(...) { ... }
processdef T4_DAConvertor(...) { ... }
processdef Camera(...) { ... }

var queue<Frame>: cam_to_t1[1];
var queue<Frame>: t1_to_t2[1];
var Frame: t2_to_t3;
var queue<Frame>: t3_to_t4[n];
var queue<Frame>: t4_to_monitor[1];
```

```

process Camera:
    camera[40ms](cam_to_t1);
@cpal:time {
  var uint32: drift = uint32.rand_uniform(999900, 1000100);
  camera.period = (40 * drift)ns;
}

process T1_PreProcessor:
    t1[cam_to_t1.notEmpty()](cam_to_t1, t1_to_t2);
@cpal:time {
    t1.execution_time = 28ms;
    /* assert(t1.bcet == t1.wcet and
              t1.wcet == t1.execution_time);*/
}

process T2_Processor:
    t2[t1_to_t2.notEmpty()](t1_to_t2, t2_to_t3);
@cpal:time {
    t2.bcet = 17ms;
    t2.wcet = 19ms;
}

```

Figure 5: Excerpt of the CPAL Code for Thales FMTV Challenge 1. Process activation conditions are specified at the definition of the processes (e.g., *t1* is activated upon the arrival of a frame from the camera). The annotations in the comments are used for the simulation and the analysis of the model. The complete code is available at http://www.designcps.com/wp-content/uploads/fmtv15.zip.

Verification outside schedulability analysis (see §4.3) in CPAL currently relies on simulation. The complete language is not amenable to formal verification by model-checking or theorem proving. Although simulation does not offer exhaustive evaluation and hence, does not equate to formal verification, it is applicable on systems of any size. Simulation of CPAL code is timing accurate through the use of timing annotations (see line 37 in Figure 5 for instance), which can be derived by measurements on target architecture or WCET analyses. We believe also that heuristics, such as biasing random number generation towards the bounds of the interval as we experimented in [Al15], and search-intensive algorithms (see [DA14]) are promising techniques to efficiently direct the simulation towards unfavorable trajectories of the system. Although this remains to be demonstrated, such worst-case oriented simulation may be a practical alternative to formal verification for some systems, especially early in the design phases.

4 Scheduling and timing correctness

4.1 Timing predictability in CPAL

The correctness of a cyber-physical system usually does not only depend on its functional behavior, but also on its timing behavior. Similarly to functional determinism, i.e., the same input always leads to the same output, we may want systems where events occurs at pre-determined points in time. This notion of *time determinism*, at the heart of the synchronous approaches, is for instance discussed and advocated in [He08].

Modern architectures with history-sensitive components such as caches and buffers, however, lead to significant variations of execution times and are increasingly complex to analyze. Despite the determinism of all individual hardware components, the complex interplay thereof appears non-deterministic if it cannot be fully comprehended. In addition, changing environmental conditions, such as temperature or EMI, will affect the functioning of the system. For instance, significant clock drifts are caused by varying temperatures [Mo11]. Complete time-deterministic systems as defined in [He08] are thus hard to achieve.

With respect to the synchronous programming models, CPAL implements a weaker version of time-determinism, still providing a form of timing-predictability sufficient in many applications while remaining closer to mainstay software development practices. Our experience is indeed that timing-correctness most often does not necessitate time-determinism. For most systems, it is sufficient if the timing of events respects a set of constraints specific to the needs of the cyber-physical system, thus allowing a substantial degree of freedom. For instance, a system may has to react to an input within a given time bound, the order of some events may be essential, or a computation may has to be repeated periodically with limited jitter. Several, distinct systems can exhibit distinct timing behaviors, which are all considered correct, and furthermore, systems can show substantial timing variations at run-time and still be considered correct. In any case, a time-deterministic system is not a necessity for timing correctness for all systems.

Instead of a fully time-deterministic system, the execution framework enforces a fixed and deterministic event ordering irrespective of the execution platform. The exact timing of an event may be subject to variations that can be evaluated by a schedulability analysis, but the order in which observable events, such as process invocation or process termination, happen shall be statically defined. We refer to this property as *event-order determinism*. This allows the CPAL program to be developed in simulation mode on a workstation and to be later run on an embedded board with an equally acceptable timing behavior. More fine-grained timing constraints such as deadline constraints can be verified with the help of schedulability analysis (see $4.3).

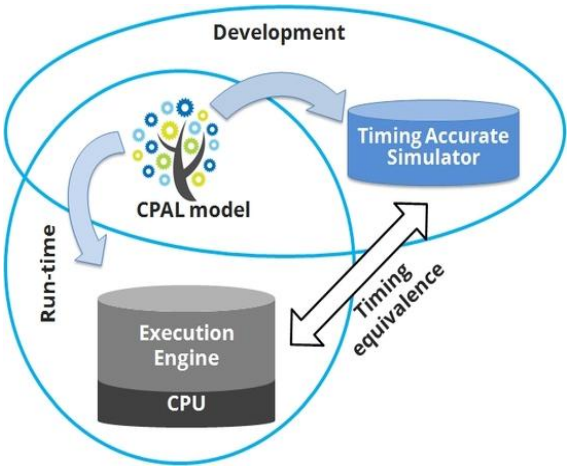


Figure 5: Event-order determinism of the task scheduling ensured by CPAL is a main dimension of timing equivalence between design time and run-time. Optionally, it is also possible to annotate a CPAL model with execution time information, obtained by measurements on target or WCET analysis, so as to achieve timing-accurate simulation of the model.

4.2 Scheduling model

If we solely concentrate on implementing a system's timing correctness, we can usually select from several scheduling policies and execution models. Among the various scheduling algorithm, we selected FIFO scheduling which is a predictable and lightweight policy particularly suited to our needs. Indeed, FIFO schedules processes non-preemptively and ensures *event-order determinism*, i.e., the order of process executions is defined statically and immutable. With this choice, we favor predictability of the run-time and simplicity of the execution engine over an optimized use of the computational resources.

In FIFO scheduling, processes are released strictly period and are executed in order of process release. Processes can be assigned priorities that serve as tie breakers in case of simultaneous process releases. Nevertheless FIFO scheduling is – in stark contrast to static-cyclic scheduling – a work-conserving scheduling policy which means that no CPU time is wasted. A system-wide clock is required to trigger process activation and to ensure determinism. All process release times are thus subject to the very same clock drifts, enforcing the unique execution order, but also restricting the system to uni-core processors or partioned multicore scheduling.

FIFO scheduling is well known to perform worse regarding schedulability than priority-driven dynamic scheduling policies such as rate-monotonic or earliest deadline first. For purely periodic systems like in CPAL, execution offsets [Mo12] can however be chosen so as to distribute the workload evenly over time, which significantly improves the ability of FIFO to meet real-time constraints [Alt15b]. Although FIFO fits well CPAL, the scheduling model of CPAL is not restricted to FIFO and can be extended to other policies such as Fixed Priority Preemptive Scheduling.

4.3 Schedulability analysis and scheduling synthesis

The CPAL execution engine possesses mechanisms to monitor and record at run-time the execution time of the processes. This feature can be taken advantage of by the designer to estimate the Worst-Case Execution Times of the processes making up the application. Since the workload submitted to the runtime environment is statically defined and fully characterized, it is possible to derive a schedulability analysis for a set of CPAL processes. We developed in [Alt15b] two schedulability analysis: an exact test based on simulation and an approximate test based on the schedulability test for non-preemptive scheduling with offsets [Pe05]. A feasibility test via simulation requires simulation up to twice the hyperperiod, which may be infeasible in many situations. In the latter case, we have to resort to the approximate schedulability test.

We believe that significant progresses in terms of development time and correctness can be achieved by further automating the design process. In the timing dimension, this can for instance be done by synthesizing a feasible scheduling solution with this two-steps approach developed in [Al15c]:

- the developer states the permissible timing behavior of the system using a declarative language for the specification of non-functional timing properties,
- a system synthesis step involving both analysis and optimization (e.g., periods and offsets) then generates a scheduling solution which at run-time is enforced by the execution environment.

The interested reader can refer to [Al15c] for a more comprehensive discussion on scheduling synthesis in the CPAL framework.

5 Related work

A motivation behind CPAL is that general purpose programming languages abstract away timing considerations, and non-functional properties at large. They also lack the domain-specific constructs that are needed to speed-up the development, facilitate the re-use and the understanding of real-time embedded software.

Synchronous programming models, be they functional like Lustre [Ha91] and Signal [Be91] or imperative like Esterel [Bo14b], propose an effective and sound answer to facilitate the correct design of reactive systems made up of concurrent tasks. Thanks to their formal semantics, they have brought major progresses to the development of safe embedded systems over the last 30 years, and are certainly well suited in some application domains such as safety-critical systems. However, the learning curve is steep for programmers used to more conventional programming languages. In addition, the complexity of the formalisms that need to be understood or manipulated is a hindrance to their adoption by the practitioner. Also the programming style and the abstractions offered by the languages do not fit all problems and programmers. In many cases, we also believe that more lightweight and less demanding programming models are equally able to guarantee the necessary timing predictability without over-constraining the design and development.

Amongst the synchronous approaches, CPAL has been inspired by Giotto [He03] which is a time-triggered architecture language. Indeed, several mechanisms available in Giotto are re-used such the task activation models (e.g., periodic process, guarded executions). In that respect, the current task activation model of CPAL can be seen as purely time-triggered. However, on the contrary of CPAL, neither Giotto defines a runtime environment nor is it a programming language to express functional behavior. In addition, although we could imagine implementing an execution semantics in CPAL compliant with Giotto, CPAL currently relaxes the programming model of Giotto in several ways:

- No system-wide mode change mechanisms as in Giotto are defined in CPAL, which supports mode changes through guarded executions at the process-level.
- Although the order of observable events is deterministic in CPAL, the outputs of a process are not produced at a predetermined point in time. They may be several output times that may be subject to variations depending on the actual execution times of the processes, but the interval where they happen can be bounded by schedulability analysis.
- Unlike in Giotto, input ports of a process are read at the actual start of the process execution and not upon (or before) its release time. The process thus works on the most recent data.
- In CPAL it is possible to explicitly re-read an I/O-mapped variable or to perform several writings to an output port during the execution of a process (e.g., to drive serial communication by bit-banging, send segmented messages, etc). This is done though the dedicated `syncIO()` function.

A more recently proposed architecture description language is Prelude [Fo09,Fo10] which extends synchronous approaches, such as Lustre, to facilitate the development of multi-rate applications with complex communication patterns between tasks. Prelude builds on the formal synchronous model to offer powerful operators (e.g., over and under-sampling) to define the flows of data between functions potentially operating at different rates. Prelude is able to perform correctness checks that ensure that the program has a deterministic semantics. Then, the Prelude compiler translates the program into a set of communicating real-time tasks scheduled in such a way as to meet the timing constraints. Like Giotto, Prelude is not a programming language to define the actual functional behavior of the tasks, neither is it an execution platform. Preliminary experiments on examples such as the flight application software in [Fo10] suggest that most of the semantics of Prelude programs can be captured in CPAL. Future work will be devoted to assess the feasibility of transforming CPAL programs into Prelude programs in order to take advantage of the data-flow verification framework readily available within Prelude.

6 Conclusion

The CPAL programming language and associated toolset is a model-driven development flow aimed at the development of timing-predictable embedded systems. Our priority in the language design has been to favor simplicity, user-friendliness and expressive power both in the functional and non-functional dimensions. In particular, CPAL provides language abstractions needed to define real-time applications and argue about the timing correctness on a high-level.

In that way, CPAL is a contribution towards addressing what Thomas Henziger called the grand challenge in embedded software design [He08]: "offering high-level programming model that exposes the execution properties of a system in a way that permit the programmer to express desired reaction and execution requirements, permits the compiler and run-time systems to ensure that these requirements are satisfied". CPAL provides a programming model, easier to handle for most programmers than synchronous approaches, which aims at ensuring timing-predictability instead of time-determinism which is over-constraining in many real-time applications.

CPAL has been already successfully used to answer several industrial problems (Al15, Ci16, Se15), as well as to teach MDD. Upcoming releases of the development environment and the CPAL interpretation engine will gradually offer an integrated support for off-line and on-line verification activity.

7 References

[Al15] S. Altmeyer, N. Navet, L. Fejoz, “Using CPAL to model and validate the timing behaviour of embedded systems”, WATERS Workshop, July 2015. Available at http://hdl.handle.net/10993/21250.

[Al15b] S. Altmeyer, N. Navet, “The case for FIFO scheduling”, technical report from the University of Luxembourg, to appear, November 2015.

[Al15c] S. Altmeyer, N. Navet, “Towards a declarative modeling and execution framework for real-time systems”, submitted to the First IEEE Workshop on Declarative Programming for Real-Time and Cyber-Physical Systems, in conjunction with IEEE RTSS, December 1, 2015.

[An12] M. Antoni, “Formal validation method and tools for computerized interlocking system”, 18th International Symposium on Formal Methods (FM 2012), Industry day, August 27-31, 2012. Slides available at http://fm2012.cnam.fr/fm2012/ID2012-Marc-Antoni.pdf.

[Be91] A. Benveniste, P. Le Guernic, C. Jacquemot » Synchronous programming with events and relations: the Signal language and its semantics”, Science of Computer Programming, 16(2), 1991.

[Bo14] F. Boniol, V. Wiels, “The landing gear system case study”, pp1-18, Proc. ABZ 2014, 2014.

[Bo14b] F. Boussinot, R. de Simone, “The Esterel language”, Proc. IEEE, 79(9), 1991.

[Br04] A. Brown, “An Introduction to Model Driven Architecture – Part1: MDA and today’s systems”, IBM technical library, 2004. Available at http://www.ibm.com/developerworks/rational/library/3100.html.

[Ci16] L. Ciarletta, L. Fejoz, A. Guenard, N. Navet, “Development of a safe CPS component: the hybrid parachute, a remote termination add-on improving safety of UAS”, to appear at ERTSS2016, Toulouse, January 2016.

[DA14] S. Di Alesio, S. Nejati, L. Briand, A. Gotlieb, “Worst-Case Scheduling of Software Tasks - a Constraint Optimization Model to Support Performance Testing”, 20th International Conference on Principles and Practice of Constraint Programming, 2014.

[Fe15] L. Fejoz, N. Navet, “The CPAL Programming Language – an Introduction”, available at http://www.designcps.com, 2015.

[Fo09] J. Forget, “Un Langage Synchrone pour les Systèmes Embarqués Critiques Soumis à des Contraintes Temps Réel Multiples”, Phd Thesis in Computer Science from the University of Toulouse, 2009.

[Fo10] J. Forget, F. Boniol, D. Lesens, C. Pagetti, “A real-time architecture design language for multi-rate embedded control systems”, 2010 ACM Symposium on Applied Computing (SAC '10), pp527-534, 2010.

[Ha91] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, “The synchronous data-flow programming language LUSTRE”, Proc. IEEE, 79(9), 1991.

[He03] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: A time-triggered language for embedded programming”, in Proceedings of the IEEE, vol. 91, n°1, pp 84–99, 2003.

[He08] T. A. Henzinger, “Two challenges in embedded systems design: predictability and robustness”, Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 366(1881):3727–3736, 2008.

[Hu03] J. Huang, J. Voeten, A. Ventevogel and L. Van Bokhoven, “Platform-independent Design for Embedded Real-Time Systems”, in Proceedings of FDL'03, pp. 318-329, 2003.

[Ma03] F. Maraninchi and Y. Rémond, “Mode-Automata: a new Domain-Specific Construct for the Development of Safe Critical Systems”, Science of Computer Programming, n°46, pp, 219-254, 2003.

[Mo11] A. Monot, N. Navet, and B. Bavoux, “Impact of clock drifts on CAN frame response time distributions”, in 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011), 2011.

[Mo12] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, “Multisource software on multicore automotive ECUs combining runnable sequencing with task scheduling”, IEEE Transactions on Industrial Electronics, 59(10):3934–3942, Oct 2012.

[Na14] N. Navet, “Lean Model-Based Design for Cyber-Physical Systems – the case for Model Interpretation”, invited talk at ABB Corporate Research, Basel, Mai 14, 2014.

[Pe05] R. Pellizzoni, G. Lipari, “Feasibility analysis of real-time periodic tasks with offsets”, Real-Time Systems, 30(1-2):105–128, May 2005.

[Se15] J. Seyler, T. Streichert, M. Glaß, N. Navet, J. Teich, “Formal Analysis of the Startup Delay of SOME/IP Service Discovery”, DATE 2015, Grenoble, France, March 9-13, 2015.

[Tr09] T. Trew, “Creating Embedded Platforms with MDA: Where's the Sweet Spot”, slides presented at ECMDA-FA, 2009.

[Wa15] Community Forum on Tools and Benchmarks for Real-Time Systems, http://ecrts.eit.uni-kl.de/forum, 2015.

