UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Towards a declarative modeling and execution framework for real-time systems

Sebastian Altmeyer, Nicolas Navet

Angaben zur Veröffentlichung / Publication details:

Altmeyer, Sebastian, and Nicolas Navet. 2016. "Towards a declarative modeling and execution framework for real-time systems." *ACM SIGBED Review* 13 (2): 30–33. https://doi.org/10.1145/2930957.2930962.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Towards a declarative modeling and execution framework for real-time systems

Sebastian Altmeyer
University of Luxembourg
Luxembourg
sebastian.altmeyer@uni.lu

Nicolas Navet
University of Luxembourg
Luxembourg
nicolas.navet@uni.lu

ABSTRACT

Our work is a contribution towards addressing what Thomas Henziger called the grand challenge in embedded software design [5]: "*offering high-level programming models that exposes the execution properties of a system in a way that permits the programmer to express desired reaction and execution requirements, permits the compiler and run-time systems to ensure that these requirements are satisfied*". In the programming model we describe here, the developer states the permissible timing behavior of the system, a system synthesis step involving both analysis and optimization generates a scheduling solution which at run-time is enforced by the execution environment. With respect to the synchronous programming models, our approach 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. This approach is currently being implemented and experimented in the CPAL language development tools and associated runtime environment.

1. INTRODUCTION

Current design practices in real-time embedded and cyber-physical systems (CPS) usually treat real-time behavior as a mere by-product of the functional implementation. For instance, in many Model-Driven Development (MDD) flows the computational resources available are even considered to be infinite and the quality-of-service that can be offered by the execution platform is just not considered. When timing verification is a matter of concern, typically the complete code is developed and bounds on the task execution times are derived by static or dynamic timing analysis later on. A scheduling analysis then checks whether all tasks can be executed on the system so that the timing requirement are met. This design process is suboptimal: it may result in time-consuming and costly modifications late in the development process, and spare computational resources are not taken advantage of in domains where most of the innovation stems from software.

We advocate an alternative design process and propose to treat the timing as a first-class citizen: the desired timing behavior is specified in a declarative fashion – already at high-level and alongside the functional behavior. At design time, timing bounds on the functional components will be derived and optimized, and the execution framework then enforces the correct timing behavior at run-time. The implementation details, i.e., how exactly the timing behavior is realized at run-time is transparent to the system developer. This is in stark contrast to traditional design and execution environments where precise knowledge about the timing and execution model is required, and often the timing implementation must be provided by the system designer. Our approach is inspired by the synchronous programming models, such as Lustre [4] and Giotto [6], but provides a weaker version of time-determinism sufficient in many applications, while offering a programming model closer to mainstay software development practices.

First, we detail our approach towards a declarative development of real-time systems, then present the modeling and execution framework that implements our approach.

2. TIMING DETERMINISM AND TIMING CORRECTNESS

The correctness of a cyber-physical system is not only defined by its functional behavior, but also by its timing behavior. Consequently, deterministic timing behavior, called *time determinism* in [5], is desired. Similarly to functional determinism, i.e., the same input always leads to the same output, we desire systems, where the timing of events is pre-determined and well-defined. 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 components of modern processors, 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 [9]. Completely time-deterministic systems as defined in [5] are thus hard to achieve.

On a side note, the demand for determinism cannot be easily transferred from the functional to the temporal domain for the following reasons: The functional behavior is independent of the target system (modulo implementation errors and hardware failures), whereas the timing behavior depends on the precise target architecture; and small changes to a system can have a tremendous impact on the timing behavior [14]. Similarly, the question of *what can be computed?* is well studied and can be answered using theoretical abstractions such as Turing machines, whereas the question *will the system react within xy ms?* requires knowledge about the run-time environment.

A system's *timing correctness* is usually not nearly as strictly defined as time-determinism. For most systems, it is sufficient if the

Copyright retained by the authors. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published at
https://doi.org/10.1145/2930957.2930962

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. Several, distinct time-deterministic 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 the timing correctness in general.

3. DECLARING TIMING CORRECTNESS

As detailed in the previous section, timing correctness does not necessarily entail a fully time-deterministic system, but it requires the fulfillment of a set of temporal constraints. Our aim is to provide a modeling framework, where the developer only needs to specify these constraints that determine the timing correctness. The developer is thus exempt from the burden of specifying how the timing correctness will be realized.

To this end, we provide a modeling and execution environment that treats the timing behavior at design time in a declarative fashion, and determines a feasible schedule – fully transparent to the user – that implements the timing correctness.

The main entities of our modeling framework are processes, which implement some functional behavior. The timing correctness of the system can be declared using the following four types of constraints:

Execution frequency: Each process must be assigned an execution frequency, or execution period. This value denotes the time between two successive releases of the process. The period can be defined as a range instead of a single value. Especially control applications often do not require exact periods, but are valid for a larger range of periods, where the highest rate of execution usually leads to a better control.
Example: process τ_a executes every $[x : y]$ seconds.

Conditional execution: This activation scheme serves to implement functions that have to interact with the system environment or to enable different functioning modes.
Example: process τ_a executes (i) if its period has elapsed and (ii) if condition C evaluates to true.

Relative deadlines: The relative deadline denotes the relative time after process invocation until the process has to finish. In contrast to the period, the deadline is a single numeric value. A range of values would be futile, as the run-time environment must ensure the system is feasible with respect to the most stringent deadline. Unless specified explicitly, a process's deadline is equal to its minimal period.
Example: process τ_a must complete within y seconds.

Temporal dependencies: A temporal dependency can be required to ensure a chronological order of processes. On top of logical ordering, a minimal and maximal offset between a process is finished and another one is started can be specified. When no temporal dependencies are specified, the execution order is irrelevant. This means that the execution framework is free to select the order of process executions.
Example: process τ_a must execute after process τ_b has finished.

The complete timing correctness is to be specified using these four types of constraints. This list may not be exhaustive, and will be enriched based on the requirements from case-studies and examples that are being developed in the context of this project (see 7).

It should be pointed out that the scheduling specific parameters, such as priorities in case of fixed-priority scheduling or time slots in case of TDMA or Round-Robin, or offsets are not required. In fact, even the scheduling policy of the system does not necessarily need to be communicated to the system designer.

4. EXECUTION ENVIRONMENT

A strong argument in favor of time-determinism remains debuggability and repeatability. Providing these features to the developer strongly improves the usability of the execution environment. Instead of a fully time-deterministic system, our execution framework shall enforce a fixed and deterministic event order. 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*.

If we solely concentrate on implementing a system's timing correctness, we can select from a large variety of scheduling and execution models. Among the various scheduling algorithm, we have selected FIFO scheduling which is a predictable and lightweight policy particularly suited to our needs: FIFO schedules processes non-preemptively and exhibits event-order determinism, i.e., the order of process executions is defined statically and immutable. With this choice, we favor predictability and simplicity over an optimized use of the computational resources. Considering other scheduling policies will be future work.

In FIFO scheduling [8], processes are released strictly period and are executed in order of process release. To this end, the scheduler maintains a FIFO queue with ready processes waiting for dispatch. 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 and is resilient to overload conditions with respect to event-order determinism. 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 partitioned multicore scheduling.

FIFO scheduling is known to perform worse than priority-driven dynamic scheduling policies, such as rate-monotonic or earliest deadline first, and is usually considered unfit for real-time systems [7]. We share this opinion in case of sporadic process release times. The critical instance is given when all processes other than process P have just been added to the FIFO queue prior to the release of P. Hence, the system is only schedulable under FIFO scheduling, if the sum of all process execution times is less than the deadline of each process. In a fully time-triggered system with a global clock, offsets can be used to distribute the workload as done in [10]. This offset optimization has great potential to alleviate the performance issues of FIFO scheduling.

The scheduling model is formally defined as follows: We assume a system composed of n processes $\{\tau_1, \ldots \tau_n\}$ running on a single processor. Each process τ_i is represented by a tuple

$$\tau_i : (O_i, C_i, T_i, D_i),$$

where O_i is the process's release offset, C_i the worst-case execution demand, T_i the process's period and D_i the deadline. A process produces an infinite sequence of jobs $\tau_{i,j}$ with $j \in \mathbb{N}$. The job

release time $r_{i,j}$ of job $\tau_{i,j}$ is given by

$$r_{i,j} = O_i + jT_i \tag{1}$$

and its absolute deadline by

$$d_{i,j} = O_i + jT_i + D_i. \tag{2}$$

The computation of the execution times is out of scope of this paper. We assume that a timing analysis, either static or dynamic, provides safe bounds on the execution times of the processes.

5. SCHEDULER SYNTHESIS

The timing correctness is defined based on a set of constraints that just cover the essential minimum of what constitutes a temporally correct system. In our approach, the execution environment is completely statically defined and thus all execution parameters are to be known. In particular, the exact period of each process and the release offsets must be selected beforehand. We bridge this gap using the following two successive steps:

(i) Period Selection: In a first step, the periods are fixed according to one of the three heuristics, in the order of increasing likelihood to determine a feasible schedule.

1. **Best Performance:** The minimal defined periods are selected, thus improving the observable system performance, i.e., highest processing frequency.
2. **Minimal Hyperperiod:** The periods are selected as to minimize the hyperperiod, defined as the least common multiple of the process periods. A minimal hyperperiod makes it easier to use an exact schedulability test based on simulation. In addition, it also eases the comprehension of the system behaviour by the designer.
3. **Lowest Processor Utilization:** The maximal defined periods are selected, thus minimizing the processor utilization and maximizing the chance to derive a schedulable system.

(ii) Offset Optimization: The offsets are selected as to distribute the workload over time and to avoid load peaks, while respecting the timing correctness constraints [10].

We offer two schedulability checks, an exact test based on simulation and an approximate test based on the schedulability test for non-preemptive scheduling with offsets [12]. A feasibility test via simulation requires simulation up to twice the hyperperiod, which may be infeasible in many situations. We therefore set a time bound to limit the analysis time. Figure 1 illustrates the flow graph of the scheduler synthesis. Our approach differs from other scheduler synthesis tools, such as [1] in that we provide a complete framework to specify, optimize and execute the timing behavior. The scheduler synthesis is thus tailored towards the framework.

6. DEVELOPMENT PROCESS

So far, we have outlined (i) how the timing correctness can be declared, (ii) how the execution environment schedules the processes, and (iii) how a feasible schedule implementing the desired timing behavior can be determined. In this section, we detail the timing declarative design flow as illustrated in Figure 2. The parts shaded in blue are the novel components described in the paper.

The model of the cyber-physical system is composed of the functional implementation and the timing declaration. The functional implementation is input to the timing analysis, which computes

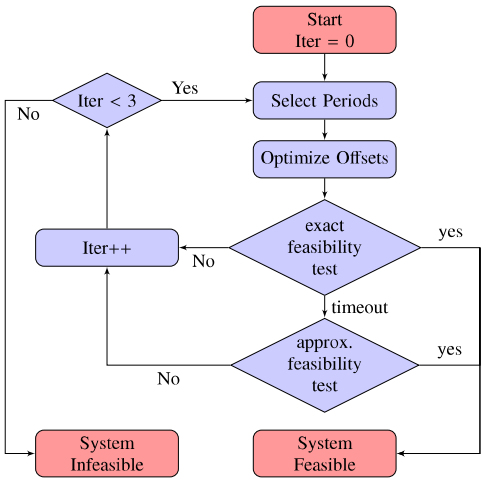

Figure 1: Flowchart of the Scheduler Synthesis.

bounds on the execution times of the functional components, denoted as processes in our framework. Note that we resort the existing static or dynamic analysis tools, as the worst-case execution time problem [13] is out of scope of this paper. The execution time bounds, as well as the timing model are inputs to the scheduler synthesis, which aims to derive a feasible scheduling configuration. If successful, the scheduling configuration is communicated to the simulator and the runtime environment, which implements the scheduling policy on the target system.

Even though the scheduler synthesis is an integral part of the framework, it is transparent to the system designer. The system designer only has access to the (functional and timing) model of the system and to the simulator. The simulator is used to present the synthesized schedule to the designer.

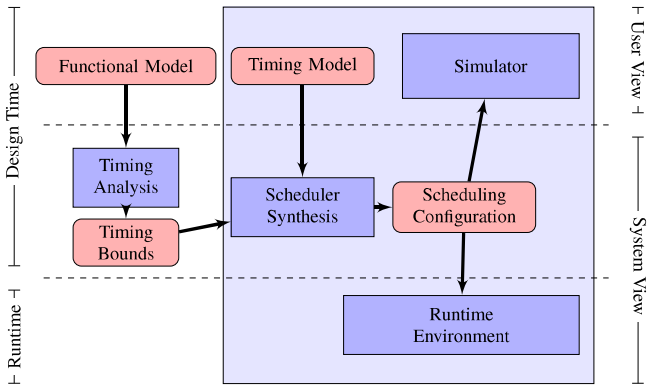

Figure 2: The timing declarative development process.

7. THE CPAL APPROACH

The ideas developed in this paper are already partially implemented in the CPAL (Cyber-Physical Action Language) modeling and development environment which aims at developing a model-driven development flow for timing-predictable embedded systems (see [11]). The vision behind CPAL is that programs can be executed and verified in simulation mode on a workstation and the exact same code can be later run on an embedded board with an equally acceptable timing behavior.

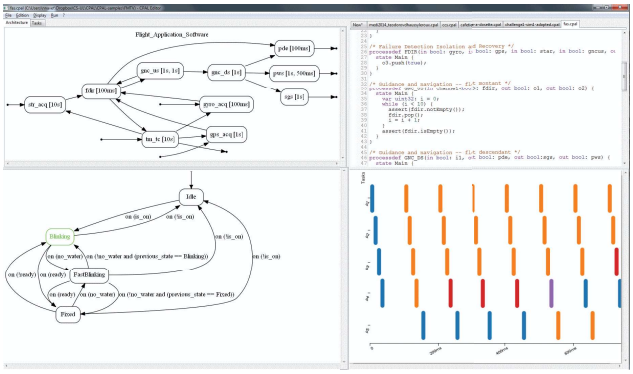

Figure 3: The CPAL editor shows the functional architecture of a system (top-left), the FSM describing the logic of a process (bottom-left) and the scheduling of the processes as seen during a simulation.

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 offers high-level abstractions that are natural for the development of CPS such as:

- *Real-time scheduling mechanisms*: processes can be activated with a user-defined period and offset relationships, with the optional requirement that a specific logical condition is fulfilled.
- *Finite State Machines (FSM)*: the logic of a process is defined as a Finite State Machine (FSM) where code can be executed in the states, or upon the firing of transitions.
- *Communication channels* to support control and data flow exchanges between processes, and read/write to hardware I/O ports with well-defined policies (FIFO or LIFO buffering, data overwriting).
- *Introspection mechanisms* that enable processes to query at run-time execution characteristics such as their activation rate and activation jitters.

CPAL development is driven by the experience gathered from realistic case-studies. For instance, a CPAL model is used in [15] to simulate the SOME/IP Service Discovery protocol used to manage service-oriented communication in automotive Ethernet networks. CPAL is also used in [2] to propose a solution to the Thales FMTV challenge (timing verification of an aerial video tracking system) and develop a smart parachute for UAVs in [3].

CPAL is a language jointly developed by our research group at the University of Luxembourg and the company RTaW. The CPAL documentation, graphical editor and simulation engine for Windows, Linux and Raspberry Pi platforms are freely available for all use from `http://www.designcps.com`.

8. CONCLUSIONS AND FUTURE WORK

This paper outlines a proposal for the development of CPS using a declarative language for the specification of non-functional timing properties. The same approach can be extended to other non-functional concerns such as power consumption. We believe that the increasing complexity of today's hardware platforms (multicore, SOC, etc) calls for methods that automate the synthesis and deployment embedded systems, and hide away from the programmer the complexity of the scheduling mechanisms and hardware platforms.

Purely synchronous development frameworks have brought major progresses to the development of embedded systems over the last 30 years, and are certainly very well suited in some application domains such as safety-critical systems. However, in many cases, we believe that more lightweight and less demanding programming models are able to guarantee the necessary timing predictability without over-constraining the design and development. The proposal discussed here, currently under implementation and experimentation in the CPAL development environment, is a contribution in that direction.

References

[1] K. Altisen, G. Gossler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1999.

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

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

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[5] 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.

[6] 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.

[7] H. Leontyev and J. H. Anderson. Tardiness bounds for FIFO scheduling on multiprocessors. In *19th Euromicro Conference on Real-Time Systems (ECRTS)*, Washington, DC, USA, 2007.

[8] S. Martin, P. Minet, and L. George. Non-premptive fixed priority scheduling with FIFO arbitration:uniprocessor and distributed cases. Research Report RR-5051, INRIA, 2004.

[9] 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.

[10] 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.

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

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

[13] R. Wilhelm and al. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, April 2008.

[14] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, July 2006.

[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 *Design, Automation & Test in Europe Conference (DATE)*, January 2015.