UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2019-14

Dissertation

Presented on 19/03/2019 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Sakthivel Manikandan Sundharam
Born on 12/04/1981 in Sathyamangalam (India)

Timing-aware Model-Based Design with Application to Automotive Embedded Systems

Dissertation defence committee

Dr. Nicolas Navet, Dissertation Supervisor
Associate Professor, University of Luxembourg (Luxembourg)

Dr. Pierre Kelsen, Chairman
Professor, University of Luxembourg (Luxembourg)

Dr. Yves Le Traon, Vice-chairman
Professor, University of Luxembourg (Luxembourg)

Dr. Sebastian Altmeyer, Dissertation Co-Supervisor
Assistant Professor, University of Amsterdam (Netherlands)

Dr. Emmanuel Grolleau, Member
Professor, ISAE-ENSMA (France)

Sakthivel Manikandan Sundharam

Timing-aware Model-Based Design with Application to Automotive Embedded Systems

Abstract

Cyber-Physical System (CPS) are systems piloting physical processes which have become an integral part of our daily life. We use them for many purposes: transportation (cars, planes, trains), space (satellite, spacecrafts), medical application, robotics, energy management, home appliance, manufacturing, and so many other applications.

Model-Driven Engineering (MDE) is widely applied in the industry to develop new software functions and integrate them into the existing run-time environment of a Cyber-Physical System (CPS), for instance, the control software for automotive engines, which are deployed on modern multi-core hardware architectures. Such an engine control system consists of different sub-systems, ranging from an air system to the exhaust system. Each of these sub-systems, again, consists of software functions which are necessary to read from the sensors and write to the actuators. In this setting, MBD provides indispensable means to model and implement the desired functionality, and to validate the functional, the non-functional, and in particular the real-time behavior against the requirements. Current industrial practice in model-based development completely relies on generative MBD, i.e., on code generation to bridge the gap between model and implementation. An alternative approach, although not yet used in the automotive domain is model interpretation. In this thesis, in the place of code generation, we investigate the applicability of model interpretation to automotive software development with a help of a control function design. We model and develop the calculation of the engine coolant temperature using interpreted model-based development CPAL (Cyber-Physical Action Language), and discuss the benefits compared to the existing code-generation practice.

The control laws of these software functions typically assume deterministic sampling rates and constant delays from input to output. However, on the target processors, the execution times of the software will depend on many factors such as the amount of interferences from other tasks, resulting in varying delays from sensing to actuating. Three approaches supported by tools, namely TrueTime, T-Res, and SimEvents, have been developed to facilitate the evaluation of how timing latencies affect control performance. However, these approaches support the simulation of control algorithms, but not their actual implementation. Further in the thesis, we present the CPAL model interpretation engine running in a co-simulation environment to study control performances while taking the run-time delays into account. The main advantage is that the model developed for simulation can be re-used on the target processors. Additionally, the simulations performed at design phase can be made realistic in the timing dimension through the use of timing annotations inserted in the models to capture the delays on the actual hardware. Introspection features natively available facilitate the implementation of self-adaptive and fault-tolerance strategies to mitigate

and compensate the run-time latencies. A DC servo controller is used as a supporting example to illustrate our approach. Experiments on controller tasks with injected delays show that our approach is on-par with the existing techniques with respect to simulation. We then discuss the main benefits of our development approach which are the support for rapid-prototyping and the re-use of the simulation model at run-time, resulting in productivity and quality gains.

As the processing power is increasingly available with today's hardware, other concerns than execution performance such as simplicity and predictability become important factors towards *functional safety* objective. The motivation towards *predictable* execution behavior, we revisited FIFO scheduling with offset and strictly periodic task activations. The execution order in this case is uniquely and statically determined. This means that whatever the execution platform and the task execution times, be it in simulation mode in a design environment or at run-time on the actual target, the task execution order will remain identical. Beyond the task execution order, the reading and writing events that can be observed outside the tasks occur in the same order. This property, leveraged by our MBD environment CPAL design flow provides a form of timing equivalent behavior between *development phase* and *run-time phase* which eases the implementation of the application and the verification of its timing correctness. Thus, the proposed development environment facilitates where also the non-experts are able to quickly model and deploy complex embedded systems without having to master real-time scheduling and resource-sharing protocols.

In practice, the design of a software component involves designers from various viewpoints such as control theory, software engineering, safety, etc. In practice, while a designer from one discipline focuses on the core aspects of the field (for instance, a control engineer concentrates on designing a stable controller), he / she neglects or considers less importantly the other engineering aspects (for instance, real-time software engineering or energy efficiency). This may cause some of the functional and non-functional requirements not to be met satisfactorily. In the thesis, we present a model-driven co-design framework based on the timing tolerance contract to address such design gaps between control and real-time software engineering. The framework consists of three steps: controller design, verified by jitter margin analysis along with co-simulation, software design verified by a novel schedulability analysis, and the run-time verification by monitoring the execution of the models on target. This framework builds on earlier mentioned CPAL design environment, which enforces a timing-realistic behavior in simulation through timing and scheduling annotations. The application of our framework is exemplified in the design of an automotive cruise control system. Through these case studies, we show that our tool enables not only to automate the analysis process at design time but also to enhance the design process by systematically combining models and analyses.

Keywords: model-based development; model-driven engineering; prescriptive modeling, descriptive modeling, real-time scheduling; schedulability analysis, worst-case execution time, timing tolerance contract; control software; controller model; control system performance; stability analysis, task release jitters; input jitters; varying execution-times; output jitters; input-to-output delay, co-simulation;

Acknowledgments

I have had the opportunity to meet, work with, and learn from a number of distinguished people who I have been impacted and indebted during my PhD. The accomplishment of this exciting and challenging journey is possible because of their support, constructive comments, and continuous feedback.

First, I would like to thank my Supervisor, Prof. Nicolas Navet for his patient guidance, advice and the encouragements which made this work achievable. His profound knowledge and abundant experience on model-driven engineering have been of great value to the research work presented in this thesis. I am always grateful for the precious knowledge and skills he has imparted me. I am also thankful to him for the opportunity to handle laboratory exercises during model-driven engineering courses, as this experience played a vital role during the work. He has has been my mentor over four years now, and I am looking forward to collaborate and work on further research directions the thesis brings in.

I wish to express my warm and sincere thanks to Prof. Dr. Yves Le Traon and Prof. Dr. Altmeyer as my thesis committee members. They have given me many insightful comments and advice. I have learnt a lot from Dr. Altmeyer through the research collaborations, especially the distinguished knowledge sharing on schedulability analyses. It has been a honor and great pleasure to work with him. I am sure I will cherish his collaboration for a long time. I am thankful to the members of the thesis defense committee, Prof. Dr. Pierre Kelsen and Prof. Dr. Emmanuel Grolleau for their time and effort to review this dissertation.

I extend my gratitude to the members of the RTaW France for the valuable discussions, tools and methods, the ideas discussed during the various meetings. Especially to thank Mr. Lionel Havet and Dr. Fejoz for their technical collaborations and tremendous support. My time at FSTC LASSY lab was enjoyable, an exciting work environment with my team Guillaume, Tingting Hu, Long Mai and others.

I dedicate this thesis to my family that have brought me so much love and unconditional support throughout my life. They have been always benevolent to me in pursuing my dreams. I would like to express my sincere thanks to the Luxembourg National Research Fund (FNR AFR project number 10053122) for their financial support to undertake this research work.

Sakthivel Manikandan SUNDHARAM
CRTES Research Group
Grand Duchy of Luxembourg 2019

Contents

List of Figures

List of Tables

List of Listings

List of Symbols

A, B, K Constants 21

C_i Worst-Case Execution Time 22

D_i Task's relative deadline 22, 25, 61, 62, 69, 72, 86

J^{τ} output jitter also known as response-time jitter 24, 86, 87, 99, 104

J^{h} input jitter also known as sampling jitter 24, 86–88, 93, 99, 103

L Task's busy-period 62, 68, 69, 72, 86

PD_i Processor demand 86

R_i^b Best-case response times 22

R_i^w Worst-case response times 22

T_i Controller task 22

Γ Task set 21

$\mathfrak{L}$ nominal input-output delay 86, 87, 99

τ input-to-output delay 24, 86, 87, 93

$\{T_1, \ldots T_n\}$ Periodic tasks 21

f_i Task's finish time 86

h_i Task's period 22

n Number of periodic tasks 21

r_i Task's release time 61, 67, 86

t_k^a k-th actuation time instance 86

t_k^s k-th sensing time instance 86

u Control signal 21

x Plant state 21

y Plant output 21

Chapter I

Introduction

Abstract

This thesis presents our research work on "Timing-aware Model Based Design with Application to Automotive Embedded Systems". In this Introduction chapter, we first present the context of the work and our research motivations. Next, we state the problems that we aim to address in this context. Subsequently, we review the related literature survey and the state-of-the-art practices. After that, we discuss the contributions which are provided in this thesis. We also detail the work hypotheses that fix the limits of these contributions. Lastly, we describe the organization of this manuscript.

I.1 Context and Motivations

Cyber-Physical Systems (CPS) are the types of embedded systems that can be found in cars, planes, robots, Unmanned Aerial Vehicle shortly UAV, medical devices, home appliances, factory and home automation, power production and distribution, etc. Such CPS, depending on the criticality of the application, are subject to certain safety, reliability and security constraints that must be verified as soon as possible in their development.

Model-Based Development (MBD), also frequently referred to as Model-Driven Development (MDD) or Model-Driven Engineering (MDE) or Model-Based Software Engineering (MBSE), denotes the use of models as the main artifacts to drive the development of such systems. In this thesis, we use MBD in general to mention these four acronyms where *model* drives the software development process. All the model-driven methods are model-based, but not the other way round. MBD has been profoundly reshaping and improving the design of software-intensive systems, and embedded systems specifically. For instance, since the introduction of hybrid and electric power-trains, it has become increasingly common case in the automotive industry to design new functions using MBD environment like Matlab/Simulink, and generate code from the models that is being used sometimes untouched in Electronics Control Units (ECUs) like in Figure I.1 of series cars.

Over the last 30 years, a great amount of research efforts in computer science has been devoted to the verification of correctness constraints, leading to important

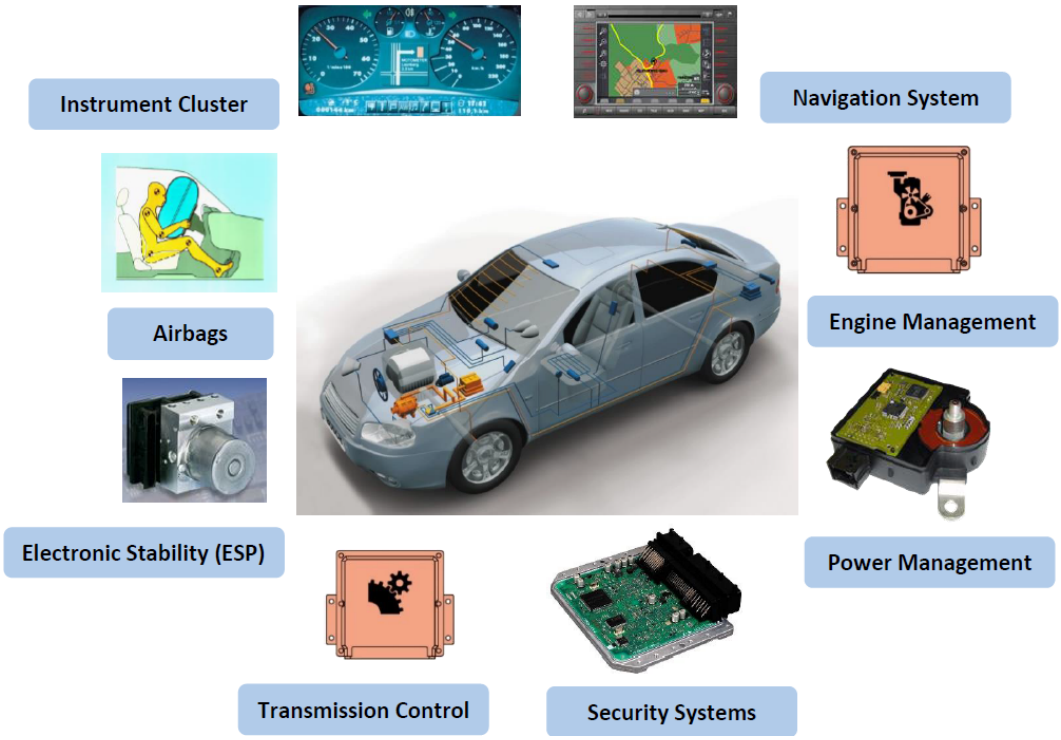


Figure I.1: Context: Automotive embedded systems.

achievements [Kreiker et al., 2011; Woodcock et al., 2009; Sifakis, 2008]. The correctness of a system is twofold: correctness in the value domain (i.e., the system produces the right outputs) and correctness in the time domain (i.e., the outputs are produced at the right points in time, typically before a given point in time). Correctness in the value domain is the realm of formal methods (model-checking, interactive theorem proving, etc [Rushby, 2006; Merz and Navet, 2008], while correctness in the time domain is usually ascertained using techniques developed in the real-time system community (formal verification methods such as worst-case execution time, schedulability analysis, etc. [Davis, 2014]). Both approaches are referred to as formal verification. The time- domain formal verification methods are mature, but are not well automated and integrated with MBD environments.

I.2 Problem Statement

A modern car typically consists of a present-generation Adaptive Cruise Control (ACC) function, a safety critical Driving Assistance System (ADAS) implemented on multi-core CPUs and distributed over several Electronics Control Units (ECUs) of the vehicle namely Radar ECU, the ABS/ESPs (braking function), Body Control Module (interaction with the driver), engine controller (engine control), etc. While the control laws in many embedded systems are not necessarily very complex, their implementation is usually very time consuming and their correctness verification are only partial. Many errors are detected very late in the development process, sometimes even once the product has been released. We believe that this is partly

due to the excessively complex run-time environment (example AUTOSAR) and partly due to the inability of the languages and development tools to efficiently support the real-time constructs. According to Feiler et al. [Feiler et al., 2009], 70% of faults have their origins at design time while 80% of them are discovered after the implementation phase.

Current design practices in real-time embedded and CPS usually treat real-time behavior as merely a by-product of the functional implementation. For instance, in many MBD flows the computational resources available are even considered to be infinite. The Quality of Service (shortly QoS), for instance *stability* property, that can be offered by the execution platform is just not considered. When timing verification is a significant concern, typically the practice is that the complete code is developed and the task execution times bounds 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 sub-optimal: 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. This high-level question is further explored in the thesis while investigating five sub-problems.

P1: Towards improving the turn around time in the development process. Existing MBD practices such as Matlab/Simulink and SCADE capture most of the aspects of model based design: requirements traceability, model design, code generation, model and code verification, test-case generation, etc. They are complex black-box software. Since they rely on code generation, they require a code generator, a compiler, a linker, a C library, and most often an operating system. The huge size of the software involved both off-line and at run-time is error prone in itself and makes it unfeasible to verify the complete software involved. This is an issue for critical systems. They are not well suited to distributed systems (i.e., no automatic configuration and deployment in the distributed case), high-performance computing architecture (SOC, multi-core, many-core, etc) and more generally to hard real-time systems where scheduling is of crucial importance. The question that is explored in this thesis is whether and how a MBD flow based on model interpretation can facilitate and speed up the automotive control software development. This includes the deployment of timing verification of applications with real-time constraints, possibly distributed over a number of nodes, and running on potentially complex hardware platforms.

P2: Invisible formal verification in MBDs. If formal verification techniques are now sufficiently mature to handle a number of industrial problems, they are far from being widely used yet, and this probably mainly because they are not sufficiently well automated and integrated within the standard development environments [Kreiker et al., 2011; Woodcock et al., 2009]. In the words of J. Rushby[Rushby, 2000, 2006], we need to "make the formal methods disappear inside traditional tools and methods". This objective is of primary importance for MBD where nowadays the support for formal verification in the time-domain is mainly non-existing, especially in the early phases of the development cycle, where the cost of repairing the errors is small. The previous works [Mraidha et al., 2014; Cervin et al., 2003a] are of notable exceptions in the direction of timing-aware MBDs which are mainly simulation

techniques. In this thesis, we investigate the problems associated in the context of integrating the time-domain correctness to the early stage of the development life cycle to mitigate the late-stage integration issues.

P3: How do we ensure real-time constraints are met during run-time? Considering an arbitrary model made up of a set of functional blocks communicating with each other that is to be executed on a complex hardware platforms, how to automatically configure the scheduling of the activities executed by the run-time environment so that the real-time constraints are met at run-time? It further drills down to whether it is possible to achieve a platform-independent temporal execution behaviour for interpreted models and how? This would imply that the models will have the same real-time behaviour between simulation and on-target execution. Once validated in timing accurate simulation, a model could thus be re-used on embedded platforms with the minimum required computational power. This would be a huge plus in terms of re-usability and would facilitate incremental certification. This research will not rely on the simplifying classical synchronous assumption that the underlying platform is infinitely fast.

P4: The lack of real-time predictability. If the real-time QoS supplied by the execution platforms is insufficient, it will directly impact the quality of the performed function, often a control function. In the best case, this may result in time consuming and costly modifications later in the development process, when all software modules are integrated. In the worst-case, this will affect the correctness of the function at run-time, and possibly jeopardize the safety of the system if the departures from the intended behavior have not been identified by tests. For instance, this could have severe consequences in typical automotive functions like Adaptive Cruise Control (ACC), Lane Keeping (LKAS), and Advanced Driving Assistance Systems (ADAS) at large. In any cases, in a context where software-functions are key enablers for the innovation, it is of primary importance for the system designer to be able to precisely determine if an electronic embedded architecture can accommodate more functions, and the margin for evolution it possesses. To extend the academic state-of-the-art and, ultimately with the aim to contribute to the improvements of the industrial practices, this thesis explores a novel approach based on model-interpretation to provide support for resource usage estimation and integrate time-domain verification in the early phases of MBD. The application domain targeted is the automotive electronics [Navet and Simonot-Lion, 2008] where the expected outcomes of this thesis could be quickly disseminated with a substantial industrial impact.

P5: The stakeholders communication gaps on *timing* during design. Control theory and software engineering are two disciplines involved in the development of control software. Traditionally, control engineers design the controller model without considering the computing platform constraints and specifications. The converse applies to software engineering, where control performance is not considered during software design. The control engineering and the software engineering are two different worlds with different objectives in mind. Consequently, the complete set of functional and non-functional requirements of the control software are usually not elicited at the control design stage. Several tools and methods developed in the past to address this problem which are essentially simulation environments that involve

a step of model-to-code transformation (typically code generation), which may risk widening the semantic gap between model and executable code, requiring additional development effort. Generally speaking, the existing co-design simulation techniques are mainly concerned with enabling the study of the effect of timing variabilities on control performance, rather than addressing the design gaps between control and software viewpoints. Other works [Lampke et al., 2015; Ziegenbein and Hamann, 2015] present co-engineering techniques where the initial controller is integrated in a virtual ECU. The behavior of the controller is then assessed through timing analysis tools whose results are injected into the controller model. This approach shares similarities with ours but it relies on expensive and proprietary timing analysis tools and remains at the model level (i.e., implementation is abstracted).

I.3 Work Hypotheses and Contributions

Model-Based Design MBD encompasses a variety of practices. The execution of models can be performed in two manners namely *generative* and *interpretative*. This thesis focuses on *interpretative*, the model interpretation, that is the direct interpretation of the design models on an interpretation engine running on top of the hardware without code generation and without an operating system. In the following, we refer this technique as *Bare-Machine Model Interpretation*, or BMMI in short. The term *Bare-Machine* stresses that no operating system is needed. The contributions are labeled as C_i and the corresponding hypotheses are denoted as H_i.

C1: Model interpretation for an AUTOSAR compliant engine control function. The first sub-problem stated in the previous section I.2 of the thesis is whether and how a MBD flow based on model interpretation can facilitate and speed-up the development. Many critical embedded systems would probably benefit, in terms of verifiability and time-to-market, from "leaner" development environment and execution platforms. If code generation is prevalent, model interpretation possesses a number of key advantages such as significantly shortening turnaround time [Völter, 2009], since it does require an explicit code generation/rebuild/retest/redeploy step, each time the model is changed. This is especially valuable in the early design phases where changes are frequent. Though model interpretation for critical systems on modern hardware platforms is very promising, it has not been thoroughly explored and experimented yet to the best of our knowledge. Towards this, we present a bottom-up approach *prototyping*, a realistic case study to investigate the applicability of model interpretation, in contrast to code generation, for the development of engine control systems. We model an engine cooling system, specifically the calculation of the engine-coolant temperature, using interpreted model based development, and discuss the benefits and drawbacks compared to the existing code-generation practice.

This contribution has been published in [Sundharam et al., 2016a] and is discussed in Chapter III *Lean Interpretation-based MBD to Design Automotive Software*.

H1: Overhead of interpretation engine. Regarding the overhead of interpretation engine, the complete source code of the prototype of the interpreter is made of only 2500 lines of C code, and complete formal verification could be envisaged.

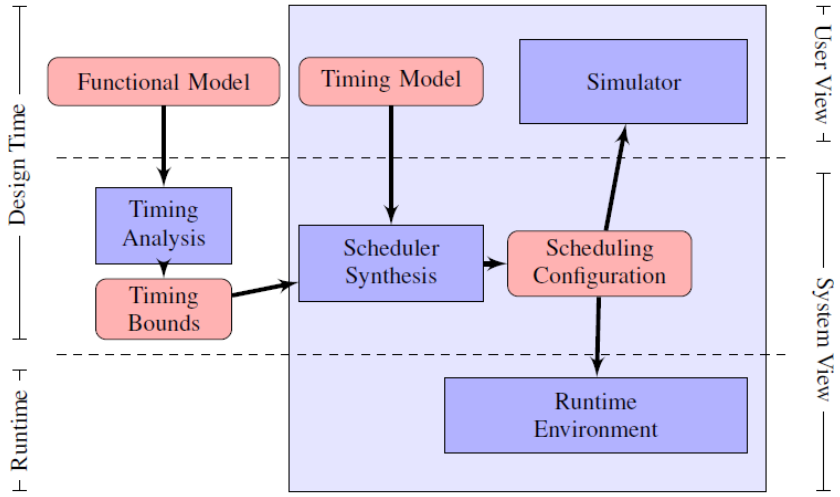


Figure I.2: The timing declarative development process.

C2: Towards a declarative modeling and execution framework. 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.

As illustrated in Figure I.2, the parts shaded in blue are the components described in the thesis. The model of the CPS is composed of the functional implementation and the timing declaration. 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 run-time 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.

This contribution has been presented in [Sundharam et al., 2016b] and is discussed in Chapter II *Background and Preliminaries*.

H2: Worst-case execution time problem. The functional implementation is input to the timing analysis, which computes 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 [Wilhelm et al., 2008a] is out of scope. The framework presented in the thesis relies on the measurements based WCET estimation but WCET is not

the contribution of the work and our approach would work with any other WCET estimation techniques such as static deterministic analysis or probabilistic analysis.

C3: A MBD co-simulation environment for rapid-prototyping of latency-sensitive control software. The experiments in this thesis are done with a novel modeling language called CPAL (Cyber Physical Action Language) [Navet and Fejoz, 2016a], well suited for research and teaching. Similar in the spirit to StateFlow®, CPAL relies on Mode-Automata [Maraninchi and Rémond, 2003] for block-level language and on the Prelude synchronous language [Pagetti et al., 2011] for the data-flow languages (communication among blocks of code) as depicted in Figure I.3. The CPAL interpreter can be executed on an embedded target, or within another simulation environment such as Matlab/Simulink. The later is called *co-simulation*, where we present a model interpretation engine running in a host simulation environment to study the control system performances while considering the run-time delays into account. Introspection features natively available facilitate the implementation of self-adaptive and fault-tolerance strategies to mitigate and compensate the run-time latencies. A DC servo controller is used as a supporting example to illustrate our approach. Experiments on controller tasks with injected delays show that our approach is on par with the existing techniques with respect to simulation. We then discuss the main benefits of our development approach that are the support for rapid-prototyping and the re-use of the simulation model at run-time, resulting in productivity and quality gains.

This contribution has been presented in [Sundharam et al., 2016d] and is discussed in Chapter IV *CPAL Co-simulation for Timing-aware Prescriptive Modeling*.

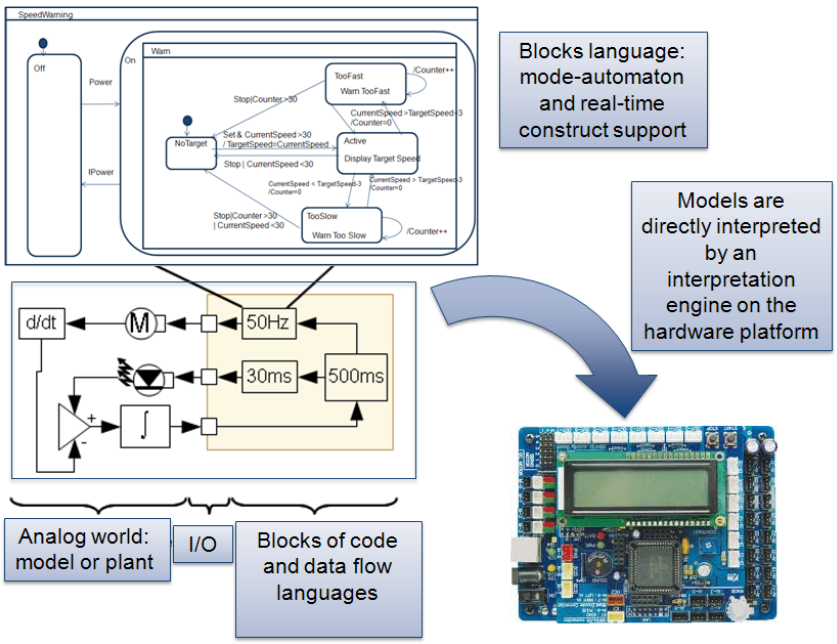


Figure I.3: Global view of the development environment presented in the thesis that relies on a language for programming functional blocks (based on mode-automata), an architecture description language with formal description of the data-flows between blocks (based on Prelude).

H3: Real-time mode vs Simulation mode. Internally CPAL relies on a Discrete Event Simulator (DES) for the simulation mode. The execution is done as fast as the CPU permits. It means that the simulator works with a logical time where the progress of time does not follow the physical time but a logical time (keeping tracks of events ordering and their time stamp). In co-simulation, the sequence of CPAL events expressed in logical time is re-injected on-the-fly into Simulink. In both modes (real-time and simulation), we have a queue of all the events in a model (tasks with their relative activation delays). In the case of simulation, the interpretation engine steps to the next event (DES) with a logical-clock. In the case of real-time mode, at each clock tick of the target hardware (in the case of BMMI, the Real-Time Clock (RTC) module provides the services to realize the actual timing), relative activation delay is decreased and once it reaches the zero, meaning task completed the execution time mentioned and steps into the next event.

C4: Timing predictability and timing equivalence bahaviour between simulation and embedded execution. This contribution explores how a early-stage timing verification can be facilitated by a timing-accurate interpretation engine (invisibly) embedded in the MBD development environment where also the non-experts are able to quickly model and deploy complex embedded systems without having to master real time scheduling and resource-sharing protocols. Especially irreproducible faults due to different timing behaviors or race conditions are a nightmare to debug. We acknowledge that techniques to avoid these problems exist, but they constitute a major obstacle for newcomers and make both design and code more complex and error-prone. When processing power is sufficient, as it is increasingly the case with modern hardware, other concerns than performance such as simplicity and predictability become important. This contribution, the revisit of FIFO scheduling for certain class of task models enables our MBD environment to provide a timing predictability and simplicity that eases the design and verification of embedded real-time systems. We provide a schedulability analysis for FIFO, both with and without task offsets. The execution order of FIFO scheduling with offset and strictly periodic task activation is uniquely and statically determined. This means that whatever the execution platform and the task execution times, be it in simulation mode in a design environment or at run-time on the actual target, the task execution order will remain identical. Beyond the task execution order, the reading and writing events, as they can be observed outside the tasks, occur in the same order.

This contribution has been published in [Altmeyer et al., 2016] and is discussed in Chapter V *Towards Timing Equivalent Behaviour*.

H4: The execution model of FIFO schedulability analysis We re-visit the FIFO scheduling under modified conditions and make a case for FIFO scheduling with strictly periodic task activation and release offsets to increase the predictability and to improve the performance.

C5: A model-driven co-design framework for fusing control and scheduling viewpoints. This contribution is a structured co-design framework to address the problem which arises during controller model design. We apply timing tolerance contract to our timing-aware MBD workflow that enables the control engineer to

consider target platform timing during controller design and provides the system integrator with the control details to explore the design space during software design (i.e.,) scheduling choices. When a new control function is required to be integrated into an existing stable, feasible, functioning Electronic Control Unit software (ECU), how can we formally validate whether the system maintains the desired performance (stable and schedulable) after integration. This work contributes three topics: firstly, how to apply timing tolerance contract to control and computing context to bridge the design gaps. Secondly, we discuss how do we define the timing tolerance contract using the formalism and semantics with in CPAL to support the design phase. We explain our approach `Analinta` (ANALysis of INtegrated TAsk) with the help of a schedulability analysis for periodic task activations. The third and last contribution is the proposed run-time verification methodology. During model on target execution, we check whether the newly integrated controller function stays within the stability margin. For this, we take advantage of CPAL introspection features to monitor the execution characteristics of a controller model at run-time. More specifically, we introspect whether the jitters and input output latencies are within the margin guaranteeing the stability and schedulability objectives.

This contribution has been presented in [Sundharam et al., 2018] and is discussed in Chapter VI *Model-driven Co-design Workflow*.

H5: Scheduling algorithm influences in the system design. The QoS provided to the tasks by the execution platform directly depends on the scheduling algorithms. Typically for control tasks jitters (input jitters, output jitters, and input-to-output delay) will be dependent on the choice of scheduling algorithm. The framework proposed in this contribution is compatible with any scheduling policy that guarantees that the deadlines will be met, although in the work we will rely on FIFO which, as explained in **C4**, facilitates the system design.

We present the proof-of-concept implementation and bench-marking on realistic case-studies. The first case-study is *Automotive thermal control function*, because there are a lot of works to rely on and compare to, such as [Fleming, 2001; Li et al., 2014]. The second case-study is an *Adaptive Cruise Control* shortly ACC [Cruise Control, 2019], which is a complex distributed function. Apart from these two case studies, we explore different application domains and contexts to which few of our contributions are pertinent. For this reason, we consider Internet of Things (IoT) application domain. In IoT, the first case-study is *Connected Motorized Riders*, and the second one is *Parking Minute* with some future works envisaged.

I.4 Literature Survey and State-of-the-art

In the literature of computing and control, there have been numerous studies and works presented. The state-of-the-art MBD practices have some shortcomings: the real-time Quality-of-Service (QoS) offered by the execution platform is just not considered. Most often, the computational resources available are considered to be infinite (code executes in zero-time, message transmissions are instantaneous, except in TrueTime [Cervin et al., 2003b] which is worthy step in the right direction but for networks only), or they are crudely estimated. However, once on the target processors, the execution time of the software will depend on the speed of the hardware platform,

the availability of the processor on the scheduling of the other tasks, the message latencies on the network transmission schedule, etc. This research, largely open in our view, will not rely on the unrealistic classical synchronous assumption, done for example in the Esterel language, that the underlying platform is infinitely fast. Nonetheless the synchronous programming approach [Altisen et al., 2002; Benveniste and Berry, 1991] will be a good starting point along with the research and return of experience on executable UMLs, TAU2 and POOSL languages [Huang et al., 2004; Geilen, 2002].

I.4.1 Related Works in Real-time Control Systems

The continuous-time signals from the sensors are periodically sampled; each sampled set of data is then processed by real-time control functions. Even though control theory assumes actuation to happen right after the sensing, since the computation of the control law consumes some amount of execution time on the processor, there exists a delay between sensing and actuation which is called input-output delay or sensing to actuation delay. In practice, for instance, due to preemptions and varying task execution times, the input to output latencies will vary over the time. Both input jitter and input to output delays will directly impact the quality of the control functions, in the worst-case, possibly jeopardizing the safety of the system. Hence, it is important to consider these delays during the design phase of the control software. In the past, approaches and tools have been developed [Torngren et al., 2006] to design control functions considering timing dimensions.

In the literature of computing and control, there has been plenty of studies and tools developed on timing irregularities effects to control design performances [Torngren et al., 2006; Cervin et al., 2003a; Morelli et al., 2014; Sundharam et al., 2016a]. On the schedulability perspective too, various schedulability analysis and tools developed. In this thesis, to explain schedulability part of contract analysis, we consider FIFO policy with offsets for our periodic sampled data systems. Proposed composite contract approach can be further extended to existing various schedulability analysis tools such as Cheddar, MAST etc. With respect to FIFO schedulability test, we found some works in the literature. Notably, George and Minet [George and Minet, 1997] presented a scheduling analysis for FIFO on a distributed system assuming sporadic task releases, and Leontyev and Anderson [Leontyev and Anderson, 2007] presented a tardiness analysis for FIFO scheduling of soft real-time tasks, also assuming a distributed system and sporadic task releases. To the best of our knowledge, FIFO with offsets for periodic task releases has not yet been analyzed.

Baruah et al. presented the jitter concerns in periodic task systems along with the feasibility analysis algorithms [Baruah et al., 1997]. In this work, authors discussed jitter tolerances mainly in the landscape of schedulability study. Subsequent to this work, Baruah et al. proposed a quantitative model for output jitter in [Baruah et al., 1999]. Cervin et al. coined the term *Jitter Margin* in [Cervin et al., 2004] where authors considered only the output jitter margin to which the system maintains the stability. Incorporating the input jitter, Cervin proposed a subsequent work to analyze combined effect of both jitters on the control performance of linear sampled-data control systems [Cervin, 2012]. Now comes the open question to our work, how do we add these input and output jitter margins into the composite contract based design which guarantees both control performance and schedulability criteria.

I.4.2 State-of-the-art Practices in Co-design of Control Function

Powerful Model-Based Development (MBD) tools such as MATLAB/Simulink (MLSL) and ASCET/MD are available for the design and development of control systems. On the other hand, there are dedicated tools such as MAST and PyCPA for analyzing and configuring real-time scheduling algorithms. Three approaches with associated tools that are presented hereafter go in the direction of a combined approach, *i.e.* to support control system design considering the influence of scheduling strategies.

SimEvents MATLAB/Simulink is a multi-domain industry standard for the design of control systems. Simulink Control Design supports the design and analysis of control systems. SimEvents is a discrete-event simulator that can be used as blocks in Simulink [SimEvents, 2019] to perform system-level simulation. It provides options to create tasks, and is able to inject network and scheduling delays with the support of the basic scheduling policies FIFO, LIFO and fixed priority scheduling. Other policies like EDF are left to the user to program. To the best of our knowledge, a significant drawback of MLSL SimEvents is that one cannot generate code from the system model consisting of SimEvents blocks. Hence, the actual realization of the controller model using SimEvents is not feasible. SimEvents is meant to model various applications where models are driven by events, starting from operation research and manufacturing processes to real-time systems. Because of this generality, it does not provide all domain-specific concepts needed for real-time systems like those available in TrueTime and T-Res.

TrueTime The MATLAB/Simulink-based tool [Cervin et al., 2003b] enables the simulation of the temporal behavior of controller tasks executed on a multitasking real-time kernel. In TrueTime, it is possible to evaluate the performance of control loops subject to the latencies of the implementation. TrueTime offers a configurable kernel block, network blocks (CAN, Ethernet, etc.), protocol-independent send and receive blocks and a battery block. These blocks are Simulink S-functions written in C++. TrueTime is an event-based simulation using zero-crossing functions. Tasks are used to model the execution of user code. The release of task instances (jobs) is either periodic or aperiodic. For periodic tasks, the jobs are created by an internal periodic timer. Aperiodic tasks can be created in response to external trigger interrupts or network interrupts. The task code is written as code segments in a Matlab script or in C++. It models a number of code statements that are executed sequentially. All statements in a segment are executed sequentially, non-preemptively, and with a simulation time that can chosen by the developer through an annotation.

T-Res This recent tool [Morelli et al., 2014] is also developed using a set of custom Simulink blocks created for the purpose of i) simulating timing delays depending on the code execution, scheduling of tasks, and communication latencies, and ii) verifying their impact on the performance of control software. T-Res is inspired from TrueTime and provides a more modular approach to design the controller model enabling to define the controller code apart from the model of the task.

Besides these three tools introduced previously, the other tools developed in the past are in general, specialized to a certain aspect of the co-design problem. For example, Jitterbug [Cervin et al., 2003b] supports statistical control-performance analysis for

a certain class of control systems. Also, these tools and methods focus only on the analysis and simulation level. They help the designer only with the study of system performance under the effects of timing delays, but not the system development. The system designer, then takes these analysis results into account to develop the actual embedded control algorithms in the next steps. This increases the possibility of distortions between the simulation model and the implementation.

A model-interpretation based run-time environment which can be used in a co-simulation environment to analyze the effects of delays on the performance of the control system, with the advantage that the controller model used in the simulation can be executed on the target hardware. This way, we eliminate the gaps between the simulation models and the executables.

I.4.3 Literature Works in Timing Contract Models

Contract based design that enables the correct by construction is discussed with motivating examples in [Benveniste et al., 2012]. In the same work, concepts and operations needed for contract framework are well explained. Various methodologies to address the system design challenges are discussed in [Sangiovanni-Vincentelli et al., 2012]. This work also exemplifies how contracts can be applied to a water flow control design. Making timing assumptions explicit using design contracts that are evolved as design approaches is illustrated in [Derler et al., 2013]. In this work the support for timing contract based design is discussed using Ptolemy and Simulink. All the three works focus on how we address gaps in system design that occur due to different viewpoints such as safety, timing, resource usage involved etc.

We apply contract concept in this thesis to timing-aware model based control design workflow that enables the control engineer to consider target platform timing during controller design and provides the system integrator with control details to explore the design space during scheduling design and system integration. In the line of reducing the control and computation engineering gaps, we found that [Lampke et al., 2015] is more related where initial control design is integrated virtually for timing analysis and the feedback is given back to get actual design.

Benveniste et al. presented a report [Benveniste et al., 2012] which discusses how to apply contract concept to design methodologies which build the system using the correct by construction approach. In this work, authors illustrated the motivating examples in the phase of requirement management and component design. In the same work, authors explained the mathematical concepts and operations necessary for the contract framework. Sangiovanni-Vincentelli et al. discussed various methodologies to address the system design challenges in [Sangiovanni-Vincentelli et al., 2012]. This work exemplified also how the contract can be applied to a practical use-case, water flow control system design. Derler et al. presented a work [Derler et al., 2013] where implicit timing assumptions made explicit using design contracts to facilitate the interaction and communication between control and software domain. In this work, authors discussed the support for timing contract based design using Ptolemy and Simulink. All the three works mentioned so far focused on the fundamental framework for design contracts such as contract algebra to system design, merging contract concept to platform-based design, and timing contract visualization in modeling environment respectively.

I.5 Thesis Organization

This thesis is organized into seven chapters plus one appendix chapter that provides auxiliary work, *product prototyping* which is loosely based on the techniques discussed in the first seven chapters.

Chapter I (Introduction) introduces the context of the works and our research motivations. Next, we state the problems that we aim to address in this context. After that, we present the literature survey state-of-the-art practices particularly relevant to our work namely, co-design of control function, real-time scheduling, Contract based-design, and MBD to IoT application development. Then we detail the work hypotheses and the thesis contributions. Lastly, we describe the organization of this thesis report.

Chapter II (Background and Preliminaries) deals with the fundamental concepts that are necessary to comprehend the issues tackled in this thesis and proposed contributions regarding methods and tools to develop real-time control software. We start this chapter with the functional and non-functional requirements of an automotive embedded system and then moving onto explain how MBD helps to realize the system requirements. We also present timing contract based design which supports the proposed timing tolerance contract based framework in the `Chapter VI (Model-driven Co-design Workflow)`. Then we present the link between control theory and real-time scheduling while developing an automotive control function. Further, we introduce the optimization framework that automates the selection and configuration of the scheduling policy. The objective is to let designers state the permissible timing behavior of the system in a declarative manner. The system synthesis step involving both analysis and optimization and then generates a scheduling solution which at run-time is enforced by the execution environment.

Chapter III (Lean Interpretation-based MBD to Design Automotive Software) We use MBD as a common practice in the automotive industry to develop real-time software based on the system constraints and requirements. In this setting, MBD provides indispensable means to model and implement the desired functionality to verify and validate the functional and non-functional requirements. Current industrial practice in MBD completely relies on *Generative*, i.e., code generation to bridge the gap between model and implementation. An alternative approach, although not yet used in the automotive domain is *Interpretative*, the direct execution of the design models using interpretation engine running on top of the hardware. In order to speed up the development time, we also investigate the applicability of our approach *Interpretative* MBD to automotive control function development with the help of a case-study which is based on an AUTOSAR design pattern and discuss the benefits and low-lights compared to the existing code-generation practice.

Chapter IV (CPAL Co-simulation for Timing-aware Prescriptive Modeling) explores the prescriptive modeling using CPAL and it's scheduling and timing annotations. We present a model interpretation engine running in a co-simulation environment to study control performances while considering the run-time delays in to account. Then we compare the three literature approaches

supported by tools, namely TrueTime, T-Res, and SimEvents to facilitate the evaluation of how timing latencies affect control performance with our proposed approach. The proposed timing-aware prescriptive modeling using CPAL eases the design and verification of embedded real-time systems.

Chapter V (Towards Timing Equivalent Behaviour) discusses the timing predictability and timing equivalence bahaviour between simulation and embedded execution. The motivation is to provide an environment where also non-experts are able to quickly model and deploy complex embedded systems without having to master real-time scheduling and resource-sharing protocols. To this front, we re-visit the FIFO scheduling policy for periodic task activations with release offsets. Under this conditions, we exhibit a schedulability analysis and present the advantages of FIFO scheduling in the context of MBD.

Chapter VI (Model-driven Co-design Workflow) presents the co-design framework that is based on the idea discussed in `Chapter II` and the timing tolerance contract to address design gaps between control and real-time software engineering. The framework consists of three steps: controller design, verified by jitter margin analysis along with co-simulation, software design verified by a novel schedulability analysis, and the run-time verification by monitoring the execution of the models on target. This framework builds on CPAL (Cyber-Physical Action Language), an MDE design environment based on model-interpretation, which enforces a timing-realistic behavior in simulation through timing and scheduling annotations. The application of our framework is exemplified in the design of an automotive cruise control system.

Chapter VII (Conclusions and Outlook) recaps the content of this thesis and summarizes the main results. This chapter also presents the future research direction along with potential extension of work carried out.

Chapter VIII (Appendix) explores the advantages of prescriptive modeling using CPAL to develop IoT applications. The descriptive modeling of high-level functional and non-functional requirements (e.g. Capella, SysML) for the unique problem under design. Multiple solutions can emerge during architecture phase which further leads to multiple solutions during development. Using design optimization, we can obtain the solution that meets the system constraints.

Chapter II

Background and Preliminaries

Abstract

This chapter presents the fundamental concepts and definitions that are necessary to comprehend the issues tackled in this thesis and proposed contributions regarding methods and tools to develop real-time control software. We start this chapter with the functional and non-functional requirements of an automotive embedded system and then moving onto explain how MBD helps to realize the system requirements. This also includes the Model-Based System Engineering (MBSE) terminologies. Then, we present timing contract based design which will be useful in explaining the proposed timing tolerance contract based framework in the `Chapter VI(Model-driven Co-design Workflow)`*. We provide the link between control theory and real-time scheduling while developing an automotive control function. After these fundamental concepts, the chapter discusses an optimization framework that automates the selection and configuration of the scheduling policy. The objective is to let designers state the permissible timing behavior of the system in a declarative manner. The system synthesis step involving both analysis and optimization then generates a scheduling solution which at run-time is enforced by the execution environment. This envisioned framework is later articulated as co-design framework in the context of control and software engineering. Scheduling is crucial in real-time control applications. For any real-time control system, the desired scheduling policy can be selected based on the scheduling problem itself and the underlying system constraints.*

II.1 Automotive Embedded Systems

In a modern automotive vehicle, many number of ECUs become evident for various reasons such as energy saving, low emission to meet the stringent emission norms, active and passive safety, comfortableness, convenience, entertainment, cost, and weight reduction. ECUs are connected with several in-vehicle networks and impose high-level of reliability and safety requirements. Due to the increasing number of various types of requirements, the amount of software in an automotive vehicle is in an ever increasing trend. Correspondingly, the hardware, for instance multi-core or high-performance computing platform, to execute the software also getting better and better in terms of performance. As the automotive embedded systems are life-critical applications, they demand strict real-time properties predictability and deterministic

execution of software functions. And the system operates under varying environmental conditions temperature and electro-magnetic interference. Few examples of automotive embedded systems are power-train control ECU, transmission control ECU, anti-lock brake system ECU, airbag ECU, instrument cluster, infotainment unit etc.

II.1.1 Automotive Software Development Life Cycle (SDLC)

A number of software development methodologies have been introduced in the past to guide and manage the development processes of software. These methodologies are usually called Software Development Life Cycle (SDLC) models. These methodologies are divided into two main groups, *Traditional* and *Agile*. Each organization follows one or more of these methods, a hybrid of two or more, or a company-tailored model based on the project needs and organizations business goals. In general, the life-cycle steps and sub-steps are not rigid, and may undergo for process improvements to reduce the cost and time. A Software Development Life Cycle (SDLC) is be defined as a step by step breakdown of the activities which are needed during a software development project. It defines a number of phases, relationship between these phases, milestones or goals for each phase and corresponding deliverables.

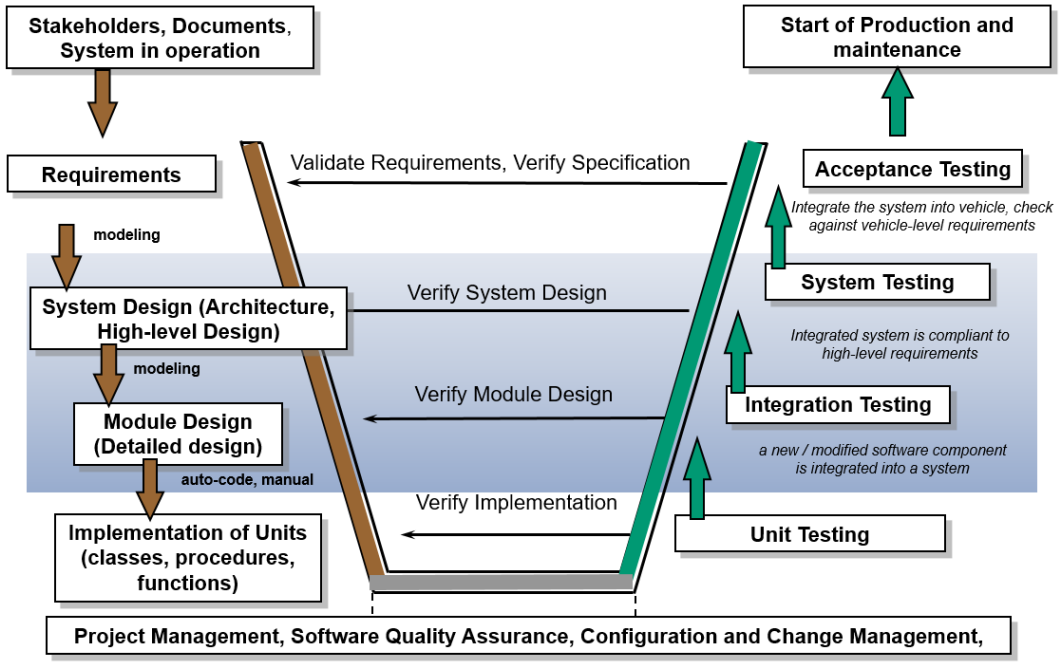


Figure II.1: Different phases of *V-model* development life-cycle to implement automotive control function.

Simple life cycles models may have only three phases: Design, Development, and Testing; while the complex life cycle models may include 20 or more phases. Typically, a life cycle model addresses these phases of a software project: requirements, design, implementation, integration, testing, operations and maintenance phase. There are a number of SDLC models which follow this approach namely Waterfall model, Incremental model and V-model etc. In the thesis the proposed lean MBD in

Chapter III (Lean Interpretation-based MBD to Design Automotive Software) is based on V-model, which is a widely used practice compared to other two. The testing of software is much more focused throughout the development life cycle in V-model than in waterfall model. The reason this model is called V-model is that the testing plans and procedures are made for all the phases preceding and including implementation/coding phase. These testing plans and procedures are then executed after the coding is done to verify those phases which precede implementation phase.

II.1.2 Functional and Non-functional Requirements

The term *requirement* is about identifying the real problem to be solved and providing a solution for it. A Requirement is considered to be a need perceived by an external stakeholder, someone with an interest in the system, typically a customer who pays for the product to be built. A requirement is also a capability or a property the system. As described by Easterbrook [Easterbrook, 2004], Requirements engineering is a set of activities concerned with identifying and communicating the purpose of a software-intensive system, and the contexts in which it will be used. IREB (International Requirements Engineering Board) defines the Requirements engineering as a systematic and disciplined approach to the specifications management for storing, changing and tracing of requirements the goals namely, understanding the stakeholders desires and needs and knowing the relevant requirements, achieving a consensus about the stakeholders about these requirements, documenting them according to given standards, and managing them systematically.

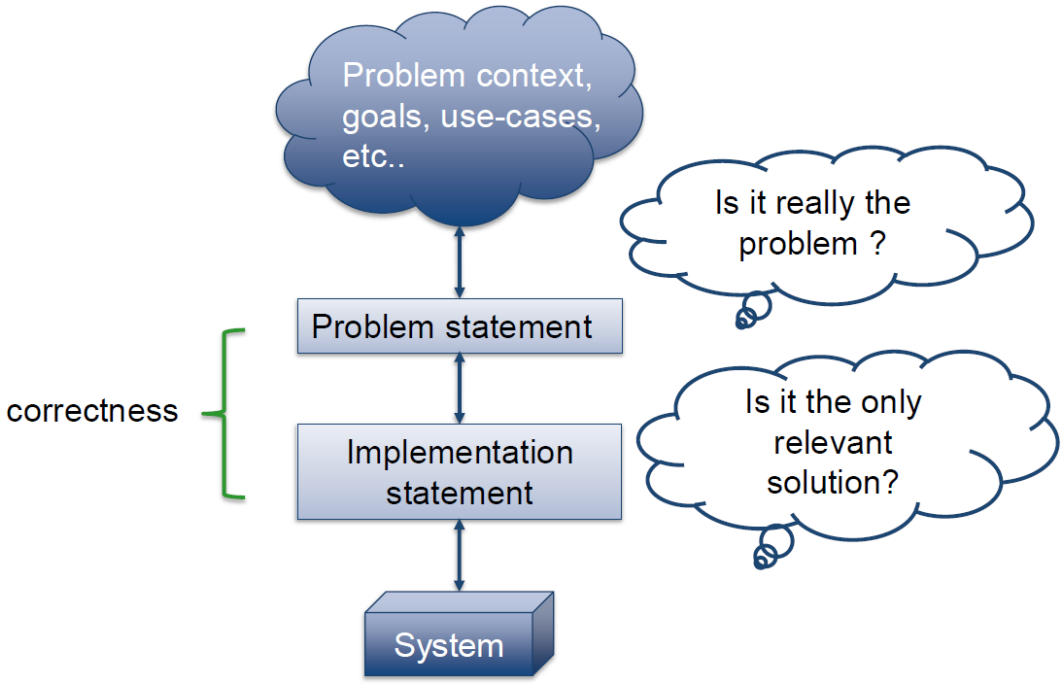


Figure II.2: Basic principle of *Requirement Engineering* is to separate the problem from the solution, an adapted picture from [Easterbrook, 2004].

The Customer requirements, standards, internal documents, error reports of legacy systems and the requirements of system in operation are translated into system

requirements. These are overall requirements of the system referring to more than one sub-system. Different types of requirements captured when a product is visioned. The requirements are majorly classified into two types namely, *Functional* and *Non-functional* requirements. The system requirements lead to system architectural design using standard descriptive languages like SysML or UML. In this thesis we use an model-based open source environment called Capella to capture the system level requirements. Since the inception of ISO 26262, along with system requirements, *Safety requirements* are captured to specify complete automotive system that adheres to the functional safety. Further the system requirements are split into hardware requirements , software requirements, mechanical requirements and so on. Typically the software requirements are derived from the system requirements either as textual or graphical (*model*) way on a requirement engineering environment, for examples IBM® Rational® DOORS®, Siemens Polarion® etc.

II.2 Model-Based Design (MBD)

As defined by INCOSE (International Council on Systems Engineering) [Haskins et al., 2006], Systems Engineering is an engineering discipline whose responsibility is creating and executing an interdisciplinary process to ensure that the customer and stakeholder's needs are satisfied in a high quality, trustworthy, cost efficient and schedule compliant manner throughout a system's entire life cycle. In the case of Model-Based System Engineering, shortly MBSE, model drives the system development. Model-Based Design shortly MBD is a development methodology to build the safety-critical (examples are automotive, aerospace etc.) embedded software based on *model*. In MBD, design becomes a central activity or a driving force behind the whole software product as depicted in the Figure II.1, the shaded grey color region.

II.2.1 Model - Definition

A model is an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system, i.e. an abstraction of an original. The model can be used to answer questions about the original system [Stachowiak, 1973]. A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [Bézivin and Gerbé, 2001]. According to the modeling language terminology, a model is a description of (part of) a system written in a well-defined languag [Kleppe et al., 2003].

In MBD, *control model* represents the embedded control software that controls the physical world represented by a *plant model*. These models typically consist of algorithms, functions and logical components and the over-all architecture comprising controller and plant are often analyzed by simulation. Upon desired simulation and analysis, a typical MBD flow recommends further the auto-code generation out of controller model and the same executes on the target hardware also referred as ECU.

II.2.2 General-Purpose Modeling Practices

The General-Purpose Modeling (GPM) is a widely used practice that uses general-purpose modeling language to depict the system. GPM helps to represent the components of a system (hardware, software mechanical, electrical and so on), life-cycle, processes etc. Examples of GPM languages are UML [Rumbaugh et al., 2017] (an industry standard for modeling the software), XML (a data modeling language, for example AUTOSAR XML also termed as *arxml* for ECU extract).

II.2.3 Domain-Specific Modeling Languages

The Domain-Specific Modeling Languages (DSML) focus on particular domain [Fowler, 2010; Ghosh, 2011] to give lot of programming advantages to practitioners. In contrast to general-purpose modeling languages, Domain-Specific Modeling Languages require less efforts and fewer low-level details to specify a system. Domain-specific modeling languages directly capture the high-level concepts of a subject domain. Eventually, DSMLs improve the efficiency of models as they are easier to understand and learn for a domain expert, but also more easily transformable, analyzable, etc. The use of a DSML is restricted to a particular domain, meaning that many DSMLs are necessary to cover all the aspects of a system.

II.2.4 Architectural Modeling

Architectural modeling is a blueprint of the automotive system to be developed. Architectural modeling is contrast to *Physical modeling* described in the next section to model mainly the architectural diagrams to illustrate the static (also called structural) and dynamic behavior of the system along with interfaces (input and output data). The Systems Modeling Language shortly SysML provides very rich and advanced expression means covering a very broad spectrum of applications, spanning from high-level architecture modeling to detailed design at the frontier of simulation. SysML is a general-purpose modeling language, an extension of UML for systems engineering applications. Inspired by SysML concepts, the *Arcadia/Capella* [Capella, 2019] used in this thesis focuses on the design of systems architectures.

II.2.5 Physical Modeling

Physical modeling helps to model automotive control system, which is a combination of physical, mechanical, electrical, hydraulic and electronic domains. Physical modeling environment provides access to physical elements, such as sensors and actuators of the system. For example, the blockset of Simulink called Simscape^{TM} enables to rapidly create models of physical systems. With *Simscape* we build physical component models based on the physical connections that directly integrate with block diagrams and other modeling paradigms. We model systems such as electric motors, bridge rectifiers, hydraulic actuators, and refrigeration systems by assembling fundamental components into a schematic. Simscape add-on products provide more complex components and analysis capabilities. The physical modeling supports simulating the multi-domain physical systems with control algorithms in Simulink. We use Simulink as physical modeling environment to host CPAL in the thesis.

II.2.6 Prescriptive and Descriptive Modeling in Industrial Practice

Rothenberg [Rothenberg et al., 1989] distinguishes between two purposes for models: descriptive models that are build to describe or explain the world and prescriptive models built to discover and prescribe optimal solutions. In other words, descriptive models are for understanding, communicating, and predicting and tend to be highly abstrative. On the other hand, prescriptive modeling is to model the specifications, and tend to be sufficiently detailed so that the intended system can be implemented. UML, SysML, ans Capella generally serve as descriptive modeling environments. Simulink, ASCET, and CPAL work as prescriptive modeling environments.

II.2.7 Generative and Interpretative Modeling in Automotive Control Function Design

Two spectrum of modeling methodologies namely *Generative* and *Interpretative* exist to develop automotive control function. Automotive suppliers and car manufacturers widely use *Generative* approach, since code and other artifacts are automatically generated from the model. The other fundamental methodology to achieve applications from models is *Interpretative*. Though, to the best of our knowledge, the technique of model interpretation remains unexplored in the automotive domain, it can facilitate and speed up the development, deployment and timing verification of applications with real-time constraints running on potentially complex hardware platforms. Verification also can be done more easily as defects will be caught earlier in the process since there is no difference between the model and the executable program.

II.3 MBD to Develop Real-time Control Software

Automotive control system employs typically Simulink, a MBD environment, which provides the capability to design and simulate the complete control system, including the control algorithm in addition to the physical plant. Simulink is especially useful for generating the approximate solutions of mathematical models that may be prohibitively difficult to solve "by hand." For example, consider that we have a nonlinear plant. A logical step is to linearize the plant for approximation and then designing the control algorithm using analytical methods. Simulink can then be employed to simulate the performance of the controller when applied to the full nonlinear model. Together with MATLAB, Simulink blocks can help to handle this non-linearity nature of the plant, which is the typical case of automotive applications.

II.3.1 Control Software Development

The first step in the controller design is to express the system in terms of mathematical model, for instance using differential equations. The mathematical model. These mathematical models are derived either from physical laws or from experimental data. Automotive control systems are dynamic, in the sense that system reactions changes over time and not fixed. For many systems, this can be stated as a set of first-order differential equations. We can represent any continuous linear time-invariant

(LTI) systems, using standard State-space representation or using Transfer Function representation. In the thesis, we use state-space representation of the system as below. Once appropriate mathematical models of a system is obtained, either in state-space or transfer function form, we may then analyze these models to predict how the system will respond in both the time and frequency domains. Control systems are often designed to achieve and maintain stability, lowest possible steady-state error, faster response (rise time) and or of minimal oscillations. In this thesis, we will show how to determine these dynamic properties from the system models when we discuss about stability property.

The system is comprised of a controller model, a plant model and platform model. Plant P is modeled by a continuous-time system of equations

$$\begin{aligned} \dot{x} &= Ax + Bu, \\ y &= Kx, \end{aligned} \tag{II.1}$$

where x is the plant state and u is the control signal. The plant output y is sampled periodically with some delays at discrete time instants. The control signal is updated periodically with some delays at discrete time instants, (i.e., actuation also happens with some delay). Quantities A, B, K are constants. The controller model is comprised of a task set Γ of n periodic tasks $\{T_1, \ldots T_n\}$ executing on a single processor.

II.3.2 Link between Control Theory and Real-time Scheduling

A control loop consists of three main parts: data collection, algorithm computation, and output transmission. In the simplest case the data collection and output transmission consist of calls to an external I/O interface, e.g. AD and DA converters or a field-bus interface. In a more complex setting the input data may be received from other computational elements, e.g., noise filters, and the output signal may be sent to other computational elements, e.g., other control loops in the case of set-point control. In most cases the control is executed periodically with a constant period, or sampling interval, T (often denoted h), that is determined by the dynamics of the process that is controlled and the requirements on the closed loop performance.

The basic control loop has two timing constraints. The first is the period which should be constant, i.e., without jitter. The second constraint involves the computational delay from the input data collection to the output transmission. This is also known as the control delay or the input-output latency. From a control point of view this delay has similar effects as a pure time delay on the input of the controlled process. An overview of control loop timing constraints is given in [Törngren, 1998]. Different approaches are possible for the control delay. The simplest approach is to implement the system in such a way that the delay is minimized and then ignore it in the control design. The second approach is to try to ensure that the delay is constant, i.e., jitter free, and take this delay into account in the controller design. One way of doing this is to wait with the output transmission until the beginning of the next sample. In this way the computational delay becomes close to the sampling period. If the controller is designed with discrete (sampled) control theory it is especially easy to compensate for this delay. However it is also relatively easy to compensate for delays that are shorter than the sampling period. The third approach is to explicitly design the controller to be robust against jitter in the computational delay, i.e., the delay is

treated as a parametric uncertainty. This approach is, however, substantially more complex than the first two.

II.3.3 Real-time Scheduling and Schedulability Analysis

In a real-time system, a process or task has schedulability indicates that the tasks are accepted by a real-time system and completed as specified by the task deadline depending on the characteristic of the scheduling algorithm. Modeling and evaluation of a real-time scheduling system concern is on the analysis of the algorithm capability to meet a task deadline. A deadline is defined as the time required for a task to be processed. A system is said to be unschedulable when tasks can not meet the specified deadlines. The performance verification and execution of a real-time scheduling algorithm is performed by the analysis of the algorithm execution times. Verification for the performance of a real-time Scheduler will require testing the scheduling algorithm under different test scenarios including the worst-case execution time.

Essentially all real-time scheduling algorithms are based on estimations of the worst-case execution time (WCET) of scheduled tasks. A WCET should provide an estimation that is a tight upper bound of the actual WCET. That is, WCET estimations should be close to, but no lower than, the actual WCET. Each controller task T_i is represented by a tuple T_i: (O_i, C_i, h_i, D_i), where O_i is the task's release offset, C_i the Worst-Case Execution Time (WCET), h_i the task's period and D_i the deadline. R_i^w and R_i^b are the worst and best-case response times. The task instances, also referred to as jobs, are scheduled non preemptively in order of their arrival. Each controller task is assumed to have three activities in the order sensing, computation and actuation. Sensing is the first activity which *reads* the data from a sensor. The computation also known as *control law execution* is the second activity. The actuation is the last activity which *writes* the data to physical devices.

II.3.4 Timing Contract Models

Control and computing are two disciplines involved in the design phase with different objectives in mind. One of the control system objectives is stability, for instance under transient and steady state analysis, system stability goals are related to achieving desired set point time, desired settling time and fewer harmonics (i.e.,) more functional properties of the controller that act on the plant. On the other hand computing system objectives are related to platform and resource constraints, timing characteristics, safety, power consumption etc. In the past, substantial work has been carried out to study the effect of timing irregularities on control performance [Cervin et al., 2003a]. In industry, the practice is that the requirements are gathered as functional and non-functional specifications. Due to the engineering process followed (different stakeholders from different backgrounds), it has been noted that there are substantial gaps during the design phase. In the past, attempts were made to close the gaps between control and computing using different approaches [Lampke et al., 2015; Ziegenbein and Hamann, 2015]. In this thesis, we propose timing contract analysis integrated with modeling environment to address the design gaps.

A contract is a set of assertions, namely assumption and guarantee. Contracts can be a crucial concept in component based design [Benveniste et al., 2012], because it

drives and synergizes the design thinking of the stakeholders from different domains. Thus, the contract can be a candidate for bridging the gaps that naturally occur when different worlds of people work for the system development. The horizontal contract can be applied to the components in the same phase of a Software Development Life Cycle (SDLC) and can be formulated to any of the phases ranging from specification to testing. The vertical contract is applied between two different phases of a software life cycle. In a nutshell, a contract can be enforced to any phase of the SDLC. In this paper, we apply contract to a context which is more of vertical nature, (i.e.,) how do we abstract the low-level platform constraints from the high-level design requirements. We define composite contract in this work which is a conjunction of two viewpoint contracts namely stability (functional) and schedulability (non-functional). These viewpoints are derived from fundamental *timing contracts* as discussed in [Derler et al., 2013].

Timing contract is a tuple of timing constraints that arise between control and computation. Timing contract can be embedded into Model Based Design (MBD) where both control and software designers agreed to accept each other's constraints (assumptions), thereby achieving each other's objectives (guarantees). Following are the timing contracts evolved in the real-time control computing over the years.

- **ZET contract**:

  Zero Execution Time contract (shortly ZET) is an ideal situation under control theory. During every sampling period instance, the sensing, control, and the actuation, all happen at the same time. This implies that no sampling jitter and computation happens at no time. Also, this means that the actuation happens instantaneously to the sensing. This is an impossible situation under computing viewpoint because computation consumes time. The addition of a new task to the existing task model is straightforward, because of input to output delay of existing tasks is not altered. The tasks are executed sequentially, concurrent execution of tasks does not happen.

- **BET contract**:

  Bounded Execution Time contract (BET) is a practical version of timing contract with respect to the actuation under software engineering perspective, in which output can happen anytime with in the period. This is not a favorable point for the control theory which expects the time-in-variant assumptions. BET may also assume that sensing happens exactly at the periodic instances which again not practically possible always(sampling jitters are quite natural, especially for low priority tasks). The addition of a new task to the system (even though no interaction with existing tasks) will affect the input output delay of the existing tasks.

- **LET contract**:

  Logical Execution Time contract (LET in short-form) is a combination of ZET and BET taking both control and computing aspects into account. The time variance part that exists in BET is now obsolete. Actuation happens exactly at the end of the period and makes control theory to assume constant sensing to actuation delay. The instances of output are delayed and buffered to use scheduling mechanisms. LET may also assume the zero (or negligible) sampling

jitter in order to strengthen further the control assumption, which is not the practical case where multiple tasks exist in a complex control system.

- **TOL contract**:

  Timing Tolerance contract (shortly TOL) is similar to BET when we consider actuation part of the system. Because the output is available in varying time (i.e) sensing to actuation delay is not constant. In addition to this, the time varying sampling jitters which are evident in multitasking environments, are included in the TOL contract. Small variations of sensing and actuation jitters do not degrade control performance much. Figure VI.4 shows how TOL contract can be defined for simulation and analysis in a modeling paradigm.

II.3.5 Timing Tolerance Contract (TOL)

Among the four different timing contracts discussed in Section II.3.4, the more pragmatic one is timing tolerance (TOL) contract where both software engineering and control engineering are considered holistically. A TOL contract is specified by a tuple (h, τ, J^h, J^τ), where h and τ are the period and sensing to actuation delay (StA) respectively. J^h and J^τ are the acceptable variations in sampling period and StA respectively. Outside to these values of J^h and J^τ system tends to behave unstable (i.e) poor performance withe the respect to control system context. Based on following assumptions $J^\tau \leq \tau$ and $J^h + \tau + J^\tau < h$.

Then the TOL contract is defined as

$$\begin{aligned} t_k^s \epsilon [k.h, k.h + J^h] \\ t_k^a \epsilon [t_k^s + \tau - J^\tau, t_k^s + \tau + J^\tau]. \end{aligned} \tag{II.2}$$

and k-th state update happens before t_{k+1}^s,for all k. In the work [Derler et al., 2013], focus was to define timing contract with variation parameters that are acceptable to ensure expected performance of the system, for instance control performance. Here in this thesis, TOL contract is further extended to include with the low level scheduling objectives thereby to formulate a composite contract that helps the designer when new function to be integrated in to the existing run-time environment, whether it maintains its properties stability and schedulability. Let $\mathbb{C}^f$ is the contract defining stability property. Here we consider transient response and the steady state response to validate the stability property of the system. For a step input signal $step(t)$, the control task T_i and sampling period h_i, four output parameters namely settling time ST, overshoot OS, rise time RT, steady-state error SS_{error} are calculated. As in Equation (II.2), assumptions are made. Along with guarantees related to stability a functional contract can be exemplified as below:

$$\mathbb{C}^f = \begin{cases} \text{variables} & : \begin{cases} \text{inputs} : C_i, h_i, step(t) \\ \text{outputs} : ST, OS, RT, SS_{error} \end{cases} \\ \text{types} & : ST, OS, RT, SS_{error} \in \mathbb{R} \\ \text{assumptions} & : \begin{matrix} t_k^s \epsilon [k.h, k.h + J^h] \\ t_k^a \epsilon [t_k^s + \tau - J^\tau, t_k^s + \tau + J^\tau] \end{matrix} \\ \text{guarantees} & : \begin{matrix} ST(5\%) \leq 10s \\ OS \leq 10\% \\ RT \leq 10s \\ SS_{error} \leq 5\% \end{matrix} \end{cases} \tag{II.3}$$

With $\mathbb{C}^f$ system stability is verified, now to include schedulability aspect, let us define the system is comprising of a task set Γ made up of n periodic tasks $\{T_1, \dots T_n\}$ executing on a single processor. Each task T_i is represented by a tuple

$$T_i \colon (O_i, C_i, h_i, D_i),$$

where O_i is the task's release offset, C_i the worst-case execution time, h_i the task's period and D_i the deadline. R_i^w and R_i^b are the task worst and best-case response times. The task instances, also referred to as jobs, are scheduled non preemptively in order of their arrival. The timing contract for scheduling aspect $\mathbb{C}^t_{SCHED}$ is as follows:

$$\mathbb{C}^t_{SCHED} = \begin{cases} \text{variables} & : \begin{cases} \text{inputs} : O_i, C_i, h_i, D_i \\ \text{outputs} : R_i^w, R_i^b \end{cases} \\ \text{types} & : O_i, C_i, h_i, D_i, R_i^w, R_i^b \in \mathbb{R} \\ \text{assumptions} & : \begin{matrix} t_k^s \epsilon [k.h, k.h + J^h] \\ t_k^a \epsilon [t_k^s + \tau - J^\tau, t_k^s + \tau + J^\tau] \end{matrix} \\ \text{guarantees} & : Feasible\ task\ set \end{cases} \tag{II.4}$$

The framework discussed in the Chapter VI (Model-driven Co-design Workflow) uses the timing tolerance contract for defining the framework to address the design gaps between control and software engineers.

II.4 Envisioning Automatic Scheduling Configuration

Real-time scheduling is now a mature and well established research field. Many scheduling policies and results have been proposed and derived over the last four decades providing efficient scheduling solutions for most hardware platforms and application-level needs. Tools and frameworks have been developed implementing these scheduling algorithms and their analyses [Altisen et al., 1999]. To the best of our knowledge, apart from few early works in that direction for specific execution platforms (e.g. [Navet and Migge, 2003]), none of these frameworks target the

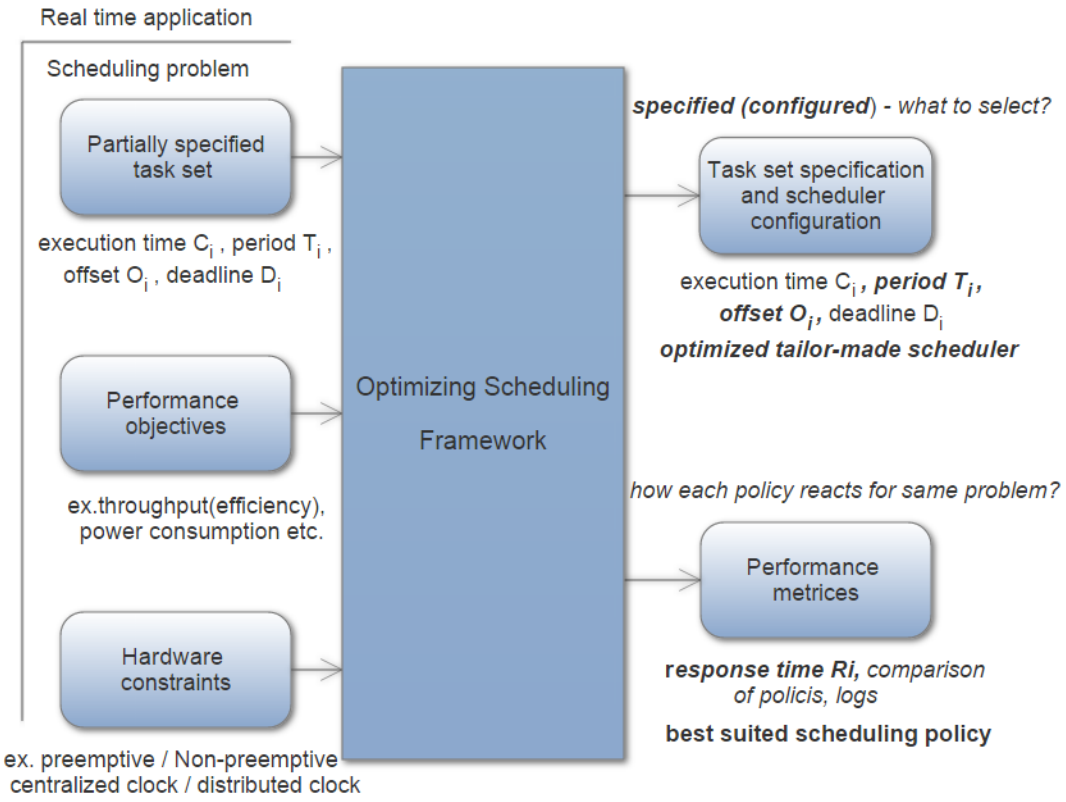


Figure II.3: Inputs and outputs of the framework.

automatic configuration and selection of the best suited policy and parameters in a systematic manner.

In this chapter, we present an optimizing framework that selects the best suited scheduling configuration for a partially specified task set and the given system constraints. In Figure II.3, we illustrate the structure of the framework. The inputs to the framework are:

- a partially specified task set (see Section II.4.1),
- the performance objectives, and
- the hardware constraints.

The framework performs the selection of the policy, optimization of the scheduling parameters, and outputs

- the complete task set specification and scheduling configuration,
- the performance metrics of the different scheduling algorithms.

This work is motivated by the ongoing research on timing-augmented Model-Based Design [Navet et al., 2016a] at the University of Luxembourg. Our aim is to develop the framework such that the system designer only focuses on the high-level timing

behavior of the system, where the implementation choices of the low-level timing behavior are taken care of by the framework. The framework fits in the early design phases as a device to automate system synthesis and hide away from the designer the complexity of the underlying run-time environments.

II.4.1 Defining the Framework

The inputs is the high-level description of the scheduling problem, whereas the output is the scheduling solution with all the required configuration parameters. In this section, we define inputs and outputs of our optimizing framework.

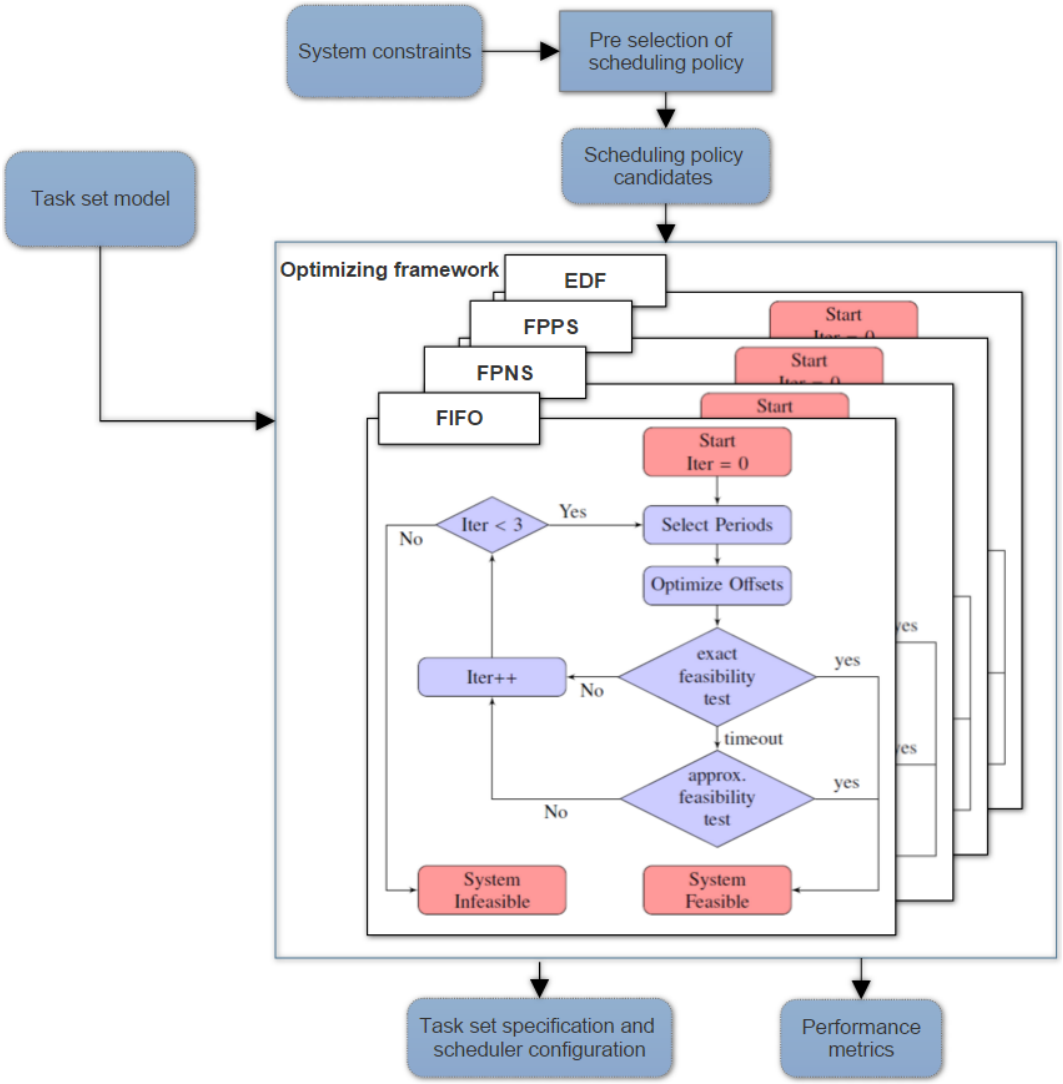


Figure II.4: Flow of the scheduler synthesis framework.

Partially specified task set:

We assume that the application is composed of n tasks $\{\tau_1, \ldots \tau_n\}$. The task set is defined partially to reflect the freedom in the selection of certain parameters.

Each task τ_i is specified by a tuple

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

a) where C_i is the worst-case execution time and is assumed to be given as single value,

b) the execution period T_i is potentially defined as a range of permissible values,

c) the deadline relative to the release time of the task, denoted by D_i, is given as single value. A range of values for the deadline would be futile, as the run-time environment must ensure the system is feasible with respect to the most stringent deadline.

Performance Objectives:

Examples for performance objectives are throughput (efficiency), power consumption, predictability constraints such as requirement of uninterrupted execution for a task, minimal activation or end-of-execution jitters, etc. These objectives are achieved for example by minimizing the periods within the allowed range in order to reduce the power consumption. If for instance throughput is to be maximized, the frequency of execution is increased.

Hardware Constraints:

The selected hardware further constraints the choices of the framework. Typical hardware, and more generally constraints of the run-time environment, are number of processing cores, preemptiveness, type of the system clock (e.g., global clock or distributed clock in the system). These inputs are accounted by the framework in the derivation of the scheduling solution.

Scheduling Configuration:

The main outcome of the framework is the scheduling solution, that is the complete specification of the task set and the scheduling configuration. This scheduling configuration covers all low-level configuration parameters and no further input is needed to execute the application on the target system. In addition to the policy, scheduling parameters include periods, offsets and possibly deadlines and priorities.

Performance Metrics:

The performance of the candidate algorithm is evaluated by selected performance metrics. Typical performance metrics are schedulability (a task set is schedulable or not under a policy), numerical values of the response times and jitters, power-consumption, or the ability of the system to grow further measured for instance by the minimum slack time.

II.4.2 Scheduler Synthesis

In this section, we explain the core of the framework, i.e., the scheduler synthesis. In contrast to other approaches towards scheduler synthesis [Grenier and Navet, 2008],

we do not generate new or non-standard scheduling policies. Instead, we focus on the selection of the most appropriate scheduling policy (including parameter optimization) amongst a set of well-studied and widely-implemented real-time scheduling policies. In Figure II.4, we illustrate the general steps of our framework.

In a first step, the framework performs a pre-selection of the scheduling policies on the basis of the system constraints and the hardware to execute the application. All policies that violate some of the requirements are excluded at this step. Policies that are compliant with the requirements under some side-constraints are considered with those side-constraints. This step results in the set of candidate policies which are subject to the actual parameter optimization.

The next step, the parameter optimization is then highly specific to the policy, and thus has to be performed for each policy individually. Also, the type of parameters to be optimized differ. However, we can build on a large variety of existing methods and techniques. For the selection of the periods, for instance, we can use a recent work by Nasri et al. [Nasri and Fohler, 2015a], offsets in case of offset-aware policies can be optimized using [Monot et al., 2012] and for the selection of priorities, we have optimality results such as [Audsley, 2001].

Real-time scheduling problems are in most contexts NP-hard. Due to the computational complexity of the problems, an optimal scheduling solution cannot be guaranteed. However, the candidate optimization techniques and heuristic algorithms have proven to be robust and lead to satisfactory solutions in many application domains (e.g., [Monot et al., 2012; Nasri and Fohler, 2015a; Navet and Migge, 2003]).

II.4.3 Illustrating Example

We illustrate our approach using the following task set Γ:

	C_i	T_i	D_i	constraints	objective
τ_1	1	[4 : 5]	4		reduce period
τ_2	2	[4 : 8]	8	non-preemptive	reduce period
τ_3	6	[15 : 24]	24	-	

Constraints and Scheduling Policies

A side constraint besides meeting deadlines is that task τ_2 has to be executed non-preemptively and the objective is to minimize the values of the periods of τ_1 and τ_2 (*i.e.*, increase frequency to achieve a better control of the system). To simplify the example, we restrain ourselves to a limited number of well-known scheduling policies: earliest deadline first (EDF), both preemptively and non-preemptively(EDF-NP), fixed-priority preemptive scheduling (FPP), fixed-priority non-preemptive scheduling (FPNP), and FIFO:

EDF	EDF-NP	FPP	FPNP	FIFO

The policy selection identifies that all policies can indeed satisfy the system constraints, but in case of the preemptive policies, i.e., EDF and FPP, further constraints are required to ensure the non-preemptive execution of τ_2:

EDF	EDF-NP	FPP	FPNP	FIFO
✓ if $D_2 \leq D_i$	✓	✓ if $pr_2 = 1$	✓	✓

All non-preemptive policies fail since the execution time of τ_3 exceeds even the largest permissible period of τ_1. Hence, the search has to concentrate only on the two remaining policies EDF and FPP (both with the appropriate side constraints).

EDF Scheduling

Using EDF we are able to achieve a processor utilization of 1 and execute tasks τ_1 and τ_2 with the smallest possible periods. This is the optimal result and it will be selected as the scheduling solution with the following parameters for the task set:

	C_i	T_i	D_i
τ_1	1	4	4
τ_2	2	4	4
τ_3	5	20	20

EDF is optimal scheduling policy for this particular scheduling problem considered as example. The modern embedded real time application experiences hardware related overheads, e.g. Cache Related Preemption Delays (CRPDs). As discussed in [Lunniss et al., 2014] if these CPRDs are considered in to account, then the benefits of EDF become negligible. This will eventually change the optimal result achieved in this illustrating example to a different scheduling policy which fits to the hardware we have. As a future work, we have planned to include these scheduling overheads in to the framework.

II.5 Chapter Conclusion

This chapter presented the fundamental concepts and the definitions that are necessary to comprehend the issues tackled in this thesis and proposed contributions regarding methods and tools to develop real-time control software. The thesis envisions an optimizing framework that considers as inputs a partially specified task set, the performance objectives of the system and the constraints of the run-time environment, importantly the hardware support. The framework synthesizes the scheduling solution that best meet the requirements. Our framework currently includes a number of basic real-time scheduling policies and ongoing work is devoted to enrich the set of available policies with customized scheduler that make optimal use of underlying execution hardware and improvements in system behavior. This work is a contribution towards a more automated design process building on the wide set of techniques and results developed within the real-time system community.

The applicability and precision of the framework is determined by the optimization algorithms and schedulability analyses. But these algorithms and heuristic techniques are limited in precision. Consequently, for some scheduling problems, the framework cannot guarantee optimality. A future work is to develop techniques such as lower bounds to estimate how far is a solution computed with the framework from the optimal solution.

Chapter III

Lean Interpretation-based MBD to Design Automotive Software

Abstract

MBD is a common practice in the automotive industry to develop real-time software based on the system constraints and requirements. In this setting, MBD provides indispensable means to model and implement the desired functionality to verify and validate the functional and non-functional requirements. Current industrial practice in MBD completely relies on Generative, i.e., code generation to bridge the gap between model and implementation. An alternative approach, although not yet used in the automotive domain is Interpretative, the direct execution of the design models using interpretation engine running on top of the hardware. We present a case study to investigate the applicability of model interpretation, in contrast to code generation, for the development of engine control systems. To this end, we model an engine cooling system, specifically the calculation of the engine-coolant temperature, using interpreted model based development, and discuss the benefits and low-lights compared to the existing code-generation practice.

III.1 Automotive Software Development Life-cycle

We begin this chapter by explaining the state-of-the-art practices of development of an automotive function for instance an engine management system. These are commonly developed using *Model Based Design* (MBD) technique. The engine is controlled by an Electronic Control Unit (ECU) that contains software functions for different sub-systems. The stakeholders requirements of the these engine functions are specified in one of the Application Life-cycle Management ALM suites and traced until its realization as ECU.

In ALM, the different tools are integrated to develop and maintain the software. For example, IBM has an ALM suite called IBM Rational Team Concert (RTC) where Rational DOORS is the requirements management tool that captures all the functional and non-functional requirements. These requirements are analyzed further to design the engine function. Popular Model Based Design (MBD) tools are MATLAB/Simulink (MLSL) from Mathworks, ASCET-MD from ETAS, and SCADE Suite from ANSYS. These industrial MBD tools further generate code for

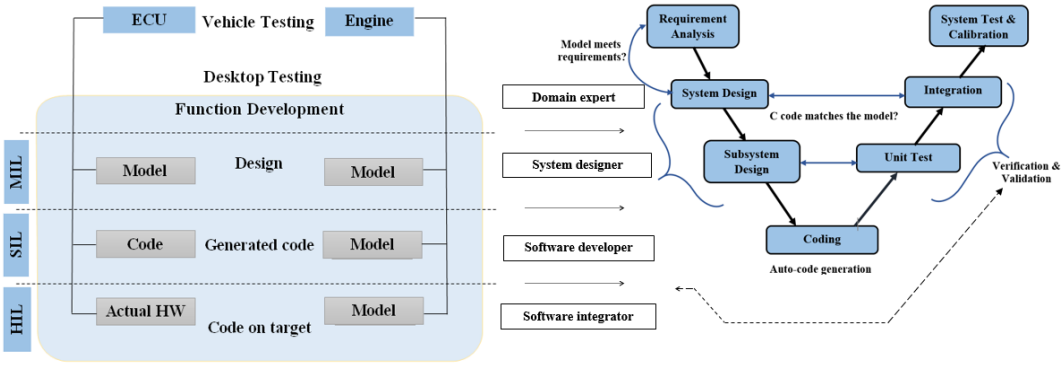


Figure III.1: Engine function development flow - Illustration of verification techniques, involved stakeholders and development phases.

engine functions using code generators. Each engine control function is further (unit-) tested and integrated into the ECU.

Figure III.1 shows the typical software function development flow practiced in the automotive industry. The system model of the engine captures the ideas and requirements. The model is an executable specification and can be simulated and rapid-prototyped to explore different design options. In the existing approach, the modeling environment is primarily used to describe the domain problem, in this case the engine function to be developed against the functional requirements. *Domain experts* and *Software designers* are involved in this phase. The controller model is tested in a simulation environment (which includes the plant model, *i.e.* the engine) and this testing is called *Model-in-the-Loop* (MiL) testing to ensure that the model meets the requirements.

In the next step, the code is generated from the model using a code generator. Then, the code is verified under an engine model. This phase is referred to as *Software-in-the-Loop* (SiL) testing. *Software developers* are involved to test each engine function individually using unit testing. Next, the function is integrated with other existing engine functions in the integration phase by the *Software integrators*, typically a tier-one supplier. The complete engine software is then ported to the ECU hardware, which can be verified using a *Hardware-in-the-Loop* (HiL) testing system, such as PT-LABCAR, which realistically emulates vehicles I/Os.

In the current practice [Broy et al., 2010], the execution environment on the target is different from the execution within the modeling environment in terms of I/Os, scheduling and even in terms of generated code. Indeed, the target-generated code will be optimized towards the platform and thus be as efficient as possible. On the negative side, the build tool-chain must be available, and it takes a substantial amount of time to produce an executable program from the designed model (build time can require several 10s of minutes). Simulink and its block sets (like Simscape, Stateflow etc.) are examples for modeling environment and Embedded coder is an example of the code generator for production code generation on a specific target processor. The generated code can be further customized to meet the requirements (e.g., with respect to safety). In the automotive software development, there is a

high probability for mixed-mode development, where generated code is integrated with manually-developed functions.

III.2 Model as Code - A CPAL Introduction

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 and RATP in France, and in UK and other countries through the Westlock interlocking system from Westingshouse. Surprisingly to the best of our knowledge, except above applications and some industrial automation (PLCs) model interpretation has not been widely explored, albeit it possesses a number of key advantages such as: the model can be directly uploaded on the target, there is no difference between the model and the code, the total software size is greatly reduced both off-line and on the target, hardware independence is ensured, etc.

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-metal 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. CPAL and its associated tools are jointly developed by our research group at the University of Luxembourg and the company RTaW since 2012. The CPAL documentation, graphical editor and execution engine for Windows, Linux, embedded Linux, and Raspberry Pi are freely available for all uses at `http://www.designcps.com`. A BMMI port of CPAL is available for Freescale FRDM-K64F boards.

III.2.1 Providing High-level Abstractions for Embedded Systems

Model-driven development is an enabling technology to fill the programming languages gaps. But still existing languages lack the high-level abstractions and automation features that would make them more productive. In addition, the design and development of embedded systems, especially ones with dependability constraints, necessitates the use of many best practices. None of the programming languages we are aware of are well suited to make the development of safe and provably correct embedded systems as quick and easy as possible.

CPAL has been designed to support the formal description, the editing, graphical representation and simulation of cyber-physical systems. Simulation is a key activity for the understanding, debugging of models and for the verification of correctness properties such as timing constraints. The main requirement when designing CPAL was to natively provide the high-level abstractions which are familiar in the domain of embedded systems, and a need to express the same in an unambiguous and concise manner domain specific patterns of functional behaviors as well as non-functional properties. A process denotes the core language entity to implement a recurrent activity with 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 execution). 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

- A key objective behind CPAL is to let designer state the permissible timing behavior of the system in a declarative manner while a system synthesis step involving both analysis and optimization then generates a scheduling solution which at run-time is enforced by the execution environment.
- Real-time scheduling mechanisms: processes are activated with a user-defined period, possibly with an offset relationship with each other, and additional execution conditions such as for instance 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.
- Communication channels to support data flow exchanges between processes, and reading/writing to hardware ports.
- Introspection mechanisms that enable processes to query at run-time their execution characteristics such as their activation rate and activation jitters.

III.2.2 CPAL to Model and Develop Control Function

Programming embedded systems implies interactions with the environment through sensors and actuators piloted by I/O ports. In CPAL, global variables and collections can be mapped to I/O ports. By default, the 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, 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()*. The CPAL programs on the embedded hardware can access GPIO, DAC (Digital-to-Analog converter), accelerometer, I2C bus and serial communication etc.

III.3 Modeling of AUTOSAR-compliant Control Software

The engine cooling system is an important part of the automotive vehicle. It is responsible for maintaining optimum operating temperature. The coolant is circulated through the engine block with the help of an electric water pump. The coolant will reduce the temperature of the engine block and then will run through the radiator equipped with a fan to remove waste heat.

III.3.1 Engine-coolant Temperature Calculation

Figure III.2 shows the physical layout of the engine-coolant temperature calculation which is considered as the use case to present our modeling approach. The engine-coolant temperature sensor plays an indispensable role in the engine cooling system. Precise information about the temperature is essential due to various reasons: The data are used by the engine control unit to adjust the fuel injection and ignition

timing. Further, the temperature value is used to control the cold starting of the engine, to control the calculation of the fuel quantity, and to control the fan speed of the electric cooling radiator. This data is also used to provide readings of the coolant temperature gauge to the dashboard to protect the engine from over-heating.

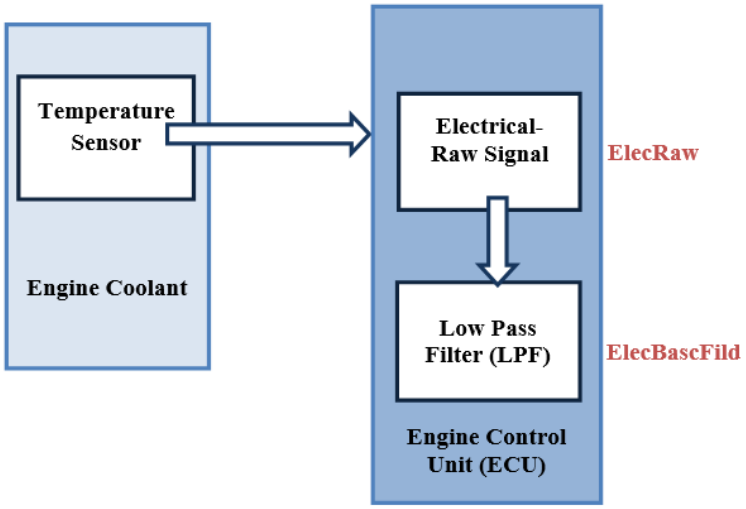


Figure III.2: Physical layout of an AUTOSAR compliant engine-coolant system function - Engine coolant temperature sensor connected to an ECU.

The engine-coolant temperature sensor is connected to the engine ECU through an analog to digital pin. The electrical output is obtained from the sensor that monitors the temperature of the engine-coolant. As per AUTOSAR design pattern [AUTOSAR, 2015] catalogue for standard sensors, the overall system consists of 3 modules as depicted in Figure III.4. *Sensor/Actuator Components* are special AUTOSAR software components which encapsulate the dependencies of the application on specific sensors or actuators. The AUTOSAR architecture takes care of hiding the specifics of the micro-controller (this is done in the micro-controller abstraction layer, MCAL, part of the AUTOSAR infrastructure running on the ECU) and the ECU electronics (handled by the ECU-Abstraction layer, also part of the AUTOSAR Basic Software).

III.3.2 Architecture of Sensor Actuator Design Pattern

AUTOSAR Sensor Actuator signal flow is depicted in the Figure III.3 that explains the way hardware components are mapped on to corresponding software layers. For example, the figure depicts Sensor software component (SWC) is a software equivalent of a physical sensor, ECU abstraction is maps to the ECU electronics and so on. The architecture of the engine-coolant temperature calculation function involves 3 AUTOSAR software components:

Electrical Device Driver Layer (DrvrSnsrElec):

The electrical value from the temperature sensor is read through the input pin and stored in the variable *ElecRaw*. The raw electrical signal (*ElecRaw*) is

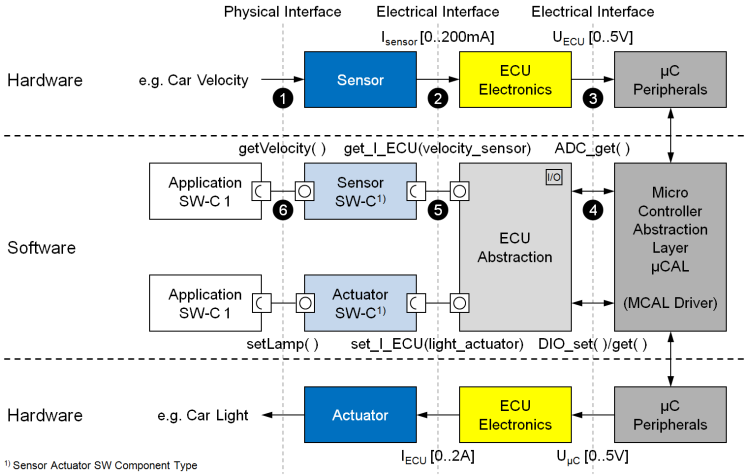


Figure III.3: Sensor actuator signal flow as described in AUTOSAR [AUTOSAR, 2015].

rugged against signal faults using the Low Pass Filter (*LPF*) and the filtered raw electrical signal (*ElecBascFild*) is obtained.

Sensor Device Driver Layer (DevDrvrSnsr):

At this stage, the raw electrical signal is converted into its physical temperature value (*Raw*) using a lookup-table, where the corresponding value is provided. The temperature value of the filtered electrical signal (*ElecBascFild*) is also obtained from the lookup-table and is provided to the next layer.

Virtual Device Driver Layer (DevSnsrVirt):

In this layer, the possible signal range check, electrical errors, cable interruption and sensor faults that may occur are identified. This is done in order that incorrect values from the sensor are not taken into account for the calculation in case of sensor malfunctioning. Other errors such as a cable interruption, short circuit to battery or sensor voltage saturation can also be detected and appropriate flags will be set:

- *ElecBascFildbit* - The electrical validity bit shows that the sensor raw value is electrical valid.
- *ElecBascFildbitCommon* - The common validity bit shows that the engine-coolant temperature as a whole is valid and can be transfered to the application Layer. Based on the temperature values calculated in this layer, the obtained temperature value (*Measd*) is compared with the estimated value (*Estimd*) from the application layer. This comparison determines the validity of the calculated value. If valid, the final temperature value (*Consld*) is sent to the application layer.

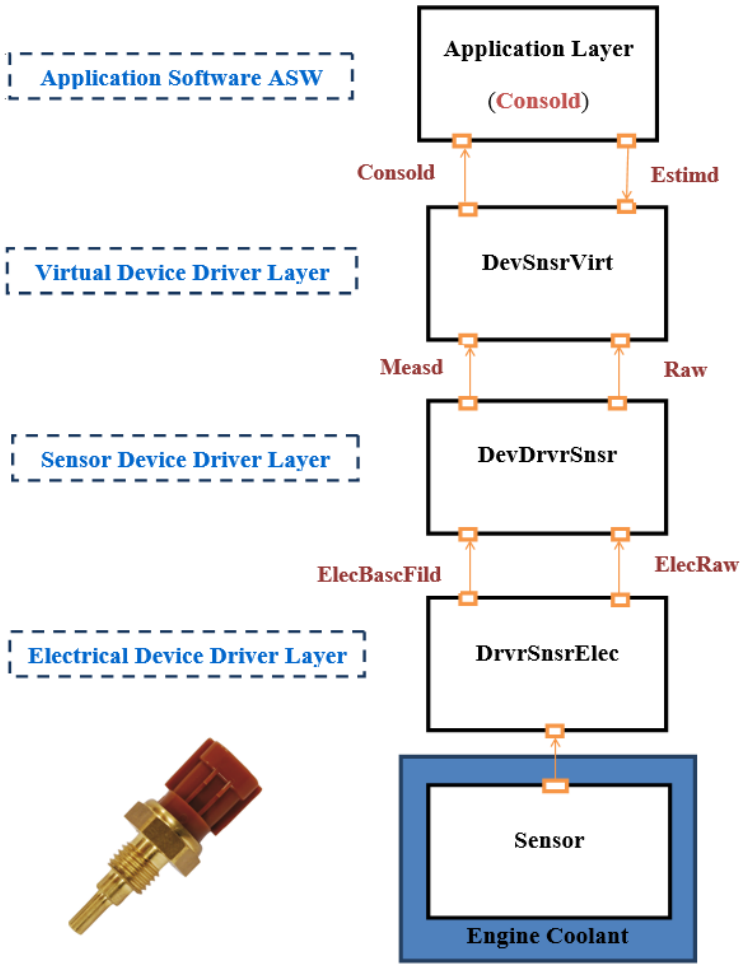


Figure III.4: AUTOSAR design pattern for a standard sensor.

III.4 A Proposed Lean MBD Approach

To the best of our knowledge, model interpretation for automotive function development has not been explored and experimented in the past. In case of model interpretation, a generic model-interpretation engine is implemented which executes the model of the engine function. As shown in Figure III.5, the modeling environment includes the execution environment. Hence, the executable artifacts (*i.e.*, model and execution engine) are available within this environment. The model interpretation can be launched within the development environment or on a target platform. In the latter case, the interpretation can run on top of an OS or directly on the hardware. There are two possible interpretation modes: simulation and real-time. Simulation mode is suited for the use in the design phase, where execution should be as fast as possible, which implies that the activation frequencies of the processes are not respected and they execute (conceptually) in zero time. Typically, executing in simulation mode is several orders of magnitude faster than in the real-time mode. Real-time mode is for the execution of the program with the actual desired temporal behavior of the application.

```
power(in float32: a,in uint32: b,out float32: pow)
{
  var uint32: i=0;
  var float32: pow1=1.0;
  while(i<b)
  {
    pow1=pow1*a;
    i=i+1;
  }
  pow=pow1;
}
log(in float32: x,out float32: lnx)
{
  var float32: taylor_s=0.0;
  var float32: temp=0.0;
  var uint32: i=0;
  var float32: temp_pow=0.0;
  while(true)
  {
    taylor_s=taylor_s+temp;
    i=i+1;
    power((x-1.0)/(x+1.0),2*i-1,temp_pow);
    temp=temp_pow/float32.as(2*i-1);
    if(i>1000)
    break;
  }
  lnx=2.0*taylor_s;
}
var bool: pin0_out;
var bool: pin1_out;
var uint32: flag;
var uint32: adcvalue;
var uint32: modelvalue;
var uint32: digit;
var uint32: mode;
var float32: ElecRaw;
var float32: Raw;
var float32: Temperature;
var uint32: model[36]=
{60,65,70,75,80,85,90,95,100,105,110,115,120,125,130,135,140,145,150,

155,160,165,170,175,180,185,190,195,200,205,210,215,220,225,230,235};

processdef Electrical_Layer(in channel<uint32>: input,out float32:
     eraw)
{
  state Main
  {
   var uint32: data;
    while (input.not_empty())
    {
      digit=input.pop();
      if(digit==101)
     {
      break;
     }
      else
     {
      data=data+digit-48;
      data=data*10;
      flag=1;
     }
    }
```


Listing III.1: Textual description of the `engine coolant` application.

```
if(flag==1)
   {
   data=data/10;
   adcvalue=data;
   eraw=float32.as(adcvalue);
   adcvalue=data-320;
   data=0;
   flag=0;
   }
   }
}
processdef Physical_Layer(in float32: eraw,out float32: raw)
{
state main
{
if(uint32.as(eraw)!=0)
{
var float32: volts; var float32: ohms; var float32: lnohm;
var float32: a =  0.002197222470870;
var float32: b =  0.000161097632222;
var float32: c =  0.000000125008328;
var float32: t1; var float32: c2; var float32: t2;
var float32: temp1; var float32: tempc;
var time64: time1; var time64: time2;
/*log variables*/
  var float32: taylor_s=0.0;
  var float32: temp=0.0;
  var uint32: i=0;
  var float32: temp_pow=0.0;
volts=(eraw*3.7)/1024.0;  /*3.3*/
ohms=((1.0/volts)*2900.0)-1000.0; /*3300*/
time1 = time64.time();
IO.println("NTC Thermistor resistance:%f",ohms*20.0);
/*log calculation*/
  while(true)
  {
    taylor_s=taylor_s+temp;
    i=i+1;
    power((ohms-1.0)/(ohms+1.0),2*i-1,temp_pow);
    temp=temp_pow/float32.as(2*i-1);
    if(i>1000)
    break;
  }
lnohm=2.0*taylor_s;
time2 = time64.time();
t1=b*lnohm;
c2=c*lnohm;
power(c2,3,t2);
temp1=1.0/(a+t1+t2);
tempc=temp1-273.15-4.0;
raw=tempc;
IO.println("Engine Coolant Temperature in Celsius:%f",raw);
}
else
raw=0.0;
}
}
processdef Virtual_Layer(in uint32:d,in float32:raw,out float32:T)
{
state main
{
```

Listing III.2: (Continued) Textual description of the `engine coolant` application

```
var float32: raw_float;
var uint32: rpm;
raw_float=raw*(9.0/5.0)+32.0;
rpm=uint32.as(((((float32.as(d)*3.3)/1024.0)/(10.0/1000.0))
    *1.8+32.0)*40.0-2400.0) ;

IO.println("Engine Coolant Temperature in Farenheit:%f,model temp
    -%u,rpm-%u",raw_float,model[rpm/200],rpm);
if(float32.as(model[rpm/200])-10.0<raw_float and float32.as(model[
    rpm/200])+10.0>raw_float)
{
pin0_out=true;
pin1_out=false;
Temperature=raw_float;
IO.println("Modeled value");
}
else
{
pin0_out=false;
pin1_out=true;
Temperature=float32.as(model[rpm/200]);
IO.println("Real value");
}
}
}
var queue<uint32>: ttyTemperature_in[2000];
process Electrical_Layer: DrvrSnsrElec[50ms](ttyTemperature_in,
    ElecRaw);
process Physical_Layer: DevDrvrSnsr[100ms,10ms](ElecRaw,Raw);
process Virtual_Layer: DevSnsrVirt[200ms,20ms](modelvalue,Raw,
    Temperature);
```

Listing III.3: (Continued) Textual description of the `engine coolant` application.

To ensure that the simulation reflects the real-time behavior on the target platform, timing annotations (e.g., execution time latencies, jitters, etc) can be introduced in simulation mode. Those timing annotations can be derived from measurements on the target architecture, from WCET analysis and, possibly, by schedulability analysis if other software components can interfere with the function under development. Timing accurate simulation thus provides benefits to identify faults in design phase itself, earlier, thus than with the traditional design process.

As the model itself can be executed, no additional artifacts are needed, and, unlike in the traditional generative MBD, no target specific code is generated. Instead, the specifics of the platform are taken care by the interpretation engine. Further steps of the application development, such as compilation of source code to object code and the linking stage to produce the executable program, are not required.

III.4.1 Usecase : Engine-coolant Control Function in CPAL

The model of the engine-coolant system is developed in the CPAL (Cyber Physical Action Language, see [Altmeyer et al., 2015; Navet et al., 2016a]), which is an embedded system language to model, simulate, verify and program Cyber Physical

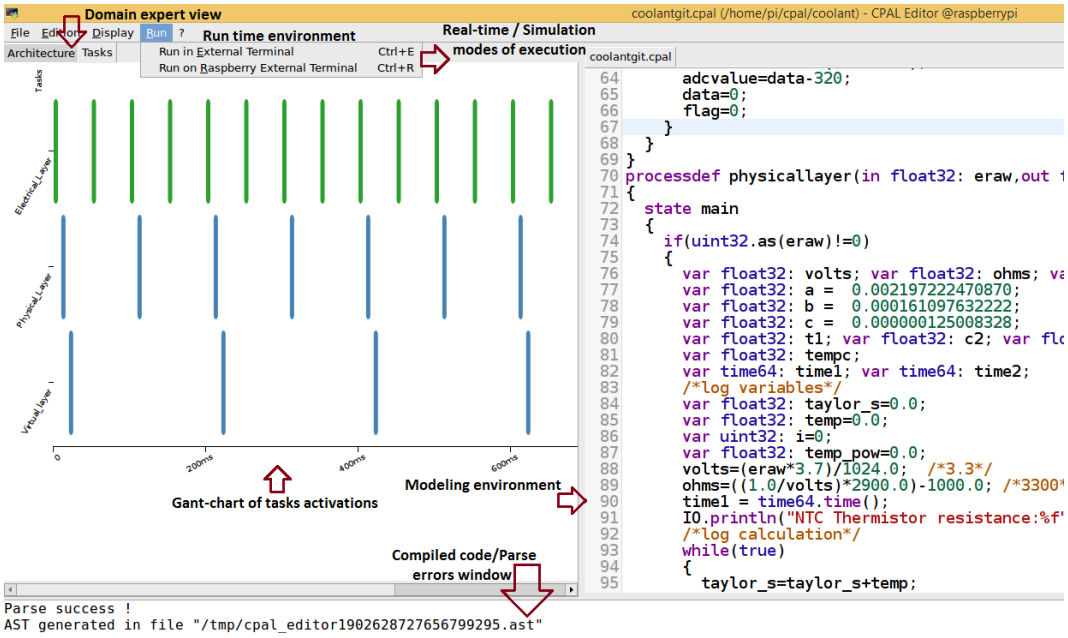


Figure III.5: An integrated environment, here the CPAL-Editor, with the code of the model, the Gantt chart of the processes activations and the possibility to execute the models in simulation and real-time mode both locally or on a target.

Systems. CPAL[1] is a language jointly developed by our research group at the University of Luxembourg and the company RTaW.

The model-based environment of CPAL consists of a single integrated development environment, i.e., the CPAL-Editor. The CPAL editor, combines the design, simulation, execution (both locally and on a target), visualization of the functional architecture and execution chronogram in one integrated environment. The model-interpretation engine is specific to the target platform. This interpretation engine can be executed on top of an operating system or without an operating system, the latter being called Bare-Metal Model Interpretation (BMMI). CPAL BMMI is available on the NXP Semiconductors Freedom-K64F, a low-cost development platform which is form-factor compatible with the Arduino R3 pin layout. The experiments in this study are performed on a Raspberry Pi equipped with a multi-core ARM Cortex-A7 processor operating at 900 MHz running Raspbian OS.

A typical engine-coolant temperature sensor can measure in the range $-40°C$ to $+150°C$. In our case study, we have considered a Negative Temperature Coefficient (NTC) type sensor with an operating voltage as 3.3V. Figure VIII.3 shows the experimental setup which aims to mimic the engine cooling system. The MCP3008 is an external ADC interface which is connected to the sensor. Since the sensor operates with the thermistor principle, a voltage divider circuit with 3.3V reference is added. ADC data from MCP3008 is communicated to the processor using the Serial Peripheral Interface (SPI). The sensor software component is modeled according to

[1]The CPAL documentation, graphical editor and the execution engine for Windows, Linux and Raspberry Pi platforms are freely available from `http://www.designcps.com`.

the AUTOSAR design catalog described in Section III.3. The speed of the electric fan is controlled based on the measured temperature.

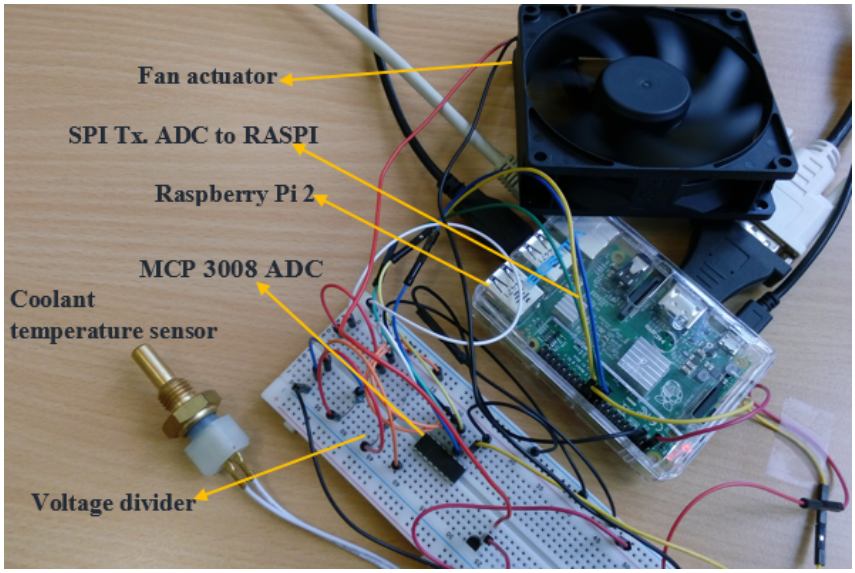


Figure III.6: Experimental set-up - Sensor interfacing to hardware.

Out of the two possible CPAL execution environments (*i.e.*, bare-metal or hosted by an OS), we use the interpretation engine on top of an OS (Raspbian on Raspberry Pi) which can also execute in real-time, although with a lesser real-time predictability than the bare-metal implementation. The engine-coolant temperature is calculated by the sensor software component modeled in CPAL. Figure III.7 shows the sample run-time environment where simulation and real-time execution are performed. Both interactive and non-interactive executions are possible. The interactive mode of execution is useful in program analysis and debugging. In interactive mode, the user has different execution options, such as a step-by-step execution, or uninterrupted execution for a pre-defined duration.

Since it is an interpretation-based execution environment, the user can list and change the values of global variables at run-time, as well as execute additional code statements. In non-interactive mode, the program is executed indefinitely or for a specified duration without requiring additional user inputs.

III.4.2 Comparison of Generative MBD and Interpretative MBD

From the case-study experience, we present our proposed development flow for function development. Figure III.8 shows the development flow of model interpreted approach to develop an engine function.

Model Interpreted Development Steps In the first step all functional, non-functional including timing requirements of the engine function are collected. These are further analyzed by domain experts. The specifications are implemented in the CPAL (step 2 - system design in Figure III.8). During the development, as soon as the function model is updated the functional architecture, and other views created

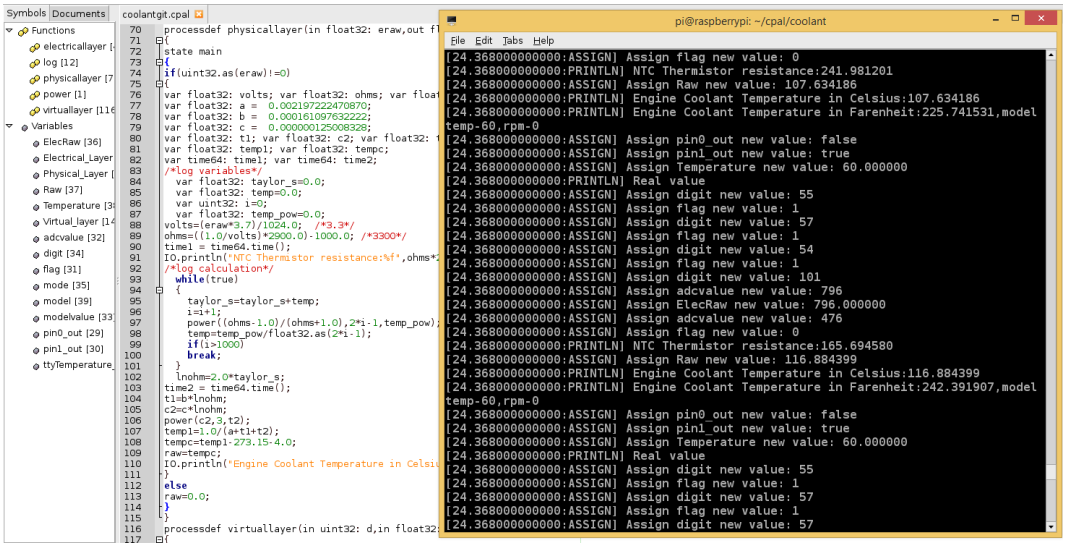


Figure III.7: CPAL model and execution environment under real-time mode.

out of the model such as execution Gantt charts, are automatically updated too (step 3) which is done in the background along with the modifications. This allows the designer to immediately visualize and understand the effects of the changes made, without the need for building the executable and running it in debug mode. The latest version of the model is always available to execute, be it in simulation mode or real-time mode, locally or on a target. Typically performed once the simulation is satisfactory (step 4), the execution in real-time mode (step 5) helps the designer to assess the performances on the target, enabling rapid-prototyping. If simulation or execution in real-time mode highlights faults, the model is refined in an iterative process. From the development of the engine-coolant temperature calculation function, we here summarize the benefits and differences against the existing generative MBD approach.

Adapting to Requirement Changes is Faster The most important benefit of model interpretation is that changes in the model do not require an explicit regeneration/rebuild/retest/redeploy step. This shortens significantly the turnaround time and, in some scenarios, the overall change management process (how changes in the requirements are implemented). Although it is not available in CPAL yet, it would be possible for models to be updated at run-time, without the need to stop the running application, hence improving productivity. Also, since no artifacts are generated, the build times can be also reduced. Depending on the specific use case, an interpreter combined with model can even require less memory than generated code.

Finding Failures in Model is Easier Failures during the testing phase, after all modules have been integrated, expose problems that are clearly in the model, since the model itself is executed. Unlike with code generation, there is no need to trace back from the generated artifacts where the failure occurred in the model,

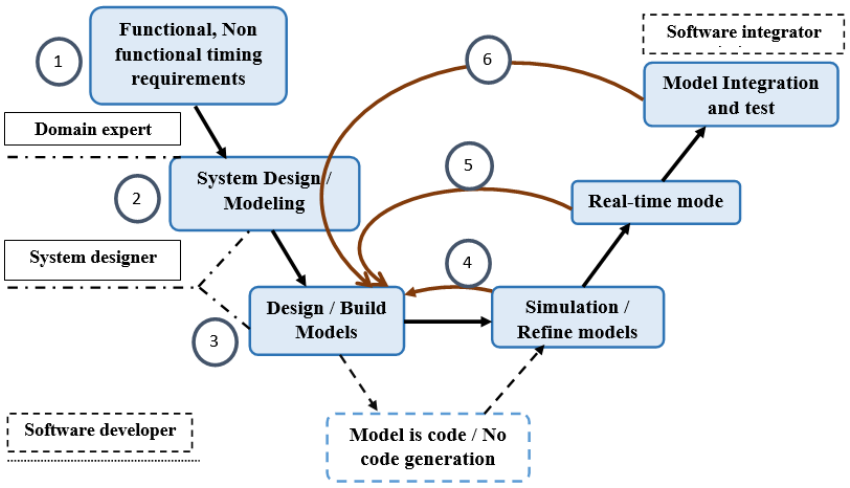


Figure III.8: Model interpreted engine function development flow - steps and stakeholders involved.

which is often hard. On the other hand, debugging models at run time is possible. Since the model is available at run-time, it is possible to debug function models by stepping through them at run-time (e.g., we can add breakpoints at the model level). When debugging at model level is possible, domain experts can debug their own models (e.g., step-by-step) and adapt the functional behavior of an application based on this debugging. This can be very helpful when, for example, complex control or data-flows are involved.

Portability and Hardware Independence Portability is another advantage of model interpretation. An interpreter in principle creates a platform independent target to execute the model. By rewriting only the hardware-specific components, it is possible to develop an interpreter which runs on multiple platforms, as it is the case for CPAL. In case of code generation, we need to make sure we generate code that is specific to the platform. In case of model interpretation, the interpreter handles the platform-specific adaptation.

A notable advantage of the model interpretation is that it hides the complexity of the hardware platform away from the programmer making it easier to configure the run-time environment and deploy the application. Indeed, easier deployment is an important difference. When code generation is used, we often see that we need to open the generated source code in an Integrated Development Environment (IDE) to analyze the program and build it from there to create the final application. In case of BMMI, we just have to upload the model and reset the target, or, when the interpreter is hosted by an OS, execute it within the development environment or in command-line (possibly on a target through a script). Hence, it is much easier for domain experts to deploy and test an application, instead of only modeling it.

Benefits of Single Integrated Environment The important difference between interpreted approach and generative is that domain experts and software developers can work together around a single integrated environment and on a single model. As shown in Figure III.9, the integrated modeling environment provides a graphical

view of the architecture of the designed function model. This model can be used by domain experts for functional analysis and verification, and by software engineers to do function development and testing from day one on.

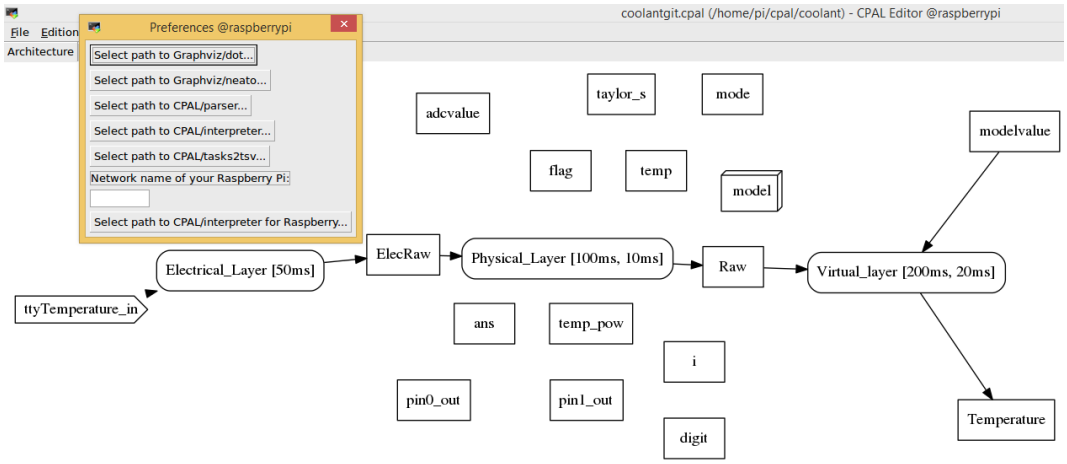


Figure III.9: Software architecture of the coolant temperature calculation.

III.5 Conclusion and Perspectives

Code generation is the widely used practice in the industry for MBD of embedded systems, and this holds true in particular for engine function development. In this chapter, we discuss a model-interpretation development flow that is exemplified with the development of an engine coolant temperature calculation by an AUTOSAR compliant software architecture. By comparison with the usual development chains relying on code-generation and based on the case-study, we discuss the benefits of model interpretation which includes simplicity, productivity and early-stage verification possibility, specifically in the time dimension. For instance, CPAL already provides the basic mechanisms to offer timing-realistic simulation early in the design process. Our ongoing work is on a method to automate the derivation of the temporal quality-of-service required by a software module and, leveraging on model-interpretation, enforce it at run-time.

Although model-interpretation brings advantages, it is not going to cover all use-cases because interpretation is intrinsically significantly slower than compiled code. There are ways to mitigate this drawback in production code such as calling binary code from interpreted code (e.g., legacy code or specialized functions) or, possibly, selectively generating code for the computation-intensive portions of the model. Interpretation and code generation are often seen as two alternatives, not as a continuum. However, one may also imagine relying on model-interpretation, and benefits from the associated productivity gains, until the function/ECU meets all functional requirements, and then switch to code-generation for production code. This remains to be investigated in future works.

From the this chapter discussion, we draw a spectrum of four development methodologies, which is ranging from *full interpretation* to *compiled code*. These are CPAL

only, CPAL with functions hard-coded in the execution engine and linked with it, logic in CPAL and some processes as external C code and full C code generated from CPAL. In most of the systems, the last two options are faster in execution to provide best performance.

Chapter IV

CPAL Co-simulation for Timing-aware Prescriptive Modeling

Abstract

This chapter deals with prescriptive modeling using CPAL and its scheduling and timing annotations. Three approaches supported by tools, namely TrueTime, T-Res, and SimEvents, have been developed in the literature to facilitate the evaluation of how timing latencies affect control performance. However, these approaches support the simulation of control algorithms, but not their actual implementation. In this chapter, we present a model interpretation engine running in a co-simulation environment to study the control performances while considering the run-time delays in to account. Experiments on controller tasks with injected delays show that our approach is on par with the existing techniques with respect to simulation. We then discuss the main benefits of our development approach that are the support for rapid prototyping and the re-use of the simulation model at run-time, resulting in productivity and quality gains. The proposed timing-aware prescriptive modeling using CPAL also eases the design and verification of embedded real-time systems.

IV.1 Latency Sensitive Control Software Development

The innovation in the field of embedded systems has been increasingly relying on software-implemented functions. The control laws of these functions typically assume deterministic sampling rates and constant delays from input to output. However, on the target processors, the execution times of the software will depend on many factors such as the amount of interferences from other tasks, resulting in varying delays from sensing to actuating. MBD has been profoundly reshaping and improving the design of software-intensive embedded systems. Traditionally, model-driven development (based on code generation) is deployed in the automotive industry. Code generation is used to generate code from a higher level model and create a working application.

As mentioned in [Broy et al., 2013], MBD is being used for series development by a majority of the automotive companies. Especially in development phases software

design and implementation model-based design is used intensively. As mentioned in [Tankovic et al., 2012], this kind of MBD used by automotive suppliers and car manufacturers is called as *generative MBD*, since code and other artifacts are automatically generated from the model. The other fundamental approach to achieve applications from models is *interpreted MBD*. Interpreted MBD can be seen as a set of platform independent models that are directly interpreted by an execution engine running on top of the hardware, with or without an operating system.

The fact that models can be directly executable helps a great deal as the development cycle time can be shortened; and there is no distortion between the model and what is executed. Though, to the best of our knowledge, the technique of model interpretation remains unexplored in the automotive domain, it can facilitate and speed up the development, deployment and timing verification of applications with real-time constraints running on potentially complex hardware platforms. Verification also can be done more easily as defects will be caught earlier in the process since there is no difference between the model and the executable program. In this chapter, we present a case-study to evaluate how *interpreted MBD* can be applied to an automotive software development scenario.

In this chapter, we present a model interpretation engine running in a co-simulation environment to study control performances while considering the run-time delays in to account. Introspection features natively available facilitate the implementation of self-adaptive and fault-tolerance strategies to mitigate and compensate the run-time latencies. A DC servo controller is used as a supporting example to illustrate our approach. Further the chapter is structured as follows. Section IV.2 provides a survey of the state of the art practices. Then, Section IV.3 describes our co-simulation approach, where the controller model is designed in CPAL and the plant model is designed in Matlab/Simulink (MLSL). In the same section, we present an automotive servo controller as the use-case, implemented in CPAL and running in the Simulink environment. In Section IV.4, we compare our model-based co-simulation timing analysis against existing approaches. Then Section IV.5 concludes the chapter.

IV.2 Review of Real-time Control System Co-simulation

Powerful Model-Based Development (MBD) tools such as MATLAB/Simulink (MLSL) and ASCET/MD are available for the design and development of control systems. On the other hand, there are dedicated tools such as MAST and PyCPA for analyzing and configuring real-time scheduling algorithms. Three approaches with associated tools that are presented hereafter go in the direction of a combined approach, *i.e.* to support control system design considering the influence of scheduling strategies.

SimEvents MATLAB/Simulink is a multi-domain industry standard for the design of control systems. Simulink Control Design supports the design and analysis of control systems. "SimEvents" is a discrete-event simulator that can be used as blocks in Simulink [SimEvents, 2019] to perform system-level simulation. It provides options to create tasks, and is able to inject network and scheduling delays with the support of the basic scheduling policies FIFO, LIFO and fixed priority scheduling. Other policies like EDF are left to the user to program. To the best of our knowledge, a significant drawback of MLSL "SimEvents" is that one cannot generate code from the

system model consisting of "SimEvents" blocks. Hence, the actual realization of the controller model using "SimEvents" is not feasible. "SimEvents" is meant to model various applications where models are driven by events, starting from operation research and manufacturing processes to real-time systems. Because of this generality, it does not provide all domain-specific concepts needed for real-time systems like those available in TrueTime and T-Res.

TrueTime The MATLAB/Simulink-based tool [Cervin et al., 2003b] enables the simulation of the temporal behavior of controller tasks executed on a multitasking real-time kernel. In TrueTime, it is possible to evaluate the performance of control loops subject to the latencies of the implementation. TrueTime offers a configurable kernel block, network blocks (CAN, Ethernet, etc.), protocol-independent send and receive blocks and a battery block. These blocks are Simulink S-functions written in C++. TrueTime is an event-based simulation using zero-crossing functions. Tasks are used to model the execution of user code. The release of task instances (jobs) is either periodic or aperiodic. For periodic tasks, the jobs are created by an internal periodic timer. Aperiodic tasks can be created in response to external trigger interrupts or network interrupts. The task code is written as code segments in a Matlab script or in C++. It models a number of code statements that are executed sequentially. All statements in a segment are executed sequentially, non-preemptively, and with a simulation time that can chosen by the developer through an annotation.

T-Res This recent tool [Morelli et al., 2014] is also developed using a set of custom Simulink blocks created for the purpose of i) simulating timing delays depending on the code execution, scheduling of tasks, and communication latencies, and ii) verifying their impact on the performance of control software. T-Res is inspired from TrueTime and provides a more modular approach to design the controller model enabling to define the controller code apart from the model of the task.

Besides these three tools introduced previously, the other tools developed in the past are in general, specialized to a certain aspect of the co-design problem. For example, Jitterbug [Cervin et al., 2003b] supports statistical control-performance analysis for a certain class of control systems. Also, these tools and methods focus only on the analysis and simulation level. They help the designer only with the study of system performance under the effects of timing delays, but not the system development. The system designer, then takes these analysis results into account to develop the actual embedded control algorithms in the next steps. This increases the possibility of distortions between the simulation model and the implementation.

In this chapter, we propose a model-interpretation based runtime environment which can be used in a co-simulation environment to analyze the effects of delays on the performance of the control system, with the advantage that the controller model used in the simulation can be executed on the target hardware. This way, we eliminate the gaps between the simulation models and the executables. In the next section, we explain our co-design approach.

IV.3 CPAL Co-simulation to Evaluate Control Performance

Digital controllers are interfaced with sensors and actuators. They are real-time systems since they require inputs and outputs to occur at the right points in time. In automotive applications, for instance, digital controllers control engines, brakes, suspensions, airbags, etc., that are referred to as "plants". The plant is in the physical world and physical quantities are sensed by sensors interfaced to Analog to Digital Converters (ADC). An ADC has two steps, namely signal acquisition and sampling. On the other side, digital controllers with Digital to Analog Converters (DAC) are connected to actuators to control the plant. The real-time computing system is implemented with control algorithms as software functions i) to capture the current (reference) state of the plant using sensors, ii) to compare against the reference or the desired state iii) to control the actuators to reach the desired state of the plant.

The continuous-time signals from the sensors are periodically sampled, each sampled set of data is then processed by the real-time control functions. If the processing is not fast enough with respect to the sampling rate, then some samples of data will be lost and the frequency of the control algorithm cannot be respected. In practice, due to for instance varying task execution times and preemption delays, the input to output latencies will vary over time. These delays will directly impact the quality of the control functions, in the worst-case, possibly jeopardizing the safety of the system. Hence, it is important to consider these delays during the design phase of the control software. To achieve this, techniques and supporting tools based on simulated models have been developed. To the best of our knowledge, none of them however support the actual implementation, and software development work is required later in the development process with the risk that the developed software is not identical to the models.

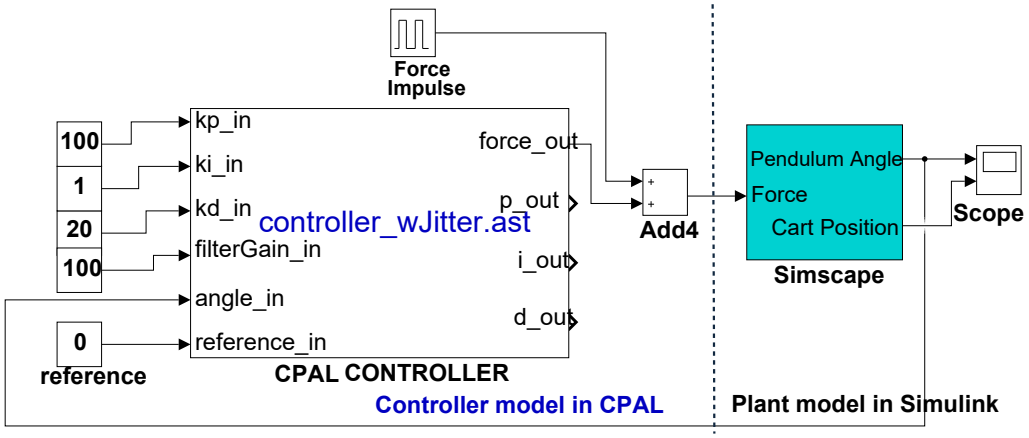


Figure IV.1: A controller model for an inverted pendulum integrated within the Simulink Environment. The CPAL control library is used to design the controller model, input and output control data are visible in the design window and can be changed without the need to access CPAL model. The *ast file* format is the binary equivalent form of the source-code of the controller model. CPAL control library allows to run CPAL code within a Simulink model

```
1
2  [...]
3
4  processdef servo_controller( out PID_data :data, in float32 :r, in
       float32 :y, out float32 : u)
5  {
6     state Main {
7     var float32 : P;
8     var float32 : I;
9     var float32 : D;
10
11    P = data.K*(data.beta*r-y);
12    I = data.Iold;
13    D = data.Td/(data.N*data.h+data.Td)*data.Dold+data.N*data.K*
          data.Td/(data.N*data.h+data.Td)*(data.yold-y);
14
15    u = P + I + D;
16    data.Iold = data.Iold + data.K*data.h/data.Ti*(r-y);
17    data.Dold = D;
18    data.yold = y;
19    }
20 }
21 var PID_data : data_control1;
22 var PID_data : data_control2;
23 var PID_data : data_control3;
24 /* simulink interfaces */
25 var float32 : r_in;
26 var float32 : y1_in;
27 var float32 : y2_in;
28 var float32 : y3_in;
29 var float32 : u1_out;
30 var float32 : u2_out;
31 var float32 : u3_out;
32
33 /* process instances */
34 process servo_controller : servo_controller1[6ms](data_control1,
       r_in, y1_in, u1_out);
35 process servo_controller : servo_controller2[5ms](data_control2,
       r_in, y2_in, u2_out);
36 process servo_controller : servo_controller3[4ms](data_control3,
       r_in, y3_in, u3_out);
37
38 init()
39 {
40    data_control1.K = 0.96;
41    data_control1.Ti = 0.12;
42    data_control1.Td = 0.049;
43    data_control1.beta = 0.5;
44    data_control1.N = 10.0;
45    data_control1.Iold = 0.0;
46    data_control1.Dold = 0.0;
47    data_control1.yold = 0.0;
48    data_control1.h = 0.006;
49    [...]
50 }
51
52 @cpal:time{
53    servo_controller1.period = 6ms;
54    servo_controller2.period = 5ms;
55    servo_controller3.period = 4ms;
56 }
```

Listing IV.1: Textual description of the `automotive servo control` application

IV.3.1 Co-simulation of CPAL in MLSL

CPAL is an embedded-system specific language designed jointly by our research group at the University of Luxembourg and the company RealTime-at-Work. CPAL is also a design-exploration platform to develop Cyber Physical Systems (CPS). It supports a Model-Driven Development (MDD) approach to model, simulate and verify systems. CPAL can be used as a stand-alone simulation environment for CPS under development or it can also be integrated with other simulation environments like MATLAB/Simulink (MLSL). The CPAL documentation, a graphical editor and the execution engine for Windows, Linux and Raspberry Pi platforms are available at `http://www.designcps.com`. The CPAL control library, needed to execute controller models written in CPAL in MLSL, and the models used in this chapter are available at `http://www.designcps.com/wp-content/uploads/cpal-simulink-control-library.zip`.

In a control-system simulation, the controller model controls the plant model. In our proposed co-simulation approach a controller model is designed in CPAL, and the plant model is designed in MLSL. Controllers can easily be designed in Simulink too. But Simulink is not offering possibilities to study the behavior of control loops subject to scheduling and networking delays. Varying execution times, preemption delays, blocking delays, kernel overheads cannot be captured in the standard Simulink environment.

In the case of the co-simulation CPAL/MLSL, Simulink acts as the primary simulator while CPAL executes the controller model as an S-function, and is being called by the Simulink engine. S-functions (system-functions) are high-level programming language description of a Simulink block written in C, C++ etc.. The CPAL control library is implemented as a *mex* (Matlab Executable) file which executes the CPAL controller model. This CPAL controller is a generic execution engine that can run any CPAL model. The CPAL source model is converted to a binary-equivalent representation using the CPAL parser. The Simulink engine interacts with the CPAL model through data flows and control flows. Data flows are for the information exchange between the Simulink engine and the CPAL S-function, while the control flows define when Simulink invokes the CPAL S-function.

Tasks and real-time schedulers are available natively in CPAL. Figure VI.4 shows the way to define the tasks using CPAL, called processes in CPAL. The default CPAL scheduling policy is FIFO, processes are executed in the order of their activation. Non-Preemptive Earliest Deadline First (NP-EDF) and Fixed Priority Non-Preemptive (FPNP) are also supported by CPAL.

Simulation of the plant's dynamic is done by computing model states at successive time steps over a specified duration. This computation is done by a solver provided by Simulink. Since our overall model is discrete, a variable step size solver is used in our co-simulation approach. The rationale behind this choice is that for the timing analysis of real-time control systems, it is necessary to reduce the step size (when needed) to increase the accuracy when model states are changing rapidly during zero crossing events. In the next section, our technique is exemplified on a use-case.

```
include controller.cpal
  /* periodic process with initial offset, 10ms is period, 2ms is offset */
process PIController: invertedPendulumController[10ms,2ms](kp_in, ki_in, kd_in,
                                                           filterGain_in, angle_in, reference_in,
                                                           force_out, p_out, i_out, d_out);
 /* annotations of task timing parameters */
@cpal:time:invertedPendulumController
{
    invertedPendulumController.jitter = time64.rand_uniform(0us,500us); /* input jitter */
    invertedPendulumController.deadline = 8ms;
    invertedPendulumController.priority = 1; /* priority if needed, here only one process it is not applicable */
    invertedPendulumController.execution_time = time64.rand_uniform(0us,2000us); /* input to output delay */
}
```

Figure IV.2: CPAL real-time task model where periods, offsets, execution times, deadlines, and release jitters are specified.

IV.3.2 Case-study: Automotive Servo Control Function

Idle Air Control Actuator also called as Idle Air Control valve (IAC actuator/valve) is a device commonly used in fuel-injected vehicles to control the engine idle RPM. This actuator is essential because during idling (when the driver is not pressing the accelerator pedal), the throttle valve is completely closed, while the engine still needs some air to prevent the engine from stalling.

Engine ECU controls this IAC actuator electrically. The actuator is fitted such that it either bypasses the throttle or operates the throttle valve directly. The actuator consists of a servo motor that controls a plunger which varies the amount of air flowing through the throttle body. The position of the servomotor and hence the amount of air bypass is controlled digitally by the engine ECU. More air means an increase in the idle speed and less air indicates the reduction in idle speed. Servo motors, by definition, run using a control loop and require feedback of some kind to attain a desired state (position, velocity, and so on).

Though different types of control loops exist, PID (Proportional, Integral, Derivative) control loops are commonly used in servo motors. Figure IV.3 shows the CPAL controller model which controls three servos where the controller task is activated with input-to-output delays. The interfaces of the CPAL controller model, basically the inputs and outputs data, are exposed in the co-design model. The input parameters can be easily changed interactively in the design model itself. When using a PID control loop, tuning of the servo motor becomes necessary. Tuning is the process of making a motor respond in the desired way. Tuning a motor can be a challenging process, but tuning has an advantage that, it lets the users have more control over the behavior of the motor. Adaptive step size for controlling the differential (D) part and integral (I) part of PID is easily achievable in CPAL by using the introspection mechanisms. Next section explains such features available in our approach in comparison to existing techniques.

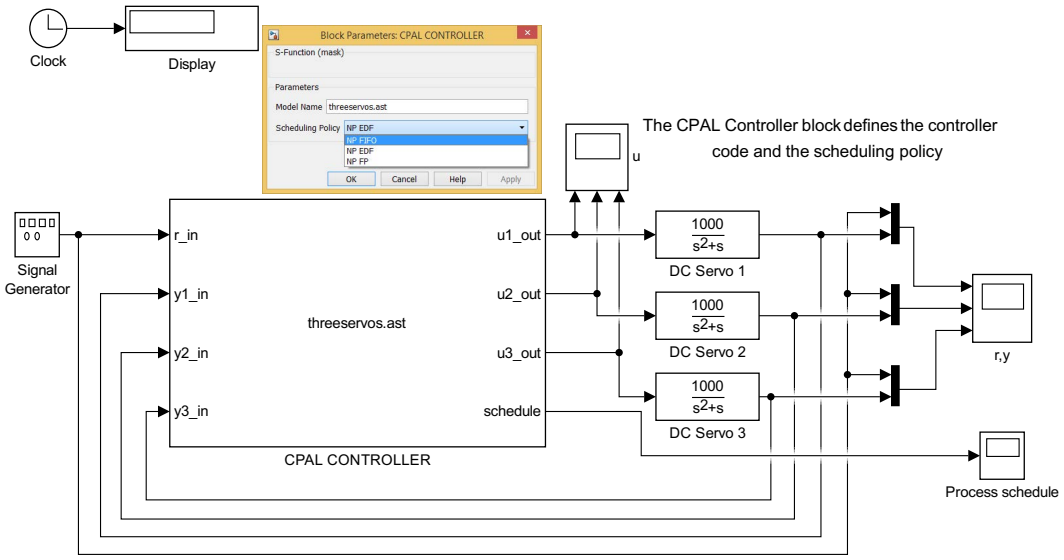


Figure IV.3: CPAL controller model which controls the three servos. The controller tasks are activated with input to output delays. The scheduling policy can be selected interactively and the servo control is specified as a transfer function. The process schedule scope displays the activation pattern of tasks, and the r,y-scope displays the control performances.

IV.4 Comparison of TrueTime, T-Res with CPAL Co-simulation Approach

IV.4.1 Scheduling and Task Model

Unlike TrueTime and T-Res, a real-time scheduling kernel is inbuilt with the CPAL interpreter. Because our ultimate research goal [Sundharam et al., 2016b] is to abstract all the low-level details from the system designer, where designer would define the initial task specifications and performance objectives such as stability, power-consumption, etc.. This chapter is in the direction of system synthesis automation, where acceptable scheduling configurations are identified by evaluating the control performance with the scheduling delays.

The control task model in TrueTime is written in the MATLAB file format or in C++. In T-Res, the task model is a modular and triggered subsystem, executed on the occurrence of a function-call event. Outputs are latched to control the plant. A task in TrueTime and T-Res is a sequence of segments. Every segment is identified by an execution time. Task description parameters such as the task type (periodic or aperiodic), the inter-arrival time (period), the deadline and the execution time are declared in MATLAB which needs to be launched before running the controller model.

In our approach, a process, also called a task, consists of the functioning logic described in the form of a Finite State Machine (FSM). FSMs, possibly reduced to a single state, is repeatedly executed. Several tasks can be executed in parallel in

which case the order of the process executions depends on the scheduling policy. The first step is to define the process, that is its list of parameters and the code itself. Then, one or several instances of the process can be created. These instances will be automatically executed at run-time by the interpreter according to the defined activation pattern.

```
/* Control code */
processdef servo_controller(out PID_data:data,in float32:r,in float32:y,out float32:u)
{   state Main {
    var float32 : P; var float32 : I; var float32 : D;
    P = data.K*(data.beta*r-y);
    I = data.Iold;
    D = data.Td/(data.N*data.h+data.Td)*data.Dold
    +data.N*data.K*data.Td/(data.N*data.h+data.Td)*(data.yold-y);
    u = P + I + D;
    data.Iold = data.Iold + data.K*data.h/data.Ti*(r-y);
    data.Dold = D;
    data.yold = y;   }
/* Timing code -  process instance defines default period, and required parameters */
process servo_controller  :servo_controller3[4ms](data_control3,r_in, y3_in, u3_out);
process servo_controller  :servo_controller2[5ms](data_control2,r_in, y2_in, u2_out);
process servo_controller  : servo_controller1[6ms](data_control1,r_in, y1_in, u1_out);
/* CPAL process timing annotations */
/* CPAL annotations do not require any control code modifications */
@cpal:time{
/*process priority and execution time, larger the number lower the priority*/
    servo_controller1.priority = 3;
    servo_controller1.execution_time =  2ms;
    servo_controller2.priority = 2;
    servo_controller2.execution_time =  2ms;
    servo_controller3.priority = 1;
    servo_controller3.execution_time =  2ms;   }
 /* Task 3 is overwritten for execution time and jitters - to show CPAL capabilities*/
 /* for the experiment in the paper below feature is not necessary and not used */
 /* this annotation gives varying (random) execution time between 1 and 2 ms */
 /* jitter annotation gives variable task scheduling jitter between 0 an 100us */
@cpal:sched:servo_controller3
{   servo_controller3.execution_time = time64.rand_uniform(1ms,2ms);
    servo_controller3.jitter = time64.rand_uniform(0s,100us);
}
```

Figure IV.4: Separation of control and timing aspects in a controller model of the servo motor example. Three servo controllers tasks are defined with associated task parameters. Here, task 3 has highest priority with an execution time and release jitter which are varying. Task 1 has lowest priority with constant execution time. Control code is decoupled from timing definitions given as annotations.

IV.4.2 Decoupling of Timing Definitions from Control Code

In TrueTime, control functionality is combined with timing definitions. There is no separation of concerns and experts in those domains have to work together to evaluate the system performance. On the other hand, T-Res provides a segregated approach, where the control aspects of the controller model are implemented using Simulink blocks and scheduling aspects are dealt with using T-Res blocks. T-Res blocks are again not meant for code-generation and the subsequent steps in development life cycle. As shown in Figure IV.4, CPAL also employs a segregated approach where

control aspects are separated from timing aspects, so that both domain designers can work independently and seamlessly.

IV.4.3 Influence of Scheduling Choices to Control Performance

The Figure IV.4 is a snippet of code of the servo-control example described in previous section, which is originally an example from the TrueTime distribution. This example is here re-used with the same parameters as in [Morelli et al., 2014] so as to check that same results are achieved with our approach. It should be noted that the CPU utilization factor with this parameter set is 1.23, the system is thus overloaded and not all task instances can be executed.

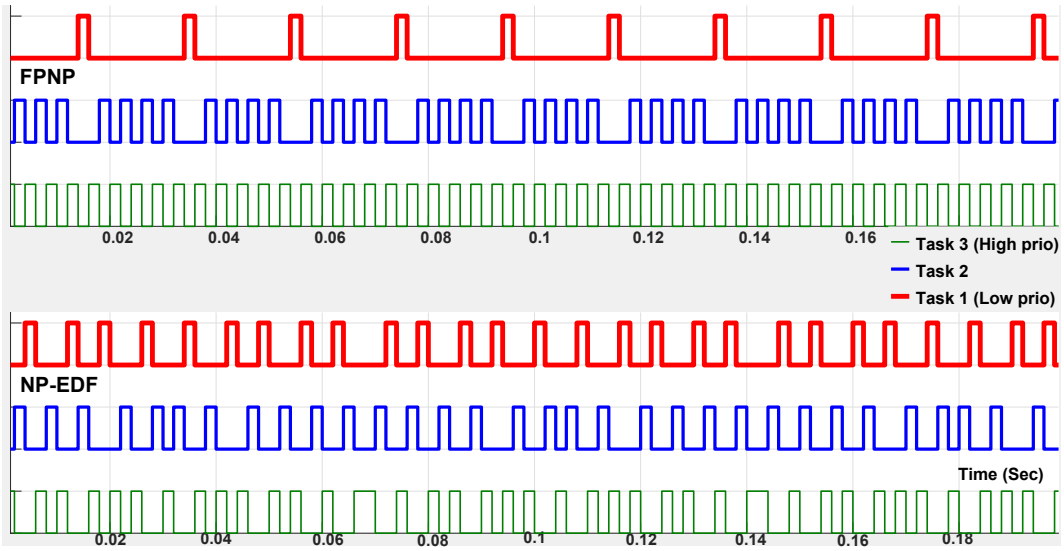


Figure IV.5: Task activations of three tasks with different levels of priorities. Task 1 has low priority with number of activations are less under FPNP scheduling. When NP-EDF becomes the scheduling choice, same Task 1 is activated better.

Figure IV.5 shows the task activation pattern of the controller tasks which control the three different servo motors under the Fixed-Priority Non-Preemptive (FPNP) policy. Task 1 with the lowest priority is activated less frequently than the two others which translates into an unstable control output (see Figure VI.10 top). When NP-EDF is used, all tasks are activated as frequently and the system is stable.

IV.4.4 Self-adapting Mechanisms

In CPAL, it is possible at run-time to query execution characteristics such as process id, period, offset, jitter, priority, deadline and activation time of the current and previous instance of any task. This feature is typically used to implement control algorithms that must adapt to their frequency of execution or their execution jitters by compensating them. Using these introspection mechanisms, it is possible to get the actual run-time period to calculate the effective step-size that will influence the differential and integral component of PID control. Introspection is also useful when

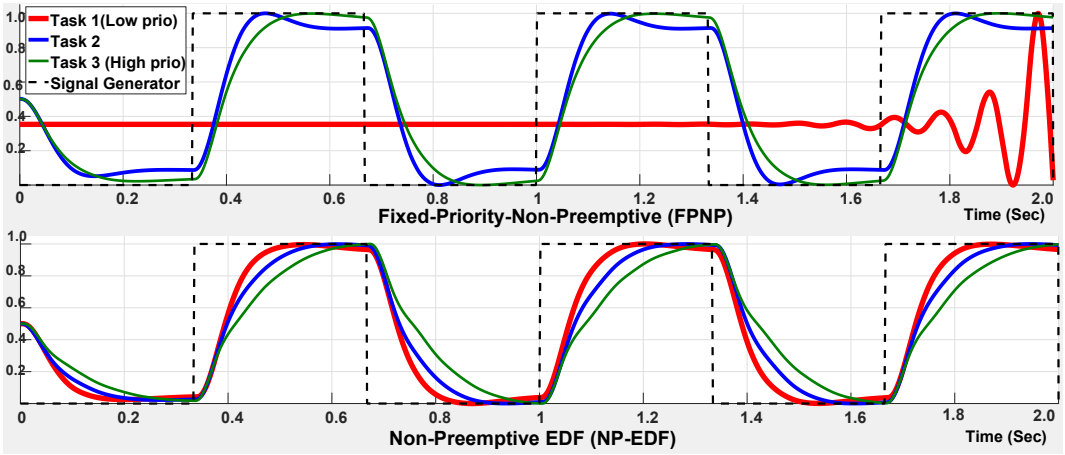


Figure IV.6: Control system performance for different scheduling policies. Under FPNP system tends to oscillate due to less number of task activations. Under NP-EDF, system performance is improved with no oscillation due to increased task activations. This example is inspired from previous works [Morelli et al., 2014] to ensure same results are achievable.

unstable behavior is observed due to a less number of low priority task executions as in the example above. By reducing at run-time the frequency of execution of the two highest priority tasks using the @cpal:sched:period annotation, we were able to achieve a stable system in the previous example.

IV.4.5 Benefits of Re-using Controller Code on Target Hardware

A key advantage of our co-simulation approach is that the same controller model used to evaluate control performance in the design phase can be re-used directly to target hardware to implement the final system. As discussed in [Sundharam et al., 2016a], this development cycle with less steps allows reduced interactions between control and software engineers.

TrueTime and T-Res only supports simulation mode and not real-time execution on target. In CPAL, the same code can be re-used on target with the difference that timing annotations (e.g., jitters, execution times) are ignored. On the other hand, the annotations that define the scheduling (e.g., priorities, deadlines) are considered as run-time parameters for the execution engine.

Table IV.1 summarizes the comparison of the approaches discussed in this chapter to evaluate the performance of latency-sensitive control systems.

IV.5 Discussions and Outlook

The timing behaviour of control tasks is a critical concern in real-time digital controllers. These delays, such as start-of-execution jitters, or missed executions, affect the system performance and need to be considered during the design phase.

Characteristics	TrueTime in MLSL	T-Res in MLSL	SimEvents in MLSL	CPAL in MLSL
Scheduling kernel	Separate kernel	Separate kernel	Built-in kernel	Built-in kernel
Task model	.m file / C++	Graphical model	Graphical model	CPAL model(C alike)
Control and timing separation	No	Yes	Yes	Yes
Scheduling policies supported	Almost all	Almost all	Only basic policies	FIFO, FPNP, NPEDF
Target Execution environment	No	No	No	Yes (Raspberry Pi, Freescale FRDM)
Simulation strategy	Discrete event simulation with variable step-size with zero crossing	Discrete event simulation with variable step-size with zero crossing	Improved Discrete event simulation	Discrete event simulation with variable step-size with zero crossing
Control system compensation	Possible	Possible	Not possible	Introspection mechanism enables tuning of control law
Rapid-prototyping	Not possible	Not possible	Not possible	Easily achievable
Code-generation	Not possible, code cannot be generated from TrueTime blocks	Not possible, code cannot be generated from T-Res blocks	Not possible, code cannot be generated from SimEvents blocks	Not necessary, controller model can run directly on target
Model – code gaps	may happen	may happen	may happen	Model is code, no gaps

Table IV.1: Comparison of CPAL co-simulation in MATLAB/Simulink with existing approaches.

The approaches developed in the past to study the performance of the control system due to run-time delays are simulation approaches, not supporting the implementation of the system. In this chapter, a model-driven co-simulation based development approach is proposed and illustrated with a servo control example. This federated approach provides the designer with both simulation and execution capabilities to define and validate the functional and non-functional constraints. The benefits of the proposed technique over the state-of-the-art are discussed in this chapter, amongst which a good support for rapid-prototyping to shorten the development cycle.

Chapter V

Towards Timing Equivalent Behaviour

Abstract

This chapter addresses the Thomas Henziger called the grand challenge in embedded software design [Henzinger, 2008]: "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". This motivates our research to provide an development environment where also the non-experts are able to quickly model and deploy complex embedded systems without having to master real-time scheduling and resource-sharing protocols. Especially irreproducible faults due to different timing behaviors or race conditions are a nightmare to debug. We acknowledge that techniques to avoid these problems exist, but they constitute a major obstacle for newcomers and make both design and code more complex and error-prone. When processing power is sufficient, as it is increasingly the case with today's hardware, other concerns than performance such as simplicity and predictability become important. This leads to the research on FIFO scheduling for a certian class of a system developed using MBD that eases the design and verification of embedded real-time systems.

V.1 FIFO for Timing Predictability

First in, first Out (FIFO) scheduling — also referred to as, first come, first serve (FCFS) — executes jobs in the exact order of job arrival. No re-ordering or pre-emption can occur, meaning that FIFO scheduling is arguably the simplest scheduling policy with minimal scheduling overhead. This simplicity comes at the cost of performance. As confirmed in the experiments presented in the later section of this chapter, FIFO is not a contender to most common real-time scheduling policies with respect to the ability to produce feasible task schedules, especially with long tasks. Only in special cases, FIFO will be able to compete with the two dominant policies, fixed-priority pre-emptive scheduling or earliest deadline first. Even amongst the set of non-preemptive work-conserving scheduling policies without offsets, where

non-preemptive EDF is optimal [Davis et al., 2015], FIFO is unlikely to achieve the same performance.

Then why FIFO in the place for discussion? The execution order of FIFO scheduling with offset and strictly periodic task activation is uniquely and statically determined. This means that whatever the execution platform and the task execution times, be it in simulation mode in a design environment or at run-time on the actual target, the task execution order will remain identical. Beyond the task execution order, the reading and writing events that can be observed outside the tasks occur in the same order. This property, leveraged by our MBD environment CPAL design flow [Navet et al., 2016a], provides a form of timing equivalent behavior between *development phase* and *run-time phase* which eases the implementation of the application and the verification of its timing correctness. Simulation form of exeecuting the controller model refers to the *development phase* and on-target form of executing the controller model refers to *run-time phase*.

Also, the FIFO scheduling is sustainable in the tasks' execution times, meaning that if a task set is deemed schedulable and the execution times of the tasks are reduced, the task set remains schedulable. These properties greatly simplify the verification and even enable simulation as a valid (even though in many cases too timing consuming) schedulability test. This is in stark contrast to most non-preemptive scheduling policies, where scheduling anomalies [Graham, 1969] prohibit the use of system simulation for validation purposes. FIFO policy is a non-preemptive by nature and provides no means to account for task priorities. Schedulability under FIFO can only be guaranteed for under-utilized systems with uniform period ranges. The focus of the research community has often been on achieving schedulability under unfavorable conditions and high task-set utilizations. As a consequence, FIFO is only considered an option for soft real-time systems, if at all. While we agree with this assessment, we are interested in other concerns than pure performance.

In this context, we re-visit FIFO scheduling under modified conditions and make a case for FIFO scheduling with strictly periodic task activation and release offsets to increase the predictability and to improve the performance. The contributions of this chapter are threefold:

- We show that FIFO with offsets is unique in the sense that it is both work-conserving and exhibits a single, well-defined execution order.
- We provide a schedulability analysis for FIFO, both with and without offsets.
- We evaluate the performance of FIFO scheduling in terms of schedulable task sets, and compare the predictability of FIFO against the two well-known non-preemptive scheduling policies fixed-priority non-preemptive scheduling (FPNS) and non-preemptive earliest deadline first (EDF_{np}) in terms of distinct execution orders.

V.1.1 Related Works of FIFO Scheduling

FIFO scheduling has received limited attention in the real-time community, probably due to its inferior performance. George and Minet [George and Minet, 1997] presented a scheduling analysis for FIFO on a distributed system assuming sporadic task releases, and Leontyev and Anderson [Leontyev and Anderson, 2007] presented a tardiness

analysis for FIFO scheduling, also assuming a distributed system and sporadic task releases. To the best of our knowledge, FIFO with offsets has not yet been analyzed.

Adding offsets to improve schedulability, has been proposed by Tindell [Tindell, 1994] for fixed-priority pre-emptive scheduling (FPPS) and has since been extended to earliest deadline first (EDF) [Pellizzoni and Lipari, 2005; Gutiérrez and Harbour, 2003], to distributed systems [Henia and Ernst, 2006] and to non-preemptive EDF (EDF_{np}) [Baruah, 2006].

All of these scheduling policies are work-conserving. Non-work-conserving algorithms, as for instance recent work by Nasri and Fohler [Nasri and Fohler, 2015b], introduce idle times to delay long tasks that would otherwise block tasks with a shorter deadline. Despite the fundamental difference to FIFO (work-conserving versus non-work-conserving), the motivation to introduce idle-time is the same as to introduce offsets, namely to establish schedulability of an otherwise unschedulable system.

In Section V.2, we explain our system and task model and introduce basic properties of FIFO scheduling, and Section V.3 provides a schedulability test for FIFO with and without offsets. In Section V.5, we evaluate FIFO in terms of performance and predictability, and Section V.6 concludes the chapter.

V.2 Real-time Scheduling Under FIFO

V.2.1 Execution Model

We now define the task set and execution parameters. We assume a task set Γ made up of n tasks $\{\tau_1, \dots \tau_n\}$ running on a single processor. Each task τ_i is represented by a tuple

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

where O_i is the task's release offset, C_i the worst-case execution time, T_i the task's period and D_i the deadline. The task instances, also referred to as jobs, are scheduled non-preemptively in order of their arrival. To this end, the scheduler maintains a FIFO queue with ready jobs waiting for dispatch.

In case of simultaneous job arrival, jobs indices are used as a tie breaker: George and Minet [George and Minet, 1997] have shown that in case of simultaneous job arrival, deadline-monotonic order is optimal. We thus assume without loss of generality that tasks are indexed in deadline monotonic order. The job with lowest index, i.e. , with the shortest deadline, is queued first.

We assume constrained deadlines, i.e. , $\forall \tau_i \colon D_i \leq T_i$ and constrained offsets, i.e. , $\forall \tau_i \colon O_i \leq T_i$. The task utilization is defined as

$$U_i = C_i / T_i \tag{V.1}$$

and the utilization of the complete task set by

$$U_\Gamma = U_i \tag{V.2}$$

The hyperperiod H of the task set is given by least common multiple of all periods

$$H_\Gamma = lcm_i\{T_i\} \tag{V.3}$$

and denotes the time after which the same arrival pattern repeats.

A task produces an infinite sequence of jobs τ_i^j with $j \in \mathbb{N}$. A job's release time is denoted by r_i^j and f_i^j denotes its finishing time. The response time r_i of task τ_i is then given by the maximal delay between the release and completion of a job of τ_i, i.e. :

$$R_i = \max_j \left\{ f_i^j - r_i^j \right\} \tag{V.4}$$

We distinguish between strictly periodic and sporadic job releases:

V.2.1.A Sporadic Release

In case of sporadic job releases, period T_i is interpreted as the minimal inter-arrival time. The job release times are thus constrained as follows:

$$r_i^0 \geq O_i \tag{V.5}$$

$$r_i^j \geq r_i^{j-1} + T_i \tag{V.6}$$

A job's deadline D_i is interpreted relative to the job's release time:

$$d_i^j = r_i^j + D_i. \tag{V.7}$$

V.2.1.B Periodic Release

In case of strictly periodic release, the job release time r_i^j of job τ_i^j is given by

$$r_i^j = O_i + jT_i \tag{V.8}$$

and its absolute deadline by

$$d_i^j = O_i + jT_i + D_i. \tag{V.9}$$

Independently of the release pattern, FIFO scheduling is work-conserving in the sense that it does not introduce any idle times when work is pending. This means that prior to any deadline miss, there must be a busy period in which the processor is not idling. The busy period can be bounded using various independent and incomparable methods:

George et al. [George et al., 1996] presented a bound based on the tasks' deadline and the utilization of the task set:

$$L_U := \max_i \left\{ D_1, D_2, \ldots, D_n, \frac{\sum_{i=1}^n (T_i - D_i) U_\Gamma}{1 - U_\Gamma} \right\} \tag{V.10}$$

Ripoll et al. [Ripoll et al., 1996] presented a bound based on the following recursive equation:

$$L_R^{a+1} := \sum_{i=1}^{n} \frac{L_R^a}{T_i} C_i \tag{V.11}$$

Since both bounds L_R and L_U are independent, we can take the minimum of both as the task set's busy period L:

$$L := \min\{L_R, L_U\} \tag{V.12}$$

Naturally, the busy period is only bounded if the task set utilization U_Γ is less than or equal to one.

V.2.2 Basic Properties of FIFO Scheduling

In this section, we formulate and prove basic properties of FIFO scheduling. As noted by Leontyev and Anderson [Leontyev and Anderson, 2007], non-preemptive execution is implied by the use of a FIFO queue. This represents a stark contrast to other scheduling policies such as EDF and DM, which exist in two variants, pre-emptive and non-preemptive. Such a distinction is not possible for FIFO as pre-emption would contradict the very nature of FIFO scheduling, and can thus only be realized outside the FIFO scheduling regime.

For the sake of completeness, we repeat the argument of George and Minet [George and Minet, 1997] that deadline monotonic priority assignment is an optimal tie breaker in case of synchronous task arrival.

Lemma 1. *Any task set schedulable with FIFO scheduling and any arbitrary tie breaker is also schedulable with deadline monotonic as tie breaker.*

Correctness of Lemma 1 is obvious. For any two synchronously released jobs, executing the job with the smallest relative deadline first will not render a schedulable task set unschedulable.

Next, we argue about the sustainability [Baruah and Burns, 2006] of FIFO scheduling:

Lemma 2. *FIFO schedulable is sustainable with respect to execution times: A schedulable task set will remain schedulable if a task's execution time decreases.*

The execution order of FIFO solely depends on the task release times. Consequently, the tasks will be executed exactly in the order in which the tasks are released (assuming a potentially unbounded FIFO queue). Reducing a task's execution time can thus only reduce, but never increase, other jobs finishing times. In case of strictly periodic task releases, we can formulate an even stronger version of Lemma 2:

Lemma 3. *Deterministic execution order: Non-preemptive FIFO scheduling with strictly periodic job releases and a deterministic tie breaker enforces a unique execution sequence, which corresponds to sequence of job releases.*

A job's τ_i^l release time r_i^l solely depends on statically defined parameters (see Equation (V.8)). Any job released prior to r_i^l has already been dispatched or waits in an earlier slot in the FIFO queue. The deterministic tie breaker completes the argumentation. Hence, simulation (assuming worst-case execution times) serves as a valid schedulability analysis, but may be prohibitively slow. We note that the uniqueness of the execution sequence does not entail a fully static schedule (e.g. , static cyclic scheduling based on schedule tables) which is non work-conserving. In contrast, FIFO is work-conserving and executes jobs whenever work is pending. An increase in a task period, however, may render a schedulable task set unschedulable as illustrates in Figure V.2. Hence, FIFO scheduling is not sustainable in the task's periods.

Lemma 4. *In case of constrained deadlines, a FIFO queue of size is n, i.e. , the number of tasks in the system, is sufficient for any schedulable task set.*

The correctness of Lemma 4 can be seen by observing that each task may have at most one job in the FIFO queue at any time. Two jobs of the same task within the FIFO queue immediately implies a deadline miss since we assume constrained deadlines, i.e. , $\forall_i D_i \leq T_i$.

V.2.3 Advantages of FIFO Scheduling

As confirmed in the experiments of Section V.5, FIFO is not a contender to most common real-time scheduling policies with respect to the ability to produce feasible task schedules, especially with long tasks. Only in special cases, FIFO will be able to compete with the two dominant policies, fixed-priority pre-emptive scheduling or earliest deadline first. Even amongst the set of non-preemptive work-conserving scheduling policies without offsets, where non-preemptive EDF is optimal [Davis et al., 2015], FIFO is unlikely to achieve the same performance.

Yet, it is a common understanding that performance is not the only criterion to select a scheduling policy. Furthermore, real-world task sets often differ strongly from the ones used to evaluate the performance of scheduling policies [Di Natale et al., 2013; Anssi et al., 2013]. In the following, we describe the non-performance related properties of FIFO that can be considered an advantage for FIFO:

- **Simplicity**: FIFO scheduling requires little more than a FIFO queue to store pending jobs and is arguably one of the scheduling policies with lowest implementation overhead. Tasks are executed non-preemptively, which also simplifies the runtime environment and the timing verification.
- **Starvation-Free**: Unless the buffer size is exceeded and jobs are dropped, each job is eventually executed. We note that fairness may be also be considered a dis-advantage since it does not provide a native solution to implement different task criticalities.
- **Work-Conserving**: Similar to EDF or priority-driven scheduling algorithms, FIFO scheduling executes ready jobs and does not introduce unnecessary idle times.

A common assumption in real-time systems are sporadic tasks with minimal inter-arrival times. We deviate from this task model and assume strictly periodic task activation with offsets. Under this restriction, FIFO also exhibits the following advantages:

- **Deterministic Execution Order**: The execution order of FIFO scheduling with offset and strictly periodic task activation is uniquely and statically determined. This means that whatever the execution platform and the task execution times, be it in simulation mode in a design environment or at run-time on the actual target, the task execution order will remain identical. Beyond the task execution order, the reading and writing events that can be observed outside the tasks occur in the same order. This property, leveraged by the CPAL design flow [Navet et al., 2016a], provides a form of timing equivalent behavior between development and runtime phases which eases the implementation of the application and the verification of its timing correctness.
- **Execution Time Sustainability**: FIFO scheduling is sustainable in the tasks' execution times, meaning that if a task set is deemed schedulable and the execution times of the tasks are reduced, the task set remains schedulable.

These properties greatly simplify verification and even enable simulation as a valid (even though in many cases too timing consuming) schedulability test. This is in

stark contrast to most non-preemptive scheduling policies, where scheduling anomalies [Graham, 1969] prohibit the use of system simulation for validation purposes.

Furthermore, we have found that FIFO is unique in that it is the only scheduling policy that combines these properties:

Theorem 1. *FIFO scheduling with strictly periodic task releases is the only scheduling policy that is both work-conserving and exhibits a unique execution order.*

Proof. We observe that only upper bounds on the task's execution time are explicitly defined. A lower bound of 0 is implicitly assumed. Consequently, the tasks' actual execution times at runtime can be arbitrary close to 0. In this case, any work-conserving scheduling policy will eventually execute jobs in order of job arrival, which corresponds to FIFO scheduling. Any re-ordering of job executions would violate the uniqueness of the execution order. Hence, we can conclude that FIFO with strictly periodic job release is the only work-conserving scheduling policy with a unique execution order.

By definition of the task model, C only represents an upper bound on a tasks execution time, whereas a lower bound of 0 is implicity assumed. We observe that when the actual task execution times approach 0, any work-conserving scheduling policy will eventually execute tasks in order of job arrival. Hence, each

The actual execution times at runtime can vary from 0 to C. If we reduce each tasks' execution time to $\epsilon > 0$,

The correctness of Theorem 1 can be shown

Each work-conserving scheduling policy behaves like FIFO in case the overall workload is low

We prove Theorem 1 by contradiction. Assume a work-conserving scheduling policy that results in a non-FIFO execution order, i.e. , two jobs, τ_i^j and τ_l^k, with $i \neq l$ are not executed in order of the job arrival. Without loss of generality, we assume that $r_i^j < r_l^k$. Since they are not executed in FIFO order, $f_i^j > f_l^k$ holds. By definition, C only defines upper bounds on the tasks' execution times □

We note that it can be argued that strictly periodic task releases imply a restriction on the work-conserving property. Although the reaction to an input event can already be computed, the processor may be idling and introduce additional delays. Figure V.1 illustrates this delay using a system with one task reacting to an external event. The

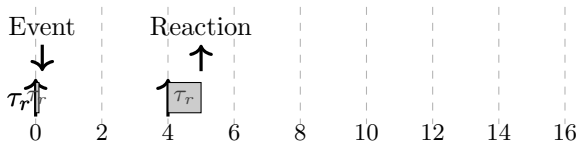


Figure V.1: Delayed event-reaction and idle time despite pending workload, in case of strictly periodic job releases.

additional delay, typically bounded by the task's response time R_r in case of sporadic job releases, is bounded by the task's response time plus period, $R_r + P_r$. Task releases and thus potentially task execution may be artificially delayed. However,

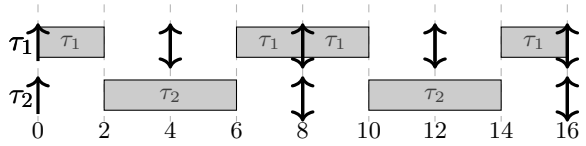
work-conservation still fully applies in overload conditions as well as in case of shorter tasks' execution times.

V.3 Schedulability Analysis

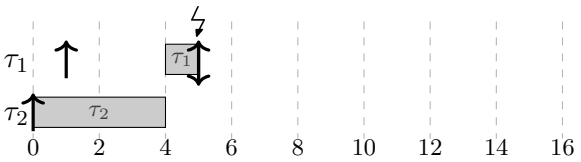
We first revisit the schedulability analysis for sporadic release times presented in [George and Minet, 1997], correct an assumption about the critical instance and then provide a schedulability analysis for the more restricted setting of strictly period job release times.

V.3.1 Sporadic Release Times

The schedulability test in [George and Minet, 1997] relies on the hypothesis that the critical instance is given when all jobs are released (i) synchronously and (ii) all subsequent jobs are released at their highest rate, i.e. , strictly periodic. The schedulability tests checks for all task release within the busy period whether or not a deadline miss occurs. Whereas a synchronous job arrival indeed constitutes the critical instance for many scheduling policies, FIFO scheduling is an exception. Figure V.2 demonstrates the optimism using a task set with harmonic periods and shows that the critical instance is not given in the case of synchronous task release.



(a) synchronous release.



(b) asynchronous release.

Figure V.2: Synchronous task release is not the critical instance for FIFO scheduling. Indeed, τ_1 meets its deadline in the synchronous case and exceeds it in an asynchronous case. $\Gamma = \{\tau_1, \tau_2\}, C_1 = 2, C2 = 4, D_1 = T_1 = 4, D_2 = T_2 = 8$.

The construction of the critical instance is the key to the schedulability test for FIFO scheduling. Yet, there is one critical instance per task, instead of one instance for all tasks. In case of sporadic task releases, any arrangement of task releases is possible.

The critical instance for task τ_i occurs when the job τ_i^j is positioned in the very last place in the FIFO queue, i.e. when the queue already contains a job of each other task. Let r_i^j be the release time of job τ_i^j. Tasks with higher priority [1] have released

[1] Higher priority means here a smaller index value which is used as the tie-breaker in case of simultaneous releases.

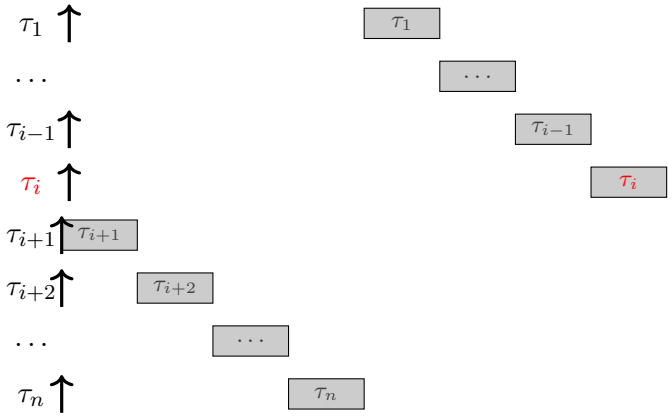


Figure V.3: Critical Instance for task τ_i. Tasks with lower priority than i are released synchronously with i, tasks with lower priority are released ϵ before.

a job synchronously, i.e.

$$\forall_{l<i} \exists k \colon r_i^j = r_l^k \tag{V.13}$$

and all tasks with a lower priority have released an job just an $\epsilon > 0$ before, i.e. :

$$\forall_{l>i} \exists k \colon r_l^k = r_i^j - \epsilon \tag{V.14}$$

Theoretically ϵ can be arbitrary small. In any realistic environment, however, the smallest interval between two job arrivals of different tasks is non-negligible and determined by the implementation of the execution environment and the scheduler.

Consequently, the response time r_i of task τ_i^j in case of FIFO scheduling with sporadic job releases and constrained deadlines is given as follows:

$$R_i = \begin{cases} \sum_{j=\{1,\dots,n\}} C_j - \epsilon & \text{if } i < n \\ \sum_{j=\{1,\dots,n\}} C_j & \text{if } i = n \end{cases} \tag{V.15}$$

The response time computation is exact and exhibits linear complexity in the number of tasks. It also highlights the low performance of FIFO scheduling: only highly underutilized systems can be deemed schedulable.

V.3.2 Strictly Periodic Release Times

Even though we are not aware of any work on FIFO scheduling with offsets, we were able to construct a schedulability analysis for this policy using already established schedulability results. In particular, the schedulability test for EDF with offsets presented by Pellizzoni and Lipari [Pellizzoni and Lipari, 2005].

We note that FIFO is work-conserving in the sense that it does not introduce any idle times when work is pending. This means that prior to any deadline miss, there must be a busy period in which the processor is not idling. As we assume arbitrary offsets and strictly periodic releases, we do not know when a deadline-miss happens and so, would need to validate all busy periods within twice the hyperperiod. To avoid this prohibitively long search, we construct for each task, a hypothetical critical instance leading to a task's first deadline miss. Let τ_i be the task to miss its deadline, and τ_i^j

released at r_i^j the corresponding job. The critical instance happens when all tasks other than τ_i release a job as close to r_i^j as possible. If we can prove that despite this pessimistic assumption, job τ_i^j will finish before its deadline d_i^j, we can conclude that no job of task τ_i will ever miss its deadline. If we can repeat the same argumentation for each task in Γ, we can conclude that the complete task set is schedulable.

V.3.2.A Construction of $\hat{\Gamma}$

Formally, we define for each task τ_i a pseudo task-set $\hat{\Gamma}$ that represents the critical instance for task τ_i. The two task sets Γ and $\hat{\Gamma}$ only differ in the task offsets, the rest of the parameters remaining identical. Let $\hat{\tau}_i^j$ be a job that misses its deadline. As we know that in a work-conserving scheduling algorithm, a deadline miss must be within a busy-period L, we set the release time as follows $\hat{r}_i^j = L$ and its deadline to $\hat{d}_i^j = L + D_i$.

We now select the task parameter of each task $\hat{\tau}_l$ with $l \neq i$ to maximize the likelihood of a deadline miss of job $\hat{\tau}_i^j$. To this end, we postpone the job release of the last job of task $\hat{\tau}_l$ executed before the deadline miss as much as possible. An earlier job release will only increase the slack time and so, reduce the pressure on the finishing time of job τ_i^j.

In case of a higher priority task, i.e. , $\hat{\tau}_l$ with $l < i$, the job must be released just before or synchronously with $\hat{\tau}_i^j$, whereas tasks with lower priority must be released strictly before $\hat{\tau}_i^j$. Since we use task priorities as a tie breaker, a lower priority task released synchronously with $\hat{\tau}_i$ would be executed after, and not before task $\hat{\tau}_i$.

Pellizzoni and Lipari presented a computation of the minimum distance between any two release times of two different tasks τ_i and τ_l (see [Pellizzoni and Lipari, 2005], Lemma 2). In contrast to their work, we are not only interested in the minimal distance, but also in the minimal distance larger than zero. We therefore repeat the computation of the minimal distance.

Let δ be distance between jth job of task τ_i and the k job of task τ_l:

$$\delta_{i,l} = j \cdot T_i + O_i - k \cdot T_l + O_l \tag{V.16}$$

By replacing T_i with $x_i \cdot \gcd(T_i, T_l)$ and T_l with $x_l \cdot \gcd(T_i, T_l)$, we get

$$\begin{aligned}\delta_{i,l} =& j \cdot T_i + O_i - k \cdot T_l + O_l \\ & j \cdot x_i \cdot \gcd(T_i, T_l) + O_i - k \cdot x_l \cdot \gcd(T_i, T_l) + O_l \\ & (j \cdot x_i - k \cdot x_l)\gcd(T_i, T_l) + O_i - O_l\end{aligned}$$

Since $j \cdot x_i - k \cdot x_l$ can take any arbitrary value, we replace it by x and get

$$\delta_{i,l} = x \cdot \gcd(T_i, T_l) + O_i - O_l \tag{V.17}$$

Now, we just need to find smallest $\delta_{i,l} \geq 0$ and the smallest $\delta_{i,l} \geq 1$, which are given by

$$x = \frac{O_l - O_i}{\gcd(T_i, T_l)}$$

and

$$x' = \frac{O_l - O_i + 1}{\gcd(T_i, T_l)}$$

Applying these values to Equation (V.17), we get

$$\Delta_{i,l} = O_i - O_l + \left\lceil \frac{O_l - O_i}{\gcd(T_i, T_l)} \right\rceil \gcd(T_i, T_l). \tag{V.18}$$

and

$$\Delta'_{i,l} = O_i - O_l + \left\lceil \frac{O_l - O_i + 1}{\gcd(T_i, T_l)} \right\rceil \gcd(T_i, T_l). \tag{V.19}$$

Finally, we can set the release time of the last job $\hat{\tau}_l^k$ of task $\hat{\tau}_l$ executed before $\hat{\tau}_i^j$ as follows:

$$\hat{r}_l^k = \begin{cases} \hat{r}_i^j - \Delta_{i,l} & \text{if } l \leq i \\ \hat{r}_i^j - \Delta'_{i,l} & \text{if } l > i. \end{cases} \tag{V.20}$$

The offset of task τ_i is given by

$$\hat{O}_i = \hat{r}_i^j \mod T_i, \tag{V.21}$$

and for all other tasks $l \neq i$ by

$$\hat{O}_l = \hat{r}_l^k \mod T_l. \tag{V.22}$$

The remaining task set parameters, i.e. , the relative deadline, period and execution time remain unchanged. Figure V.4 illustrates the task set parameters.

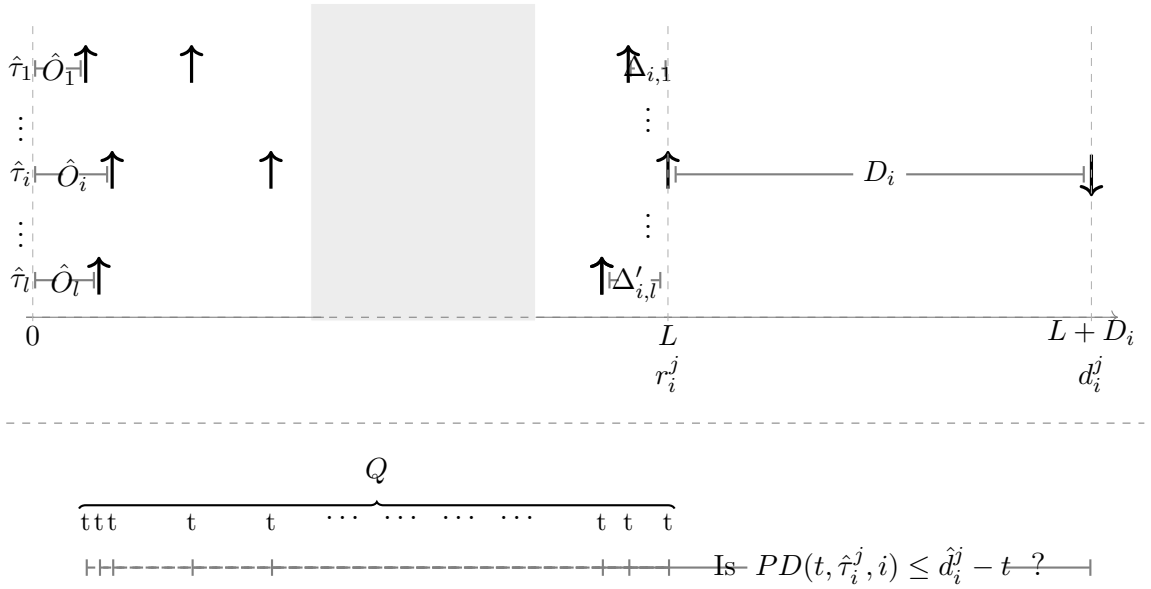


Figure V.4: Illustration of the pseudo task set $\hat{\Gamma}$ and the schedulability test of τ_i with $1 < i < l$. The release time $\hat{r}_i^j$ of job $\hat{\tau}_i^j$ is set to L and its deadline $\hat{d}_i^j$ to $L + D_i$. For all other tasks, the release time of the last job executed before $\hat{\tau}_i^j$ is moved as close to L as possible. For each task release t within $[0 \colon L]$, the schedulability analysis needs to verify that the processor demand does not exceed the available processor time.

Theorem 2. *A deadline miss of task τ_i in Γ entails a deadline miss of job $\hat{\tau}_i^j$ within task set $\hat{\Gamma}$*

$$\exists k \colon f_i^k > d_i^k \Rightarrow \hat{f}_i^j > \hat{d}_i^j$$

Proof. Let k be the job index, so that $f_i^k > d_i^k$ holds. We know that prior to the deadline miss of $\tau_{i,k}$, there must be a busy period without any idle time. Let t be the length of this busy period. The number of job releases N_l of task τ_l within $[r_i^k - t\colon r_i^k]$ is given by

$$N_l = \left\lfloor \frac{t - dist_{i,l}}{T_l} \right\rfloor + 1$$

where $dist_{i,l}$ is the distance between r_i^k and the last job of τ_l executed before r_i^k. Analogously, the number of job releases $\hat{N}_l$ of task $\hat{\tau}_l$ within $[\hat{r}_i^k - t\colon \hat{r}_i^k]$ is given by

$$\hat{N}_l = \left\lfloor \frac{t - \hat{dist}_{i,l}}{T_l} \right\rfloor + 1.$$

Since there is a deadline miss at d_i^k and an idle time before $r_i^k - t$, we know that

$$\sum_l N_l \cdot C_l > t + D_i.$$

By construction, $\hat{dist}_{i,l} \leq dist_{i,l}$, and hence $N_l \leq \hat{N}_l$. Therefore, we can conclude that $\sum_l \hat{N}_l \cdot C_l \geq \sum_l N_l \cdot C_l > t + D_i$, and so, there must be a deadline miss at $\hat{d}_i^k$, which concludes our proof. □

V.3.2.B Schedulability of $\hat{\tau}_i^j$

Using Theorem 2, it is sufficient to validate the schedulability of $\hat{\Gamma}$: if $\hat{\tau}_i$ in $\hat{\Gamma}$ is schedulable with FIFO, so is τ_i in Γ. Furthermore, since we know which job of task $\hat{\tau}_i$ will miss its deadline in case of a deadline miss, it is sufficient to concentrate on the jth job $\hat{\tau}_i^j$, which allows us to reduce the analysis time. If we are able to prove or disprove a deadline miss of job $\hat{\tau}_i^j$, we can immediately abort the schedulability analysis of task τ_i. Consequently, we concentrate only on job $\hat{\tau}_i^j$ and ignore all others. First, we define the number of job releases that may postpone the completion of task i within a given time interval.

The function $\eta_l^{inc}(t_1, t_2)$ denotes the number of job releases of task τ_l within the time interval $[t_1\colon t_2]$, i.e. , including t_2 and is given as follows:

$$\eta_l^{inc}(t_1, t_2) = \left\lfloor \frac{t_2 - \hat{O}_l}{T_l} \right\rfloor + 1 - \left\lceil \frac{t_1 - \hat{O}_l}{T_l} \right\rceil \tag{V.23}$$

The function $\eta_l^{exc}(t_1, t_2)$ denotes the number of job arrivals of task τ_j within the time interval $[t_1\colon t_2)$, i.e. , excluding t_2 and is given as follows:

$$\eta_l^{exc}(t_1, t_2) = \left\lceil \frac{t_2 - \hat{O}_l}{T_l} \right\rceil - \left\lceil \frac{t_1 - \hat{O}_l}{T_l} \right\rceil \tag{V.24}$$

Using these two functions, we define the processor demand PD within time interval $[t_1\colon t_2]$ that can delay the completion of a job of task $\hat{\tau}_i$ released at t_2:

$$PD(t_1, t_2, i) = \sum_{l \leq i} \eta_l^{inc}(t_1, t_2) \cdot C_l + \sum_{l > i} \eta_l^{exc}(t_1, t_2) \cdot C_l \tag{V.25}$$

Again, we distinguish between tasks with higher priorities and tasks with lower priorities to correctly account for the tie-breaking policy in case of synchronous job arrivals.

Theorem 3. *A deadline miss is preceded by a busy period, in which the processor demand exceeds the available computation time:*

$$\hat{f}_i^j > \hat{d}_i^j \Leftrightarrow \exists t \in [0\colon \hat{r}_i^j]\colon PD(t, \hat{r}_i^j, i) > \hat{d}_i^j - t$$

Proof. We prove both directions separately.
$\Leftarrow$: We select t so that $PD(t, \hat{r}_i^j, i) > \hat{d}_i^j - t$. By construction, $PD(t, \hat{r}_i^j, i)$ is the computational demand of all jobs released within $[t\colon \hat{r}_i^j]$ with job $\hat{\tau}_i^j$ being the last to be executed. Hence, the finishing time $\hat{f}_i^j$ of $\hat{\tau}_i^j$ is at least $PD(t, \hat{r}_i^j, i) + t$, i.e. , $\hat{f}_i^j \geq PD(t, \hat{r}_i^j, i) + t$ and so, we can conclude that $\hat{f}_i^j > \hat{d}_i^j$.
$\Rightarrow$: We assume that $\hat{f}_i^j > \hat{d}_i^j$. Let t be the beginning of the busy period prior to the deadline miss. If $PD(t, \hat{r}_i^j, i) \leq \hat{d}_i^j - t$, there is either an idle time prior to $\hat{d}_i^j$, or the processor is not idle prior to t. Both contradict our assumption that t is the beginning of a busy period. Hence, we can conclude that $PD(t, \hat{r}_i^j, i) > \hat{d}_i^j - t$, which finishes our proof. □

Using Theorem 3, we can test for a deadline miss of job $\hat{\tau}_i^j$ as follows:

$$\forall t \in [0\colon \hat{r}_i^j, i]\colon PD(t, \hat{r}_i^j, i) \leq \hat{d}_i^j - t \Rightarrow \hat{f}_i^j \leq \hat{d}_i^j \tag{V.26}$$

To reduce the number of test, we observe that $PD(t_1, t_2, i)$ only changes at the time of a job release, which means that we only need to validate the schedulability at these points:

$$Q = \{t | \exists l, k\colon t = k \cdot T_l + \hat{O}_l \wedge t \leq L - D_i\} \tag{V.27}$$

Hence, we can validate the schedulability of task τ_i as follows:

$$\forall t \in Q\colon PD(t, \hat{r}_i^j, i) \leq \hat{d}_i^j - t \Rightarrow \hat{f}_i^j \leq \hat{d}_i^j \tag{V.28}$$

The schedulability analysis for job $\hat{\tau}_i^j$ is illustrated in Figure V.4. We note that the schedulability test is sufficient but not necessary. Theorem 2 does not provide an equivalence between the schedulability of Γ and $\hat{\Gamma}$. The schedulability analysis can falsely deem a schedulable task set unschedulable, but not the inverse.

V.3.2.C Schedulability Test

Algorithm 2 shows the complete schedulability test in pseudo-code. Function *computeBusyPeriod* implements Equation (V.12) and function *computeMinDistance* Equation (V.18) and (V.19). In the outer loop (line 4 to 22), the algorithm iterates over all tasks and generates for each task the pseudo task set $\hat{\Gamma}$ (line 7 to 10). In line 12 to 18, the algorithm checks for each point in Q, if the schedulability condition (see Equation (V.28)) is met. The algorithm terminates either when all tasks are found to be schedulable, or if one unschedulable task is found.

	FIFO with offsets	FIFO w/o offsets	FPPS with offsets	FPNS w/o offsets	FPNS	EDF	TT
predictability	+	-	-	-	+	-	-
det. exec order	+	-	-	-	-	-	-
work conserving	+	+	+	+	-	-	-
performance	0	-	-	-	+	-	-

Table V.1: Properties of common scheduling algorithms

V.3.2.D An Example to Illustrate the Algorithm

We illustrate Algorithm 2 using the second task τ_2 of the following example task set Γ:

	C_i	D_i	T_i	O_i
τ_1	1	4	4	0
τ_2	2	4	4	2
τ_3	1	8	9	2

The longest busy period L of Γ is 8. To validate the schedulability of τ_2, we construct $\hat{\Gamma}$ as follows: The release time $\hat{r}_2^j$ of job of $\hat{r}_2^j$ is set to L and the deadline of that job to $L + D_2$:

$$\hat{r}_2^j = L = 8 \qquad \hat{d}_2^j = L + D_2 = 8 + 4 = 12$$

The release times of the jobs $\hat{r}_1^k$ and $\hat{r}_3^k$ are set according to Equation (V.20):

$$\hat{r}_1^k = \hat{r}_2^j - \Delta_{2,1} = 8 - 2 = 6$$
$$\hat{r}_3^k = \hat{r}_2^j - \Delta'_{2,3} = 8 - 1 = 7$$

The set Q is then filled with the release times of jobs of all three tasks within the interval $[0\colon 8]$:

$$Q = \{0, 2, 4, 6, 7, 8\}$$

For each element in Q, we have to validate Equation (V.28):

t	$PD(t, \hat{\tau}_2^j, 2)$	$\hat{d}_2^j$	$\hat{d}_2^j - t$	$PD(t, \hat{\tau}_2^j, 2) \leq \hat{d}_2^j - t$?
0	10	12	12	✓
2	8	12	10	✓
4	7	12	8	✓
6	5	12	6	✓
7	4	12	5	✓
8	2	12	4	✓

Using Theorem 2 and 3, we can conclude that no job of task τ_2 will ever miss deadline when scheduled according to FIFO with the chosen offsets.

V.4 FIFO Scheduling Versus Other Scheduling Regimes

V.4.1 Mixed Periodicity

A prerequisite for the implementation of a strictly periodic scheduler is a common system-wide clock, that affects all release times equally. Any jitter in the system

is thus applies to all tasks equivalently and does not need to be considered in the scheduling analysis. A further prerequisite is that tasks

Algorithm 1 FIFO Schedulability Test

1: $i = 1$
2: isSchedulable = true
3: L = computeBusyPeriod
4: **while** $i \leq n \wedge$ isSchedulable **do**
5: $\quad \hat{r}_i^j = L$
6: $\quad \hat{O}_i = r_i^j \mod T_i$
7: $\quad$ **for all** l **do**
8: $\quad\quad \hat{dist}_{i,l}$ = computeMinDistance(i, l)
9: $\quad\quad \hat{O}_l = r_i^j - \hat{dist}_{i,l} \mod T_i$
10: $\quad$ **end for**
11: $\quad Q = \{t | \exists l, k \colon t = k \cdot T_l + \hat{O}_l \wedge t \leq L\}$
12: $\quad$ **for all** $t \in Q$ **do**
13: $\quad\quad$ **if** $PD(t, \hat{r}_i^j, i) - t > \hat{d}_i^j$ **then** isSchedulable = false
14: $\quad\quad$ **end if**
15: $\quad\quad$ **if** $\neg$isSchedulable **then break**
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: $\quad i = i + 1$
19: **end while**
20: **return** isSchedulable

V.4.2 Offset Optimization

We have so far assumed immutable offsets provided a priori. This assumption may hold, for instance, when precedence constraints are implicitly realized via offset relationships. Often, however, offsets can be considered mutable and offset optimization can provide a means to either improve system performance or even to establish schedulability in the first place. We note that systems with mutable offsets are referred to as *offset free systems* [Goossens, 2003].

In recent years, several offset optimization techniques have been developed, such as for instance [Goossens, 2003; Grenier et al., 2006; Monot et al., 2012]. The techniques were typically tailored towards a particular scheduling policy (FPPS with offsets [Goossens, 2003; Grenier et al., 2006]) or towards a dedicated application domain (e.g. , automotive runnables [Monot et al., 2012] or avionics networks [Li et al., 2011]) and are thus not immediately applicable to FIFO scheduling. Yet, the evaluation in [Grenier et al., 2006] indicates that randomization provides sub-optimal yet satisfactory results. With randomization we refer to a completely random distribution of all task offsets. The feasibility of such a random offset assignment will be validated until either a fixed number of assignments has been unsuccessfully validated, in which case the task set is considered unschedulable, or until a feasible offset assignment has been found. As future work, we intend to evaluate the existing offset optimization techniques towards their usability for FIFO scheduling, and/or to develop an offset optimization specifically tailored towards it. Both research topics, however, are out-of-scope of this chapter.

We note that offset optimization does not violate event order determinism: exactly one pre-defined event order is permissible at runtime, only which of the permissible event orders will be selected is decided statically at design-time.

V.5 Performance of FIFO Scheduling

In this section, we evaluate the behavior of FIFO scheduling with respect to

- its ability to lead to feasible schedules compared to other scheduling polices,
- the precision and analysis time of the schedulability test presented in Section V.3, and
- the predictability of FIFO scheduling.

We acknowledge that the performance of scheduling polices is highly sensitive to the choice of parameters. This is in particular true for non-preemptive policies. To achieve transparency and to ease the reproduction of the results, the source code of the programs used in our experiments, including the schedulability test, is available online[2]. This source code enables the reproduction of the experiments presented in this section, as well as evaluation for different parameter settings.

V.5.1 Experimental Setup

To evaluate the performance of FIFO scheduling in terms of schedulable task sets, we have randomly generated 10000 task sets for each task set utilization from 0.025 to 0.975 in steps of 0.025, We have found that the dominant parameters in case of FIFO are (i) the period range, (ii) the type of periods and (iii) the granularity. The period range, or to be precise, the difference between smallest and largest period is a common performance indicator for non-preemptive scheduling policies [Short, 2010]. With granularity of a task set, we refer to the greatest common divisor of all periods and offsets. We have conducted three sets of experiments, one for each of the following period types:

- Random: task periods T_i were randomly chosen in the range [1000 : 100.000]ms,
- Loosely-harmonic: the task periods T_i were set to $x \cdot 1000$ms with x randomly chosen in the range [1 : 100],
- Harmonic: task periods T_i were set to $2^x \cdot 1000$ms with x randomly chosen in the range [0 : 7].

Random-period and harmonic-period tasks are on the opposite sides of the spectrum with respect to period randomness, while loosely-harmonic tasks are representative of the common situation in practice where tasks are a multiple of a time quantity larger that the intrinsic granularity of time, such as 5 or 10ms.

The remaining parameters were set as follows:

[2] `https://www.designcps.com/wp-content/uploads/cpal_codesign_framework.zip`

- the granularity was set to 100ms,
- the task set size was 10,
- task utilizations U_i were generated using UUnifast[Bini and Buttazzo, 2005],
- task execution times C_i were set $C_i = T_i \cdot U_i$,
- deadlines were implicit, i.e. $D_i = T_i$,
- task offsets were randomly chosen in the range $[0 \colon T_i]$.

The schedulability of each task set has been assessed using the following approaches:

- Fixed-priority pre-emptive scheduling FPPS without offsets (red line),
- Fixed-priority non-preemptive scheduling FPNS without offsets (green line, filled circle),
- FIFO with strictly period task releases, resp. with initial random offset (pink line, empty square) and optimized offsets (dark blue line, triangle),
- FIFO with aperiodic releases (light blue line, filled square).

Optimized offsets means here that we have tried up to 1000 random offset assignments for each task set before concluding that the task was unschedulable.

In addition to the purely analytical schedulability tests, we have performed simulation to twice the hyperperiod to evaluate the feasibility of FIFO with offsets (yellow line) for harmonic and loosely-harmonic periods.

In addition to the analytical schedulability tests presented in Section V.3, we have performed simulation to twice the hyperperiod to evaluate the feasibility of FIFO with offsets (black line, empty circle) for harmonic and loosely-harmonic periods. For random periods, simulation is computationally infeasible due to the length of the hyperperiod.

It should be noted that when it can be performed, simulation provides an exact schedulability test for FIFO scheduling with offsets and thus can be used to evaluate the accuracy of the schedulability test in Section V.3. Simulation is however not usable with FPNS with offsets which is not sustainable in the execution times and thus can exhibit scheduling anomalies. We have not yet found in the litterature a feasibility test for FPNS with offsets for the exact same system model (i.e. , not using the transaction model that is for instance used in the CAN network schedulability analysis [Yomsi et al., 2012]).

We note that only for FIFO scheduling with offsets, simulation provides a feasible schedulability test; FPNS with offsets may exhibit scheduling anomalies so that a feasibility analysis using simulation is either unreliable or computationally infeasible even for task sets with a short hyperperiod.

V.5.2 Schedulability Under FIFO

The number of task sets that are schedulable with the different policies under study out of 10000 randomly generated task sets are shown in Figure V.5 for random

periods, in Figure V.6 for loosely-harmonic periods and in Figure V.7 for harmonic periods.

As expected, FIFO with aperiodic task releases performs poorly and is only able to schedule task sets at a low processor utilization, irrespective of period types. In contrast, the performance of the schedulability test for FIFO with strictly period task releases (with and without offset optimization) strongly depends on the type of periods. For random periods, FIFO with offsets performs only slightly better than aperiodic FIFO scheduling. Due to the high variability of the periods, the minimal distances between two job releases of different tasks decreases and the analysis has to assume unfavorable release-scenarios for the pseudo tasks sets $\hat{\Gamma}$. For loosely-harmonic and harmonic periods, adding offsets to FIFO scheduling significantly increases the number of schedulable task sets. Especially for harmonic periods, FIFO with offsets can profit from the very regular periods and is able to compete with FPNS (without offsets). Furthermore, we observe that the regularity of the periods also improves the precision of the analysis; in case of harmonic (Figure V.7) and loosely-harmonic periods (Figure V.6), the analysis is able to deem a similar number of tasks as the, more much more computationally intensive, simulation, and only fails to compete at high processor load. For instance, as can be seen in Figure V.6, results with simulation and analysis remain identical up to 0.65 load.

V.5.3 Weighted Schedulability Measure

To further evaluate the sensitivity of the scheduling policies in terms of (i) number of tasks, (ii) period factor, and (iii) granularity, we use the weighted schedulability measure [Bastoni et al., 2010]. The weighted schedulability measure condenses an otherwise three-dimensional graph to two dimensions. It takes the average of the schedulability ratio weighted by the utilization U. Let $S(\Gamma, p)$ be the result of the schedulability test for task set Γ and parameter p. $S(\Gamma, p) = 1$ indicates that the task set is schedulable, otherwise $S(\Gamma, p) = 0$. The weighted measure is defined as follows:

$$W(p) = \frac{\sum_{\forall \Gamma} U \cdot S(\Gamma, p)}{\sum_{\forall \Gamma} U}, \tag{V.29}$$

The weighted schedulability measure (for 1000 task sets per utilization) for a varying number of tasks is shown here for random task sets in Figure V.8, for loosely-harmonic task sets in Figure V.9, and for loosely-harmonic task sets in Figure V.9.

FPNS profits from an increasing number of tasks. Non-preemptive scheduling is highly dependent on the difference between shortest and longest task, which tends to decrease as the number of task increase. FIFO, even though also non-preemptive, can not profit from an increasing number of tasks. The positive effect of the decrease in the difference between the tasks' execution times is diminished by the higher pessimism of the schedulability analysis; Indeed, the higher the number of tasks, the more pessimistic the schedulability test due to cumulated conservative assumptions. Consequently, the weighted measure for FIFO scheduling (in all cases) remains largely constant starting from a task set size of 8 onwards. It is also noteworthy that the offset optimization scheme by randomization is only marginally efficient (e.g. , 5% more schedulable tasks sets than without optimization with 14 tasks).

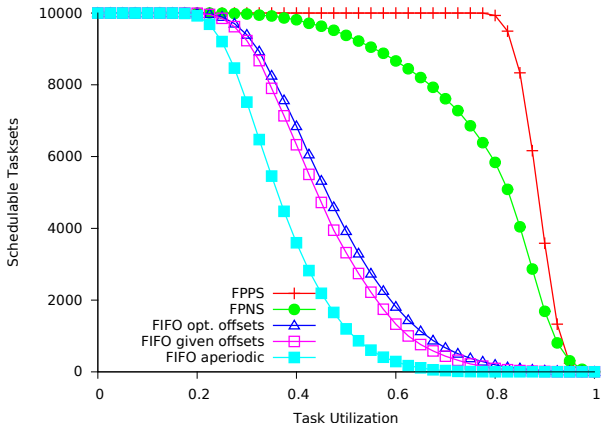


Figure V.5: Evaluation of the base configuration, random periods.

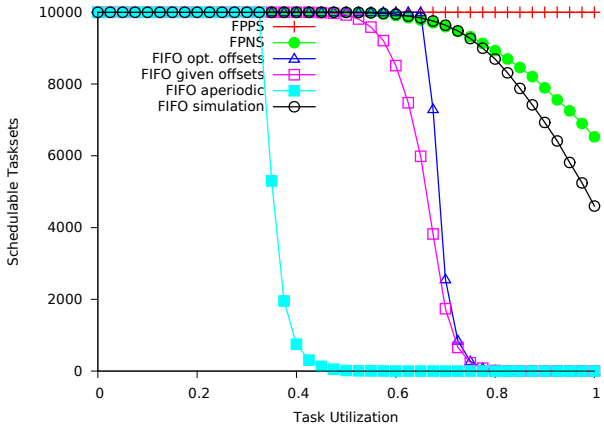


Figure V.6: Evaluation of the base configuration, loosely-harmonic periods.

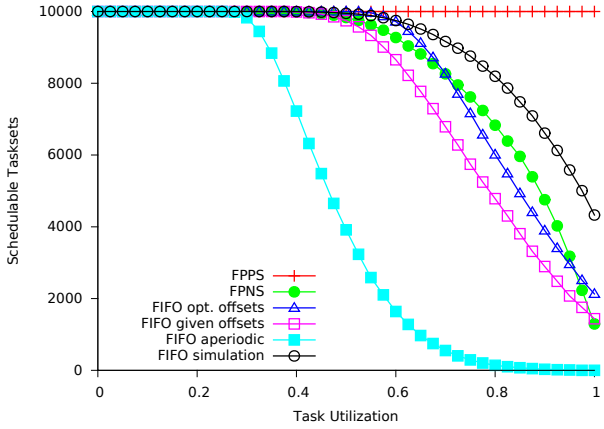


Figure V.7: Evaluation of the base configuration, harmonic periods.

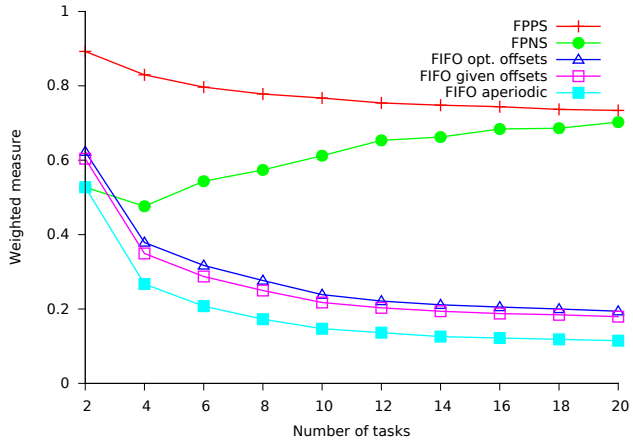


Figure V.8: Weighted schedulability measure, varying number of tasks, random periods.

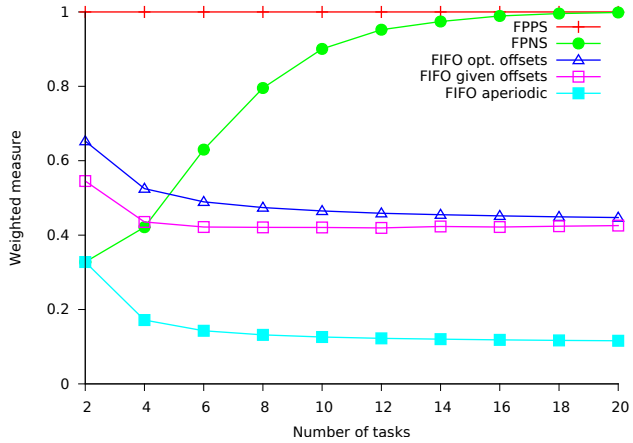


Figure V.9: Weighted schedulability measure, varying number of tasks, loosely-harmonic periods.

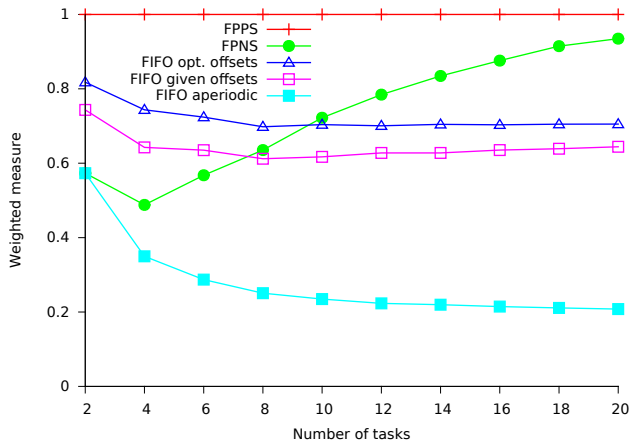


Figure V.10: Weighted schedulability measure, varying number of tasks, harmonic periods.

The evaluation of varying period ranges for loosely-harmonic task sets is shown in Figure V.11 to V.13. All approaches except for FPPS are similarly affected by an increase in the period ranges due to the long task problem [Short, 2010].

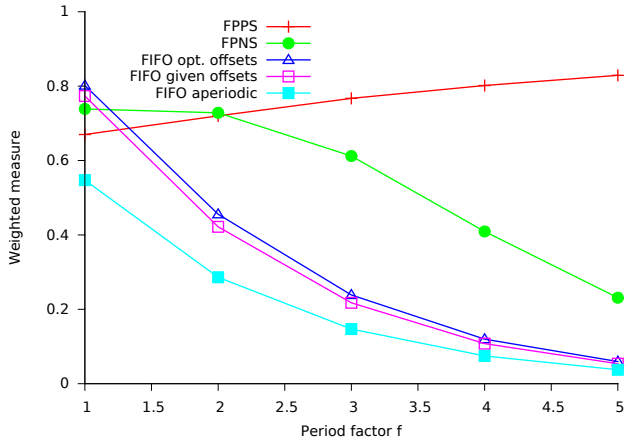


Figure V.11: Weighted schedulability measure, varying period factor ($T_i \in [100 \colon 100 * 10^f]$), random periods.

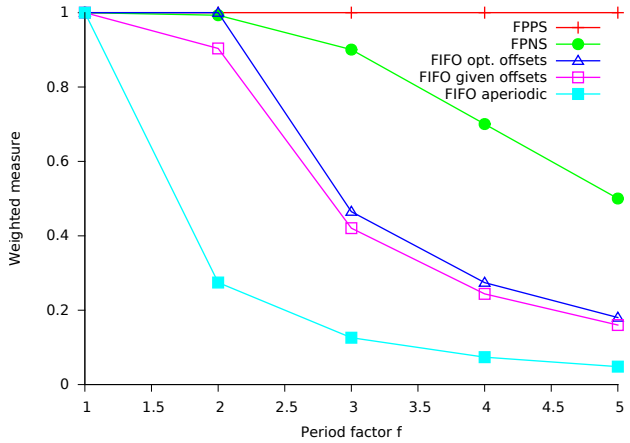


Figure V.12: Weighted schedulability measure, varying period factor ($T_i = x \cdot 1000$, $x \in [1 \colon 10^f]$), loosely-harmonic periods.

The evaluation of varying time granularities, defined as the greatest common divisor of all periods and offsets is given in Figure V.14 for random periods. Only the results for random periods may profit from an increases in the granularity. For harmonic and loosely-harmonic period, the periods are already regular and changing the offsets has only a limited effect. We observe that the performance of all scheduling policies improve when the granularity increases, with a small peak at a granularity of 500ms. The minimal period is set to 1000ms, which means that setting the granularity to 500 leads to more regular periods than for instance a granularity of 600. In fact, by increasing the granularity, we gradually move from random periods to loosely-harmonic periods. The weighted measures for harmonic and loosely-harmonic periods remain largely constant (see Figure V.15 and V.16).

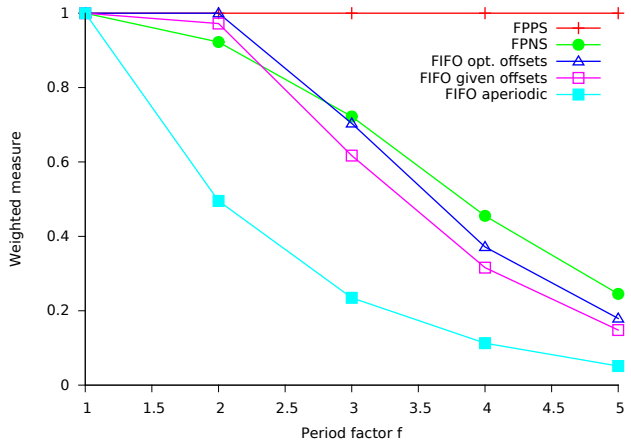


Figure V.13: Weighted schedulability measure, varying period factor ($T_i = 2^x \cdot 1000$, $x \in [0\colon f]$), harmonic periods.

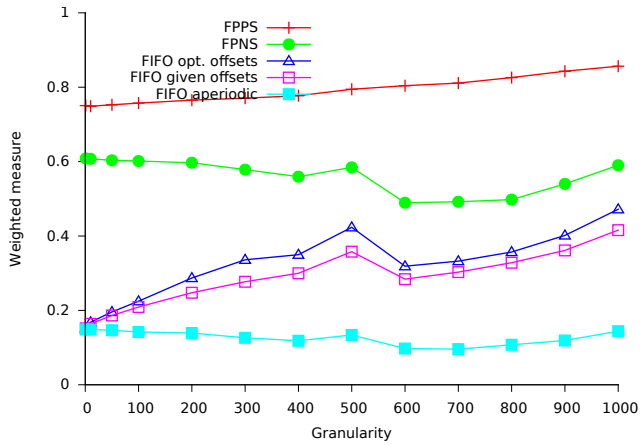


Figure V.14: Weighted schedulability measure, varying granularity, random periods.

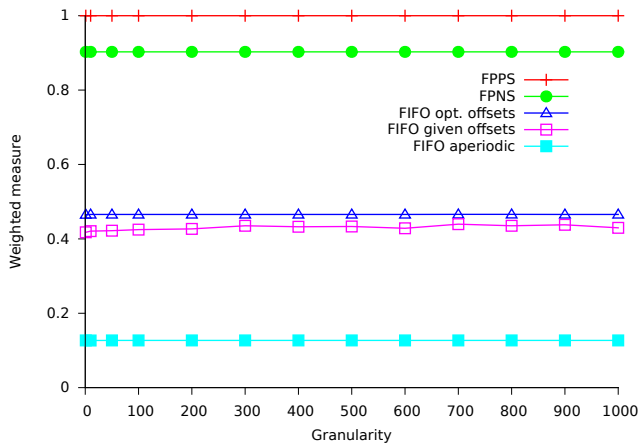


Figure V.15: Weighted schedulability measure, varying granularity, loosely-harmonic periods.

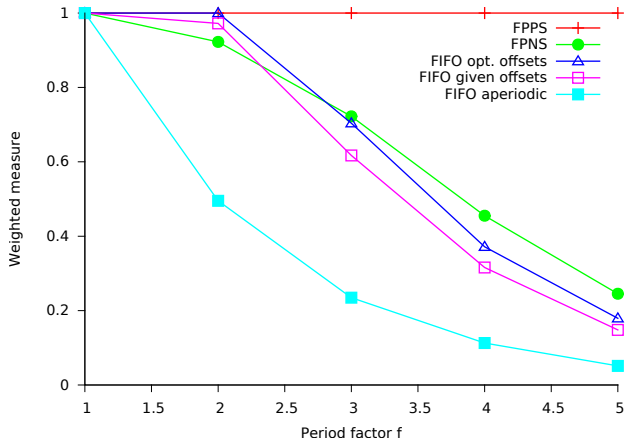


Figure V.16: Weighted schedulability measure, varying granularity, harmonic periods.

V.5.4 Predictability Concerns

FIFO with strictly periodic task releases exhibits a unique execution sequence, or event order, which strongly eases verification and validation. To further evaluate the predictability concerns, we have derived the number of distinct event order for two of the dominant non-preemptive scheduling policies, EDF_{np} and FPNS, both with offsets and strictly period task releases. The task activation pattern is thus the same for all three policies and event orders can only differ in the order of task executions, but not task release. Also, we do not record the timing of an event, but only the order of events. For each task set utilization, we have generated 100 task sets and performed 1000 simulations to twice the hyperperiod per task set and scheduling policy. To vary the execution times of the tasks, we have randomly selected a value $C_i' \in [C_i/2 : C_i]$ with a uniform distribution. The results are shown in Figure V.17 for loosely-harmonic periods and in Figure V.18 for harmonic periods. As already discussed by Buttazzo

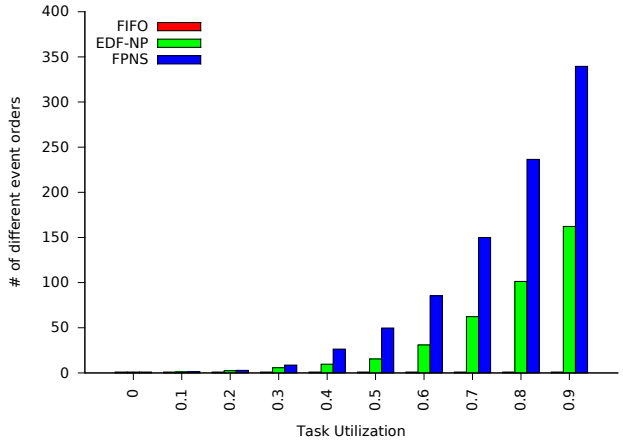


Figure V.17: Number of different event orders, harmonic periods.

in [Buttazzo, 2005], FPNS shows a higher variability than EDF_{np}, despite the static priorities of FPNS. Also, quite surprisingly, harmonic periods lead to an order of magnitude higher number of event orders, both for EDF_{np} and FPNS, compared to loosely-harmonic periods. In case of loosely-harmonic periods, task releases are

stretched over a longer period of time compared to harmonic period, which reduces the freedom to re-order task executions.

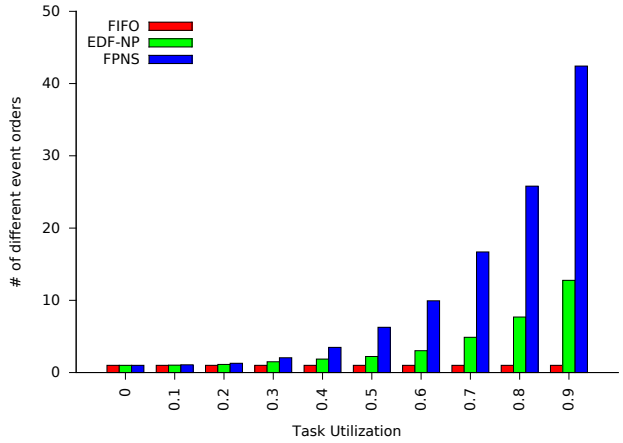


Figure V.18: Number of different event orders, loosely-harmonic periods.

V.6 Chapter Conclusion

In this chapter, we provided a schedulability test for FIFO with and without offsets and made the case that FIFO scheduling, with strictly periodic tasks and offsets, is a competitive scheduling policy when predictability and simplicity matter. FIFO is non-preemptive by construction and provides no means to account for task priorities. Consequently, FIFO is in general not able to schedule a competitive number of task sets at high processor utilizations as for instance FPNS, let alone pre-emptive scheduling policies.

Adding offsets and enforcing strictly periodic task releases provides two significant advantages to FIFO: (i) The performance issues are mitigated and a higher number of task sets are schedulable, even at high utilization rates, and especially for task sets with harmonic or loosely-harmonic periods and (ii) a unique execution order, defined by the order of job arrivals is enforced which greatly simplifies validation and testing. We have shown that FIFO with offsets is unique in the second property amongst all work-conserving algorithms. It is thus a good fit for our CPAL design flow which aims at automating system synthesis and hiding away from the designer the complexity of the underlying runtime environments, lowering thus the barriers to designing and modeling provably-safe real-time systems. In future works, we plan to consider and evaluate the scheduling overheads of FIFO, but also of the other scheduling policies, and design an offset optimization strategy tailored to FIFO.

Chapter VI

Model-driven Co-design Workflow

Abstract

In the previous chapters, we have discussed how MBD is applied in the industry to develop new software functions and integrate them into the existing run-time environment of a Cyber-Physical System (CPS). The design of a software component involves designers from various viewpoints such as control theory, software engineering, safety, etc. In practice, while a designer from one discipline focuses on the core aspects of his field (for instance, a control engineer concentrates on designing a stable controller), he neglects or considers less importantly the other engineering aspects (for instance, real-time software engineering or energy efficiency). This may cause some of the functional and non-functional requirements not to be met satisfactorily. In this chapter, we present a co-design framework based on the idea discussed in `Chapter II (Background and Preliminaries)` *and the timing tolerance contract to address such design gaps between control and real-time software engineering. The framework consists of three steps: controller design, verified by jitter margin analysis along with co-simulation, software design verified by a novel schedulability analysis, and the run-time verification by monitoring the execution of the models on target. This framework builds on CPAL (Cyber-Physical Action Language), a MBD environment based on model-interpretation, which enforces a timing-realistic behavior in simulation through timing and scheduling annotations. The application of our framework is exemplified in the design of an automotive cruise control system.*

VI.1 Introduction

Control theory and software engineering are two disciplines involved in the development of control software. Traditionally, control engineers design the controller model without considering the computing platform constraints and specifications. The converse applies to software engineering, where control performance is not considered during software design. The control engineering and the software engineering are two different worlds with different objectives in mind. Consequently, the complete set of functional and non-functional requirements of the control software are usually

not elicited at the control design stage. Hence, as discussed in [Lampke et al., 2015], substantial design-gaps may exist during the design of a control software.

The control software executes on an Electronic Control Units (ECU) interfaced with various sensors and actuators. The continuous-time signals are periodically sampled; each sampled set of data is then processed by real-time control functions. Control theory typically assumes deterministic and periodic sampling. However in practice, for instance, due to preemptions and varying task execution times, there exists a varying delay between sensing and actuation, which is called input-to-output delay or sensing-to-actuation delay. A control designer typically assumes this input-to-output delay to be zero or constant which is an unrealistic assumption. The input-to-output delay depends on the time at which sensing and actuation takes place. The sensing time may also vary over time typically due to the interference of higher priority tasks, and the variability of sensing times is called *input jitter*. There are also jitters in the actuation times, called *output jitters* caused by varying execution times and preemptions.

These jitters directly impact the quality of control functions, and, in the worst-case, they might jeopardize the safety of the system. Hence, it is important to consider these delays during the design phase of the control software. This work addresses the case where the input data acquisition is done locally on one node. It can be extended like in TrueTime [Cervin et al., 2003a] to cover the case of networked control systems, including "Industrial Internet of Things" (IIoT) applications, where data are transmitted over a network, which would increase the input jitters, as well as the input-to-output delays.

This chapter is structured as follows. In Section VI.2, we explain the system model and the steps involved in the framework for fusing control and scheduling viewpoints. Section VI.3 presents the proposed co-modeling and simulation environment as well as jitter analysis tools and methods. In Section VI.4, we explain the verification of timing tolerance assumptions using WCET measurements and the schedulability analysis. In Section VI.5, we evaluate the framework using the example of a cruise control system. In the same section, we discuss the stability verification using the jitter margin concept and the CPAL co-simulation in Simulink. The section also details the scheduling configuration and run-time introspection features. Section VI.6 provides the related work. Section VI.7 concludes the chapter.

VI.2 Workflow for Fusing Control and Scheduling Viewpoints

In this chapter, we discuss a framework that supports our co-design modeling environment for both controller and control software development. The framework provides schedulability and control performance analysis along with simulation capabilities. We underpin the proposed framework with the help of timing contracts introduced in [Derler et al., 2013] which are sets of timing characteristics that ensure the targeted control performance. The timing contract can be a crucial concept in component-based design because it drives and synergizes the design thinking of the stakeholders from different viewpoints. We use the timing contract as a candidate to bridge the control software design-gaps. During the application of a timing contract, we observe a vertical type contract [Nuzzo et al., 2015] in our proposed framework

as the timing contract is applied between two phases of the Software Development Life Cycle (SDLC), in this case between controller design and software development.

The co-design framework presented in this work encompasses three steps of the development cycle: (i) controller design; (ii) software scheduling and execution platform configuration; and (iii) run-time monitoring. Firstly, we discuss scheduling and stability viewpoint analyses supporting the proposed co-design and simulation environment. We rely on our timing-aware model-driven environment called Cyber-Physical Action Language (CPAL) for co-design in Simulink. We then present the CPAL constructs and timing annotations, central to our approach, which enable us to reproduce the timing irregularities of interest, such as jitters and varying input-to-output delays. CPAL provides the timing dimension to the controller design, which acts on the plant model in Simulink. We provide the CPAL execution platform for Simulink as open access for experimentation. Along with existing jitter analysis tools, the proposed co-design platform helps designing stability guaranteed controller models by integrating the target-platform timing behavior. Furthermore, it provides software engineering with the control information needed to bound the space of feasible software design solutions. The stability verification itself is done with the help of the jitter margin concept and the co-simulation of CPAL execution in the Simulink environment.

The second contribution is the verification of the timing tolerance contract assumptions made during controller design. The verification is specifically useful when a new control function is integrated into an existing stable and functioning ECU. How can we analytically validate whether the system maintains the desired performance (stable and schedulable) after integration? To this end, we propose a novel schedulability analysis for a certain class of task and execution models in real-time scheduling. To assign a realistic execution time to the controller task, we estimate the Worst-Case Execution Time (WCET) beforehand using measurements of the model running on the target hardware.

The third and last contribution is the proposed run-time verification methodology. During model on target execution, we check whether the newly integrated controller function stays within the stability margin. For this, we take advantage of CPAL introspection features to monitor the execution characteristics of a controller model at run-time. More specifically, we introspect whether the jitters and input output latencies are within the margin guaranteeing the stability and schedulability objectives.

System designers in the industry are typically highly knowledgeable in their own fields (control systems, software engineering, scheduling, etc.) but contracts among design teams are not necessarily well established and communicated among the stakeholders. Our objective is to define a structured framework, with clear interfaces, which can be agreed upon and followed by all. The framework proposed in this section highlights the issues faced at each step of the design and we propose possible solutions. Our framework may not be suitable for all industrial settings, but it addresses the gap between control models and their implementation, and can serve as a basis for context-specific design frameworks.

VI.2.1 System Model

We propose an integrated framework which combines the tools and methods necessary to design a model of the system. Table VI.1 provides a quick reference for the notations used in this chapter. The system is comprised of a controller model, a plant model and platform model. Plant P is modeled by a continuous-time system of equations

$$\begin{aligned} \dot{x} &= Ax + Bu, \\ y &= Kx, \end{aligned} \tag{VI.1}$$

where x is the plant state and u is the control signal. The plant output y is sampled periodically with some delays at discrete time instants. The control signal is updated periodically with some delays at discrete time instants, (i.e., actuation also happens with some delay). Quantities A, B, K are constants. The controller model is comprised of a task set Γ of n periodic tasks $\{T_1, \ldots T_n\}$ executing on a single processor.

Table VI.1: Notations used in the Chapter *Model-driven Co-design Workflow*.

task-set	$\Gamma = \{T_1, \ldots, T_n\}$
pseudo task-set	$\Gamma = \{\hat{T}_1, \ldots, \hat{T}_n\}$
number of tasks	$n \in N$
job index	$i, j \in N$
task worst-case execution time with no interference	$C_i \in R$
task period	$h_i \in R$
task relative deadline	$D_i \in R$
task absolute deadline	$D_i \in R$
task release time	$r_i \in R$
task finish time	$f_i \in R$
task worst-case response time	$R_i^w \in R$
task best-case response time	$R_i^b \in R$
task processor demand	$PD_i \in R$
task busy-period	$L \in R$
input jitter also known as sampling jitter	$J^h \in R$
output jitter also known as response-time jitter	$J^\tau \in R$
input-to-output delay also known as StA latency	$\tau \in R$
k-th sensing time instance	$t_k^s \in R$
k-th actuation time instance	$t_k^a \in R$
nominal input-output delay	$\mathfrak{L} \in R$

Each controller task T_i is represented by a tuple T_i: (O_i, C_i, h_i, D_i), where O_i is the task's release offset, C_i the Worst-Case Execution Time (WCET), h_i the task's period and D_i the deadline. R_i^w and R_i^b are the worst and best-case response times. The task instances, also referred to as jobs, are scheduled non preemptively in order of their arrival. Each controller task is assumed to have three activities in the order sensing, computation and actuation. Sensing is the first activity which *reads* the data from a sensor. The computation also known as *control law execution* is the second activity. The actuation is the last activity which *writes* the data to physical devices.

The variability in the times at which the control software reads and writes the input and output data is called jitter. Jitters have a major impact on the performance

of some control systems. To formally define the jitters that must be respected by an execution platform, the authors in [Derler et al., 2013] introduce four timing contracts namely Zero Execution Time (ZET), Bounded Execution Time (BET), Logical Execution Time (LET) and Timing Tolerance (TOL) contract. In this work, we consider the latter contract which is more general than ZET and BET, and does not imply strong implementation constraints like LET [Kirsch and Sokolova, 2012]. A Timing Tolerance TOL contract implies that the following conditions hold:

$$\begin{aligned} t_k^s \ &\epsilon \ [k.h, k.h + J^h], \\ t_k^a \ &\epsilon \ [t_k^s + \tau - J^\tau, t_k^s + \tau + J^\tau], \end{aligned} \tag{VI.2}$$

where J^h is the tolerable input jitter. τ is the tolerable input-to-output delay also known as tolerable Sensing-to-Actuation delay (StA delay). The nominal input-to-output delay $\mathfrak{L}$ is a minimum delay experienced between input to output. J^τ is the tolerable output jitter. The tolerances J^h and J^τ are also referred to as margins, namely input jitter margin and output jitter margin.

VI.2.2 Framework Steps

To bridge the control-computing gap, we propose a framework that fuses the control and scheduling viewpoints in the context of model-based system design. Figure VI.1 shows the overall step-by-step flow of the framework.

Step 1: Controller Design Based on the functional and non-functional requirements, the first stage of the framework is the control study, that determines the control equations that will potentially allow the system to achieve the required control performances. This control study relies on the designer's expertise with the help of control-system simulators like MATLAB/Simulink, which include the plant model. We note that at this stage the timing issues are not considered, and in particular the implementation delays are ignored, which may require to revisit the choice of the control law later in the design flow.

The next stage is to model the control law in CPAL, which provides native support for Finite State Machines (FSMs) to describe the logic of the algorithm, in a similar way as StateFlow. The CPAL model controls the plant model designed in the Simulink environment. At this stage, timing delays are introduced: the controller tasks are activated with input-to-output delays using timing annotations in the CPAL model (input and execution time jitter). The timing annotations are also useful for defining tasks' periods, deadlines, priorities of execution, and the scheduling policy. The CPAL interpreter, which can be seen as an execution engine, runs the controller model within the Simulink environment that hosts the CPAL/Simulink co-simulation. The simulation results such as control performance, task activation diagram and values of the outputs are all available within Simulink.

As discussed in [Aminifar et al., 2012], the controller design can be done using two analytical methods: expected control performance and worst-case control performance. The Jitterbug toolbox [Lincoln and Cervin, 2002] is used to calculate the expected value of quadratic control costs. The Jitter margin toolbox is used to calculate the worst-case control cost, as explained in details in Section VI.5.2.VI.5.2.A. For a given control performance, this tool determines the tolerable jitter margins. In turn,

these jitter margins provide admissible deadlines for the controller tasks. Using the proposed co-simulation, we verify the tolerable input jitter margin J^h and tolerable input-to-output delay (StA delay) under which the system maintains an acceptable stability performance. We also fine-tune the obtained deadline for step response expectations when required. Further, using simulations, we study the effect of these tolerable jitter margins on control performance.

Step 2: Software Design At the end of Step 1, each controller model consists of a single task performing sensing, computation and actuation. Note that this task can be integrated with other existing tasks ("Software components" block in Figure VI.1). At Step 2, a suitable scheduling solution, i.e., a scheduling policy and the associated parameters should be selected so as to meet the real-time constraints expressed as deadlines derived at the first step. This can be achieved, for instance, using the optimization framework in [Sundharam et al., 2016b], a form of scheduler synthesis. Schedulability analysis has to be performed under some Worst-Case Execution Time (WCET) assumptions for all tasks. These values can be obtained by analysis or, as in our approach, approximated with on-target measurements. If this scheduling configuration meets the timing performance needed to provide the necessary control performance to the controller task then the design flow moves on to Step 3. Otherwise, we return to Step 1 and redesign the control law.

The same CPAL model executed in the simulation environment (in the previous step) is now interpreted directly on the target to measure the execution time of the task. Schedulability analysis can then be performed, and we propose a novel schedulability analysis for FIFO policy with offsets in Section VI.4. Although FIFO is outperformed by most policies in terms of meeting deadlines [Altmeyer et al., 2016], it has the advantage that the scheduling order does not depend on the execution times, irrespective of the platform. The schedulability analysis checks whether the controller task we integrate with the existing software components remains schedulable or not.

This stage, if successful, ensures that the timing constraints coming from the control laws are met by the software and execution platform. If unsuccessful, we can first try to optimize the CPAL code. This may include breaking down the controller task into sub-tasks, for instance one for sensing, one for computation and one for actuation, which is a classical strategy to increase the schedulability of control systems [Gerber and Hong, 1997], but in some cases the suitable strategy has to be specific to the application. If still unsuccessful, the process returns to Step 1 for a redesign or fine-tuning of the controller. In any cases, the model used at Step 1 for functional simulation will be the one used for execution on the target hardware.

Step 3: Model Introspection From Step 2, we obtain a functional CPAL controller along with the scheduling parameters to be configured for on-target execution. These parameters have been derived from the models. To make sure that there is no distortion between the model's assumptions and the execution, task characteristics such as period, offset, jitter, priority, deadline as well as the activation time of the current and previous instances are monitored during execution using the CPAL introspection features. In Section VI.5, we discuss the monitoring of CPAL model execution at run-time, especially the monitoring of timing tolerance specifications such as input jitters, output jitters and the input-to-output delays.

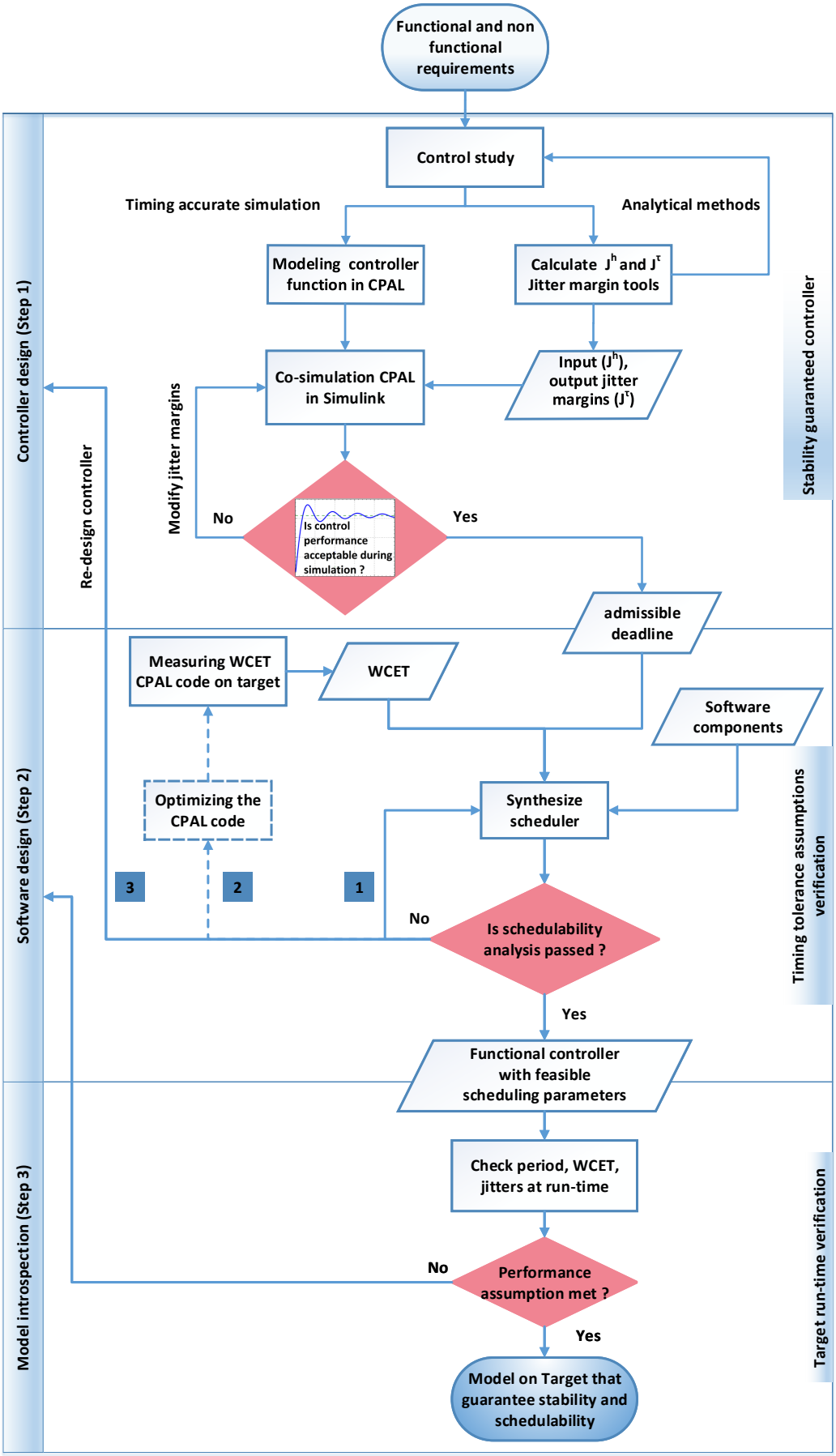


Figure VI.1: Illustration of framework flow for fusing control and scheduling view-points. The dashed part in the software design step is out-of-scope of this chapter.

VI.3 Analysis and Co-Simulation of Controller Design

This section explains the controller design using analytical methods and co-simulation. The result of this stage is a controller whose stability and more generally performance are guaranteed under certain assumptions on the worst-case timing behavior of the software implementation.

VI.3.1 Jitter Analysis

Jitter analysis is performed using two evaluations, namely the evaluation of the expected control performance, and of the worst-case control performance. For instance, the Jitterbug toolbox [Lincoln and Cervin, 2002] can be used to calculate the expected value of quadratic control costs. This measure in the general case is not sufficient to guarantee the stability of the plant [Aminifar et al., 2012], but stability can be verified through worst-case control performance analysis. In our framework, the technique presented in [Cervin, 2012] and implemented in the jitter margin toolbox is used for the derivation of the jitter margins, both input and input-to-output delays, ensuring stability under the worst-case control performance. The calculated jitter margins imply the maximum deadline for a controller task. This theoretical bound on the deadline derived by analysis may be further fine-tuned by simulation as explained in the next subsections.

VI.3.2 Controller Modeling in CPAL

CPAL, short for Cyber-Physical Action Language, is a modeling and discrete-event simulation language for cyber-physical systems [Navet et al., 2016b]. CPAL serves as a design-exploration platform with graphical representation. The models can be executed both in simulation mode as well as in real-time mode on an embedded target. CPAL is a lightweight execution engine (around $10,000$ lines of C code) designed for timing predictability that can run on top of an OS or without any OS, and thus without the interferences the OS would create.

In case of simulation, execution is as fast as possible according to a logical clock and not the physical time (see [DesignCPS, 2019]). Typically, executing in simulation mode is several orders of magnitude faster than in real-time mode. The controller code executes in zero-time during simulation, except if it uses predefined CPAL timing annotations. The simulation mode CPAL interpreter is an execution engine hosted by an operating system. The simulation execution can be carried out in a stand-alone built-in simulation environment [Fejoz et al., 2016b] or it can be used in co-simulation environments, for instance as in this work integrated in MATLAB/Simulink as an S-function. CPAL aims to achieve the same temporal behavior in simulation mode and real-time mode on the target. This property is referred to as *timing equivalence*. It can be achieved through timing annotations to inject delays in the simulation model. Figure VI.2 illustrates the CPAL timing annotations to inject input and output jitters in a control model.

Like other modeling environments for control programs such as StateFlow, CPAL provides support for Finite State Machines (FSMs) with conditional and timed transitions. As can be seen in Figure VI.3, transitions can happen either when a

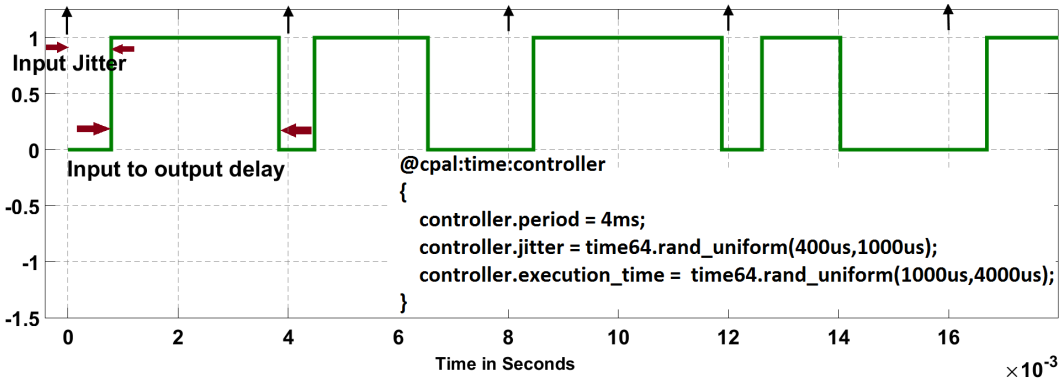


Figure VI.2: Simulating random input and output jitters affecting a CPAL controller model using timing annotations. Level 1 means that the controller is being executed.

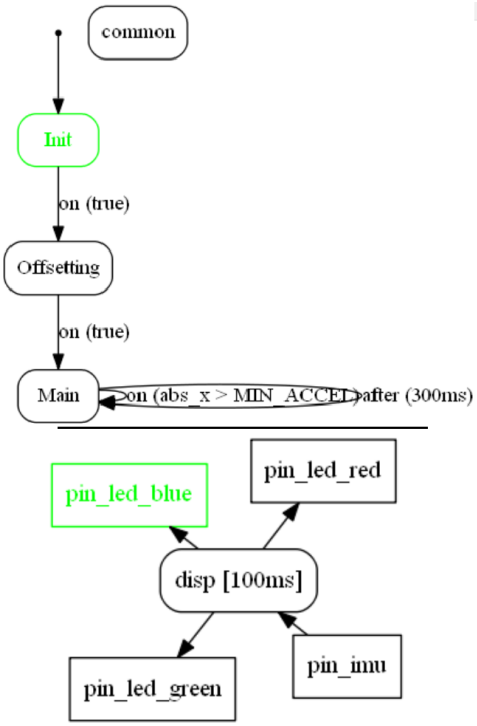
boolean condition is true, after a certain time duration is spent in the active state, or the conjunction of both. A distinctive feature of CPAL is that it relies on model interpretation: a CPAL model verified by simulation can be executed directly on an embedded target such as ARM Cortex - M4 (FRDM K64F) and ARM Cortex - A7 (Raspberry Pi). Model-interpretation is well suited for rapid-prototyping [Sundharam et al., 2016a] and prevents any distortion between models and code that could be introduced during code generation. A disadvantage of model interpretation is that it is slower than compiled code. For that reason, it is not always a practical solution for on-target execution. For the purpose of simulation on desktop machines, the execution time of the control part is however not an issue, especially in a co-simulation environment where simulating the plant is by far the most time-consuming task.

The CPAL documentation, a graphical editor and the execution engine for various desktop and embedded platforms are freely available at `http://www.designcps.com`. The CPAL control library that is needed to execute CPAL model in Simulink environment and the models to reproduce the experiments of this chapter are freely available at `https://www.designcps.com/wp-content/uploads/cpal_codesign_framework.zip`.

VI.3.3 Co-simulation in MATLAB/Simulink

In our proposed co-simulation approach, a controller model is designed in CPAL, and the plant model in Simulink. Controllers can easily be designed in Simulink too. However, Simulink out-of-the-box is not offering possibilities to study the performance of control loops subject to scheduling and networking delays. Indeed, varying execution times, preemption delays, blocking delays, kernel overheads cannot be captured in the standard Simulink environment. This can be done only with TrueTime [Cervin et al., 2003a], which, to the best of our knowledge, is the most widely used tool in the real-time and control communities to study control performance subject to timing irregularities. One should also cite T-Res [Morelli et al., 2014], a more recent and modular version of TrueTime.

In [Sundharam et al., 2016d], we have discussed how to integrate the timing behaviour of the controller into Simulink models. In this work, we have studied how CPAL timing-accurate interpretation in Simulink compares against TrueTime and T-Res.




```
121  state Init {
122    IO.println("Initialising...");
123    set_red(red, green, blue);
124  }
125  on (true) to Offsetting;
126  state Offsetting {
127    offsets.accel_x = -imu.accel_x;
128    offsets.accel_y = -imu.accel_y;
129    offsets.accel_z = 2048 - imu.accel_z;
130    IO.println("offsets: %i %i %i", offsets.accel_x
131  }
132  on (true) to Main;
133  state Main {
134    if (abs_x < MIN_ACCEL and abs_y < MIN_ACCEL) {
135      green = false;
136      blue = false;
137    } else {
138      if (abs_x > abs_y) {
139        if (imu.x < -MIN_ACCEL) {
140          set_white(red, green, blue);
141        } else {
142          set_blue(red, green, blue);
143        }
144      }
145    }
146  }
147  on (abs_x > MIN_ACCEL or abs_y > MIN_ACCEL) {
148    IO.println("Resetting timer...");
149  } to Main;
150  after (300ms) {
151    red = not red;
152  } to Main;
```

Figure VI.3: CPAL program illustrating the native support for FSM, conditional and timed state transitions. The top-left graphic is the representation of the FSM embedded in a process, while the bottom-left graphic is the functional architecture with the flows of data, as both seen in the CPAL-editor.

The important difference, and also the advantage of our co-modelling approach is that the same model used during simulation can be used on target, whereas TrueTime and T-Res are simulation environments. Also, like Simulink, CPAL is a high-level embedded systems specific language which favors productivity and correctness by providing domain-specific constructs and abstractions [Navet and Fejoz, 2016a]. In the case of the co-simulation of CPAL within MLSL, Simulink acts as the primary simulator while CPAL executes the controller model as an S-function, and is being called by the Simulink engine. The S-functions (system-functions) are high-level programming language description of a Simulink block written in C, C++ etc. The CPAL control library is implemented as a *mex* (Matlab Executable) file, which executes the CPAL controller model. This CPAL controller is a generic execution engine that can run any CPAL model. Before execution, the CPAL source model is converted into a binary-equivalent representation (an Abstract Syntax Tree, shortly ast file format) using the CPAL parser. The Simulink engine interacts with the CPAL model through data flows and control flows. Data flow, for instance *force_out*, are used for the exchange of information between the Simulink engine and the CPAL controller, while the control flows define when Simulink invokes the CPAL S-function.

The implementation is discrete-event-based simulation using Simulink built-in zero-crossing detection. The concept of tasks and real-time schedulers are available natively in CPAL. The default CPAL scheduling policy is FIFO, but CPAL also supports Non-Preemptive Earliest Deadline First (NP-EDF) and Fixed Priority

Non-Preemptive (FPNP). In Figure VI.4, we show the instantiation of a controller task and the task parameters with the delays and jitters. A timing annotation can also specify the scheduling policy if the controller consists of several tasks. Simulation of the plant dynamics is carried-out by computing model states at successive time steps over a specified duration. This computation is done by a solver provided in Simulink. Since our overall model is discrete, a variable step size solver is used in our co-simulation approach. The rationale behind this choice is that for the timing analysis of real-time control systems, it is necessary to reduce the step size (when needed) to increase the accuracy when model states are changing rapidly during zero crossing events. Section VI.5.1 presents an example co-simulation of a simplified *cruise control system*.

```
include controller.cpal
  /* periodic process with initial offset, 10ms is period, 2ms is offset */
process PIController: invertedPendulumController[10ms,2ms](kp_in, ki_in, kd_in,
                                                           filterGain_in, angle_in, reference_in,
                                                           force_out, p_out, i_out, d_out);
 /* annotations of task timing parameters */
@cpal:time:invertedPendulumController
{
    invertedPendulumController.jitter = time64.rand_uniform(0us,500us); /* input jitter */
    invertedPendulumController.deadline = 8ms;
    invertedPendulumController.priority = 1; /* priority if needed, here only one process it is not applicable */
    invertedPendulumController.execution_time = time64.rand_uniform(0us,2000us); /* input to output delay */
}
```

Figure VI.4: Snippet of CPAL code instantiating a controller of period 10 ms and offset 2 ms and specifying the variation of the input jitter J^h and the input-to-output delay τ during a simulation run. This is achieved through a timing annotation executed in simulation, but ignored once on target.

VI.4 Timing Verification Using Schedulability Analysis

The next step in the framework is the timing verification of the controller model designed in the previous step. From the jitter margins, we derive the deadlines of the controller task(s). Typically, it will be a single task, but the controller can also be implemented as several tasks such as an input task, a computation task and an output task. The deadlines will be used for the scheduler synthesis and schedulability analysis. To obtain realistic Worst-Case Execution Times (WCET) for the schedulability analysis, we use a measurement-based technique in which the controller model is executed on the target hardware.

VI.4.1 Worst-Case Execution Time (WCET) Measurement

The CPAL controller model which we executed earlier in the co-simulation environment is now uploaded to the target platform to estimate the WCET by measurements. The CPAL model-interpretation engine is specific to a target platform, it can be executed on top of an Operating System (OS) or without an OS, the latter being called Bare-Metal Model Interpretation (BMMI). There are two ways to estimate the WCETs: using a logic analyzer or taking advantage of CPAL in-built execution-time measurement feature. The latter possibility is only available when CPAL is hosted by an OS, as freeRTOS, embedded Linux or Raspbian. It does not require connecting the target to an external measurement device and instrumenting the code, and thus provides a quick method to estimate the WCET. It is, however, less accurate than

measurements using logic analyzer, since it involves additional run-time overhead in the interpretation engine.

For the discrete-time PID controller used in Section VI.5, the measured WCET of the CPAL controller task using logic analyzer is 34.4 µs on a Raspberry Pi2 model B Figure VI.5. This can also be obtained using the in-built feature of CPAL *--stats*, a command-line option to be used when we execute the model on target, Figure VI.6. When we remove the code of the actual control algorithm, leaving just the skeleton of the tasks, we can observe the scheduler overhead, which amounts to 155 µs. When we execute the model as it is, we observe the scheduler overhead plus the execution time of the task to be 189 µs. The difference between these two values would then provide the execution time of the task, 34 µs, which is indeed observed also on the logic analyzer. With an ARM Cortex-A7 core at 900 MHz, Raspberry Pi is a cost-effective development platform to experiment with CPAL but it is not suited for executing real-time applications due to large timing variabilities (e.g., jitters in task release times). The best supported platform with respect to timing predictability is the NXP FRDM-K64F, a SOC on which the CPAL execution engine runs on the bare hardware, thus without any interference and latency from an OS. As provided in the Supplementary Materials (both WCET measurement and jitter measurements), we experiment the same controller model on FRDM-K64F target too, which is a BMMI target. Despite BMMI, due to inferior hardware configuration, we observe that the same task takes 340 µs to execute on the FRDM-K64F, about 10 times more than on the Raspberry Pi. We present the model on target experiments of Section VI.5 with Raspberry Pi because we could output the jitter measurements on the console at run-time through process introspection features. CPAL on FRDM-K64F does not have a facility to provide console outputs. In this case, a logic analyzer helps us to monitor the model executed on the target.

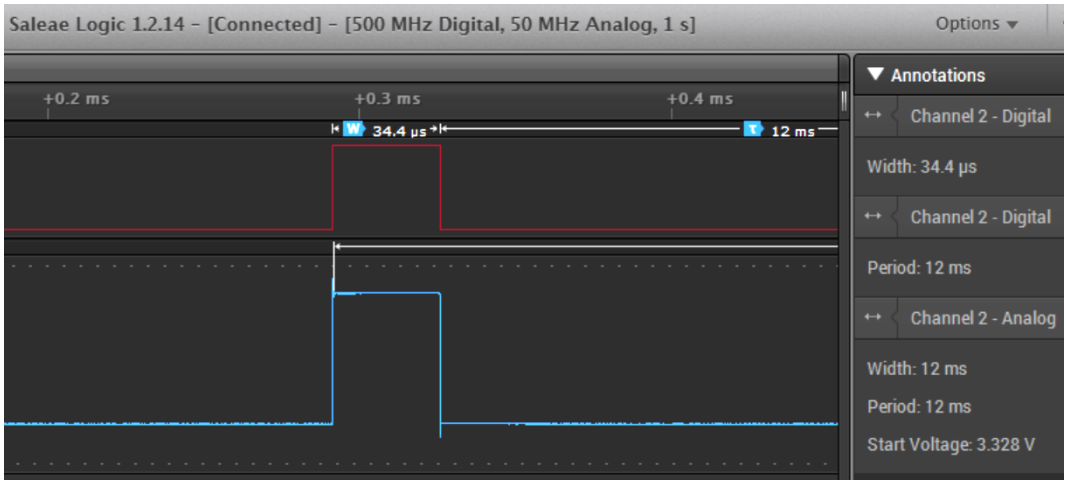


Figure VI.5: WCET measurement of a controller model executed on an ARM Cortex-A7 core using logic analyzer.

Deriving safe and precise WCET bounds is a difficult issue in itself (see [Wilhelm et al., 2008b] for a survey), and determining WCET estimates using state-of-the-art techniques and tools is outside of the scope of this work. Although it is a practical approach widely employed in the industry, using measurements as done in this work carries the risk of being unreliable because the worst-case situation might not

have been observed. This becomes especially true for complex systems, with many tasks and architectures including multiple cores and multiple levels of caches. In such settings, more advanced WCET estimation techniques must be employed. Our framework would however work with any other WCET estimation techniques such as static deterministic analysis or probabilistic analysis. For instance, it is possible on the basis of the measurements to provision for a safety margin, typically using probabilistic arguments [Cazorla et al., 2013]. This margin can for instance account for cache latencies which have not been considered here. Another option is to employ an analytic WCET analysis, generally considered safer than measurement-based techniques, although much more conservative.

```
3. 10.145.15.132 (pi)
process controller: p1[12ms](data_control1, r_in, y1_in, T1_out);
init()
{
        data_control1.K = 0.96;
        data_control1.Ti = 0.12;
        data_control1.Td = 0.049;
        data_control1.beta = 0.5;
        data_control1.N = 5.0;
        data_control1.Iold = 0.0;
        data_control1.Dold = 0.0;
        data_control1.yold = 0.0;
        data_control1.h = 0.012;
}
@cpal:time:p1{
        p1.priority = 10;
        p1.offset = 0ms;
        p1.period = 12ms;
        system.sched_policy = Scheduling_Policy.FIFO;
        pin0_out = true;
              IO.sync();
}
pi@raspberrypi:~/Downloads/CPAL_121 $ sudo chrt 60 ./cpal_interpreter_raspberry -r -t 5000 -z --stats wcet.ast
setup
  => Digital pin found: 0, output
Quiet mode enabled
EXIT (SUCCESS) AT 7mn2s272ms139us309ns (422272139309000)
@cpal:time {
  p1.wcet = 183us593ns;
}
pi@raspberrypi:~/Downloads/CPAL_121 $ sudo chrt 60 ./cpal_interpreter_raspberry -r -t 2000 -z --stats wcet.ast
setup
  => Digital pin found: 0, output
Quiet mode enabled
EXIT (SUCCESS) AT 7mn7s831ms326us443ns (427831326443000)
@cpal:time {
  p1.wcet = 188us698ns;
}
pi@raspberrypi:~/Downloads/CPAL_121 $ sudo chrt 60 ./cpal_interpreter_raspberry -r -t 3000 -z --stats wcet.ast
setup
  => Digital pin found: 0, output
Quiet mode enabled
EXIT (SUCCESS) AT 7mn16s135ms932us80ns (436135932080000)
@cpal:time {
  p1.wcet = 199us322ns;
}
```

Figure VI.6: WCET measurement for a controller model executed on an ARM Cortex-A7 core in real-time mode. Command-line option - *-stats* indicates that the WCET of the controller model measured during execution.

VI.4.2 FIFO Scheduling to Simplify Design and Verification

We are interested in devising an environment that eases the design and verification of embedded real-time systems. A main goal is to provide an environment where also the inexperienced designers are able to quickly model and deploy trustworthy embedded systems without for instance having to master real-time scheduling theory and resource-sharing protocols. Especially corner case faults due to different timing behaviors or race conditions can be a nightmare to debug. We acknowledge that

techniques to avoid these problems exist, but they require experience and make both the design and the code more complex and error-prone. When processing power is sufficient other concerns than performance, such as simplicity and predictability, can be considered. In our context, as shown in [Altmeyer et al., 2016], FIFO exhibits two properties which greatly eases the verification:

- Deterministic execution order: the execution order of FIFO scheduling with offset and strictly periodic task activation is uniquely and statically determined. This means that whatever the execution platform and the task execution times, be it in simulation mode in a design environment or at run-time on the actual target, the task execution order will remain identical. Beyond the task execution order, the reading and writing events that can be observed outside the tasks occur in the same order. This property, leveraged by the CPAL design flow [Navet et al., 2016b], provides a form of timing equivalent behavior between development and run-time phases which eases the implementation of the application and the verification of its timing correctness.
- Execution time sustainability: FIFO scheduling is sustainable in the tasks' execution times, meaning that if a task set is deemed schedulable and the execution times of the tasks are reduced, the task set remains schedulable.

The latter property allows simulation as a valid technique for schedulability verification. In practice, however, the simulation time required can be unpractical if the least-common multiple of the task periods is too large. A schedulability analysis does not suffer from this limitation. In this context, we derive a schedulability analysis for FIFO scheduling on uniprocessor systems with strictly periodic task activation and tasks having release offsets. It should be noted that the use of offsets is a technique which increases the ability of FIFO to meet deadlines, no matter if the offset of a task is unique as in this work (see the experiments in [Altmeyer et al., 2016]) or may vary, as in [Nasri et al., 2018]. With offsets, FIFO becomes a candidate scheduling policy for low-memory embedded hardware with constrained run-time overheads.

We proposed in [Sundharam et al., 2016b] a scheduling synthesis approach, where performance, hardware and functional constraints only need to be specified to derive a feasible low-level scheduling configuration. The framework proposed in this chapter is compatible with any scheduling policy that guarantees that the deadlines will be met, although in the remainder of this chapter, we will rely on FIFO which, as explained, facilitates the system design.

VI.4.3 FIFO Schedulability Analysis

In the previous Chapter we discussed the schedulability analysis of FIFO with offsets for periodic task set. Algorithm 2 consolidates the analysis presented so far. Using this algorithm, we can derive the worst-case response times of all task. To check schedulability, we verify that these response times are less than or equal to the fine-tuned deadlines, which we have obtained from the previous step. To achieve transparency and to ease the reproduction of the results, the source code of the programs used in our experiments, including the schedulability test, is available online (`https://www.designcps.com/wp-content/uploads/cpal_codesign_framework.zip`). The source code enables the reproduction of the experiments presented in this chapter, as

well as evaluation for different parameters settings. The tool *cpal2x* (see [DesignCPS, 2019] for usage), which is available in the CPAL distribution, extracts the timing information (timing and scheduling annotations) from the controller function designed at Step 1. This constitutes the system task model which is then inputted to the presented schedulability analysis.

Algorithm 2 Worst-Case Response time R_i^w

$i = 1$
isSchedulable = true
L = computeBusyPeriod
while $i \leq n \wedge$ isSchedulable **do**
  $\hat{r}_i^j = L$
  $\hat{O}_i = r_i^j \mod h_i$
  for all l **do**
    $\hat{dist}_{i,l}$ = computeMinDistance(i, l)
    $\hat{O}_l = r_i^j - \hat{dist}_{i,l} \mod h_i$
  end for
  $Q = \{t | \exists l, k \colon t = k \cdot h_l + \hat{O}_l \wedge t \leq L\}$
  for all $t \in Q$ **do**
    if $PD(t, \hat{r}_i^j, i) - t > \hat{d}_i^j$ **then** isSchedulable = false
    end if
    if isSchedulable **then break**
    end if
    $\hat{f}_i^j = \{PD(t, \hat{r}_i^j, i) + t\}$
  end for
  $\hat{f}_i = \max\{\hat{f}_i^j\}$
  $R_i^w = \hat{f}_i - \hat{r}_i$
  $i = i + 1$
end while
return isSchedulable
return R_i^w

VI.5 Evaluation and Results

We now evaluate the framework with the help of an automotive control system. Before presenting the evaluation, we describe the system model. As depicted in the framework of Section VI.2, the evaluation consists of three steps. Firstly, we calculate the tolerable jitter margin values under which the system remains stable using jitter margin analysis. The calculated output jitter margin provides the maximum deadline for the controller task. This deadline is further fine-tuned using co-simulation that

provides additional and more fine-grained information about the control performance. Secondly, we evaluate the schedulability of the controller task when executed with other tasks in the system. Finally, in the third step, we use CPAL introspection to check that the run-time behavior of the controller task complies with the design assumptions.

VI.5.1 Motivating Example : Cruise Control ECU

A Cruise Control system maintains the speed of a car at a desired level. For that, the system uses a servo mechanism that takes over the throttle of the car to maintain a steady speed as set by the driver. The system model used is taken from the Simulink reference examples [SimEvents, 2019], but the controller model is replaced by a CPAL implementation. Without injecting run-time delays, both the CPAL implemented version of the controller and the Simulink version provide the same outputs. This comparison is available in the Supplementary Materials provided. Figure VI.7 shows the architecture of the co-simulation model. The proposed co-simulation environment provides the control performance with the run-time delays due to execution times and interferences from higher priority tasks, and facilities the visualization of the task scheduling. In addition, the same controller model developed for simulation can be executed on the target by the CPAL execution engine.

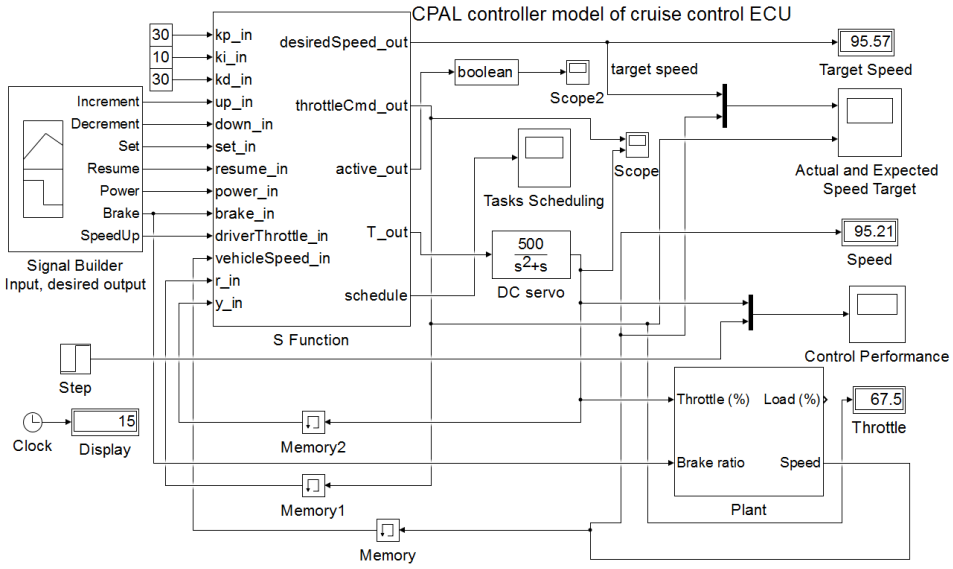


Figure VI.7: Illustration of a CPAL controller in Simulink. Here, the CPAL model controls the servo which in turn actuates the engine throttle. The controller task is executed with simulated input-to-output delays.

In our implementation, different tasks and variables are defined within the controller model. We consider tasks, namely *set point manager*, *cruise control manager* and *sensors manager*. The label *T_out* in Figure VI.7 is the controller tasks' output which actuates the DC servo mechanism controlling the throttle valve. We model the DC servo with the transfer function $P(s) = \frac{500}{(s^2+s)}$. The controller developed relies on a

PID control algorithm with proportional gain $K_p = 0.96$, derivative gain $K_d = 0.049$, integral gain $K_i = 0.12$ and filter divisor $N = 5.0$.

VI.5.2 (Step 1) Controller Design

The evaluation of the controller designed consists of two steps, namely the analytical jitter margin method and the co-simulation technique.

VI.5.2.A (Step 1. a) Stability Verification Using Jitter Margin Concept

For a given controller and a nominal input-to-output delay $\mathfrak{L}$, the Jitter margin toolbox [Cervin, 2012] computes the tolerable level of jitter for which stability is guaranteed (like phase margin and gain margin computations of control systems). This toolbox provides the stability curve that determines the maximum tolerable output jitter J^{τ} and maximum tolerable input jitter J^{h}, based on the nominal input-output delay $\mathfrak{L}$. Figure VI.8 shows the worst case control cost which is a H_{∞} (H-infinity) performance metric calculated for different input and output jitters. For example, for the PID controller of the previous sub-section with a sampling period of 12 ms, the nominal (minimum) input-to-output delay $\mathfrak{L}$ is equal to 5.6 ms, the input jitter margin J^{h} is 3.64 ms, the output jitter margin J^{τ} is 5.45 ms, while the control cost H_{∞} is 72.13 ms. The input-to-output delay, which is the sum of encountered jitters during the execution of the controller task is then 9.09 ms. The control cost we use is H_{∞}, a gain parameter calculated when we apply a disturbance input to the plant and the corresponding output amplifies.

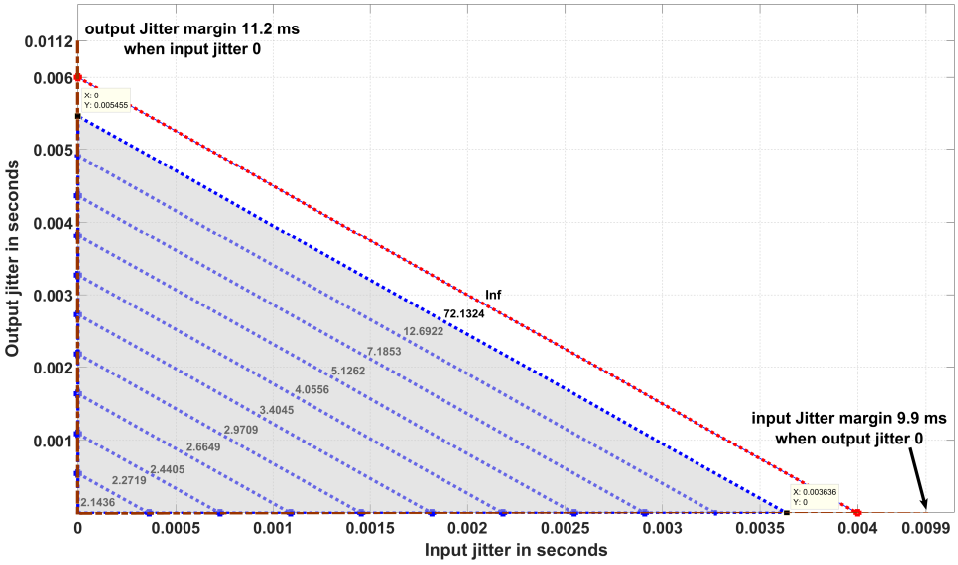


Figure VI.8: The worst-case control cost calculation during various input and output jitter occurrences. The control cost mentioned here is H_{∞}, a gain parameter. Finite control costs indicate that the system is stable while an infinite value $'Inf'$ indicates that the system tends to be unstable. At zero input and zero output jitter, the highest performance is achieved. The control cost increases when jitters increase.

When we get a finite value for the gain parameter H_{∞}, it indicates that the system remains stable. Beyond the jitter margin, we observe that the gain becomes infinite,

which means that the system tends to be unstable. During the ideal situation where the controller task executes with zero input jitter and zero output jitter, we obtain the highest possible control performance with control costs H_∞ equal to 2.14. If we want to guarantee a certain control performance, expressed in terms of H_∞, we have to design the system such that the experienced jitters are within the jitter margins leading to H_∞ being no greater than the target. For instance in Figure VI.8, the jitters must remain in the shaded region to ensure that H_∞ remains equal to 72.13. This allows to deduce that the controller task deadline must be less than 9.09 ms. Now, to fine-tune the deadline and also to study the effect of scheduling choices on the control performance, the co-simulation approach is used. In the implementation of the cruise-control system, the controller task, denoted $Task$ 1, is activated every 12 ms. $Task$ 2 is another task with the same period, always activated before $Task$ 1. In case both are released at the same time, the execution time generates an input jitter for $Task$ 1. This latency, plus the varying execution time of $Task$ 1 itself, induce the output jitter.

Figure VI.9 illustrates the execution of the tasks under First-In First-Out (FIFO) policy. As we can see from Figure VI.9, $Task$ 1, the controller under design, experiences an input jitter of 3.64 ms. This is realized by means of an execution time annotation (see Section VI.3.3) of an interfering task activated immediately before. By setting the execution time of $Task$ 1 to 5.45 ms, combined with the input jitter, we enforce an input-to-output delay of 9.09 ms. This is the tolerance level beyond which the system performance degrades significantly, as shown in Figure VI.10.

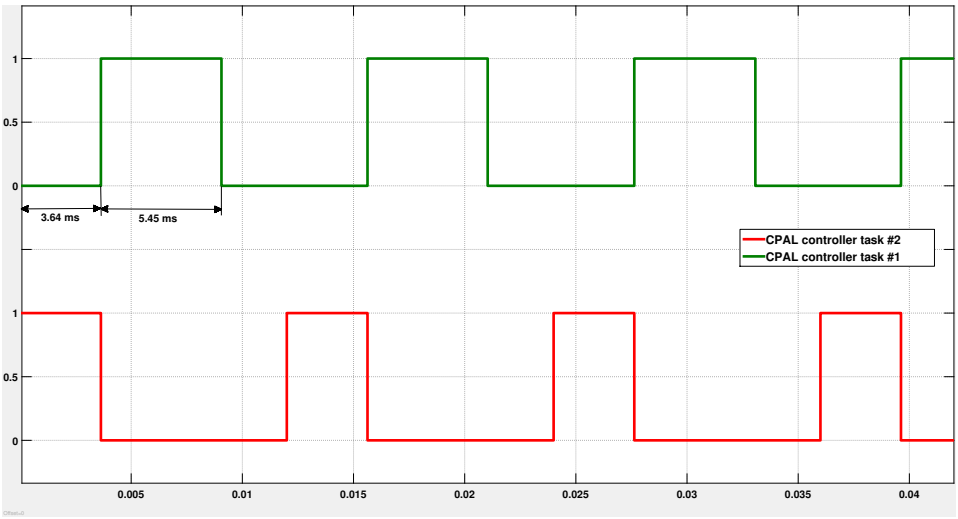


Figure VI.9: Successive activations of two tasks under FIFO. $Task$ 1 is the controller we design with a period of 12 ms. $Task$ 2 is the *cruise control manager* also with a 12 ms period. As $Task$ 2 is of higher priority, it is activated first when both tasks are released simultaneously. Using varying execution time annotations for $Task$ 1 and $Task$ 2, we enforce an input-to-output delay of at most 9.09 ms for $Task$ 1, which is the bound obtained from jitter margin analysis.

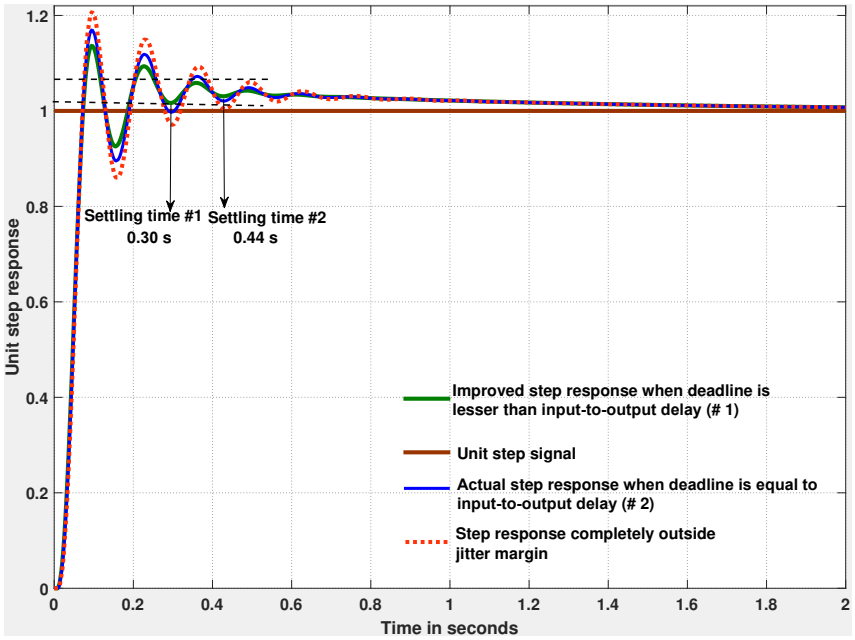


Figure VI.10: Control performance using step response for different deadline assignments: equal, less and greater than the input-to-output delay (resp. blue, green and red curves). The green curve (reduced overshoot one) is obtained with a deadline value equal to 8.2 ms chosen such that the settling time within 2% of the steady-state value is less than 0.3 s. When the control task deadline is greater than the jitter margin, logically the system performs poorer with increased oscillations and overshoots.

VI.5.2.B (Step 1. b) Co-Simulation CPAL/Simulink

The co-simulation of CPAL in the Simulink environment serves two purposes: fine-tuning of the deadline and selection of the scheduling policy. Although the jitter bound derived by jitter margin analysis helps to assign the deadline, in practice a system designer may want to evaluate the control performance with the response to an input elementary signal such as impulse or a step signal. For this purpose, we feed an unit step signal in the co-simulation model to study the step response of the system. Based on the step response characteristics such as rise time, settling time and overshoot, we can decide whether a fine tuning of the deadline is necessary. For instance, if the control requirement is to achieve a desired settling time, defined as the time taken to settle within 2% of the steady state value, equal to 0.3 s, then the deadline should be no greater than 8.2 ms (versus 0.44 s with a deadline of 9.09 ms). In our previous work [Sundharam et al., 2016d], we have exemplified the co-simulation of CPAL in Simulink to study the control system performance for different scheduling options.

VI.5.3 (Step 2) Software Design

As explained earlier in Section VI.4.1, the controller model we designed at step 1 is now uploaded on the target platform to estimate a WCET bound. For the specifications of the controller with the sampling period of 12 ms (see Section VI.5.1), the execution

time of the CPAL controller task measured using a logic analyzer is 34.4 µs. As explained in Section VI.4, WCET estimation can also be conveniently performed using the CPAL in-built *--stats* feature. For the controller task developed, the maximum execution time value observed is around 200 µs including the scheduler overhead. When there are no preemptions as here, or a bounded number of preemptions, it is possible to include the scheduler overhead in the WCET of the task. To provision for a safety margin, we consider the execution time along with the scheduler overhead. We use this WCET and the admissible deadline of 8.2 ms obtained from the previous step to test the schedulability of the system. The schedulability analysis presented in Section VI.4.3 tells us whether the integrated task set (the controller under design plus the existing tasks on the ECU) is feasible or not. In our experimental setup, the task set passes the schedulability test.

Once we obtain a stable controller model, we verify its run-time behavior on the target hardware. At run-time, it is possible in CPAL for a process instance to query its id, period, offset, both input and output jitters, priority, deadline and the activation times of the current and previous activations. Statistics can be collected and analyzed off-line, but it is also possible to visualize at run-time the variation of these quantities. Figure VI.11 shows a snippet of the code of the two monitoring tasks of $Task$ 1, one for the input jitter and one for the output jitter, as well as their scheduling parameters. Here the choice has been made to have external tasks monitoring the jitters in order to not clutter the controller code. Although FIFO is the scheduling policy, simultaneous task releases are broken with the *priority* attribute (see Figure VI.11).

```
        IO.sync();
        IO.println("id %u input jitter %t Priority %u Period %t Offset %t", p.pid,
         p.current_activation - p_mon_init.current_activation, p.priority, p.period, p.offset);
        IO.println("id %u output jitter %t Priority %u Period %t Offset %t", p.pid,
         self.current_activation - p.current_activation - p.bcet, p.priority, p.period, p.offset);
        pin = false;
        IO.sync();
        }
}
@cpal:time{
          p2_mon_in.priority = 6;
          p2_mon_in.offset = 0ms;
          p2.priority = 5;
          p2.offset = 0ms;
          p2.period = 12ms;
          p2_mon_out.priority = 4;
          p2_mon_out.offset = 0ms;
          p1_mon_in.priority = 3;
          p1_mon_in.offset = 0ms;
          p1.priority = 2;
          p1.offset = 0ms;
          p1.period = 12ms;
          p1_mon_out.priority = 1;
          p1_mon_out.offset = 0ms;
          system.sched_policy = Scheduling_Policy.FIFO;
}
pi@raspberrypi:~/Downloads/CPAL_121 $ sudo chrt 60 ./cpal_interpreter_raspberry -r -t 10000 -q observer.ast
setup
  => Digital pin found: 0, output
  => Digital pin found: 1, output
  => Digital pin found: 2, output
  => Digital pin found: 3, output
  => Digital pin found: 4, output
  => Digital pin found: 5, output
Quiet mode enabled
[2137952.754274563000:PRINTLN] id 0 input jitter 133us177ns Priority 5 Period 12ms Offset 0
[2137952.754930812000:PRINTLN] id 0 output jitter 162us396ns Priority 5 Period 12ms Offset 0
[2137952.755765551000:PRINTLN] id 1 input jitter 1ms441us561ns Priority 2 Period 12ms Offset 0
[2137952.756385186000:PRINTLN] id 1 output jitter 116us510ns Priority 2 Period 12ms Offset 0
```

Figure VI.11: Code snippet of the two monitoring processes, including their scheduling parameters.

VI.5.4 (Step 3) Introspection Features for Run-Time Verification

During model on target execution, the run-time monitoring tasks are respectively executed before the start of the controller task and immediately after. This can be for instance ensured by setting the priority attribute so that the input-monitoring process is at a higher priority than the controller task (3 in our case), while the output-monitoring task is at the immediate lower priority (1 in our case). The controller $Task$ 2 is the *cruise control manager* of higher functional importance. It is activated first when both $Task$ 1 and $Task$ 2 are released simultaneously. The lower part of Figure VI.11 shows a sample console display of the input and output jitters during command-line execution in real-time mode (i.e. option $-r$ in the command line) of the CPAL controller model with quiet option $-q$ enabled. Here, jitters are recorded for 10 s on a Raspberry Pi2 model B with an ARM Cortex A7 processor.

Figure VI.12 shows the input jitter measurements of the two controller tasks, $Task$ 1 and $Task$ 2, over a duration of 10 s. Even if $Task$ 1 suffers delays from $Task$ 2, we observe that its input jitters are well within the input jitter margin value of 3.64 ms (see Section VI.5.2.VI.5.2.A). Likewise, Figure VI.13 shows the output jitter measurements for both controllers, and the cruise-control system meets the 5.45 ms output-jitter margin. These experiments, along with logic analyzer measurements confirm the design assumptions related to jitters. The logic analyzer set-up and captures files are available as additional references within the supplementary materials provided with this report.

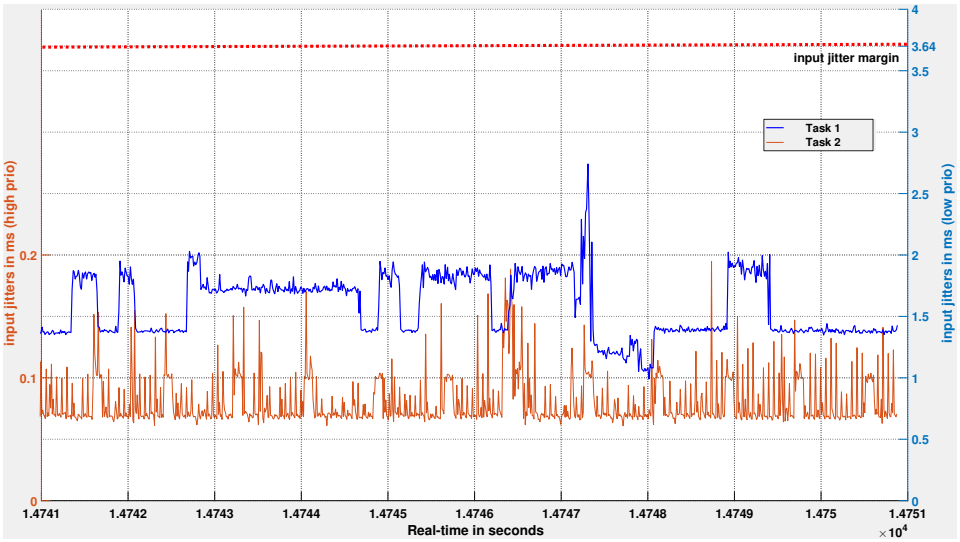


Figure VI.12: Global view of input jitter measurements of $Task$ 1 and $Task$ 2. The input jitter J^h of $Task$ 1 (blue curve, right y-axis) varies over time below 2 ms, except in rare cases where it reaches 2.7 ms. The input jitter of $Task$ 2 (red curve, left y-axis) is bounded by 0.2 ms. The design assumption of input jitters for $Task$ 1 is less than 3.64 ms is met by the implementation.

We enabled $-q$ (quiet) option during model execution to get only the necessary console outputs, which are the jitter values during run-time. We record these jitter values for the purpose of visualization and to study whether the jitters are within the margins. To cross-check, we also measure the jitters using a logic analyzer with a 100 MHz sampling rate for about 10 s, as shown in Figure VI.14. For a particular job

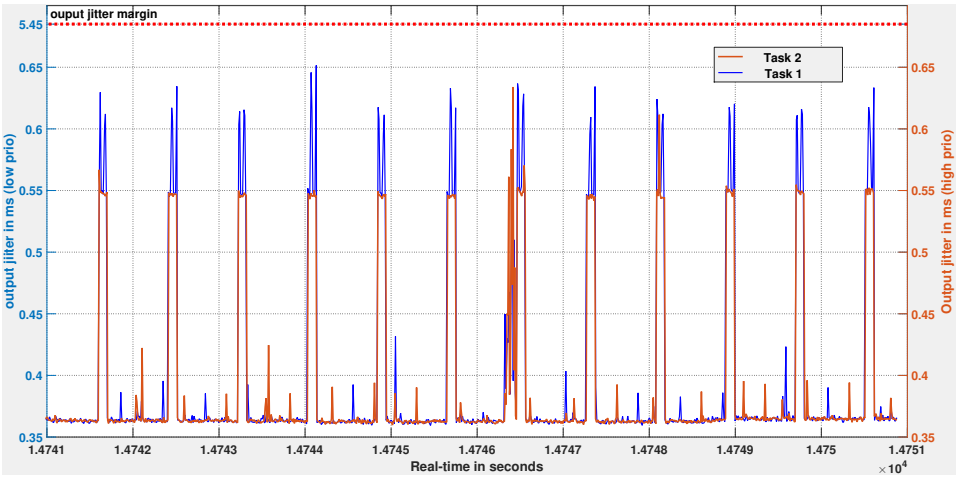


Figure VI.13: Global view of output jitter measurements of $Task$ 1 and $Task$ 2. The output jitter J^{τ} of $Task$ 1 (blue curve, left y-axis) varies over time but remains below 0.65 ms. The output jitter of $Task$ 2 (red curve, right y-axis) is bounded by 0.63 ms. The design assumption of output jitters for $Task$ 1 is less than 5.45 ms is met by the implementation.

instance (zoomed portion of the figure), we measure an execution time of 34.58 µs for controller $Task$ 1 and 25.19 µs for controller $Task$ 2, which both run with a period of 12 ms. The monitoring processes (input and output) are here to help measure the input jitters, output jitters and input-to-output delay. We observe that when we do not include the printing of jitter values on the console, both input monitor and output monitor tasks (i.e., channels 1, 3 for $Task$ 2 and channels 5, 7 for $Task$ 1) consume less than 4 µs. Note that these monitoring tasks can be removed for the production code once the design is finalized to avoid overhead.

VI.6 Related Works of Control Function Co-design

In the literature of computing and control, there have been numerous studies on the effects of timing irregularities on control performance [Torngren et al., 2006; Cervin et al., 2003a; Morelli et al., 2014; Sundharam et al., 2016d]. Cervin et al. coined the term *jitter margin* in [Cervin et al., 2004], where the authors considered the output jitter margin under which the system still maintains its stability. In a subsequent work [Cervin, 2012], Cervin extended the analysis to account for both the input and output jitters on the control performance of linear sampled-data control systems. In this chapter, we integrate this analysis in a tool-supported design flow which guarantees the control performance on a given execution platform.

A technical contribution needed in this work is a FIFO schedulability analysis for periodic tasks with offsets. Closely related are the results by George and Minet published in [George and Minet, 1997], who proposed a scheduling analysis for FIFO on a distributed system assuming sporadic task releases, and the results by Leontyev and Anderson [Leontyev and Anderson, 2007], who developed a tardiness analysis for FIFO scheduling of soft real-time tasks, also assuming a distributed system and

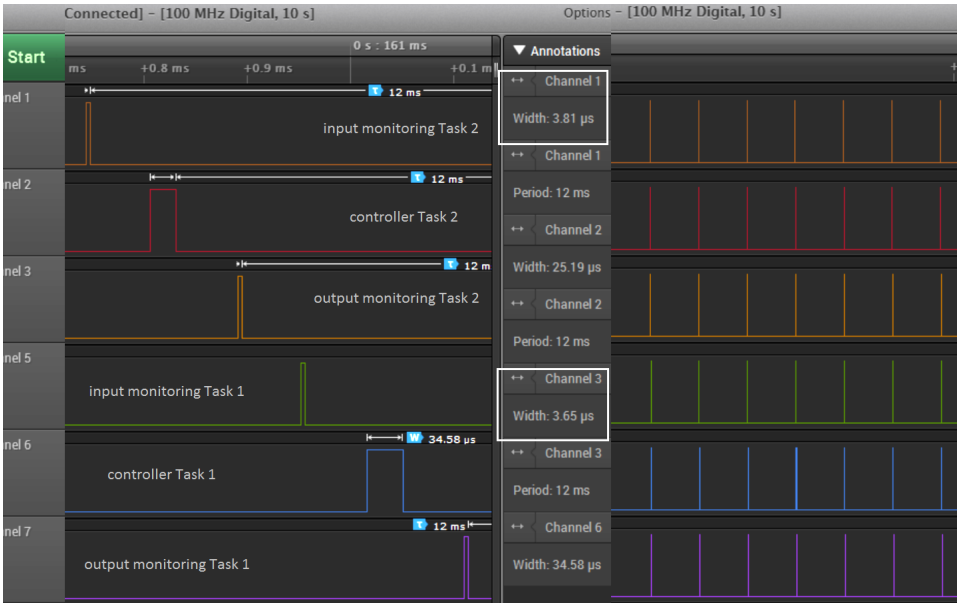


Figure VI.14: The input and output monitoring task activations for two controller tasks captured using a logic analyzer. Both the controller tasks *Task* 1, *Task* 2 are activated with a period of 12 ms. Both are different control algorithms which run for an execution time of 34.58 µs and 25.19 µs, respectively at the highlighted job instant. The monitoring tasks execute only a fraction of the controller's computation time, typically less than 4 µs.

sporadic task releases. The two latter works did not apply directly to our task model, i.e., periodic task with release offsets.

Sangiovanni-Vincentelli et al. discussed various methodologies to address the system design challenges in [Sangiovanni-Vincentelli et al., 2012]. This work highlights the importance of *Assume/Guarantee* contracts during component design and explains how a contract can be applied to the design of a water flow control system. Derler et al. proposed in [Derler et al., 2013] that implicit timing assumptions are made explicit using design contracts to facilitate the interaction and communication between control and software domains. The authors discussed the support for timing-contracts-based designs using Ptolemy and Simulink. Benveniste et al. proposed in [Benveniste et al., 2012] to apply contracts to design methodologies. Importantly, the authors explained the mathematical concepts and operations necessary for the contract framework. All these works mentioned in this paragraph focused on the fundamental framework for design contracts, such as contract algebra applied in system design, and timing contract visualization in modeling environment. In this work, we are concerned with the application of timing tolerance contract in our Model-Based Design flow used to develop control software, thus focusing on scheduling and implementation issues.

In terms of related design environments, we identify two approaches with associated tools aiming to support control system design considering the influence of scheduling strategies:

- *TrueTime*: this MATLAB/Simulink-based tool [Cervin et al., 2003a] enables the simulation of the temporal behavior of controller tasks executed on a multi-tasking real-time kernel. In TrueTime, it is possible to evaluate the performance

of control loops subject to the latencies of the implementation. TrueTime offers a configurable kernel block, network blocks, protocol-independent send and receive blocks and a battery block. These blocks are Simulink S-functions written in C++. TrueTime is an event-based simulation using zero-crossing functions. The tasks are used to model the execution of user code and are written as code segments in a MATLAB script or in C++. It models a number of code statements that are executed sequentially.

- *T-Res*: this more recent tool [Morelli et al., 2014] is also developed using a set of custom Simulink blocks created to simulate timing delays dependent on code execution, scheduling of tasks and communication latencies, and verifying their impact on the performance of control software. T-Res is inspired from TrueTime and provides a more modular approach to the design of controller models enabling to define the controller code independently from the model of the task.

These tools and methods focus on simulation and analysis. They both help the designer to study the control system performance under the effects of timing delays. The system designer then takes simulation analysis results into account to develop the embedded control algorithms in the next steps. This increases the possibility of distortions between the simulation model and the implementation. An advantage of our co-simulation modelling approach is that the same controller model used to evaluate the control performance during design phase can be re-used directly on the target hardware (in the coding and testing phase) to implement the system. As discussed in our previous work [Sundharam et al., 2016a,d], the reduced development cycle favors efficient interactions between control and software engineers. The reader is referred to [Sundharam et al., 2016d] for a review of CPAL in Simulink, TrueTime and T-Res development environments.

VI.7 Conclusion and Future Work

The timing behavior of control tasks is a critical concern in real-time digital controllers. The delays, such as input jitters, or missed executions due to temporary overload, affect system performance and are to be accounted for in the design phase. Model-driven engineering has been successful for capturing the functional requirements during design, but non-functional requirements such as timing have been traditionally overlooked. This leads to a late verification of controller timing and, in the best case, to corrections at a stage when they are costlier. This work is a contribution towards conceiving a design environment for embedded control systems that capture all the necessary functional and non-functional requirements, while providing analysis, simulation and run-time capabilities.

In this chapter, we presented a framework based on timing tolerance contracts which fuses the stability and scheduling viewpoints during controller design. The three steps of the framework have been described: controller design verified by stability analysis and co-simulation, software design verified by schedulability and WCET estimation, and lastly, the implementation checked through run-time verification. The crucial advantage of our co-simulation approach based on model interpretation is that the same controller model verified in the design phase can be ported directly

(without the need for code generation) to target hardware to implement the final system. This feature will ease the deployment and the update of code on distributed nodes, for instance in Industrial Internet of things (IIoT) applications.

To exhibit the framework flow, we have presented the scheduling viewpoint using novel FIFO schedulability analysis for periodic task activations with offsets. As future work, we plan to extend the framework to other schedulability analyses using tools such as Cheddar [Singhoff et al., 2004a] and MAST [Harbour et al., 2001] to support more scheduling options during scheduler synthesis. Another objective is to extend the approach to other important non-functional properties, foremost power consumption for next-generation Cyber-Physical Systems, which will require both analysis and modeling language support.

Chapter VII

Conclusions and Outlook

Abstract

The correctness of CPS usually does not only depend on its functional behavior, but also on its timing behavior. Similar to functional determinism, i.e., the same input always leads to the same output, we may want systems where events occurs at predetermined points in time. This timing determinism is the central point discussed in the thesis and provided the research findings on "Timing-aware Model Based Design with Application to Automotive Embedded Systems". MBD has been profoundly reshaping and improving the design of software-intensive systems, and embedded systems specifically. Though MBD encompasses a variety of practices, we focused on model interpretation featuring early stage timing analysis to develop control software faster and efficiently. This chapter summarizes our research contributions and gives a perspective for future research activities in the area.

VII.1 Summary of the thesis

In this dissertation we have addressed the challenges as discussed in Chapter I (Introduction) during automotive control software development by making the following contributions presented in consecutive chapters:

Improving the turn around time in the development process. Code generation is the standard practice in the industry for MBD of embedded systems, and this holds true in particular for engine function development. In this thesis, we discussed a model-interpretation development flow that is exemplified with the development of an engine coolant temperature calculation by an AUTOSAR compliant software architecture. By comparison with the usual development chains relying on code-generation and based on the case-study, we discuss the benefits of model interpretation which includes simplicity, productivity and early-stage verification possibility, specifically in the time dimension. For instance, CPAL already provides the basic mechanisms to offer timing-realistic simulation early in the design process. This leads to automate the derivation of the temporal quality-of-service required by a software module and, leveraging on model-interpretation, enforce it at run-time.

Although model-interpretation brings advantages, it is not going to cover all use-cases because interpretation is intrinsically significantly slower than compiled code. There are ways to mitigate this drawback in production code such as calling binary code from interpreted code (e.g., legacy code or specialized functions) or, possibly, selectively generating code for the computation-intensive portions of the model. Interpretation and code generation are often seen as two alternatives, not as a continuum. However, one may also imagine relying on model-interpretation, and benefits from the associated productivity gains, until the function/ECU meets all functional requirements, and then switch to code-generation for production code. This remains to be investigated in future works.

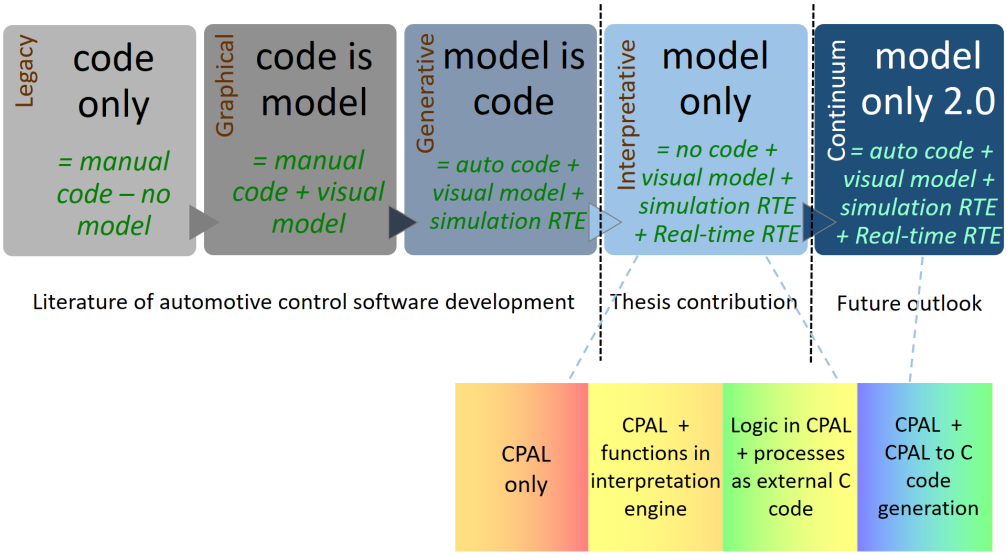


Figure VII.1: The spectrum of automotive control software development methodologies, ranging from ”code only” method to ”model only” method. The future outlook is to include the code generator to complement the advantages of ”model only” method discussed in this thesis, refer to as *Continuum* in the picture.

From all these discussions, we draw a spectrum of development methodologies as depicted in the Figure VII.1, which are ranging from *code only* to *model only* and *full interpretation* to *compiled code*. The approach *model only* also referred as *full interpretation* further leads to four different options that will bring complementary advantages of *generative* MBD. These are CPAL only, CPAL with functions hard-coded in the execution engine and linked with it, logic in CPAL and some processes as external C code and full C code generated from CPAL. In most of the systems, the last two options are faster in execution to provide best performance.

Towards a declarative modeling and execution framework. This thesis proposed a declarative modeling and execution framework using CPAL for the development of CPS to specify the 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 (multi-core, 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 current implementation and experimentation in the CPAL development environment is a contribution in that direction.

Rapid-prototyping of latency-sensitive control software. While the control laws in many embedded systems are not necessarily complex, their development is usually time consuming and their correctness verification only partial (the dynamic behaviour errors are detected very late in the integration phase, sometimes even once the product has been released). We believe that this is partly due to the excessively complex run-time environment and the inability of the languages and development tools to efficiently support the real-time constructs. Many critical embedded systems would probably benefit, in terms of verifiability and time-to-market, from *leaner* development environment and execution platforms. The timing behaviour of control tasks is a critical concern in real-time digital controllers. These delays, such as start of execution jitters, or missed executions, affect the system performance and need to be considered during the design phase. The approaches developed in the past to study the performance of the control system due to run-time delays are simulation approaches, not supporting the implementation of the system. In this thesis, a model-driven co-simulation based development approach is proposed and illustrated with an automotive servo control example. The proposed federated approach provides the designer with both simulation and execution capabilities to define and validate functional and non-functional behaviors. The benefits of the proposed technique over the state-of-the art are discussed in this paper, amongst which a good support for rapid-prototyping to shorten the development cycle.

Timing equivalence bahaviour between simulation and embedded execution. We re-visited FIFO scheduling under modified conditions and make a case for for timing equivalence bahaviour between simulation and embedded execution. FIFO scheduling with strictly periodic task activation and release offsets to increase the predictability and to improve the performance. We have shown that FIFO with offsets is unique in the sense that it is both work-conserving and exhibits a single, well-defined execution order. Further, we provided a schedulability analysis for FIFO, both with and without offsets. Finally, we evaluated the performance of FIFO scheduling in terms of schedulable task sets, and compare the predictability of FIFO against the two well-known non-preemptive scheduling policies fixed-priority non-preemptive scheduling (FPNS) and non-preemptive earliest deadline first (EDF_{np}) in terms of distinct execution orders. FIFO with and without offsets for strictly periodic tasks is a competitive scheduling policy when predictability and simplicity matter. Adding offsets [Nasri et al., 2018] and enforcing strictly periodic task releases provides two significant advantages to FIFO: (i) The performance issues are mitigated and a higher number of task sets are schedulable, even at high utilization rates, and especially for task sets with harmonic or loosely-harmonic periods and (ii) a unique execution order, defined by the order of job arrivals is enforced which greatly simplifies validation and testing.

This second advantage is a promising property of FIFO and provides a form of timing equivalent behavior between development and run-time phases. It means that the sequence of reading and writing events will be the same in simulation and on target, whatever the relative execution times of the tasks. This ensures that precedence constraints (as imposed by data flows between tasks), if verified in simulation, will be met on target. This is not true with EDF where the execution orders will dynamically depend on the WCETs (the execution order may also change at run-time from one job instance to another). If precedence constraints are not met, then the impact on input jitters can be huge because the control algorithms will have to use outdated data (produced at the last cycle), and those kinds of issues can happen even if all deadlines are met.

We have shown that FIFO with offsets is unique amongst all work-conserving algorithms. It is a good fit for our CPS design flow which aims at automating system synthesis and hiding away from the designer the complexity of the underlying run-time environments, lowering thus the barriers to designing and modeling provably-safe real-time systems. In future works, we plan to consider and evaluate the scheduling overheads of FIFO, but also of the other scheduling policies, and design an offset optimization strategy tailored to FIFO. The idea behind CPAL is to provide a virtual platform, on which design exploration can be performed, which executes an application in simulation mode in the exact same manner as it executes on target (in terms of timing behavior). This is what we call "timing-equivalence" behavior between simulation and target hardware.

Model-driven co-design framework for fusing control and scheduling viewpoints. Model-driven engineering has been successful for capturing the functional requirements during design, but non-functional requirements such as timing have been traditionally overlooked. This leads to a late verification of controller timing and, in the best case, to corrections at a stage when they are costlier. This work is a contribution towards conceiving a design environment for embedded control systems that capture all the necessary functional and non-functional requirements, while providing analysis, simulation and run-time capabilities.

In this report, we presented a framework based on timing tolerance contracts which fuses the stability and scheduling viewpoints during controller design. The three steps of the framework have been described: controller design verified by stability analysis and co-simulation, software design verified by schedulability and WCET estimation, and lastly, the implementation checked through run-time verification. The crucial advantage of our co-simulation approach based on model interpretation is that the same controller model verified in the design phase can be ported directly (without the need for code generation) to target hardware to implement the final system.

VII.2 Future Outlook

Several interesting areas and direction for the future research can be extended based on the research presented in this thesis. This dissertation sets the basis to follow such research directions in the future.

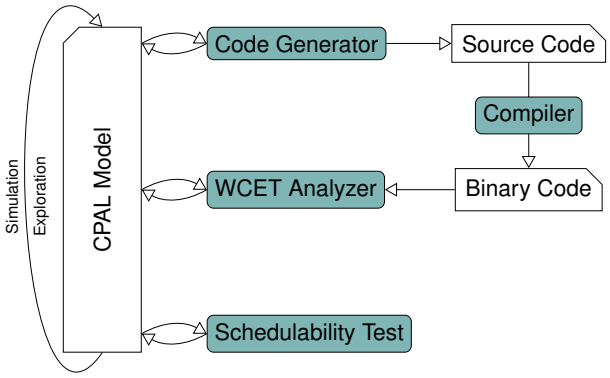


Figure VII.2: Extending the model interpretation based co-design framework to include code generator and WCET analyses.

Extending the modeling framework towards automotive distributed use-cases. The methods and techniques applied in the thesis are mainly with a consideration that the sensing and processing are centralized and not distributed. But what is inevitable on automotive use-case is that several sensors and actuators distributed across the communication network. While we design the controller model using CPAL, the system modeling assumption is that sensing, control algorithm and actuation all the three activities are within one software component. In the case of distributed, these three activities can be modeled as individual software component. Using the same CPAL as modeling environment, we can simulate the distributed control system. Our interest in the thesis is scheduling of computational tasks and computing latencies, and this can be further applicable to add communication latencies. In that case, along with *execution time* of a task in a computing platform, the system model includes the *transmission time* of a frame in the network, provided modeling of network is established.

Spectrum of MBD, code generation to complement the model interpretation. Another possible extension is that as depicted in the Figure VII.2 and refering also the Figure VII.1 for *model only 2.0*, for the high-level controller model, which readily provides a rich notion of timing, we could tightly integrate a code generator, a WCET analysis tool [Ballabriga et al., 2010], and advanced schedulability tests into the framework presented. This tight integration will enable the **exchange of rich timing information between the tools** and even allow us to **propagate relevant information back to the high-level model**. This will allow to better account for the characteristics of the actual low-level implementation early on during high-level modeling and simulation.

Applying the research outcomes to other industrial domains and aspects. The lean model-driven development life-cycle eases the deployment and the update of functional code on distributed embedded nodes on any networked use-cases, for instance in Industrial Internet of things (IIoT) applications. To exemplify the framework flow in this thesis, we have presented scheduling viewpoint using novel FIFO schedulability analysis for periodic task activations with offsets. A perspective is to extend the framework to other schedulability analyses using tools such as

Cheddar [Singhoff et al., 2004b] and MAST [Harbour et al., 2001] to support more scheduling options during scheduler synthesis. Another perspective is to extend the approach to other important non-functional properties, foremost *power consumption* for Cyber-Physical Systems, which will require both analysis and modeling language support. In this thesis we also exhibit two use-cases Connected Motorized Riders shortly *ConMoR* and the *Parking Minute* in Appendix chapter which will be further used for extending the CPAL framework towards incorporating *power consumption* non-functional property. This way we would exemplify the applicability of framework to other application domains.

High level requirements low-level model automated synthesis. Figure VII.3 illustrates an extension to CPAL modeling, the high-level requirements modeling using *Capella* [Capella, 2019], an open source *descriptive modeling* environment to capture the stakeholder requirements, specifications and viewpoints.

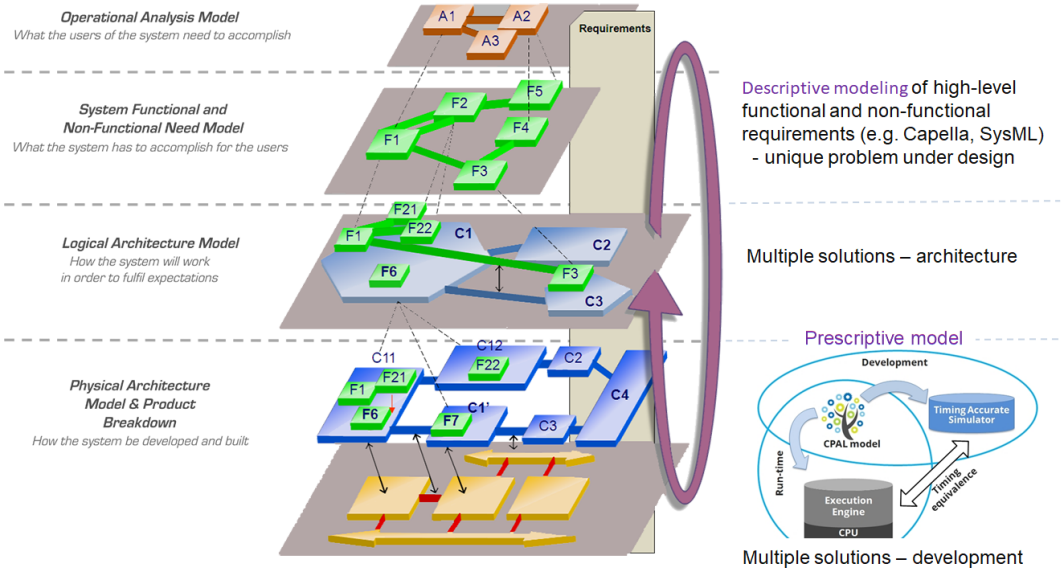


Figure VII.3: Extending the *prescriptive modeling* based on CPAL model-interpretation to include *descriptive modeling* to capture the high-level system requirements, design space exploration to provide optimized development solution. The left side part of this figure is from [Capella, 2019].

Multiple solutions for unique problem under study could emerge during architecture phase. One can design and develop multiple solutions using *prescriptive modeling* like the one presented in this thesis CPAL. Based on various constraints and viewpoints, a design space exploration would yield then an optimized solution.

Chapter VIII

Appendix

Abstract

This Appendix chapter presents the prescriptive modeling using CPAL and its support to develop IoT applications. Also, we discuss the descriptive modeling of high-level functional and non-functional requirements (e.g. Capella, SysML) of unique problem under design. We exhibit two IoT prototypes namely ConMoR and Parking Minute to discuss how MBD is useful during development. The concept of applying modeling *to IoT applications is part of the research stay [Sundharam, 2016] during Short Term Scientific Mission (STSM) of ICT-COST action TACLe (Timing Analysis on Code-Level).*

VIII.1 State-of-the-art Practices in MBD to IoT

We present two case-studies to demonstrate model driven approach applying to IoT applications, namely *Connected Motorized Riders* [Sundharam et al., 2016c] and *Parking Minute*. The application development for IoT-compliant use cases may be carried out using methods such as traditional hand written code-development, model-driven development and the mashups. IoT use cases often employ mashups, which permit visual, interactive modeling of the message flow between devices. On the other hand, model-driven approaches which are also being employed for IoT application development, in general, permit expressive modeling of the systems, possibly with code generation. However, a majority of IoT applications for embedded devices, often employ hand-written code in programming languages such as C/C++ (e.g. ARM-mbed environment [ARM, 2018]) Despite being an emerging field, none of the existing approaches for IoT application development deal with the analysis of performance characteristics of the IoT-enabled software system. For instance, existing approaches do not address the aspect of early timing validation of IoT based use cases.

While mashup tools provide a quick prototyping environment, modeling approaches permit very expressive modeling of the systems, possibly with code generation [Prehofer and Chiarabini, 2015]. Several development toolkits, which combine both mashup and model-driven development paradigm, were recently introduced [Pramudianto et al., 2014]. However, such tools support prototype code generation in the Java programming language only, which is not a commonly used programming

language for embedded software development, though involving IoT aspects. On the other hand, among the Model Driven Development (MDD) approaches for IoT-compliant embedded software development, ThingML [Department of Informatics, 2018] is a practical model-driven software engineering tool-chain, which targets resource constrained embedded systems such as low-power sensor and microcontroller based devices. In [Christoulakis and Thramboulidis, 2016], an UML profile for the IoT, with targeting cyber-physical manufacturing systems, is introduced. This approach automates the generation process of the IoT compliant layer for legacy and new mechatronic components alike, to exploit the IoT connectivity. However, support for evaluation of NFPs integrated with the overall development approach is not available in [Christoulakis and Thramboulidis, 2016; Department of Informatics, 2018]. The UML4IoT profile introduced in [Christoulakis and Thramboulidis, 2016] is also claimed to be applicable to legacy code, after Reverse Engineering (RE) and annotating the resulting UML diagrams.

RE applied to software can be considered as a process of analyzing the subject system (e.g. hand-written code) to create representations of the system at a higher level of abstraction (e.g. UML models). RE is often employed as a process of examination only, wherein the subject software system is not modified. Similarly, in the workflow proposed in this paper, RE is used to primarily extract models at a higher level of abstraction (e.g. UML diagrams) from the embedded software system (e.g. hand-written C++ code) for IoT specific use cases. The UML models, thus obtained, are annotated with the NFPs for performance evaluation, as proposed in [Iyenghar et al., 2016]. In the UML domain, profiles such as Modeling and Analysis of Real Time and Embedded Systems (MARTE) [OMG, 2018] are specially developed to address such application areas. Despite the fact that many timing requirements could be validated and analyzed on model level, most of the UML tools do not have the capability to handle this. Specialized timing validation tools address the aforementioned aspect. Therefore, an interfacing framework between UML and such timing validation tools is needed. For embedded IoT compliant use cases incorporating timing dimensions especially using model interpretation is not available. Similarly, a NFP model for IoT applications presented in [Sicari et al., 2014] may be considered as a starting point for the development of IoT privacy-aware solutions. However, specifics pertaining to addressing the IoT timing-aware solution are not available. Further, at this juncture, a majority of the embedded IoT applications are developed as hand-written code (e.g. rapid prototyping). For example, the mbed platform is a lightweight C/C++ microcontroller development environment that facilitates the user to quickly write programs, compile and download them to various platforms. We present a timing aware MBD, the CPAL coupled with high level descriptive modeling to architect the IoT case studies that we prototype.

VIII.2 Case-study : Connected Motorized Riders shortly (ConMoR)

The Smart Cities Mission has been launched in India in 2015 to develop 100 cities, with smart mobility being one of the main topics in the mission. As urban areas are flooded with two (motorcycles) and three wheelers (auto-rickshaws), introducing smart control of such vehicles may reduce the congestion on the roads and the number

of accidents. Indeed, over-speeding and drunken driving are common traffic violations. In this project we propose an IoT-based smart mobility system which tracks data, such as the vehicle location, vehicle speed, alcohol level of the driver, etc. efficiently over the internet. Our system has been conceived with CPAL, a high-level language meant to simulate and execute Cyber Physical Systems including IoT applications. A prototype running on ARM mbed IoT hardware, shows the feasibility of our concept. We believe that more efficient and interactive traffic management, more disciplined driving behaviors, reduction in accident rate, more controlled pollution, increased passenger safety can be achieved if systems like the one prototyped in this work deployed contributing to smarter cities.

VIII.2.1 Introduction and the Problem

The data from WHO Global status report [WHO, 2019] on road safety 2015 states that 72% of vehicles in India are motorized two and three wheelers. Because of this, the two and three wheelers are major contributors in creating traffic congestion in urban areas. The report also states that among all the categories of road deaths, motorized two and three wheelers are predominant ones with 34% cases. The main reasons for two and three wheeler accidents are over-speeding and alcohol consumption. Indeed, the urban speed limit is often ignored by commuters and manual checks for alcohol consumption is not a systematic enough procedure to reduce the accidents.

In addition, to the best of our knowledge, there is currently no technology supported system in India to know the density of motorized riders in a particular region or an urban area. Also, there exists no automated system to know whether vehicle speed and alcohol consumption level are within the legal limits. These needs motivate us to propose a smart system where two and three-wheelers are equipped with set of sensors and are connected to the internet. Next section gives the overview of proposed solution. Remaining sections of the chapter explain the solution in details.

VIII.2.2 Solution: ConMoR

Two-wheelers (i.e. motorcycles) and three-wheelers (auto-rickshaws) are low-cost and easy commuting means in India. The dashboard in these vehicles is simple, available either as analog or as digital meter. But new vehicles are released in the market with digital dashboards. Our smart mobility solution can be integrated in the dashboard either as additional embedded hardware or as software in existing hardware. A prototype device has been built to demonstrate the feasibility of the idea. The device is connected to internet and the web platform captures the live data as depicted in Figure VIII.1.

In this work, CPAL [DesignCPS, 2019] is employed as modeling and simulation environment in the design phase of the application. The final implementation has been realized using the ARM mbed development platform that includes a SDK and software libraries. In the Section VIII.2.3, a timing aware model-driven engineering paradigm that supports the IoT system development is presented. Section VIII.2.4 elaborates the hardware details of the prototype, IoT software development flow, information security and device connectivity to internet.

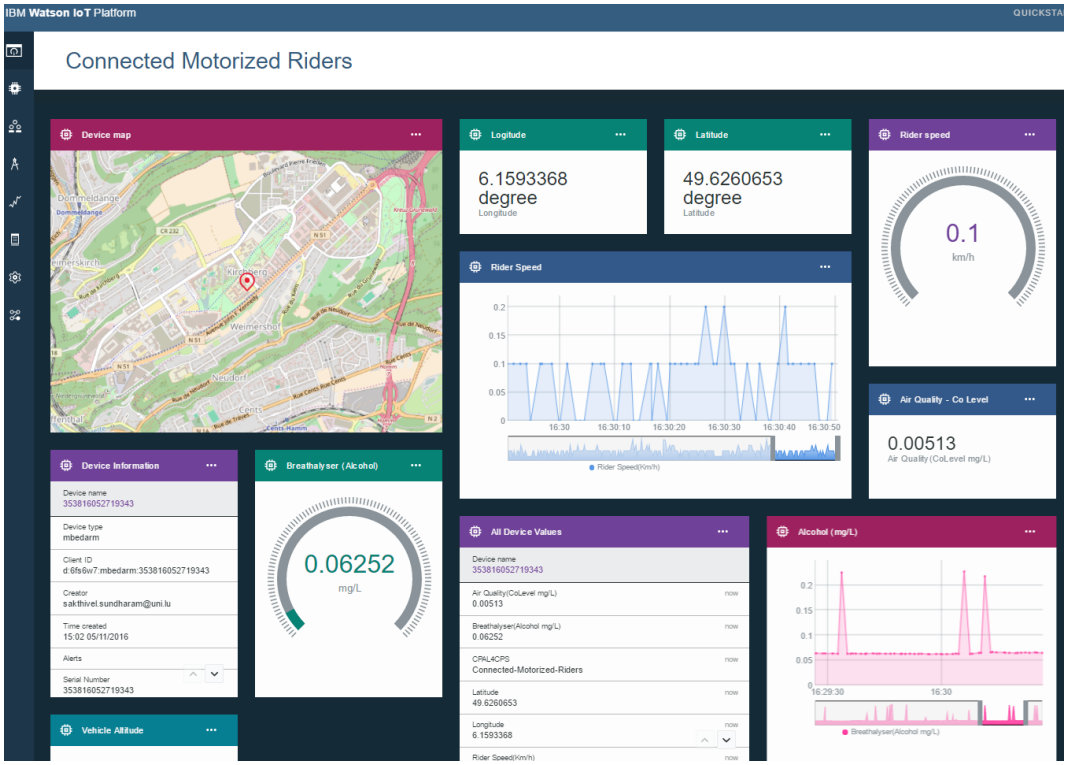


Figure VIII.1: Connected Motorized Riders - A smart mobility system for connecting two and three-wheelers to internet. Data such as latitude, longitude, altitude, speed, alcohol blood level are tracked and monitored in IBM Watson IoT platform as shown in the screen-shot.

Figure VIII.2: CPAL model of the application. The left-hand side is the view of the functional architecture with the functions and the data flows making up the application. The CPAL code is in the center. The right-hand side shows a Gantt chart of the activation of the tasks implementing the application. This model can be executed on a workstation and without any changes interpreted (Bare-Machine Model Interpretation) on the ARM mbed Cortex M3 processor of the U-Blox C027 IoT board.

VIII.2.3 CPAL - System Engineering to Build IoT Applications

VIII.2.3.A Need for a Timing Aware Model Driven Development

Innovation in the development of systems relies increasingly on software, which is disrupting complete industries like the transportation industry. Model-Driven Development (MDD) has proven to be a cost and time-effective method. But still non-functional properties such as time, safety and energy concerns are not well integrated in the existing MDD environments. The industry practice is that the design model is constructed, then the code is generated from the model and ported to the target hardware. Then run-time tests and measurements are performed during testing phase, but that is where the time and costs of fixing the problems are the largest.

This motivates the need for timing-aware design flows. Indeed, existing MDD flows are simulation-centric environments that assume that the computational resources available are infinite, or, at best, they are crudely estimated. This tends to create a gap between design phase verification and target testing. This problem motivates a new modeling paradigm. Our initiative is CPAL [Navet and Fejoz, 2016b] which is a language to model, simulate, verify and execute Cyber-Physical Systems as used in automotive, aerospace, IoT, etc. These systems are typically interacting with numerous sensors, computing resources and the actuators.

VIII.2.3.B Model-Driven Approach to IoT System Development

Unlike the traditional MDD where execution is achieved by generating code from a model, and then compiling it, CPAL programs are executed by a real-time model interpretation engine. As shown in Figure VIII.2, a key advantage of using CPAL for IoT application development is that it provides during the design phase a view of the functional architecture of the application which is the set of processes, how they are activated, and the data flow among them. This is to the best of knowledge not possible with the MBDed development environment. In addition, simulation possibility strengthens the early stage functional verification.

An IoT enabled CPAL model interpretation engine has been developed for the ARM Cortex M3 microcontrollers [Fejoz et al., 2016a]. On this platform, the interpreter runs on the bare hardware, that is without an OS, what is referred to as Bare-Metal Model Interpretation (BMMI). BMMI helps to reduce the memory footprint and avoid errors that can be introduced throughout the design flow (e.g., at code generation or compilation stage, or errors at the OS level). BMMI eases also the verification, because the model is what actually executed, and not code generated from the model.

A benefit of interpretation is that the low-level services needed for the product development can be offered by the interpretation engine (e.g., MQTT protocol support). Only the high-level application is to be developed as per the functional and non-functional requirements. Hence the development efforts to interface sensors, GPS and MQ-3 to hardware is reduced, leading to shorter time to market.

CPAL executes on platforms that have very different capabilities in terms of Input/Output ports (e.g., General Purpose Input/Output, I2C, serial communication, etc). CPAL interpretation engine is executed on hardware such as Raspberry Pi,

platforms running on embedded Linux and FRDM-K64F [Fejoz et al., 2016a]. In the case of ARM mbed hardware, CPAL interpreter is available as a binary file which is further merged with the CPAL IoT application (as CPAL binary format called the .ast file).

VIII.2.4 Product Prototype and IoT Platform

VIII.2.4.A Device Overview

The speed of the vehicle is generally estimated by the wheel speed sensor and is already available in a dashboard. In the prototype we have used the GPS speed sensor which provides the speed but also the location of the vehicle. An ARM mbed-enabled IoT kit with U-Blox C027 (ARM Cortex M3 core) is used as central unit to run the application. We have used U-BLOX Max 7Q series GPS module to capture the location data and speed calculation. Global Navigation Satellite System (GNSS) antenna is used to connect the GPS module.

A MQ-3 ethanol sensor is used to detect the alcohol content in the driver's breath. A MQ-7 Carbon mono-oxide (Co) sensor is connected to the board to measure the air quality level while the vehicle is on the move. SARA-G350 GSM GPRS modem with cellular SIM card at the back side of the board enables the connectivity using standard mobile data network.

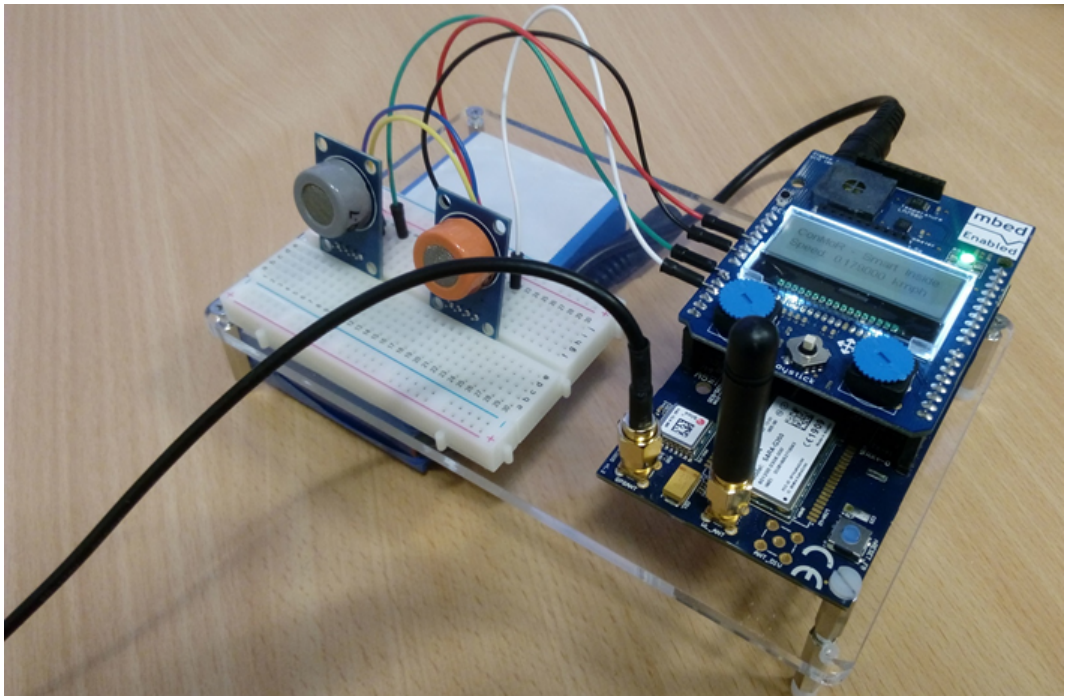


Figure VIII.3: Prototype set-up for interfacing sensors to ARM mbed IoT hardware (right) consisting of the application shield and the base board U-Blox C027 ARM Cortex M3, the MQ-7 Carbon mono-oxide sensor (left on the breadboard) and the MQ-3 ethanol sensor (righton the breadboard), U-BLOX-Max 7Q GPS module (not shown in the picture) with GNSS antenna, and the GSM modem with antenna.

Figure VIII.3 shows the prototype of the *connected motorized riders* device built using ARM mbed enabled IoT hardware. The IoT board consists of two parts: the application shield and the base board U-Blox C027 ARM Cortex M3. The sensors such as GPS and MQ-3 are interfaced to application shield of the board. Since the product is for mobility applications, to connect to the internet either IoT network or cellular connectivity is required. We have carried out our experiments using cellular GSM/GPRS modem to connect the device to the internet wirelessly.

VIII.2.4.B ARM mbed IoT Flow

ARM mbed IoT platform provides the OS, cloud services, tools and libraries to facilitate the creation and deployment of large-scale commercial IoT solutions based on standards. The ARM mbed device connector [mbed, 2019] service connects the device (i.e) motorized rider to IBM cloud efficiently and easily with inbuilt security functions enabled. IBM Watson IoT platform [IBM, 2019] which is used in our project to manage connected vehicles is one of the cloud-hosted services offered by the IBM Bluemix cloud infrastructure.

The mbed device connector service is connected to the IBM Watson platform to send/receive the IoT device events, using API keys and tokens. We have defined the mbed connector bridge to connect to the platform. Events (send/receive) between devices (vehicles) and the Watson IoT platform are created using Node-RED application which is a visual tool (see Figure VIII.4) for wiring the IoT objects such as hardware devices, APIs, and online services in an intuitive way.

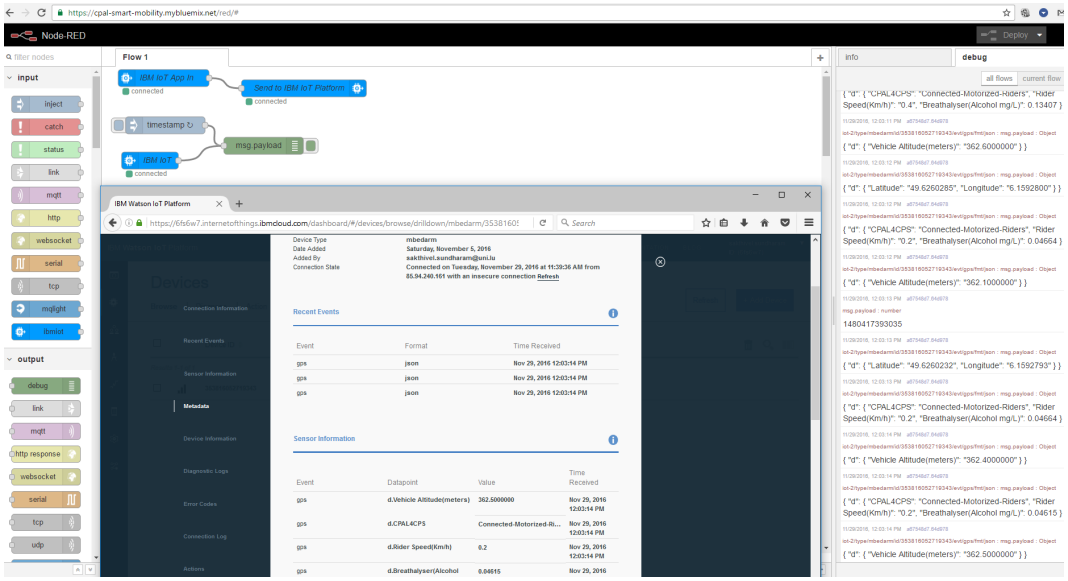


Figure VIII.4: Node-RED provides a browser-based flow editor that makes it easy to wire together the data flows coming from the devices using a wide range of predefined nodes. Once defined in Node-RED, the application is then deployed and executed in the IBM Watson IoT platform.

VIII.2.4.C Information Security Mechanisms

Watson IoT Platform runs within the Bluemix cloud. Watson fulfills *ISO27001* standard for the information security management processes. Security is achieved in Watson platform at many levels. Like mentioned in Figure VIII.5, secured information management within the organization, secured device and application credentials enable strong information security. Device credentials with organization ID, device type, device ID (in our case IMEI number of the cellular modem), authentication token are integrated into the embedded program. Application credentials such as API keys and authentication tokens are also hard coded into the embedded program.

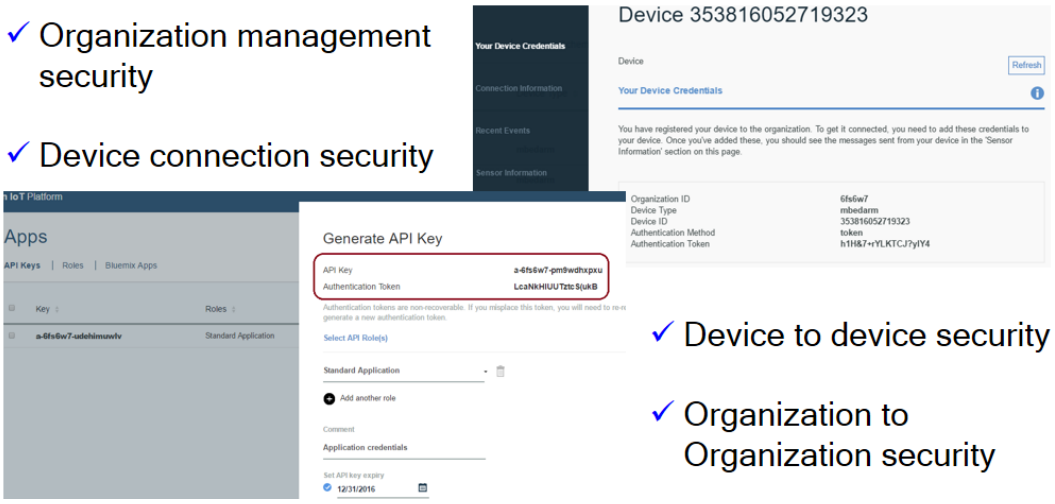


Figure VIII.5: Various levels of security mechanisms enabled in Watson IoT platform to ensure secured connections.

VIII.2.4.D Connectivity to Internet

Since vehicles are moving objects wireless connectivity is required. Wireless LAN and cellular connectivity can be considered if power consumption of the object is not a main constraint. If the power consumption of the device is a significant requirement, then wireless connectivity is more efficiently achieved through low-power Wide Area Networks such as LoRaWAN and SIGFOX. The product we prototyped can be viably deployed when low-power networks are established. SIGFOX for instance uses ISM band 868MHz and comply with regulation ETSI300-220. SIGFOX [Sigfox, 2019] wireless networks is meant to offer connectivity to objects such as electricity meters, smart watches and washing machines, which need to transmit very regularly small amounts of data. During the Grand Challenge day of *IEEE ISED2016* conference [Sundharam et al., 2016c], the demonstration of the product has been done with cellular (GSM) connectivity.

The device has been built and tested on a real vehicle. Speed and location data, both on the device and in the cloud platform, were verified against the data captured on the dashboard and a smart phone respectively. Also, the sensitivity and response of the MQ-3 sensor to alcohol content has been verified. Since location and speed data are

tracked online, it would be possible to identify over-speeding behaviors and possibly learn from the patterns of behaviors observed. This can be done anonymously or, if the Vehicle Identification Number (VIN) is mapped to the device ID, with the identification of the vehicle's owner.

To measure the air quality, the device is equipped with a MQ-7 sensor which measures the Carbon mono-oxide (CO) level in the air. Other emission gas sensors such as CO2, NOx, can be interfaced to the device which would enable value added features such as real-time pollution monitoring in a particular area. The experiments presented in this work, where the data were transmitted from the device to the internet, can be extended to other use-cases where data are sent from the internet to the device. For instance, when the vehicle is near schools or hospitals, the driver could be requested to reduce the speed.

VIII.3 Case-study: Parking Minute

In this section, we discuss the descriptive modeling using Capella [Capella, 2019] and its process ARCADIA. Descriptive modeling supports in capturing high-level functional and non-functional requirements (e.g. Capella, SysML) of the problem under design.

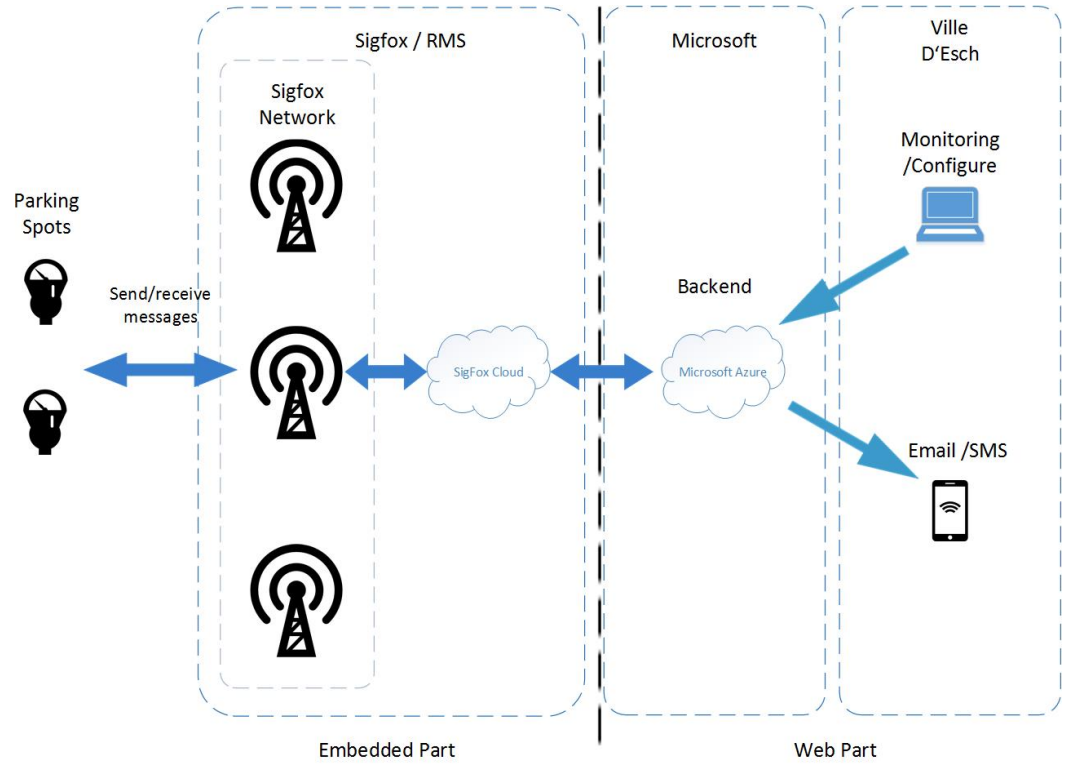


Figure VIII.6: Parking minute overview.

As shown in the Figure VIII.6, Parking minute system uses a smart sensor system, which detects the parked car and sends a message through the SIGFOX network to the control center after the parking period has elapsed. A smart sensor is similar to

an ultrasonic sensor with a small processing unit. This processing unit filters out faulty values and/or generates alerts which can be sent over network infrastructure to a server. In the case of the "Parking Minute", the ultrasonic sensor used in the system is a smart sensor. It calculates automatically the distance in centimeters between the sensor and the first recognized object in the sensing area.

VIII.3.1 Architectural Modeling of High-level System Requirements

The set of functional and non-functional requirements from stakeholders help to define further the system requirements to build the product as shown in Figure VIII.7. The system requirements lead to software, hardware and mechanical requirements.

Figure VIII.7: Parking minute pole-mounted prototype.

VIII.3.2 Functional Requirements

VIII.3.2.A REQ_F_001 Car detection

Use-case: Car detection
Actor: Sensor
Description: The system should detect if a car stands on the parking spot. The check will be performed every x minutes (x will be defined during the implementation).

VIII.3.2.B REQ_F_002 Message send when time limit reached

Use-case: Send message when time limit reached
Actor: Airboard
Description: The Airboard should send a message over the SIGFOX shield to the SIGFOX back-end when a car exceeds the allowed parking time limit as shown in Figure VIII.8.

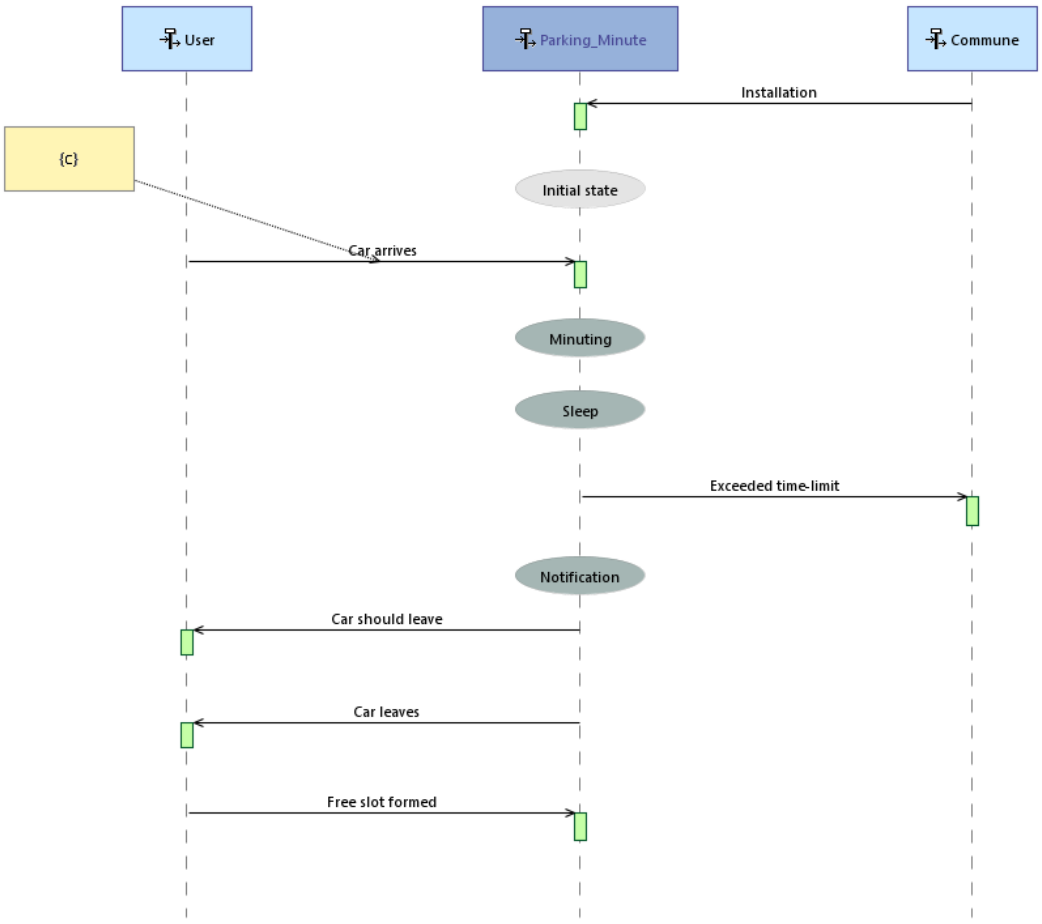


Figure VIII.8: Exchange scenario (ES), sequence of different parking scenarios.

VIII.3.2.C REQ_F_003 Messages send over SIGFOX network

Use-case: Messages send over SIGFOX network
Actor: Airboard
Description: The communication with the backend will be done over the SIGFOX network.

VIII.3.2.D REQ_F_004 Sensor detection accuracy

Use-case: Sensor detection accuracy
Actor: Sensor
Description: The sensor should only detect vehicles. When a person passes by the sensor, the system should not be triggered.

VIII.3.2.E REQ_F_005 SIGFOX shield turned off to save power

Use-case: SIGFOX shield turned off to save power
Actor: Airboard
Description: The SIGFOX shield will always be turned off except when sending a message. Before sending a message, the shield will be turned on followed by a wait state for one second. Then the message will be sent, also followed by a wait state for three (3) seconds. Lastly the shield will be turned off again.

VIII.3.2.F REQ_F_006 Deep sleep mode

Use-case: Deep sleep mode
Actor: Airboard
Description: The Airboard will fall asleep when not processing some data. The sleep will be interrupted every x minutes (x will be defined during the implementation) to check if something changed (Car arrived/left or time limit reached).

VIII.3.2.G REQ_F_007 Initializing the system

Use-case: Initializing the system
Actor: Administrator
Description: When the system is started for the first time, a connection to the SIGFOX network will be established and the configured values for the time limit, time tolerance and the operation time will be pulled from the back-end. If the connection cannot be established on the system, then it will use the default values.

Capella is structured on consecutive engineering phases which establishes a clear separation between needs (operational need analysis and system need analysis) and the solutions (logical and physical architectures). As a first step of the process, the system operational need analysis is performed and the Figure VIII.9 illustrates the context *what the users need to accomplish.*

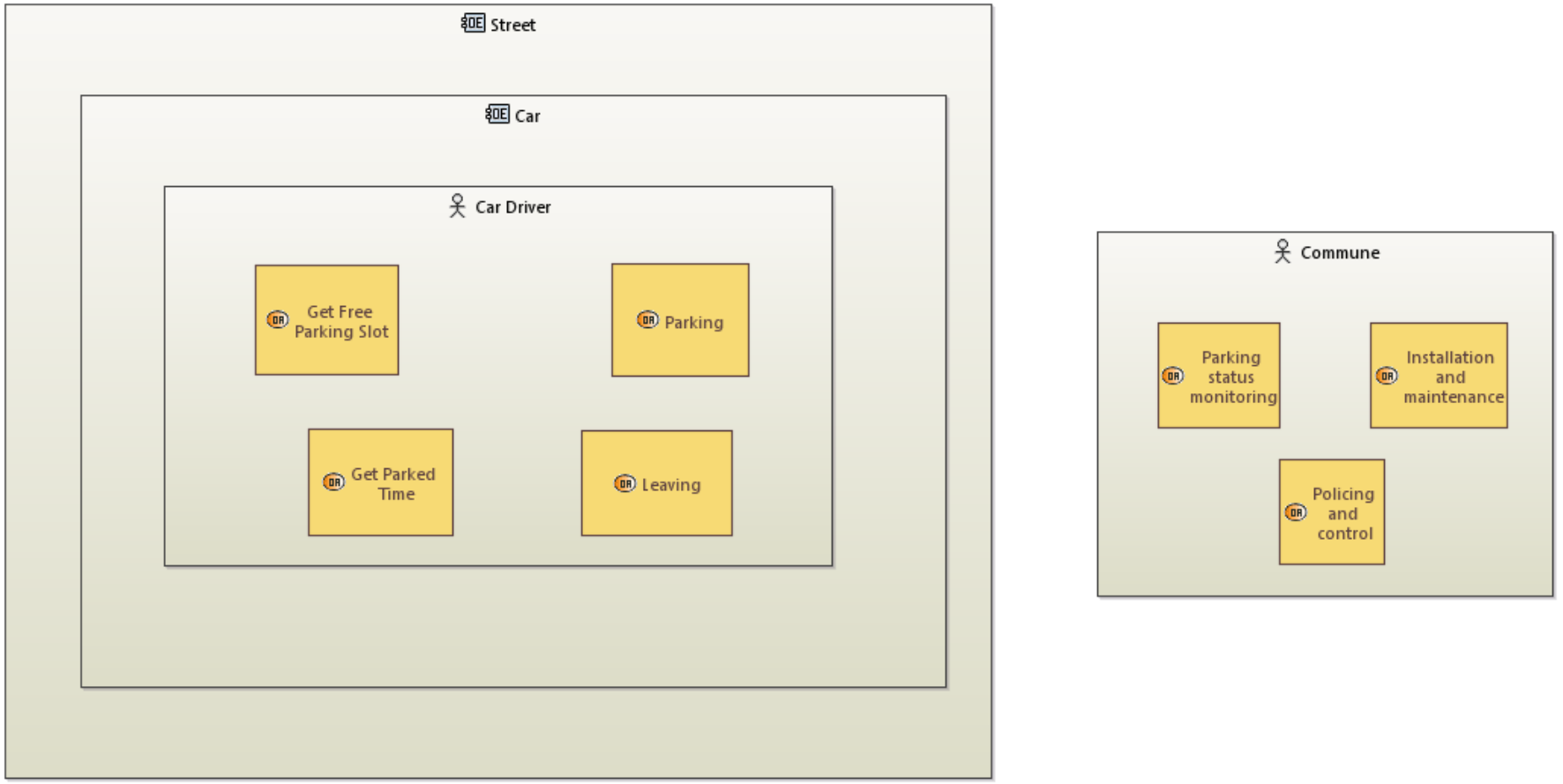


Figure VIII.9: Operational entity blank diagram of Parking minute.

VIII.3.2.H REQ_F_008 Configure time limit

Use-case: Configure time limit
Actor: Administrator
Description: The time limit can be configured for each station separately on the back-end.

VIII.3.2.I REQ_F_009 Configure time tolerance

Use-case: Configure time tolerance
Actor: Administrator
Description: The time tolerance can be configured for each station separately on the back-end.

VIII.3.2.J REQ_F_011 Statistic messages

Use-case: Statistics messages
Actor: Airboard
Description: A message should be sent to the SIGFOX backend when the state of the parking spot changes. (Car arriving or leaving) This data will be used to make statistics when the parking spots are used the most. The operational flow of the system is depicted in the Figure VIII.10.

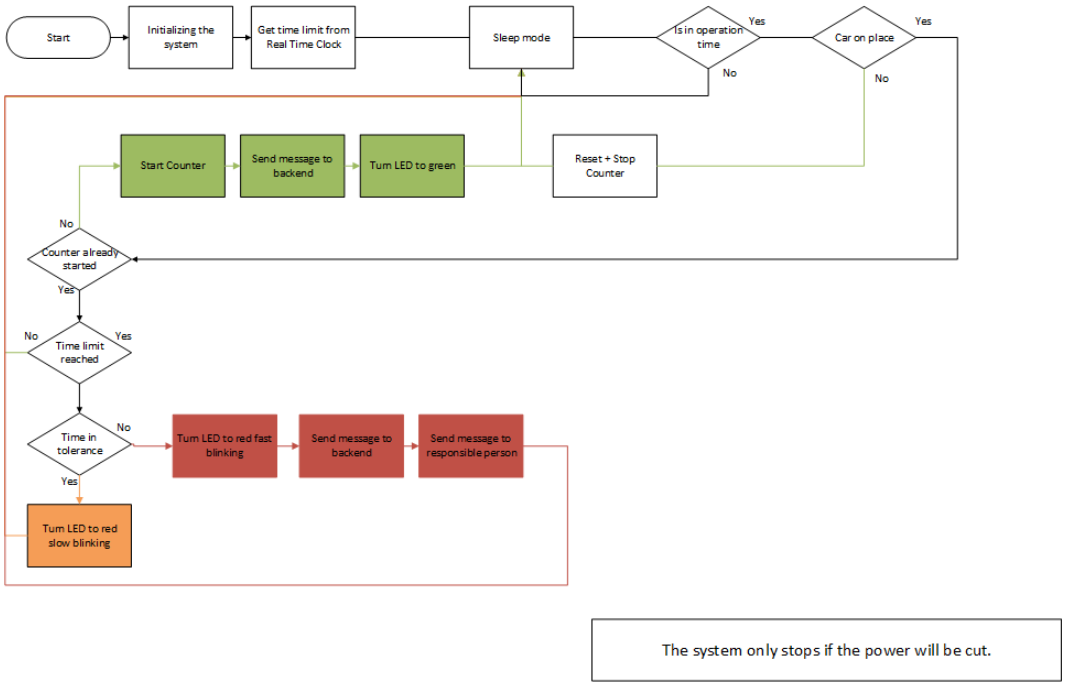


Figure VIII.10: Operational flow of the Parking minute.

VIII.3.2.K REQ_F_010 Configure operation time

Use-case: Configure operation time
Actor: Administrator
Description: The operation time can be configured for each station separately on the back-end.

VIII.3.3 Non-functional Requirements

VIII.3.3.A REQ_NF_001 Detection of the vehicle

The system must detect a vehicle with the help of a sensor which could be an inductive or an ultrasonic sensor.

VIII.3.3.B REQ_NF_002 Detection prevention of other things

The system shall detect only vehicles. Objects like persons and trash bins etc. shall not be detected.

VIII.3.3.C REQ_NF_003 Sending message to responsible person

The system must send a message to the responsible person when the time limit is reached.

VIII.3.3.D REQ_NF_004 Signalization within time limit

The system must symbolize to the client and the responsible person with a green LED that the car is within the time limit which can be configured.

VIII.3.3.E REQ_NF_005 Signalization closely over the time limit

The system must symbolize to the client and the responsible person with an orange LED that the car is just over the time limit. The range can be configured.

VIII.3.3.F REQ_NF_006 Signalization long over time limit

The system must symbolize to the client and the responsible person with a red LED that the car is over the time limit which can be configured.

VIII.3.3.G REQ_NF_007 Easy installation

The system must be installed without drilling a hole in the ground. Additionally, the system shouldn't need a power line. The system must be designed, that it can be removed and installed in a different location without damaging the system.

VIII.3.3.H REQ_NF_008 Low power consumption

The system should use components which don't use too much power, so it can be powered by a battery.

VIII.3.3.I REQ_NF_009 Power consumption of a battery

The system must use a battery as a power supply. A battery must be used because the system should be portable and shouldn't use a power line.

VIII.3.3.J REQ_NF_010 IP56 conform case

The system's case must be IP56 conform, which means it should be dust protected and waterproofed. (`http://www.enclosurecompany.com/ip-ratings-explained.php`)

VIII.3.3.K REQ_NF_011 Display collected data

The system must have a backend which presents the collected data (Arrival time, departure time and ticket time). The information can be shown as charts from all or a selected station.

VIII.3.3.L REQ_NF_013 Configurable time limit

The arrival time, departure time and tolerance time has to be configurable for each station over the backend.

VIII.3.3.M REQ_NF_014 Operation time

Each station should be able to get the operation time of the backend. Operation time means in which time period the system is activated.

VIII.3.3.N REQ_NF_015 Client alert (Optional)

The system should be able to inform the client when the time limit is reached. Here the client can scan a QR-code with his/her phone to send the phone number to the system.

VIII.3.3.O REQ_NF_016 Low maintenance

The system must run autonomously. A general maintenance should be done once a year. Additionally, the components must be easy accessible and easy to replace.

VIII.3.3.P REQ_NF_017 Arrival and departure message

The system should send a message when the car arrives and a message when the car leaves the parking place. This feature will be used to make statistics.

VIII.3.3.Q REQ_NF_018 Battery recharging over a solar panel

The battery must be recharged by a solar panel installed on the system VIII.11.

Figure VIII.11: Battery and sensor casing.

The core program on the embedded system interacts with the sensor(s) and the SIGFOX backend, to send or receive messages. The core program checks with the help of the sensor(s), if there is a car or not. When the program notices that a car has arrived, he begins to start the timer and sends a message with the required information to the SIGFOX backend.

Additionally, the system turns the green LED on to signalize that the car is within the time limit. When the car leaves the parking spot, the program stops the timer and sends a new message to the backend. If the time limit is reached and is within the configured tolerance time, the orange LED lights up and the green LED turned off. If the car stands even longer than the tolerance time, the red LED lights up and signalizes that it's time for a ticket. The system sends a message to the backend which contacts the responsible person for writing a ticket. The micro-controller could be an *Arduino* or an *ARM* chip. The system gets powered by a solar panel and a

battery, a stakeholder requirement to consider as oppose to the static power line which are in general incurs more installation costs.

The second part is the web interface. It is possible to list the different sensors and its messages which are saved on the SIGFOX back-end servers. We provide the possibility to configure the time limit, tolerance time and the operation time of each individual station. As an extension of work, we plan to list on the geographical map the different parking stations which are overdue and to develop an algorithm to optimize the walking distance to check the parking stations.

VIII.3.4 Operational Flow

This section provides the operational flow of the Parking minute.

VIII.3.4.A Initialization

In the initialization phase, the system will retrieve the date and time from the RTC (Real time clock). Additionally the different ticks will be initialized to zero and all unused ports will be turned off. After everything is prepared the system will be set in the deep sleep mode.

VIII.3.4.B System Functions - Different Parking Scenarios

A car comes and leaves within the time limit The system is normally in the deep sleep mode. It awakes every 5 seconds. In the project we check every 30 seconds if a car stands on the parking spot. When a car stands on the spot, the counter will be started if the counter isn't already running. Additionally, the system turns on the SIGFOX shield and sends a message to the back-end. After sending, the system will wait three seconds and turn off the SIGFOX shield to save power. Lastly the system blinks the green LED to signalize that the car is within the time limit.

Now the system returns to the deep sleep mode. Every 30 seconds the system checks if the car still stands on the spot and if the time limit is not already reached. If the system checks and the car has left, the counter will be stopped and reset. The next step is that the system sends a message to inform the backend of the departure of the car. Lastly the system returns to the deep sleep mode and waits for the next car.

A car comes and leaves within the tolerance time The system detects the car and starts the counter like in the case before. After the system realizes that the time limit is reached, the system blinks the red LED slowly if the car is within the tolerance time. If the car leaves in the tolerance time, only the message of the departure will be sent to the backend and the system returns to the deep sleep mode.

A car stays too long When the system realizes that the time limit and the tolerance time was exceeded, the red LED will blink fast and signalizes that this car should get a ticket. Additionally, the system sends a message to the back-end, which contacts the responsible person for writing a ticket. Then the system returns to the

deep sleep mode and checks when the car will leave. When the car leaves, a message will be sent for the departure. This data will be used for statistics.

VIII.3.5 Descriptive Modeling of High-level System Requirements using Capella

Capella is an open source Model-Based System Engineering (MBSE) tool. ARCADIA stands for Architecture Analysis and Design Integrated Approach from PolarSys, an Eclipse industry working group. Capella provides tools facilitating the daily work of the system engineer (transitions between architectures, filters and layers on diagrams, modeling acceleration tools, synthetic business view of relationships between model elements, etc.) and guaranteeing the quality of the designed architectures (impact analysis, model validation, etc.). Capella facilitates capability-driven model organization with scenarios and functional chains, functional analysis and allocation to components and resources Interfaces, bit-precise data models, behaviors, etc.

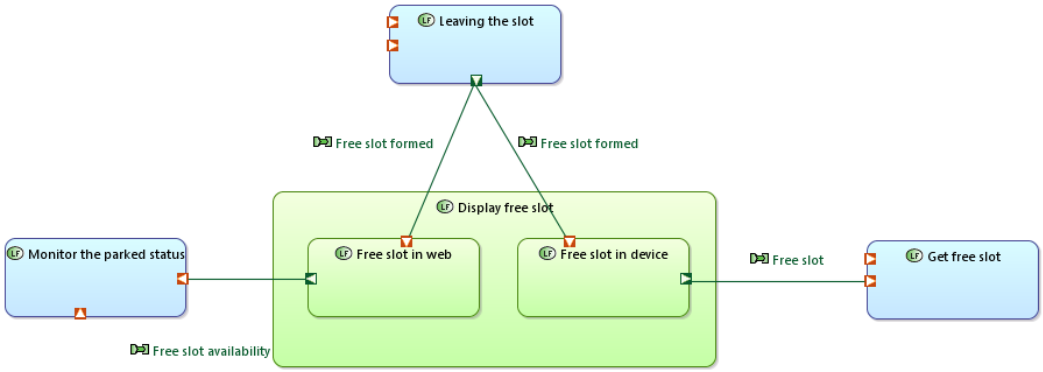


Figure VIII.12: Logical functions and their interfaces, Logical Data Flow Blank.

The Capella descriptive modeling [Voirin, 2017] environment has five different engineering levels as well as associated models. The first one called *Operational analysis (or OA)* focuses on what users of the system need to accomplish that includes issues of users, by identifying actors that have to interact with the system, their goals, activities, constraints and the interaction conditions between them. The second one is *System needs analysis (or SA)* concentrates on what the system has to accomplish for the users, which is constructed based on the operational analysis and input requirements.

The next level is *Logical architecture (or LA)*, *LA* is created in response to the needs expressed by the two previous levels. Thus it provides the first choices of solution design using an internal functional analysis of the system. It describes the functions involved in *Parking Minute* as shown in Figure VIII.12 to be performed and assembled in order to implement the service functions identified in the previous step. Then it continues with the identification of the operational components as depicted in Figure VIII.13, implementing these solution functions, integrating the non-functional constraints that we chose to be addressed at this level.

The overall logical architecture model of *Parking Minute* is shown in Figure VIII.14. In the perspective of *Physical architecture (or PA)*, a fourth level, which has the

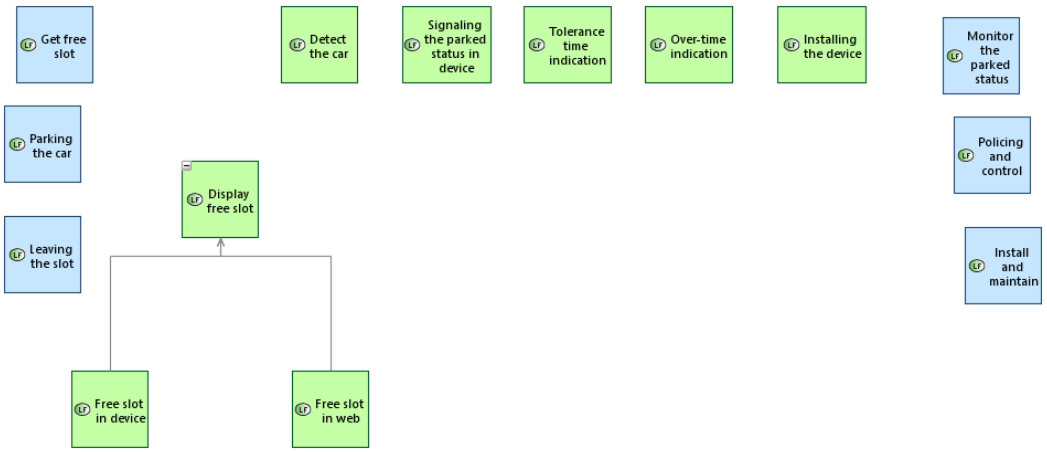


Figure VIII.13: Logical functions and their breakdown of operational components.

similar objective as the logical architecture, except that it defines the finalized architecture of the system, as it should be completed and integrated. It adds the functions required by the implementation and technical choices, and states the behavioral components that perform these functions. These behavioral components are then implemented using implementation components. The last level *Product building strategy (or BS)* concludes, from the physical architecture, conditions that each component must adhere to satisfy architecture design constraints, achieved in the previous steps. It also specifies the integration, verification and validation strategy of the whole system.

VIII.3.6 Dashboard and Sensor of the Device

The dashboard of the device VIII.15 consists of a mounted sensor, LEDs, and the SIGFOX antenna. The dashboard is nicely fit inside the whole set-up and only the sensing part and display of LED are visible outside.

In order to show the current status of a parked car, the parking system uses 2 LEDs. The green LED flashes every 15 seconds as long as the car is within the valid period of time. The red light starts flashing slowly if the car has reached a certain tolerance time limit. The red light then flashes more quickly if the car has eventually exceeded the allowed parking duration. The selected sensor for the prototype is an MB7389 ultrasonic sensor form MaxBotix. The external battery is connected over an micro-USB cable to the Airboard. This external battery extends the life of the Airboard. The RTC-module is connected to the pin *A5* and *A4* of the analog port to calculate the real-time information to the Airboard. The ultrasonic sensor is connected with the PWM pin of the Airboard, where it receives the raw values of the distance. Every power supply for each component is connected to an digital port to save energy.

This sensor is IP56 rated and has a feature which suits for the project. This sensor is able to detect the largest target in a range and not the first one. This is helpful in a situation where a person stands in front of the sensor, in which it still detects the car. The used antenna for the project is the Sigbee from ATIM. This antenna

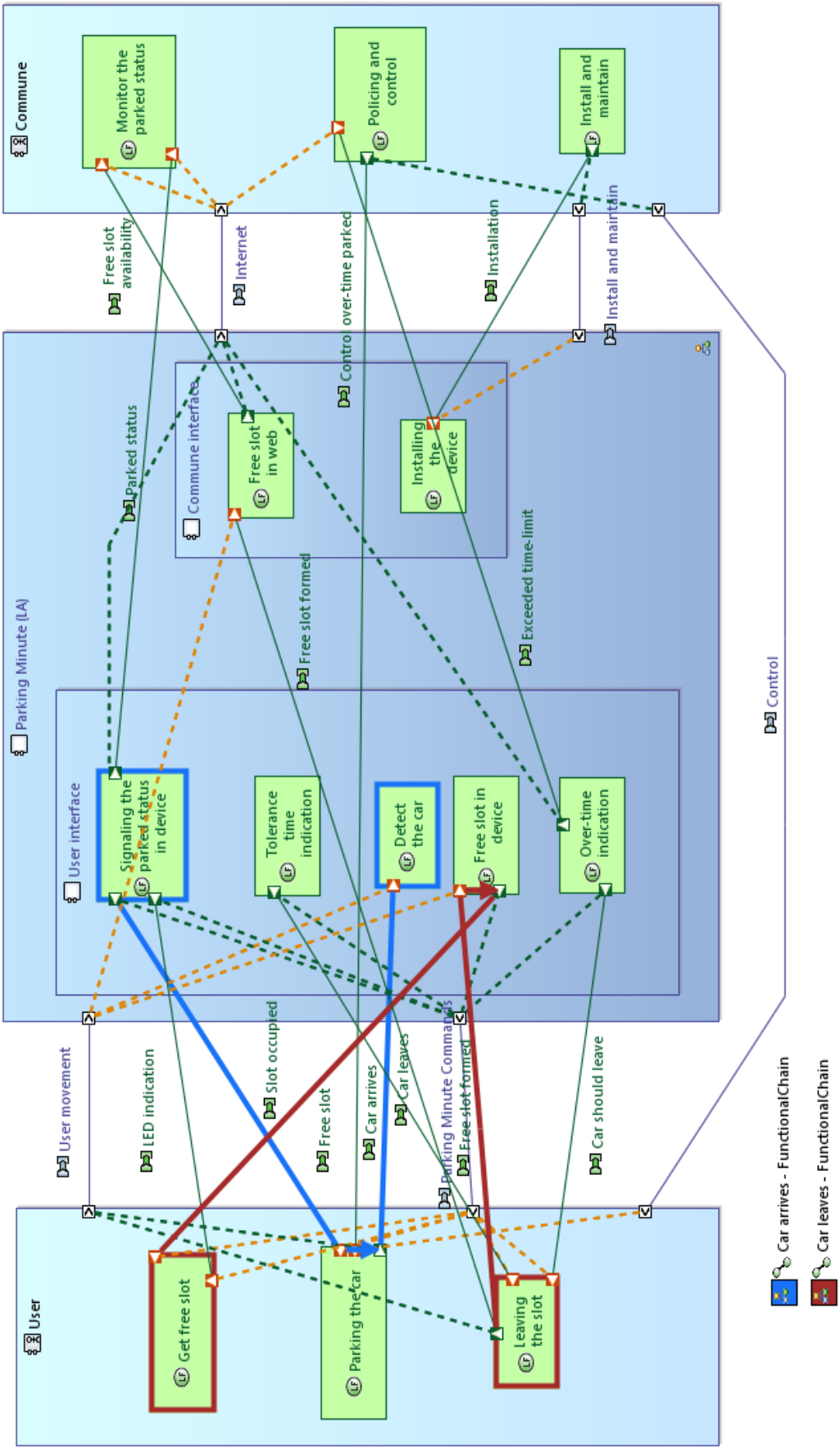


Figure VIII.14: Logical architecture of Parking minute.

Figure VIII.15: Dashboard of the prototype - Parking minute.

module fits only to the ariboard. The module has a low power consumption and thus it is the best module to send data.

The message will be sent over the SIGFOX network to an SigFox antenna station. The Sigbee module, has his own identification number. The RTC-module is a real time clock, which is used to have the current date and time. This RTC-Module has been used because the Airboard doesn't come with an internal real time clock, which continues counting when the board is shut down or put into deep sleep mode. The Airboard has an integrate small battery, which doesn't have a long autonomy for our needs. The prototype uses an external battery to extend the autonomy. In the field testing a power-bank has been used with an output voltage of 5V and an energy consumption of 6200mAh.

The hardware components installed on the IP56-conform case as shown in VIII.11. This case is installed inside of the system's pole stand which is made out of hard PVC and thus assures that the system is protected from any weather conditions. Furthermore, the hardware is protected against thievery since the bonnet of the case is locked by a key. The auxiliary battery which extends the internal battery of the Airboard is attached next to the actual hardware case. This case is also waterproof up to IP56.

We tested the system on a big parking lot of Belval Luxembourg and commune of Esch-sur-Alzette Figure VIII.16. The first test-case was whether the sensor detects the car accurately. Over exhaustive run of the system and parking the car in the

slot in different positions, the correct positioning of system was identified. Also the test-case, different angles of the system was tested to check whether the sensor still detects the car. Furthermore, the distance to which the sensor can able to detect the car, was measured and further calibration is performed for best possible operation of the system. The Table VIII.1 provides the field test data stored on the SIGFOX back-end shown in Figure VIII.17.

Table VIII.1: Understanding the field test data of Parking minute.

Time-out Reached : indicating tolerance time reached	TR
Car Goes : indicating car left the slot	CG
Car Arrives : indicating a new car arrives the slot	CA
3.46	*remaining battery level*
1CCAE5	*Device ID*
Location	*Geo position of the device*

(b) Dashboard of the prototype with "Green" indicates a car is parked winthin the valid period of time

(a) Field testing in a public parking place

Figure VIII.16: Parking minute field testing.

Device 1CCAE5 - Messages

					Base station reception attributes					
Time	Delay (s)	Header	Data / Decoding	Location	Base station	RSSI (dBm)	SNR (dB)	Freq (MHz)	Frames	Callbacks
2017-10-16 11:30:37	1.2	0000	5452332e343600 ASCII: TR3.46.		2E45 2E4E 2E60	-123.00 -122.00 -142.00	27.94 29.10 9.03	868.1501 868.1499 868.1506	1/3 1/3 1/3	
2017-10-16 10:58:28	< 1	0000	4341332e343600 ASCII: CA3.46.		2E45 102B 2E60	-118.00 -144.00 -142.00	32.70 7.53 8.80	868.1402 868.1402 868.1407	1/3 1/3 1/3	
2017-10-16 10:45:45	< 1	0000	4347332e343500 ASCII: CG3.45.		2E4E 2E45 102A	-130.00 -124.00 -126.00	21.43 27.20 24.68	868.1299 868.1300 868.1301	1/3 1/3 1/3	
2017-10-16 10:43:34	< 1	0000	5452332e343600 ASCII: TR3.46.		2E4E 2E45 3D70	-112.00 -123.00 -123.00	39.26 27.68 28.42	868.1286 868.1288 868.1284	1/3 1/3 1/3	
2017-10-16 10:11:31	< 1	0000	4341332e343500 ASCII: CA3.45.		102A 2E4E 102B	-124.00 -113.00 -144.00	26.64 37.84 7.58	868.1297 868.1295 868.1297	1/3 1/3 1/3	
2017-10-16 10:10:39	1.6	0000	4347332e343500 ASCII: CG3.45.		102A 2E45 2E4E	-126.00 -123.00 -115.00	24.80 28.42 35.95	868.1457 868.1456 868.1455	1/3 1/3 1/3	
2017-10-16 09:39:35	< 1	0000	5452332e343500 ASCII: TR3.45.		2E4E 102A 2E45	-112.00 -126.00 -117.00	39.51 25.31 33.63	868.1537 868.1540 868.1539	1/3 1/3 1/3	
2017-10-16 09:07:00	1.1	0000	4341332e343400 ASCII: CA3.44.		2E45 102A 2E4E	-126.00 -127.00 -114.00	25.43 23.85 36.95	868.1462 868.1463 868.1460	1/3 1/3 1/3	
2017-10-16 09:06:09	< 1	0000	4347332e343400 ASCII: CG3.44.		102B 3DB1 2E45	-141.00 -137.00 -121.00	10.46 14.60 30.40	868.1142 868.1139 868.1142	1/3 1/3 1/3	

Figure VIII.17: Field test data of Parking minute.

Chapter IX

List of Publications

[Altmeyer et al., 2016] Sebastian Altmeyer, Sakthivel Manikandan Sundharam, and Nicolas Navet. The Case for FIFO Real-time Scheduling. Technical report, University of Luxembourg, 2016.

[Fejoz et al., 2016a] Loïc Fejoz, Nicolas Navet, Sakthivel Sundharam, and Sebastian Altmeyer. Demo Abstract: Applications of the CPAL Language to Model, Simulate and Program Cyber-Physical Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 59, 2016.

[Sundharam et al., 2016a] Sakthivel Manikandan Sundharam, Sebastian Altmeyer, and Nicolas Navet. Model interpretation for an AUTOSAR compliant engine control function. In *7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, July 2016.

[Sundharam et al., 2016b] Sakthivel Manikandan Sundharam, Sebastian Altmeyer, and Nicolas Navet. Work in progress : An Optimizing Framework for Real-time Scheduling. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016.

[Sundharam et al., 2016c] Sakthivel Manikandan Sundharam, Loic Fejoz, and Nicolas Navet. Connected Motorized riders—a Smart Mobility System to Connect Two and Three-wheelers. In *Embedded Computing and System Design (ISED), 2016 Sixth International Symposium on*, pages 345–348. IEEE, 2016.

[Sundharam et al., 2016d] S. M. Sundharam, L. Havet, S. Altmeyer, and N. Navet. A Model-based Development Environment for Rapid-prototyping of Latency-sensitive Automotive Control Software. In *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, pages 228–233, Dec 2016.

[Sundharam et al., 2018] Sakthivel Manikandan Sundharam, Nicolas Navet, Sebastian Altmeyer, and Lionel Havet. A Model-driven Co-design Framework for Fusing Control and Scheduling Viewpoints. *Sensors*, 18(2), 2018.

[Sundharam, 2016] Sakthivel Manikandan Sundharam. Short Term Scientific Mission Report on Applying Timing Analysis Metamodel to Co-simulation Model Based Development Environment. Technical report, ICT COST Action IC1202, European Cooperation in Science & Research, 2016.

Bibliography

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

[Altisen et al., 2002] Altisen, K., Gößler, G., and Sifakis, J. (2002). Scheduler modeling based on the controller synthesis paradigm. *Real-Time Systems*, 23(1-2):55–84.

[Altmeyer et al., 2015] Altmeyer, S., Navet, N., and Fejoz, L. (2015). 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.

[Altmeyer et al., 2016] Altmeyer, S., Sundharam, S. M., and Navet, N. (2016). The case for FIFO real-time scheduling. Technical report, University of Luxembourg.

[Aminifar et al., 2012] Aminifar, A., Samii, S., Eles, P., Peng, Z., and Cervin, A. (2012). Designing high-quality embedded control systems with guaranteed stability. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 283–292. IEEE.

[Anssi et al., 2013] Anssi, S., Kuntz, S., Gérard, S., and Terrier, F. (2013). On the gap between schedulability tests and automotive task model. *Journal of Systems Architecture*, 59(6):341–350.

[ARM, 2018] ARM (2018). ARM mbed development platform for IoT. `https://www.mbed.com/en/` (accessed September, 2018).

[Audsley, 2001] Audsley, N. C. (2001). On priority asignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44.

[AUTOSAR, 2015] AUTOSAR (2015). Autosar 4.2.2 design patterns catalogue. `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_TR_AIDesignPatternsCatalogue.pdf` (accessed 31 January, 2019).

[Ballabriga et al., 2010] Ballabriga, C., Cassé, H., Rochange, C., and Sainrat, P. (2010). Otawa: an open toolbox for adaptive WCET analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer.

[Baruah and Burns, 2006] Baruah, S. and Burns, A. (2006). Sustainable scheduling analysis. In *proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, pages 159–168.

[Baruah et al., 1999] Baruah, S., Buttazzo, G., Gorinsky, S., and Lipari, G. (1999). Scheduling periodic task systems to minimize output jitter. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 62–69. IEEE.

[Baruah, 2006] Baruah, S. K. (2006). The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1):9–20.

[Baruah et al., 1997] Baruah, S. K., Chen, D., and Mok, A. K. (1997). Jitter concerns in periodic task systems. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 68–77. IEEE.

[Bastoni et al., 2010] Bastoni, A., Brandenburg, B., and Anderson, J. (2010). Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '10)*, pages 33–44.

[Benveniste and Berry, 1991] Benveniste, A. and Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282.

[Benveniste et al., 2012] Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.-B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., and Larsen, K. G. (2012). *Contracts for System Design*. PhD thesis, Inria.

[Bézivin and Gerbé, 2001] Bézivin, J. and Gerbé, O. (2001). Towards a precise definition of the omg/mda framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE.

[Bini and Buttazzo, 2005] Bini, E. and Buttazzo, G. C. (2005). Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154.

[Broy et al., 2010] Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., and Ratiu, D. (2010). Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526–545.

[Broy et al., 2013] Broy, M., Kirstan, S., Krcmar, H., Schätz, B., and Zimmermann, J. (2013). What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310.

[Buttazzo, 2005] Buttazzo, G. C. (2005). Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29(1):5–26.

[Capella, 2019] Capella (2019). Polarsys. `https://www.polarsys.org/capella/` (accessed 31 January, 2019).

[Cazorla et al., 2013] Cazorla, F. J., Quiñones, E., Vardanega, T., Cucu, L., Triquet, B., Bernat, G., Berger, E., Abella, J., Wartel, F., Houston, M., et al. (2013). Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94.

[Cervin, 2012] Cervin, A. (2012). Stability and worst-case performance analysis of sampled-data control systems with input and output jitter. In *American Control Conference (ACC), 2012*, pages 3760–3765. IEEE.

[Cervin et al., 2003a] Cervin, A., Henriksson, D., Lincoln, B., Eker, J., and Årzén, K.-E. (2003a). How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3).

[Cervin et al., 2003b] Cervin, A., Henriksson, D., Lincoln, B., Eker, J., and Årzén, K.-E. (2003b). How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3).

[Cervin et al., 2004] Cervin, A., Lincoln, B., Eker, J., Arzén, K.-E., and Buttazzo, G. (2004). The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, pages 1–9. Citeseer.

[Christoulakis and Thramboulidis, 2016] Christoulakis, F. and Thramboulidis, K. (2016). IoT-based integration of iec 61131 industrial automation systems: The case of UML4IoT. In *Industrial Electronics (ISIE), 2016 IEEE 25th International Symposium on*, pages 322–327. IEEE.

[Cruise Control, 2019] Cruise Control (2019). Speed cruise control system using simulink® and stateflow®. `https://nl.mathworks.com/help/plccoder/examples/speed-cruise-control-system-using-simulink-and-stateflow.html` (accessed 31 January, 2019).

[Davis, 2014] Davis, R. I. (2014). A review of fixed priority and edf scheduling for hard real-time uniprocessor systems. *ACM SIGBED Review*, 11(1):8–19.

[Davis et al., 2015] Davis, R. I., Thekkilakattil, A., Gettings, O., Dobrin, R., and Punnekkat, S. (2015). Quantifying the exact sub-optimality of non-preemptive scheduling. In *proceedings of the Real-Time Systems Symposium, (RTSS '15)*, pages 96–106.

[Department of Informatics, 2018] Department of Informatics, U. o. O. (2018). ThingML modeling language for embedded distributed systems. `http://thingml.org/` (accessed September, 2018).

[Derler et al., 2013] Derler, P., Lee, E. A., Törngren, M., and Tripakis, S. (2013). Cyber-physical system design contracts. In *Cyber-Physical Systems (ICCPS), 2013 ACM/IEEE International Conference on*, pages 109–118. IEEE.

[DesignCPS, 2019] DesignCPS (2019). Cyber physical action language (CPAL). `https://www.designcps.com/wp-content/uploads/cpal-intro.pdf`(accessed 31 January, 2019).

[Di Natale et al., 2013] Di Natale, M., Dong, C., and Zeng, H. (2013). Reality check: the need for benchmarking in RTS and CPS. In *proceedings of the 4th Real-Time Scheduling Open Problems Seminar (RTSOPS '13)*, pages 18–19.

[Easterbrook, 2004] Easterbrook, S. (2004). Csc2106s: Requirements engineering. `http://www.cs.toronto.edu/~sme/CSC2106S/slides/01-intro.pdf`.

[Feiler et al., 2009] Feiler, P. H., Hansson, J., De Niz, D., and Wrage, L. (2009). System architecture virtual integration: An industrial case study. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[Fejoz et al., 2016a] Fejoz, L., Navet, N., Sundharam, S., and Altmeyer, S. (2016a). Demo abstract: Applications of the CPAL language to model, simulate and program cyber-physical systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, page 59.

[Fejoz et al., 2016b] Fejoz, L., Navet, N., Sundharam, S. M., and Altmeyer, S. (2016b). Demo abstract: Applications of the cpal language to model, simulate and program cyber-physical systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[Fleming, 2001] Fleming, W. J. (2001). Overview of automotive sensors. *IEEE sensors journal*, 1(4):296–308.

[Fowler, 2010] Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

[Geilen, 2002] Geilen, M. C. W. (2002). *Formal techniques for verification of complex real-time systems*. Technische Universiteit Eindhoven.

[George and Minet, 1997] George, L. and Minet, P. (1997). A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*, pages 441–448.

[George et al., 1996] George, L., Rivierre, N., and Spuri, M. (1996). Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report 2966, Institut National de Recherche et Informatique et en Automatique (INRIA), France.

[Gerber and Hong, 1997] Gerber, R. and Hong, S. (1997). Slicing real-time programs for enhanced schedulability. *ACM Trans. Program. Lang. Syst.*, 19(3):525–555.

[Ghosh, 2011] Ghosh, D. (2011). *DSLS IN ACTION*. Wiley India Pvt. Limited.

[Goossens, 2003] Goossens, J. (2003). Scheduling of offset free systems. *Real–Time Systems*, 24(2):239–258.

[Graham, 1969] Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429.

[Grenier et al., 2006] Grenier, M., Goossens, J., and Navet, N. (2006). Near-Optimal Fixed Priority Preemptive Scheduling of Offset Free Systems. In *proceedings of 14th International Conference on Real-Time and Networks Systems (RTNS '06)*, pages 35–42.

[Grenier and Navet, 2008] Grenier, M. and Navet, N. (2008). Fine tuning MAC level protocols for optimized real-time QoS. *IEEE Transactions on Industrial Informatics, special issue on Industrial Communication Systems*, 4(1).

[Gutiérrez and Harbour, 2003] Gutiérrez, J. C. P. and Harbour, M. G. (2003). Offset-based response time analysis of distributed systems scheduled under EDF. In *proceedings of the 15th Euromicro Conference on Real-Time Systems (ECRTS '03)*, pages 3–12.

[Harbour et al., 2001] Harbour, M. G., García, J. G., Gutiérrez, J. P., and Moyano, J. D. (2001). Mast: Modeling and analysis suite for real-time applications. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 125–134. IEEE.

[Haskins et al., 2006] Haskins, C., Forsberg, K., Krueger, M., Walden, D., and Hamelin, D. (2006). Systems engineering handbook. In *INCOSE*.

[Henia and Ernst, 2006] Henia, R. and Ernst, R. (2006). Improved offset-analysis using multiple timing-references. In *proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, pages 450–455.

[Henzinger, 2008] Henzinger, T. A. (2008). 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.

[Huang et al., 2004] Huang, J., Voeten, J. P., Ventevogel, A., and Van Bokhoven, L. (2004). Platform-independent design for embedded real-time systems. In *Languages for system specification*, pages 35–50. Springer.

[IBM, 2019] IBM (2019). IBM watson IoT platform. http://www.ibm.com/internet-of-things/iot-solutions/watson-iot-platform/ (accessed 31 January, 2019).

[Iyenghar et al., 2016] Iyenghar, P., Noyer, A., Engelhardt, J., Pulvermueller, E., and Westerkamp, C. (2016). End-to-end path delay estimation in embedded software involving heterogeneous models. In *Industrial Embedded Systems (SIES), 2016 11th IEEE Symposium on*, pages 1–6. IEEE.

[Kirsch and Sokolova, 2012] Kirsch, C. M. and Sokolova, A. (2012). The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer.

[Kleppe et al., 2003] Kleppe, A. G., Warmer, J. B., and Bast, W. (2003). *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional.

[Kreiker et al., 2011] Kreiker, J., Tarlecki, A., Vardi, M. Y., and Wilhelm, R. (2011). Modeling, Analysis, and Verification - The Formal Methods Manifesto 2010 (Dagstuhl Perspectives Workshop 10482). *Dagstuhl Manifestos*, 1(1):21–40.

[Lampke et al., 2015] Lampke, S., Schliecker, S., Ziegenbein, D., and Hamann, A. (2015). Resource-aware control-model-based co-engineering of control algorithms and real-time systems. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 8(2015-01-0168):106–114.

[Leontyev and Anderson, 2007] Leontyev, H. and Anderson, J. H. (2007). Tardiness bounds for FIFO scheduling on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS '07)*, pages 71–82.

[Li et al., 2014] Li, H., Huang, Y., Dai, X., and Hu, M. (2014). Design and application of the ecu application software components library for diesel engine. Technical report, SAE Technical Paper.

[Li et al., 2011] Li, X., Scharbarg, J.-L., Ridouard, F., and Fraboul, C. (2011). Existing offset assignments are near optimal for an industrial afdx network. *SIGBED Review*, 8(4):49–54.

[Lincoln and Cervin, 2002] Lincoln, B. and Cervin, A. (2002). Jitterbug: A tool for analysis of real-time control performance. In *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, volume 2, pages 1319–1324. IEEE.

[Lunniss et al., 2014] Lunniss, W., Altmeyer, S., and Davis, R. I. (2014). A comparison between fixed priority and edf scheduling accounting for cache related pre-emption delays. *Leibniz Transactions on Embedded Systems*, 1(1).

[Maraninchi and Rémond, 2003] Maraninchi, F. and Rémond, Y. (2003). Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of computer programming*, 46(3):219–254.

[mbed, 2019] mbed, A. (2019). ARM mbed connector. `https://connector.mbed.com/` (accessed 31 January, 2019).

[Merz and Navet, 2008] Merz, S. and Navet, N. (2008). *Modeling and Verification of Real-Time Systems-Formalisms and Software Tools*. ISTE Publishing.

[Monot et al., 2012] Monot, A., Navet, N., Bavoux, B., and Simonot-Lion, F. (2012). Multisource software on multicore automotive ecus; combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942.

[Morelli et al., 2014] Morelli, M., Cremona, F., and Di Natale, M. (2014). A system-level framework for the evaluation of the performance cost of scheduling and communication delays in control systems. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*.

[Mraidha et al., 2014] Mraidha, C., Tucci-Piergiovanni, S., and Gerard, S. (2014). Schedulability analysis at early design stages with MARTE. In *Embedded Systems Development*, pages 101–119. Springer.

[Nasri et al., 2018] Nasri, M., Davis, R. I., and Brandenburg, B. B. (2018). FIFO with offsets: High schedulability with low overheads. In *to appear in the Proceedings of 24th IEEE Real-Time and Embedded Technology and Applications Symposium, (RTAS 2018)*. IEEE.

[Nasri and Fohler, 2015a] Nasri, M. and Fohler, G. (2015a). An efficient method for assigning harmonic periods to hard real-time tasks with period ranges. In *27th Euromicro Conference on Real-Time Systems (ECRTS 2015)*, pages 149–159. IEEE.

[Nasri and Fohler, 2015b] Nasri, M. and Fohler, G. (2015b). Non-work-conserving scheduling of non-preemptive hard real-time tasks based on fixed priorities. In *proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15)*, pages 309–318.

[Navet and Fejoz, 2016a] Navet, N. and Fejoz, L. (2016a). Cpal: High-level abstractions for safe embedded systems. In *Proceedings of the International Workshop on Domain-Specific Modeling*, pages 35–41. ACM.

[Navet and Fejoz, 2016b] Navet, N. and Fejoz, L. (2016b). CPAL: High-level abstractions for safe embedded systems. In *Proc. 16th Workshop on Domain-Specific Modeling*. ACM.

[Navet et al., 2016a] Navet, N., Fejoz, L., Havet, L., and Altmeyer, S. (2016a). Lean model-driven development through model-interpretation: the CPAL design flow. In *Embedded Real-Time Software and Systems (ERTSS2016)*.

[Navet et al., 2016b] Navet, N., Fejoz, L., Havet, L., and Sebastian, A. (2016b). Lean model-driven development through model-interpretation: the cpal design flow. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

[Navet and Migge, 2003] Navet, N. and Migge, J. (2003). Fine tuning the scheduling of tasks through a genetic algorithm: Application to Posix1003.1b compliant OS. *IEE Proceedings Software*, 150(1):13–24.

[Navet and Simonot-Lion, 2008] Navet, N. and Simonot-Lion, F. (2008). *Automotive embedded systems handbook*. CRC press.

[Nuzzo et al., 2015] Nuzzo, P., Sangiovanni-Vincentelli, A. L., Bresolin, D., Geretti, L., and Villa, T. (2015). A platform-based design methodology with contracts and related tools for the design of cyber-physical systems. *Proceedings of the IEEE*, 103(11):2104–2132.

[OMG, 2018] OMG, O. M. G. (2018). The UML profile for modeling and analysis of real-time and embedded systems (MARTE). `https://www.omg.org/omgmarte/` (accessed September, 2018).

[Pagetti et al., 2011] Pagetti, C., Forget, J., Boniol, F., Cordovilla, M., and Lesens, D. (2011). Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338.

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

[Pramudianto et al., 2014] Pramudianto, F., Kamienski, C. A., Souto, E., Borelli, F., Gomes, L. L., Sadok, D., and Jarke, M. (2014). IoT link: An internet of things prototyping toolkit. In *Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom)*, pages 1–9. IEEE.

[Prehofer and Chiarabini, 2015] Prehofer, C. and Chiarabini, L. (2015). From internet of things mashups to model-based development. In *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*, volume 3, pages 499–504. IEEE.

[Ripoll et al., 1996] Ripoll, I., Crespo, A., and Mok, A. K. (1996). Improvement in feasibility testing for real-time tasks. *Real-Time Sytstems*, 11(1):19–39.

[Rothenberg et al., 1989] Rothenberg, J., Widman, L. E., Loparo, K. A., and Nielsen, N. R. (1989). The nature of modeling. *in Artificial Intelligence, Simulation and Modeling*.

[Rumbaugh et al., 2017] Rumbaugh, J., Booch, G., and Jacobson, I. (2017). *The unified modeling language reference manual*. Addison Wesley.

[Rushby, 2000] Rushby, J. (2000). Disappearing formal methods. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 95–96. IEEE.

[Rushby, 2006] Rushby, J. (2006). Tutorial: Automated formal methods with pvs, sal, and yices. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 262–262. IEEE.

[Sangiovanni-Vincentelli et al., 2012] Sangiovanni-Vincentelli, A., Damm, W., and Passerone, R. (2012). Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European journal of control*, 18(3):217–238.

[Short, 2010] Short, M. (2010). The case for non-preemptive, deadline-driven scheduling in real-time embedded systems. In *proceedings of the World Congress on Engineering (WCE '10)*, pages 399–404.

[Sicari et al., 2014] Sicari, S., Rizzardi, A., Coen-Porisini, A., and Cappiello, C. (2014). A nfp model for internet of things applications. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2014 IEEE 10th International Conference on*, pages 265–272. IEEE.

[Sifakis, 2008] Sifakis, J. (2008). The quest for correctness: Beyond verification.

[Sigfox, 2019] Sigfox (2019). Sigfox IoT network. `https://www.sigfox.com/en/coverage/` (accessed 31 January, 2019).

[SimEvents, 2019] SimEvents (2019). MATLAB/Simulink, SimEvents. `http://nl.mathworks.com/products/simevents/` (accessed 31 January, 2019).

[Singhoff et al., 2004a] Singhoff, F., Legrand, J., Nana, L., and Marcé, L. (2004a). Cheddar: A flexible real-time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM.

[Singhoff et al., 2004b] Singhoff, F., Legrand, J., Nana, L., and Marcé, L. (2004b). Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM. Software available at `http://beru.univ-brest.fr/~singhoff/cheddar/`.

[Stachowiak, 1973] Stachowiak, H. (1973). Allgemeine modelltheorie.

[Sundharam, 2016] Sundharam, S. M. (2016). Short term scientific mission report on applying timing analysis metamodel to co-simulation model based development environment. Technical report, ICT COST Action IC1202, European Cooperation in Science & Research.

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

[Sundharam et al., 2016b] Sundharam, S. M., Altmeyer, S., and Navet, N. (2016b). Work in progress : An optimizing framework for real-time scheduling. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

[Sundharam et al., 2016c] Sundharam, S. M., Fejoz, L., and Navet, N. (2016c). Connected motorized riders—a smart mobility system to connect two and three-wheelers. In *Embedded Computing and System Design (ISED), 2016 Sixth International Symposium on*, pages 345–348. IEEE.

[Sundharam et al., 2016d] Sundharam, S. M., Havet, L., Altmeyer, S., and Navet, N. (2016d). A model-based development environment for rapid-prototyping of latency-sensitive automotive control software. In *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, pages 228–233.

[Sundharam et al., 2018] Sundharam, S. M., Navet, N., Altmeyer, S., and Havet, L. (2018). A model-driven co-design framework for fusing control and scheduling viewpoints. *Sensors*, 18(2).

[Tankovic et al., 2012] Tankovic, N., Vukotic, D., and Zagar, M. (2012). Rethinking model driven development: analysis and opportunities. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference on*, pages 505–510. IEEE.

[Tindell, 1994] Tindell, K. (1994). Adding time-offsets to schedulability analysis. Technical report, University of York.

[Törngren, 1998] Törngren, M. (1998). Fundamentals of implementing real-time control applications in distributed computer systems. *Real-time systems*, 14(3):219–250.

[Torngren et al., 2006] Torngren, M., Henriksson, D., Arzen, K.-E., Cervin, A., and Hanzalek, Z. (2006). Tool supporting the co-design of control systems and their real-time implementation: Current status and future directions. In *Proceedings of the Conference on Computer Aided Control Systems Design, CACSD*, pages 1173–1180. IEEE.

[Voirin, 2017] Voirin, J.-L. (2017). *Model-based System and Architecture Engineering with the Arcadia Method*. Elsevier.

[Völter, 2009] Völter, M. (2009). Best practices for dsls and model-driven development. *Journal of Object Technology*, 8(6):79–102.

[WHO, 2019] WHO (2019). Who global status report 2015. `https://www.who.int/violence_injury_prevention/road_safety_status/2015/en/` (accessed 31 January, 2019).

[Wilhelm et al., 2008a] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. (2008a). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36.

[Wilhelm et al., 2008b] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008b). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53.

[Woodcock et al., 2009] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36.

[Yomsi et al., 2012] Yomsi, P. M., Bertrand, D., Navet, N., and Davis, R. I. (2012). Controller Area Network (CAN): response time analysis with offsets. In *proceedings of 9th IEEE International Workshop on Factory Communication Systems (WFCS '12)*, pages 43–52.

[Ziegenbein and Hamann, 2015] Ziegenbein, D. and Hamann, A. (2015). Timing-aware control software design for automotive systems. In *Proceedings of the 52nd Annual Design Automation Conference*, page 56. ACM.

List of Glossaries

ECU Electronic Control Unit 1, 2

ABS Anti-lock Braking System 2

ACC Adaptive Cruise Control 2, 4, 9

ADAS Advanced Driving Assistance Systems 2, 4

ALM Application Life-cycle Management 31

AUTOSAR AUTomotive Open System ARchitecture 2, 19, 35, 36, 41, 44

BMMI Bare Machine Model Interpretation 5, 8

ConMoR Connected Motorized Riders 115

CPAL CPAL is an acronym for Cyber-Physical Action Language iii, iv, 7, 9, 13, 14, 19, 33, 34, 40–43, 45, 47, 48, 50, 52–58, 60, 64, 82–85, 87, 88, 90–94, 96–98, 100–103, 106–117

CPS Cyber-Physical Systems 1, 3, 6, 33, 107, 108, 110

DES Discrete Event Simulator 7, 8

DSML Domain-Specific Modeling Languages 19

ECU Electronic Control Unit 1, 2, 5, 8, 15, 16, 18, 19, 31, 32, 35, 45, 53, 84, 85, 102, 108

ESP Electronic Stability Program 2

FIFO First In First Out iv, 8–11, 13

FSM Finite State Machine 34, 87

GPM General-Purpose Modeling 18, 19

GPML General-Purpose Modeling Languages 19

IoT Internet of Things 9, 113–120

LED Light-Emitting Diode 129, 130, 132

LET Logical Execution Time 23

LKAS Lane Keeping Assist Systems 4

LTI Linear Time Invariant System 21

MBD Model-Based Design iii, iv, 1, 3–6, 8–10, 13, 15, 16, 18, 20, 23, 31, 41, 42, 44, 47, 48, 59, 60, 83, 107, 108, 111, 113, 114, 117

MBSE Model-Based System Engineering 15, 18, 130

PWM Pulse Width Modulation 132

QoS Quality of Service 3, 4, 9

RTC Real-Time Clock 8, 130, 132

SDLC Software Development Life Cycle 16, 23

SIGFOX SigFox is a French company that builds wireless networks to connect low-energy objects 120, 121, 123, 124, 126, 129, 130, 132, 134

SysML System Modeling Language 14, 18–20, 113, 121

UAV Unmanned Aerial Vehicle 1

UML Unified Modeling Language 18–20, 114

WCET Worst-Case Execution Time 22, 40, 84–86, 88, 93–95, 101, 102, 106