Matteo Morelli

A System-Level Framework for the Design of Complex Cyber-Physical Systems from Synchronous-Reactive Models

Scuola Superiore Sant'Anna
di Studi Universitari e di Perfezionamento

Anno Accademico
2014-2015
Corso di perfezionamento
in Tecnologie Innovative

A System-Level Framework for the Design of Complex Cyber-Physical Systems from Synchronous-Reactive Models

Autore
Matteo Morelli
Tutor
Prof. Marco Di Natale

Abstract

THE goal of this research is the definition of a design process, supported by tools, for the development, verification and deployment of time-sensitive cyber-physical systems (CPS) control applications. Safe interaction between a robotic arm and a human co-worker, robotic-assisted surgery, flight control of an aerial robot; but also, data acquisition and coordination of smart traffic light systems, advanced driver assistance systems in cars and operation of smart grids in public infrastructures — all of these are examples of time-sensitive CPS control applications. They are realized as software algorithms (control software) whose correct functioning depends not only upon the logical correctness of control actions, but also upon the time in which these actions are performed.

Control software is in turn realized as a set of software tasks exchanging messages on top of networks of embedded computers running real-time operating systems (execution platform). The realization of control software is difficult, because a number of factors may affect the final performances, such as, how control functions are mapped into tasks, how tasks are deployed onto the computing nodes, the types of resources for local communication, networks and protocols for communication among remote nodes. And it is made even more challenging by the need to keep the software development costs low and the time-to-market short, that mandates careful selection and efficient usage of hardware/software resources.

As of today, embedded control application design is carried out in stages. In the first stage, a Model-Based Design approach is used, where models of control functionality are designed and verified by simulation in a virtual environment. In most cases functional models are based on a Synchronous-Reactive (SR) execution paradigm. All the computations and communications are assumed to complete within the interval between two events in logical time and implementation aspects (including the time delays introduced by the execution platform) are not considered. In the next stage, a task implementation (code) that realizes the control law is produced. Then, the software code is analyzed onto prototypical (or even the final target) hardware to verify that timing constraints are satisfied. This approach impedes design-space exploration, and defers the validation of the selected implementation (hardware/software) until late, in the integration/testing stage.

In this work, we propose a Model-Driven Design process and tools, encompassing the tight integration of control, hardware and real-time software architectures from the very beginning, at model level. This framework supports the transition from the functional model to the code implementation (with the preservation of the original model semantics), and enables designers to explore tradeoffs between delays (of task scheduling and messages) and control performances when a semantics-preserving implementation of functionality is not achievable. The framework is based on *standards* (Simulink and OMG's SysML/MARTE, MOFM2T and M2M) and *open* (EMF-based) tools (Papyrus, Acceleo and QVTo).

We define a common semantic domain and rules for the integration of SR models of control with models representing the execution platform and the software tasks and messages that realize the functions. Models incorporate and expose appropriate information to allow accurate prediction of control performance and timing properties of a candidate software implementation. The design cycle is a sequence of tool-assisted stepwise refinements which is iterated until the implementation satisfies the initial specification. Models and rules are defined to generate semantics-preserving deployment of control functionality on top of the real-time capable, component-based middleware Orocos-RTT that is very popular among robotics practitioners (researchers and industry).

When the full preservation of functional model semantics cannot be guaranteed by the implementation, designers may resort to simulation to gather important information about the estimated impact of time delays (dependent on code execution, scheduling of tasks, and network communication latencies) on the control performances. System-level prediction of timing behavior is enabled by T-Res, a Simulink-based co-simulation framework which integrates external simulation engines for real-time scheduling and network communication. The advantages of this approach are demonstrated onto a model of a simulated rotorcraft Unmanned Aerial Vehicle (UAV).

Finally, we define a simulation-driven optimization process for the automated synthesis of software architecture configurations on single-core platforms, in those situations where there are no feasible task-sets for the deadlines determined in the control design stage. The approach couples MILP-based optimization with simulation and extend the traditional design flows to include the exploration of relaxed deadlines and order-of-execution constraints. The experiments conducted on a simulated UAV case study show the benefits of the approach.

To Annalisa and Leandro.

Acknowledgements

THIS work would not have been possible without the advice and support of many. Firstly, I want to thank my advisor, Marco Di Natale, for having helped me to get into the domain of Model-Driven Design and for having always been available to answer my often confusing questions and discuss concepts in detail. Marco, I greatly appreciated your style, your constructive criticism and the freedom you gave me to work independently.

I want also to thank Giuseppe Lipari for having actively supported the work on RTSim and Simulink, and for the interesting discussions on design patterns in Object-Oriented Programming, which have profoundly influenced the way I now develop software. Many thanks go also to Mauro Marinoni for his endless patience and for having always provided immediate support with hardware issues and related discussions. Thanks to Enrico Bini and Giorgio Buttazzo, who, though less directly involved in my research, have repeatedly provided valuable suggestions and feedback.

A Special thank goes to Fabio Cremona, who worked with me to develop T-Res. Fabio, I appreciated your pragmatism and determination to deliver a working system. Thank you for the enjoyable hours of fighting to embed the OMNeT++ simulation kernel into Simulink. It has been a pleasure to work with you.

I want to thank my colleagues of LISE Lab. at CEA-List Yasmina Seddik, Sara Tucci-Piergiovanni and Chokri Mraidha, who have co-authored the work on the automated synthesis of real-time control tasks. Thanks Sara and Yasmina for the friendly welcome in Paris and also for the practical help at the beginning of my stay in France.

I wish to thank Federico Moro, Luigi Palopoli, Tizar Rizano and Daniele Fontanelli, of DISI Lab. at the University of Trento for having shared (and explained) their Simulink models and C++ code for the (ongoing) project on the robotic car testbench.

I am very grateful to Paolo Gai who shared his experience as a former PhD student of ReTiS Lab. and pointed me in the right direction at the end of my Master's Degree program. And thanks also to Simone Mannori and Roberto Bucher, who first introduced me into the world of code-generation, since the days of the Scicos-RTAI Code Generator and the Robotics Toolbox for Scilab/Scicos.

Thanks to Alessandro Passaro, for the helpful discussions on Acceleo at the beginning of my PhD work, and to Dario Di Stefano and Riccardo Schiavi (Evidence Srl.), for their support with E4Coder. In general, thanks to all the friendly and smart guys at Evidence Srl. (Errico, Davide, and many others).

Thanks to all the people of ReTiS Lab., who were next to me in this long project and adventure: Andrea and Riccardo (who enrolled in the PhD Program with me in November 2011); the former PhD students Christian, Matteo, Claudio, Giulio, Francesco, Juri, Mario, Stefano, Marco and Daniele; the current PhD students Youcheng, Paquale, Carmelo, Alessandra, Alessandro, Andrea, David, Davide, Simone and Luca; the research fellows Anna Lina, Gabriele, Gianluca, Daniel, Paolo, Mariano; and all the people of the staff Igor, Claudio, Valentina, Annalisa, Stefania, Sabrina, Elena, Antonio, Francesca, Valeria, Ketty and Isabella.

Last, but not least, I desire to thank my family. My parents, Paolo and Paola, who have been encouraging throughout my entire lifetime, and for all their help. Flavio and Rosanna, for coming to Pisa so many times to help Annalisa when I was traveling.

Annalisa and Leandro, this work is dedicated to you. *Grazie Annalisa per essere sempre dalla mia parte e per il tuo incessante supporto. Grazie per la determinazione con cui lavori e ti prendi ogni giorno cura della nostra famiglia (inclusi ovviamente Spike, Jackie, Sansone, Elvis e Priscilla—grazie anche a voi!), adesso che io sono lontano. Il tuo esempio mi spinge ogni giorno a dare il massimo. Sei una moglie e una madre meravigliosa. Leandro, grazie per prenderti cura della mamma e per i tuoi incoraggiamenti. Ho stretto i denti come mi hai detto di fare tante volte e... Visto? Ce l'abbiamo fatta! Ti voglio bene, piccolo mio* ♡

Contents

Abstract iii

Acknowledgements vii

List of Figures xiii

List of Tables xvii

Chapter 1. Introduction 1
- 1.1. Cyber-Physical Systems (CPS) 1
- 1.2. Current Design Workflows 2
- 1.3. Platform-Based Design (PBD) 3
- 1.4. Research Objectives 5
  - RI.1. Models and abstractions for the verification of implementation properties 6
  - RI.2. Semantics-preserving application deployments 7
  - RI.3. Simulation tools to predict the system-level timing behavior 7
  - RI.4. Automatic software architecture configuration 7
- 1.5. Thesis Outline 8

Chapter 2. Methods and Tools for System-Level Modeling and Design 9
- 2.1. Introduction 9
- 2.2. Model-Based Design (MBD) 9
- 2.3. Model-Driven Engineering (MDE) 10
- 2.4. Frameworks for Heterogeneous-Model Integration 11
- 2.5. System-Level Approaches in Automotive and Robotics 11
- 2.6. Simulation of Platform's Delay Effects 13
- 2.7. Discussion and Conclusions 15

Chapter 3. Design-Process Flow and Platform Meta-Models 17
- 3.1. Introduction 17
- 3.2. Process Flow Based on Standard Technologies 20
- 3.3. Functional Modeling 21
  - 3.3.1. Functional Modeling in Simulink and EMF 21
  - 3.3.2. M2M Transformation to SysML 22
- 3.4. Platform Modeling 26
  - 3.4.1. Physical Platform Modeling 26

3.4.2. BSW Resources Modeling 28
3.5. Software Architecture Modeling 30
3.6. Summary 32

Chapter 4. Generation of Semantics-Preserving Robot Controls from Simulink Models 35
4.1. Context and Positioning 35
4.2. Constraints in the Implementation of Synchronous (Simulink) Models 37
4.2.1. Model Assumptions and Basic Formalization of the Synchronous Semantics 37
4.2.2. Software Implementation and Preservation of Data-Flows 38
4.3. The Orocos-RTT Run-Time Environment 39
4.4. The Code-Generation Process 40
4.4.1. Generation of Task Synchronization Infrastructure 41
4.4.2. Generation of Task Code 45
4.4.3. Implementation of the Functional Communication Links 45
4.5. Summary 46

Chapter 5. Platform-Aware Control Simulations in Simulink Through Co-Simulation 49
5.1. Context and Positioning 49
5.2. How the Simulink Engine Simulates a Dynamic System 51
5.3. Platform Simulators and Execution Models 52
5.3.1. Discrete-Event Platform Simulators 52
5.3.2. Execution Model of Real-Time Tasks 52
5.3.3. Execution Model of Network Communication 54
5.4. The T-Res Co-Simulation Environment 56
5.4.1. Architecture 56
5.4.2. Simulink Implementation of Platform Execution Models 57
5.4.3. Interface to Other Platform Simulators 62
5.5. Application Examples 65
5.5.1. PID Control of Three Networked DC-servo Systems 65
5.5.2. Scheduling-Aware Design of Attitude Control for a Simulated Quadrotor 67
5.6. Integration in the Proposed System-Level Design Flow 71
5.7. Summary 75

Chapter 6. Simulation-Driven Process for Automated Software Synthesis 77
6.1. Context and Positioning 77
6.2. System Model 79
6.2.1. Definition of Mapping 79
6.2.2. Response-Time Analysis 79
6.3. Optimization Model 80
6.3.1. Optimization Variables 80
6.3.2. Constraints 81
6.3.3. Optimization Metrics 83

6.4. Simulation 84
6.5. Application Example 84
6.5.1. Case Study Definition 84
6.5.2. Computation of Deadline Approximations 84
6.5.3. Exploration Strategies and Application of MILP 86
6.5.4. Evaluation of Candidate Designs 87
6.6. Summary 90

Chapter 7. Conclusions 93
7.1. Contributions 93
7.2. Limitations (Ongoing Work) 95
7.3. Opportunities for Future Research 97

List of Publications 99

Bibliography 101

List of Figures

1.1 CPS: typical realization and real-time guarantees of control software stack. 2

1.2 The PBD concept (reproduced from [**DNSV10**]). 4

1.3 The levels of platform abstractions that are of concern in this thesis. 6

3.1 Platform models, mapping process and application scenarios for the proposed design flow. 17

3.2 Examples of safety-critical and performance-sensitive systems in automotive. 18

3.3 Heterogeneous model integration and code generation by the framework tools. 21

3.4 The Ecore meta-model for the functional part. 22

3.5 The SysML meta-model for the functional part. 23

3.6 Sequence of mapping operation calls within the QVTo transformation entry point. 24

3.7 SysML profiles for the execution platform with their dependencies. 26

3.8 `HwControlUnit` model's structure. 27

3.9 `HwComputing` model's structure. 27

3.10 `HwCommunication` model's structure. 27

3.11 Portion of `HwIO` model's structure. 28

3.12 `BswRTOS` model's structure. 29

3.13 Portion of `BswCoreDataTransfer` and `BswCOM` models structures. 30

3.14 Structure of mapping model's SysML profiles. 31

4.1 Heterogeneous model integration and code generation by the framework tools. 36

4.2 Execution of a pair of connected blocks with direct feedthrough according to the SR semantics ($i_j(k) = o_i(m)$). 38

4.3 Evaluation of blocks at simulation time and at runtime (single- and multi-task implementations). 38

4.4 Schematic representation of deployed Orocos-RTT components. 40

4.5 `BswRobotMW` model's structure. 40

4.6 Structure of `Mapping::Synchronization` package (extends that of Figure 3.14c). 41

4.7 Mapping model and automatically-generated BSW model representing the deployment onto a single-core processor. 42

4.8 Mapping model and automatically-generated BSW model describing the deployment of subsystems and threads onto a dual-core processor. 43

4.9 Threads activation constraints in the dual-core architecture. 44

4.10 Synchronization between producer/consumer Orocos-RTT components on a dual-core architecture. 44

4.11 Mapping model of functional communication links into signal variables (*ComImpl*s), and automatically-generated BSW model. 46

4.12 Automatically-generated code for the platform-dependent realization of functional links. 47

5.1 T-Res in a Simulink-based PBD-like flow for real-time distributed embedded control systems development. 50

5.2 Simplified view of S-function callback methods invoked by the Simulink engine during the simulation loop[1]. 51

5.3 Execution model of a simulated real-time task in Simulink. 53

5.4 Timing for the transmission of periodic mode messages objects. 55

5.5 The middleware task TxTask executes with period t_{T_x}, reads message objects and enqueues messages at the driver level. 55

5.6 Co-simulation of the functional controls (e.g., ADAS, dark-gray blocks), the plant (car) and the task scheduling and network communication parts (light-gray/white blocks), in a simplified representation. 57

5.7 Simulink implementation of time-consuming task computations execution model. 58

5.8 Instances of T-Res blocks for the representation of kernel and tasks (bottom side), with respect to the simplified view in Figure 5.6 (top side). 59

5.9 Simulink implementation of finite-communication time network execution model. 61

5.10 Instances of T-Res blocks for the representation of network and messages (bottom side), with respect to the simplified view in Figure 5.6 (top side). 62

5.11 Extensible architecture for interfacing T-Res with other platform DES simulators (closer view of top side of Figure 5.6). 63

5.12 Implementation of software architecture of Figure 5.11 using the object adapter design pattern. 63

5.13 Simulation loop of Kernel S-function (pseudo-code). 64

5.14 The application example from TrueTime [**CHL+03**], PID control of three DC-servo systems. 65

5.15 Simulink model of DC-servo control system with back-annotations. 66

5.16 Verification of DC-servo control system back-annotated model. 67
5.17 The IRIS quadrotor (left) and the PX4FMU Autopilot (right) 68
5.18 Models used for the quadrotor flight-control scheme. 68
5.19 Attitude control with models of RT kernel and tasks from T-Res. 69
5.20 Definition of type and timing properties of tasks. 70
5.21 Execution trace of `FP#1` (200ms), clearly showing that the task-set is non-schedulable. 71
5.22 Simulation results of first candidate design solution (`FP#1`), with respect to the control performance of model of Figure 5.18a (`Functional`). 71
5.23 Execution trace of `FP#2` (200ms): task-set is ***again non-schedulable***. 72
5.24 Simulation results of refined candidate design solutions (`FP#2` and `EDF`), with respect to the control performance of model of Figure 5.18a. 72
5.25 SysML IBD describing the deployment of quadrotor's control functions and threads onto the single-core FMU board running a FP real-time scheduler. 73
5.26 Acceleo instructions for the generation of the `Kernel` block. 74

6.1 Flow of the simulation-driven automated software synthesis process. 78
6.2 The T-Res setup to evaluate the maximum acceptable delay for f_2. 86
6.3 Values of delays for f_6 (top) and f_2 (bottom) for which the performance is significantly compromised. 86
6.4 Difference with respect to ideal case. 89
6.5 R_{od} model, comparison of task configurations and optimization metrics ($U = 92\%$). 90

List of Tables

3.1 Transformation rules for the `Subsystem` `EClass` and its `EAttributes`. 23

3.2 Transformation rules for the *`Port`* `EClass` and its `EAttributes`. 25

3.3 Transformation rules for the `Signal` `EClass` and its `EAttributes`. 25

6.1 Subsystem names and corresponding function names for the quadrotor example. 85

6.2 Case study configurations for the design exploration. 85

6.3 Table of all computed mappings. 88

CHAPTER 1

Introduction

1.1. Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) integrate computation, networking, and physical processes. The physical process, or plant, is the physical part of the CPS. Embedded computers and networks are the cyber part. The plant is the *controlled system* and may include mechanical parts, human operators, etc. The cyber components *monitor and control* the plant.

Robotics is one of the most immediate applications of CPS, with future robots that are expected to move out of structured environments and cooperate extensively with humans in homes, offices, and novel industrial facilities designed for flexible manufacturing. Not only this, but also robotic-assisted surgery, autonomous driving and flight-control of Unmanned Aerial Vehicles (UAVs) are further emerging trends for CPS applications. Besides robotics, CPS have applications in domains such as agriculture, energy, defense, aerospace and building. Rajkumar *et al.* [**RLSS10**] describe CPS as *the next computing revolution* and discuss some technical challanges for their widespread adoption. Lee *et al.* [$\mathbf{L}^+$] provide a interesting conceptual chart that characterizes CPS, their core enabling technologies and applications.

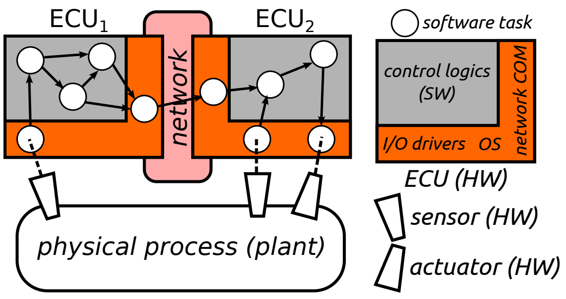
The typical realization of a CPS has the structure sketched in Figure 1.1a. Each embedded computer (Electronic Control Unit, or ECU) is made up of

- a set of hardware elements, including sensors, actuators, processors and network devices;
- an embedded software layer called Basic Software (BSW), including device drivers, operating system (OS), communication stack and middleware services.

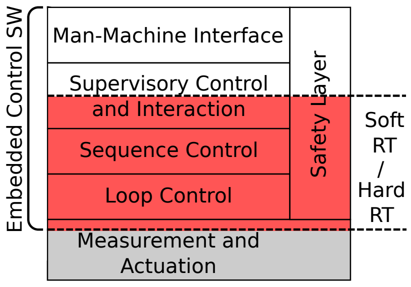
Networked ECUs use sensors to measure the dynamics of plant. Processors execute the control logics; based on the sensors' data (feedback), they collaboratively determine the actions to influence the plant dynamics. Actuators perform these actions on plant.

The software algorithms that implement the control logics are called the *control software*. The control software is organized in layers, as shown in Figure 1.1b. Each software layer requires at least one type of real-time guarantees (as described in classical real-time textbooks, e.g., [**But11**]): *non-real-time*, *soft*, *hard real-time*. Note that most layers do not fall into exactly one category of the real-time guarantees.

In this work, we focus on ***time-sensitive CPS control applications***, for which the correct functioning depends not only upon the logical correctness of control actions, but also upon the time in which these actions are performed. Control software is implemented as a set of tasks executed periodically within timing constraints (deadline or latency) and exchanging messages on networks.



(A) Example structure of a CPS.



(B) Embedded control software stack.

FIGURE 1.1. CPS: typical realization and real-time guarantees of control software stack.

Broenink *et al.* [**BN12**] give a clear characterization of real-time embedded control software stack, which is summarized as follows. Time-sensitive applications/systems are operated by software at *supervisory*, *sequence* and *loop control* level. The *loop control* is at the lowest level, and is the software part that commands actuators. It is usually classified as hard real-time. The *sequence control* layer is a “task-level” controller that computes the set points for the loop controllers. Sequence control runs at a lower frequency than loop control. It can be classified either as hard or soft real-time, depending on application requirements. The *supervisory control* layer is a “strategy” controller. It instructs the sequence controller with the next task. Its calculations may take considerable amount of time and is typically classified as soft real-time.

For a robotic arm, an example of loop control is the (digital) Proportional-Integral-Derivative[1] (PID) controller to follow voltage/current setpoints for motor driving. Inverse dynamics or other PID controllers compute the setpoints for the lower-level PIDs from desired motion trajectories. These are examples of sequence controllers. If the arm is in close interaction with a human operator, reaction strategies upon unexpected collision or active control of arm’s mechanical compliance will also be included in the computations of setpoints. The desired motion trajectories are computed by supervisory controllers performing, e.g., visual tracking, obstacle detection and collision avoidance tasks.

1.2. Current Design Workflows

V-Shaped Lifecycle

The traditional real-time embedded system development process follows the standard *V-shaped lifecycle*. V-cycle splits the product development process into a design and an integration phase. Instead of moving down in a linear way, the process steps are bent upwards after the implementation. Each phase of the development life cycle has its associated phase of testing.

[1]`https://en.wikipedia.org/wiki/PID_controller`

While the way the V-cycle is actually realized varies significantly among practitioners in embedded systems, a number of limitations are common to all realizations. First, a very small amount of Verification and Validation (V&V) is performed that is supported by formal methods, model checking or other similar techniques. Timing verification starts after implementation and integration, with all the negative consequences this entails (issues due to timing are difficult to detect and even more difficult and expensive to fix). Second, the transition between the different stages requires careful manual inspection and cross-checking, and this is frequently error-prone. Third, it produces inefficient testing methodologies.

Multiple-V-Shaped Lifecycle

The *multiple-V-shaped lifecycle* addresses the above issues. It is developed in a sequence of three consecutive V-shaped development cycles (model, prototypes, and final product). The first V covers the definition and simulation of the overall system functionality. Software-in-the-Loop simulation is the primary methodology applied, and implementation aspects (including the time-triggered nature of the application) are not considered. The second V is characterized by rapid prototyping based on Hardware-in-the-Loop simulation. This phase covers the mapping of application tasks to computer nodes and the determination of among messages nodes. The third V addresses the system development for the final target hardware.

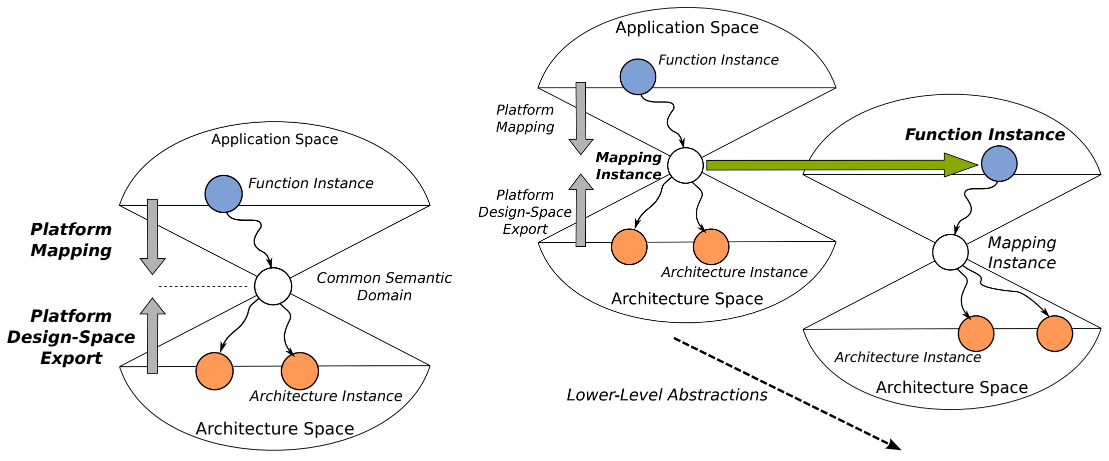
Gaps among V-cycles are the major drawbacks of multiple-V-cycle. There is a gap between the first and the second V: a distributed control application running stable at the first V might experience excessive delay due to message passing between computer nodes and fail (i.e., provide unacceptable Quality-of-Service or even exhibit unstable behavior) at the second V. There is a gap between the second and the third V: deadlines met by the oversized prototypical hardware (second V) might not be met on the target (third V).

1.3. Platform-Based Design (PBD)

Platform-Based Design (PBD) originates from Electronic Design Automation (EDA) industry, where it is in use since some years. In the embedded systems industry, it has been promoted and advocated by A. Sangiovanni-Vincentelli [**SV02**]. Further literature references on this subject include [**SSVDB$^+$05**, **B$^+$06**, **DNSV10**]. The application of PBD to the development of complex CPS is introduced in [**SV07**] and recently discussed in [**SVS14**].

PBD introduces the concept of platform for virtual exploration in which some abstraction of the execution infrastructure is used in the earlier phases of the design flow in support of the exploration. In fact, PBD enables a *multi-level virtual exploration.* The rationale behind this design approach is that there is no reason to require that all parts of a system be explored simultaneously with the same level of granularity. The PBD cycle is a sequence of tool-assisted stepwise refinements that go from the initial specification towards the final implementation using models of platform at various level of abstraction.

Platforms are libraries of model components representing the design space that can be explored. For the design of CPS, libraries contain (at different levels of abstractions):



(A) Platforms. (B) The PBD process.

FIGURE 1.2. The PBD concept (reproduced from [**DNSV10**]).

- models of computational components that carry out the appropriate system functionality;
- models of architectural elements that represent the execution medium, such as microprocessors, memories, networks, sensors, actuators;
- models of execution software layers, that abstract the architectural elements to high level interfaces (APIs) that computational components can use.

Each element has a characterization in terms of performance parameters together with the functionality it can support. The selection of a particular collection of library components whose parameters are set defines a *platform instance*, i.e., a potential design solution at the level of abstraction the platform represents.

For every abstraction level, there is a set of methods used to map the upper level platform into an instance of the lower platform and propagate constraints (top-down process), and a set of methods used to estimate performances of lower level abstractions and to build (or refine) an upper level platform (bottom-up process). PBD is meet-in-the-middle process, as it can be seen as the combination of these two efforts (top-down and bottom-up), as shown in Figure 1.2a. The Figure shows that if designers are given a system platform, then several applications can be mapped into it and the parameters obtained by the design space export can be used to estimate the performance of the application onto the platform of choice. By the same token, if the application space is known, then the platform instance could be optimized according to the needs of the application space.

The mapping can be performed to optimize cost, timing constraints, energy consumption, reliability, etc. Figure 1.2b visualizes the refinement process. The mapped functionality of the system to be designed becomes the "function" at the lower level of the refinement. Another platform is then considered side-by-side with the mapped instance. This process is applied at all levels of abstraction, and the process is iterated until all the components parameters are fully instantiated in their final form.

One important characteristic that makes PBD particularly suitable for the design of complex CPS is that it accounts for the interplay of top-down constraint propagation and bottom-up performance estimation. Another key benefit of adopting the PBD methodology is the elimination of costly design iterations (short time-to-market cycles), because PBD is a structured methodology that limits the space of exploration, and fosters design re-use of hardware and software at all abstraction levels.

1.4. Research Objectives

The complexity of CPS poses significant challenges for their design and verification. This thesis aims to contribute to the ongoing discussion on how to deal with this complexity by applying the methodological framework of Platform-Based Design (PBD) to the development and deployment of complex, time-sensitive control applications for CPS.

The PBD approach is completely general. It fosters a virtual exploration of the system design space on multiple levels and enables the specification of multiple goals for the exploration process. To put this work into a context that otherwise would be intractable in the most general case, we focus on the ***three levels*** that are of most immediate interest in the majority of instances of CPS design. These levels are the functional and the architecture platform layers, represented as the two cones in Figure 1.3, and the mapping layer, represented as the vertex of the two cones in the same figure. The mapping defines a software-architecture layer of tasks and communication resources that define the implementation of the (control) functionality on top of the architecture platform. The importance of these platform levels derives from the fact that most of the critical design choices are taken in the early stages of the design, and misconceptions in these stages produce costly and time-consuming re-design cycles. With a PBD approach considering those levels, fair exploration of design space can be performed, since crucial aspects such as functional specifications (of control) and their implementation characteristics are both handled formally and early in the design process.

We assume that functionality is modeled using ***Synchronous-Reactive (SR) models***, that dominate the market of embedded control application design and on which widely used products like Simulink and SCADE (commercial) and Scicos (open-source) are based.

Finally, the gray arrow in Figure 1.3 indicates that we restrict our focus to ***timing issues*** as the primary design concern.

The model-driven approach of PBD is extremely flexible. It can serve as the backbone for the definition of a design flow where models from different (standard) tools are shared and the integration effort is reduced. This is of paramount importance, especially in modern CPS, where applications are designed by teams of engineers with different specialties (control, software, firmware and hardware) which work together but use different modeling and development tools.

As model-driven approach that complies to principles of PBD, the design process proposed in this work must support the transition from the functional model directly to the code implementation. A fundamental part of this problem is to guarantee that the generated implementation

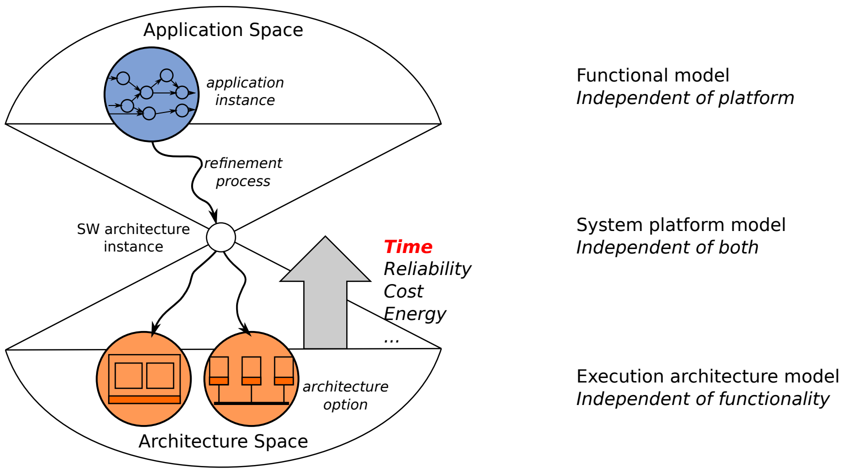


FIGURE 1.3. The levels of platform abstractions that are of concern in this thesis.

preserves the semantics of functional model. Designers must understand under what conditions this may actually happen, and they must also realize the implications of an incorrect implementation. If a semantics-preserving implementation of functionality is not achievable, can designers define an implementation in which delays are deterministic and added to the model? In general, the proposed design process must be supported by tools that enable designers to explore tradeoffs between additional delays, feasibility and control performances.

The definition of such a design process, with the corresponding models, methods and tools is a challenging task, and it was subject to ongoing work throughout the entire thesis. The following introduces the research issues defined as the basic building blocks towards the realization of a supporting framework for the design of complex, time-sensitive control applications for CPS.

RI.1. Models and abstractions for the verification of implementation properties

A prerequisite for adopting the PBD approach is the definition of a common semantic domain where the platform models and the mapping process (Figure 1.3) can be represented formally. Models and abstractions must be isolated from lower-level details but, at the same time, must provide enough information to allow accurate prediction of the properties of the implementation.

Because our primary design concern is on timing issues, platform models at the three levels (control functions, architecture and mapping) must formalize attributes related to time. The model of the functions must be complemented by the formalization of constraints such as end-to-end deadlines. The architecture and the mapping models must formalize the properties of computation and communication resources that have an impact on the timed execution of control functions. For the architecture model, this requires a detailed representation of the basic software (BSW) and of the execution hardware (HW). For the mapping model, it requires the description of activation policies and priorities of tasks and messages, and the worst-case blocking time on shared communication resources.

The formalization of time-related attributes in platform models enables designers to devise system properties such as schedulability and average-case performance, and is at the core of the design of time-sensitive control applications for complex CPS. This research issue is investigated in Chapter 3.

RI.2. Semantics-preserving application deployments

In modern CPS, system properties of control functions (e.g., stability at steady-state, overshoot and settling time during the transient phase) are verified on a model of functional components by simulation or model checking, or other formal means. These properties remain valid after the implementation provided that the refinement of functional model into executable code be performed in such a way that the original semantics is preserved. This is especially important in safety-critical applications, where software certification plays a key role.

When the functional model is a synchronous model (as we assume in this work), available commercial code generation solutions produce code implementations for a single-core execution, and under restrictive assumptions about the scheduling policy. Novel tools are needed that support the code-generation phase in a PBD flow for, at least, multi-core systems. This research issue is investigated in Chapter 4.

RI.3. Simulation tools to predict the system-level timing behavior

When the full preservation of functional model semantics cannot be guaranteed by the implementation, designers may resort to simulation to gather important information about the estimated impact of time delays on the control performances.

It is worth to make the clear statement on this scenario, that simulation alone is not sufficient to achieve software certification of implementation. But there exist numerous time-critical applications in many application domains of CPS that do not need to undergo a rigorous certification process, and for which predicting the system-level timing behavior (latencies and jitter) is a crucial competitive advantage for designers.

Tools are needed that enable such a kind of analysis when the functional model is a synchronous model. Moreover, these tools must comply to the key principle of PBD, i.e., the separation of concerns between control functions and architecture. Separation of concerns requires that model artifacts representing the software architecture model be added to the functional model as annotations through automatic transformations. This research issue is investigated in Chapter 5.

RI.4. Automatic software architecture configuration

The mapping of the functional model into the execution platform is a crucial step in the PBD process. In fact, several possible options exist for the definition of software architecture (mapping) once the control functionality and the architecture platform are defined. In distributed architectures, the design of the software architecture is a very complex task that may require several iterations and is often delegated to the most experienced designer.

The procedure for (software) architecture selection and evaluation is today a "what-if" iterative process [**SVDN07**]. First, the designer provides an initial set of architecture options. Then, the designer uses a set of metrics to evaluate if the architecture options fit to the exploration goals. If the designer is not satisfied with the result, a new set of candidate architectures is evaluated. The iterative process continues until a solution is obtained.

Because the search space is extremely large in most cases, it is very likely that designers have to settle for solutions not only not optimal, but possibly far from optimality. To improve the current situation, automated tools should provide guidance in the definition of the optimal configuration of the software architecture when evaluating an execution platform option, and in the analysis of the results. This research issue is investigated in Chapter 6.

1.5. Thesis Outline

Each of the core Chapters 3–6 is based on peer-reviewed conference or workshop papers. These Chapters do not include discussions of related work, because all the literature review relevant to the research objectives of this thesis is discussed in Chapter 2.

Chapter 3 defines formally the process flow and introduces models and abstractions for the three levels of platform, namely, functional (synchronous models), architecture and mapping. This chapter lays the groundwork for successive Chapters, that present tools for timing-analysis and code-generation which operate on these platform models.

Chapter 4 introduces models and a formal process (code-generation) enabling the implementation of (robot) control applications with the preservation of the SR flows on single-/multi-core computing architectures. The target of the code-generation process is the real-time capable, widely used robotic middleware Orocos-RTT [**Bru**].

Chapter 5 describes models and automated tools to integrate the simulated execution of system scheduling (of real-time tasks and network messages) with control simulation, allowing for the analysis of the impact of computation and communication delays on control performance. The attitude control of a simulated Unmanned Aerial Vehicle (UAV) is used as case study.

Chapter 6 presents a simulation-driven optimization process for the automated synthesis of the software system architecture. The process aims at exploring the interplay between the control performance and the real-time behavior under relaxed constraints (e.g., task deadlines), and at evaluating the quality of different software implementations. The method is applied to a UAV case study.

Chapter 7 concludes the thesis, by summarizing contributions, impact and limitations of this work, as well as suggesting opportunities for future research.

CHAPTER 2

Methods and Tools for System-Level Modeling and Design

2.1. Introduction

This Chapter provides a review of existing methods and tools related to the research objectives formulated in Section 1.4. We introduce the methodologies and the supporting modeling languages and simulation tools that are today in use in the industrial domain or explored in the academic-research domain. We discuss their strengths and weaknesses and identify the basis for the proposed system-level design flow described in the next Chapter.

2.2. Model-Based Design (MBD)

The Model-Based Design (MBD) approach enables behavioral modeling based on a mathematical formalism and executable semantics. In a mathematical-based language, the Model of Computation (MoC) describes the semantics of computation and communication among model elements, and assumes paramount importance. A wide variety of formalisms exist, as reported in [**LSV$^+$98**].

In the domain of CPS control applications development, MBD is the reference approach for the analysis of the system, its verification by simulation, the documentation of the design and the automatic generation of a code implementation. Functional models are based on a Synchronous-Reactive (SR) execution paradigm. Examples of available commercial tools are Simulink [**Theb**] (which is the *de-facto standard*), SCADE [**Est**] and LabVIEW [**Nat**]. Open source and research tools include Scicos [**INR**]/Xcos [**Sci**] and Ptolemy [**EJL$^+$03**] (which supports multiple MoCs). These tools are feature-rich and allow the modeling of continuous-, discrete-time, and also hybrid systems. The control functionality is typically defined using dataflow formalism (possibly in combination with an extended finite-state machine formalism). The controls are simulated against a model of the controlled system (plant), and then validated. The development of complex functionality may require several iterations, but the availability of a virtual simulation environment allows to speedup the verification of each solution (as opposed to traditional testing) and to verify scenarios that would be impractical or even impossible to setup on the actual system.

A common limitation to most commercial modeling and simulation tools is that the functionality is represented in abstract terms, that is, independent from the execution platform that will support the execution of the controls as a set of software tasks and network messages. A sound development methodology should ensure that the computation and communication delays that occur when the system is implemented on a (possibly distributed) execution architecture with finitely available resources preserve the execution assumptions that are part of the simulated (and verified) model.

Sometimes, a semantics preserving implementation is not feasible for the selected execution platform, and the only option is to account for the computation and communication delays in the simulation.

Verifying the preservation of the SR semantics or estimating the delays requires the construction of a model of the execution platform and providing an assumption on how the functionality will be implemented on the computation resources.

In addition, in a sound model-based flow, the software implementation of the model should be automatically generated by a model compiler. This is desirable not only for improving efficiency, but for guaranteeing that no errors are introduced in a manual coding stage. Unfortunately, current code generators only produce a code implementation for a single core execution, and under restrictive assumptions about the scheduling policy.

2.3. Model-Driven Engineering (MDE)

Model Driven Engineering (MDE) is a software development approach that promotes the use of models and their transformations in the software development process. The central idea in MDE is the definition of domain-specific models (called *meta-models*) that capture the aspects relevant to a particular domain. In other words, domain-specific models represent the knowledge and activities that govern a particular application domain. Concrete models conformant to the meta-model can then be analyzed, validated, transformed and executed.

MDE is a modeling *paradigm.* One of most relevant standardization efforts of it is the Model Driven Architecture (MDA) initiative [**(OMb**] by the Object Management Group (OMG). MDA prescribes a design process in three stages. First, a Platform-Independent Model (PIM) defines the system functionality independently of the software platform it will execute onto. Next, the PIM is transformed to the Platform-Specific Model (PSM) by means of a Platform-Definition Model (PDM). Finally, the PSM is transformed in a Platform-Specific Implementation (PSI). The first transformation is *Model-to-Model* (M2M): it transforms the PIM (source model) into a model of the software execution platform (target model). The second transformation is *Model-to-Text* (M2T), because it transforms the PSM into final code implementation (PSI).

OMG defines several other standards to effectively support MDA. The Meta-Object Facility [**(OMg**] (MOF) specification describes the meta-meta modeling language and the rules that specify meta-models. The Query/View/Transformation [**(OMe**] (QVT) standard defines M2M transformation languages, which operate on models conform to MOF. A key component of QVT is the Object Constraint Language [**(OMd**], (OCL) that allows the specification of constraints on models. M2T transformations are standardized separately as MOF Model to Text Transformation Language [**(OMc**] (MOFM2T).

The Unified Modeling Language [**(OMh**] (UML) defines diagram types for modeling a number of software aspects, including *behavior.* However, UML lacks of a rigid semantic formalization, and only recently OMG has released a behavioral semantics specification with the foundational UML [**(OMf**] (fUML) and Action Semantics language [**(OMa**] (ALF).

Within MDE paradigm, we can distinguish two modeling approaches: profiling and meta-modeling. *Profiling* is the mechanism standardized by the OMG that allows adaptation of the

UML meta-model for a specific application domain. *Stereotypes* and their attributes enable the representation of concepts for that specific domain. Today, many profiles exist for numerous domains. The System Modeling Language [**Obj12**] (SysML) and the Modeling and Analysis of Real-Time and Embedded Systems [**(OM11**] (MARTE) profile are relevant in the context of this work.

SysML is a general-purpose modeling language for systems engineering applications. SysML offers concrete advantages over UML for tasks like capturing system requirements and specifying quantitative constraint on the system. However, its behavioral semantics is not completely and formally specified (like for UML).

MARTE introduces UML extensions describing a variety of non-functional properties, and aims to provide support for specification, design, and verification/validation of real time and embedded systems.

Meta-modeling is the approach of defining a new meta-model from scratch. As of today, the Eclipse Modeling Framework [**Ecla**] (EMF) is the most widespread framework that supports meta-modeling. EMF enables the definition of custom metamodels using its Ecore meta-modeling language, which is the de-facto reference implementation of (a subset of) OMG's MOF[1].

2.4. Frameworks for Heterogeneous-Model Integration

In agreement with the principle of separation of the functional and architecture design concerns, and with respect to the subject of heterogeneous models integration, several approaches, methods and tools have been proposed. GME [**KML$^+$04**] and MetroII [**DDM$^+$07**] propose the use of a general meta-model as an intermediate target for the model integration. However, they do not provide simulation capabilities or a code generation path for distributed implementation based on open source tools and open standards.

Raghav et al. [**RGR$^+$10**] and Hugues et al. [**HZPK08**] proposed two similar MDA methods for describing the functional behavior according to a reference architecture and then comparing the deployed system with respect to the reference to check whether the performance (delay) target is guaranteed.

GeneAuto [**Gen**], ProjectP [**Pro**], the Rubus Component Model [**HMTN$^+$08**] and AADL [**AAD**] put emphasis on the modeling of task sets and their interactions and the code generation infrastructure, without including simulation capabilities or an explicit formal metamodel for the internal behavior of tasks.

2.5. System-Level Approaches in Automotive and Robotics

Automotive (AUTOSAR)

AUTOSAR [**AUT**] is a development partnership of automotive OEMs (Original Equipment Manufacturers) and suppliers. The partnership aims to develop an open industry standard for automotive software architectures. The standard covers all software levels, from the BSW to the

[1]`https://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-100007.html`

specification of platform-independent application-level components. A meta-model formalizes all the definitions provided by the standard.

AUTOSAR is perhaps the most successful application of MDA paradigm in the industry. The AUTOSAR standard defines a virtual integration environment for the software components and a separate model for the distributed execution architecture, later merged in a deployment stage supported by tools. However, the current implementation of the standard lacks the support of a formal MoC, and hence is not suitable for the early-stage control validation. Also due to this, the integration of AUTOSAR with MoC-based executable languages (e.g., Simulink) presents several difficulties.

Robotics

In robotics, the most common design paradigm for application development is the component-based SW engineering. Frameworks such as ROS [**Ope**], OpenRTM-aist [**AIS**] and Orocos-RTT [**Bru**] act as middlewares, providing abstractions to encapsulate active control threads and communications among them. Orocos-RTT is specifically oriented toward programming and executing component-based applications on top of *Real-Time* Operating Systems (RTOSes) and relies on *lock-free* communication to guarantee a deterministic execution time for all in-process inter-component data exchange.

However, this situation is changing and MDE approaches are becoming increasingly popular. In the last few years, several MDE Integrated Development Environments (IDEs) and Domain-Specific Languages (DSLs) have been made available. BRIDE is an IDE based on Eclipse developed in the BRICS project [**web**]. It targets the automatic generation of platform-specific code for component-based frameworks from a graphical (abstract) model of the system architecture and its SW components (the BRICS Component Model [**BKH$^+$13**]). BRIDE uses model-to-model (M2M) transformations to generate framework-specific code for the communication, configuration, composition and coordination of ROS and Orocos-RTT components. The declarative description of robotics architectures and SW deployment using a DSL is described in [**HGS$^+$13**] with a hierarchy of architectural concepts for HW and SW, inspired by AADL [**AAD**]. However, the properties of HW and SW that define the timing behavior of components are not included. The SmartSoftMDSD toolchain [**SSBK10**] supports non-functional properties for design-time real-time schedulability analysis. The framework allows the graphical modeling of applications and provides M2M transformations to construct a platform-specific model for schedulability analysis using Cheddar [**SLNM04**].

Some IDE provide DSLs for the algorithmic description of behaviors. V^{3}CMM [**AVCO$^+$10**] is a modeling language that provides a simplified version of UML activity diagrams, to model the sequential flow of execution within components. RobotML [**DKS$^+$12**] is a DSL aiming at the design of robotic applications and their deployment to multiple target execution platforms (and simulators). It uses a specialization of UML state machines for the modeling of the behavior of generated component implementations. RobotML enables (simplistic) modeling of platform-specific non-functional properties of SW components, that are used to create models for third-party real-time schedulability analyzers. Hence, it suffers the same drawbacks of the SmartSoftMDSD toolchain.

Virtual Path [**NJH09**] is a HW-SW co-Design method that includes Simulink in the development flow, to create executable models representing the controls. In [**WNBG12**], Wätzoldt *et al.* adapt the automotive toolchain to the development of robotic systems. The design methodology uses Simulink for the simulation of robot functionalities, and Embedded Coder for the generation of the implementation. AUTOSAR models and tools (e.g., SystemDesk [**dSP**]) are used to combine hard and soft real-time tasks in a system view and analyze the scheduling feasibility.

2.6. Simulation of Platform's Delay Effects

Models can capture the architecture-level details and enable the study of system's timing behavior by means of timing analysis techniques (e.g., using MARTE). Another approach is simulation. Simulators can provide support for the evaluation of effects that computation and communication delays have on the performance of the system.

Network simulators can aid in the evaluation of communication delays (due to, e.g., message transmission). A large variety of simulation tools are available from the industry and the academia. Most of them are geared towards the use in a specific domain. NS-3 (Network Simulator) [**NS-**] and OMNeT++ [**OMNa**] are freely available discrete-event computer network simulators. They support several communication protocols and are extensible for the inclusion of new ones. In [**MMT**$^+$] a simulation environment for CAN-Ethernet networks is presented as example of extension to OMNeT++.

Real-time scheduling simulators support the evaluation of computation delays (finite execution times, scheduling delays, etc). A huge number of projects target the evaluation of scheduling policies and the analysis of task implementations (more than 6 million hits when searching the keywords *Real-Time Scheduling Simulator* in Google). A necessarily incomplete list includes Yartiss [**CFM**$^+$**12**], Storm [**UDT10**], ARTISST [**DP02**], Cheddar [**SLNM04**], Schesim [**MSHT12**], Stress [**ABRW94**].

Simulators must be used in conjunction with MBD tools that support the definition of control applications, to effectively support the designer in the architecture evaluation process. The following lists the possible options for use with Simulink.

Options for Simulink

Co-Simulation with Third-Party Programs

A number of research works address the problem of combined execution of Simulink together with an external program for network simulation. Relevant works on this topic include [**HA04**, **Kac10**]. They share the idea of using sockets (TCP/UDP) to establish a bidirectional communications between the two simulators. This requires the implementation of special coupling components, that slow down the simulation and realize the coordination between the simulators in ad-hoc fashion. Furthermore, these implementations are tied to the specific network simulator and are difficult to interface with others.

Co-Simulation with Ad-Hoc Implementations of Scheduling and Network Simulators: The TrueTime Toolbox

In Simulink, a possible solution for the simulation of platform's computation and communication delays is provided by the TrueTime toolbox [**CHL$^+$03**]. TrueTime is already used by many research groups worldwide to study the (simulated) impact of lateness and deadline misses on controls. Relevant literature on this subject includes [**ÅW11**] and [**CVMC11**].

TrueTime enables the simulation of control functions considering their software task and message implementations, including scheduling and resource management policies. It provides two special-purpose Simulink blocks representing a model of multi-tasking real-time kernels and of a network, respectively. The kernel model supports many popular scheduling policies for single-core systems, such as Earliest Deadline First (EDF) and Fixed Priority (FP), including Rate Monotonic (RM). Presently, multi-core architectures are not supported. For what concerns the network standards, TrueTime supports Ethernet, CAN and FlexRay, and wireless network standards such as 802.11b WLAN and 802.15.4 ZigBee.

In TrueTime, the kernel models in fact a *computer node* together with A/D and D/A converters, external interrupt inputs and network interfaces. The corresponding block is configured via an initialization script (usually written in Matlab code), where a specific API is used by the designer to create tasks, timers and interrupt handlers and define the scheduling policy and the communication resources. The model of task code is represented by *code functions* that are written in either Matlab or C++ code. A TrueTime developer has two options: hand-code the control logic and lose availability of Simulink (control) toolboxes, or call external discrete-time Simulink models from within the code functions using a mechanism based on the MATLAB built-in operator `sim()` with several limitations. First, signal-generator blocks that use the simulation time and blocks for which it is not possible to specify the sample rate (e.g., the Discrete Derivative block) cannot be used. Second, data connections among Simulink models need to be implemented in code using a purposely offered API and the application of a TrueTime Scheduler to an already existing Simulink model of controls requires substantially rewriting, mixing the controller functionality, the model of the task set, the scheduler, and the physical execution platform. Finally, because of the monolithic architecture and the number of code artifacts that are needed for system configuration (e.g., initialization script and code functions), the current TrueTime implementation is hardly compatible with an automatic model generation and a M2M transformation flow.

Native Simulink Discrete-Event Simulation: SimEvents

SimEvents [**Thea**] is a commercial toolbox developed by The MathWorks, providing a discrete-event simulation engine and component library for Simulink. It enables event-driven communication modeling between Simulink components to analyze and optimize end-to-end latencies, throughput, packet loss, and other performance characteristics. SimEvents is modular and its component library comes with many blocks that allows the designer to customize processing delays, prioritization, and other operations, to represent systems that range from manufacturing processes, to hardware architectures and sensor/communication networks.

Because of its generality though, SimEvents does not provide an explicit model of task, real-time scheduler, network protocol or hardware component. They need to be built as libraries using blocks representing priority queues and servers.

2.7. Discussion and Conclusions

Concepts and methodologies reviewed in Sections 2.2 and 2.3 can be summarized as follows. On one side, SR languages (e.g., Simulink) allow for the simulation of controller-plant interactions and behavioral code generation (of control), but offer no support for the representation of complex system architectural aspects and execution platforms. On the other hand, MDA languages are very good at representing architectural aspects, can be easily extended and provide mechanisms to transform models expressed in a language into another; but their behavioral semantics is weak (not completely and formally specified) and they are not suitable for modeling continuous-time systems. Their strengths and weaknesses are complementary. Therefore, *MBD and MDA approaches are good choices to form the backbone of the PBD design flow.*

Within MDA, MARTE is today the best option for the definition of distributed embedded systems with real-time constraints, although with some limitations. MARTE purposely lacks the complete specification of a number of HW and SW concepts, that is left up to model-library designers. However, when there is the intent of processing a model annotated with MARTE stereotypes to analyze it or to generate code/models from it, the introduction of new language constructs becomes necessary [**SG14**]. Moreover, MARTE defines a number of stereotypes to represent BSW concepts that are mostly cumbersome, come with a large number of properties and are quite difficult to be mastered by the system designer. As a result, this work requires *custom taxonomies of stereotypes extending MARTE.*

Despite much research, such an *integrated MBD/MDA framework, based on SR and SysML/-MARTE models, is not ready available* (Section 2.4). Among possible alternatives, the reviewed works that provide a clear separation of the functional and platform models (GME, MetroII) typically do not provide model-based simulation capabilities or a code generation path for distributed implementation based on open source tools and open standards. AUTOSAR (Section 2.5) does not have any feature for modeling the behavior of the functions. Therefore, an exetrnal tool or the actual code is needed for functional modeling. The same consideration holds for the majority of the frameworks today available for model-driven development in robotics. Those that provide support for behavioral modeling are based on UML and, consequently, model execution and simulation are tool-specific. Nearly all lack any kind of support for timing analysis. Others (e.g., SmartSoftMDSD) portray an approximate (and sometimes inaccurate) view of the schedulability problem as a binary decision process ("ok"/"not ok").

Finally, tools reviewed in Section 2.6 do not represent a viable solution to the problem of simulating computation and communication delays in SR models. This demands *the development of a novel, modular and extensible co-simulation framework to be integrated in the proposed model-driven flow.*

CHAPTER 3

Design-Process Flow and Platform Meta-Models

3.1. Introduction

As model-driven approach that complies to principles of PBD, the design process proposed in this work integrates executable, synchronous models of control with structural models of the execution platform in a sequence of stepwise refinements from the initial specification towards the final implementation. The mapping of control functions to the platform is performed through an intermediate layer, in which functions are implemented by tasks and communication between components maps into local communication (internal or between tasks) and network messages. The top side of Figure 3.1) represents (conceptually) the process of mapping the functions model onto a model of the execution architecture through the intermediate SW architecture model.

Two major application scenarios are considered in this thesis, namely those of design of *safety-critical* and *performance-sensitive* systems. In both the scenarios, the use of models is a consolidated practice to improve the quality of the system and to speed-up the development process (albeit with several limitations). The link between the proposed PBD process and the application scenarios is visualized in Figure 3.1) (bottom side). The mapping model is represented as the *foundation* part in the successive refinement view of PBD for both the scenarios, since

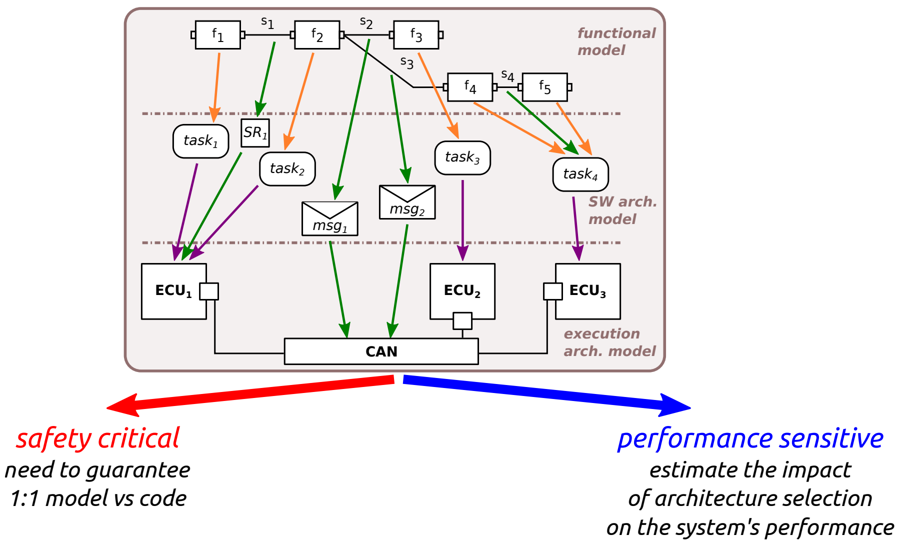
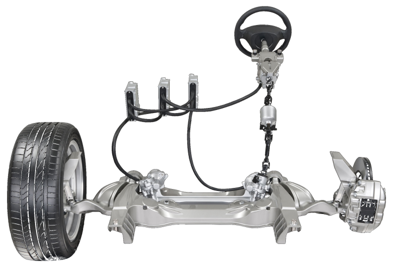


FIGURE 3.1. Platform models, mapping process and application scenarios for the proposed design flow.


(A) Steer-by-Wire[1]

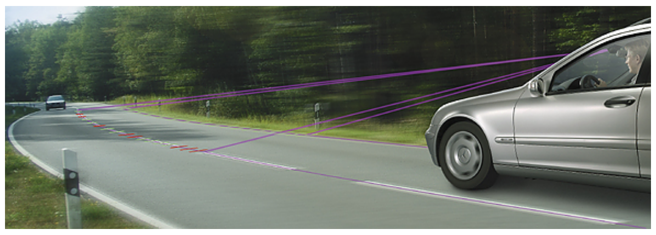

(B) Lane Departure Warning[2]

FIGURE 3.2. Examples of safety-critical and performance-sensitive systems in automotive.

only after mapping is described one can verify whether constraints on non-functional properties of the design are satisfied.

In ***safety-critical*** CPS, failure or malfunction may result in death or serious injury to people, loss of or severe damage to equipment, and/or environmental harm. Examples from the aerospace/automotive and robotics domains include X-by-Wire (e.g., Steer-by-Wire, Figure 3.2a) and physical Human-Robot Interaction (pHRI) systems, respectively.

Safety-critical systems demand precise guarantees that the final implementation will satisfy a number of properties, formally verified during the design phase with the use of model checkers or other formal means. They must pass thorough certification processes to be put in operation. Examples of safety standards are ISO26262 [**Int**], for road vehicles functional safety, and DO-178B [**RTC**], for SW considerations in airborne systems. Therefore, a process addressing their design at system-level must support the transition from the functional model directly to the code implementation in a way that the generated implementation is guaranteed to preserve the semantics of functional model (SR). This includes overcoming the limitations of current commercial code generation solutions, that, for synchronous models, can produce semantics-preserving code implementations only for a single-core execution, and under restrictive assumptions about the scheduling policy.

The other major application scenario for the proposed framework is that of ***performance-sensitive*** systems design. This kind of systems are time-critical, i.e., subject to demanding timing constraints, and often have the potential for very high consequences of failure. In fact, in many cases the distinction between safety-critical and performance-sensitive systems is subtle, and mostly depends on whether the law mandates system designers to show compliance with an applicable (safety) standard. Examples from the automotive domain include Advanced Driver Assistance Systems (ADAS), such as Adaptive Cruise Control (ACC) and Lane Departure Warning (LDW) (Figure 3.2b) systems.

[1]`http://www.caranddriver.com/features/electric-feel-nissan-digitizes-steering-but-the-wheel-remains-feature`

[2]`http://www.continental-automotive.com/www/automotive_de_en/themes/passenger_cars/chassis_safety/adas/`

It is very common that performance-sensitive systems are classified as hard real-time systems. Therefore, their design is carried out according to the classical separation of concerns between the correctness of control functionality and the verification of the time properties of the computations.

In reality, many of these systems (including fuel injection [**But12**]) are tolerant to delays and deadline misses in a way that is different from a simple "safe/not safe" outcome. Therefore, when the full preservation of functional model semantics cannot be guaranteed by the implementation, the evaluation of the impact of computation and communication delays on the performance of controls in the simulation environment (at design time) gains fundamental importance. Satisfactory (simulated) performance gives designers reasonable expectations on the validity and effectiveness of the final SW implementation. On the other hand, performances excessively degraded may trigger design iterations, where the architecture configuration may be modified or the mapping decisions may be changed. When iterations are required on the functional model, a different selection of the execution periods of the functions, or different synchronization and communication solutions may be explored.

The proposed approach enables design space exploration with a fairly accurate prediction of (time and control) properties of the implementation upon condition that platform models incorporate and expose appropriate information. Many factors influence the capability of predicting the timing behavior (latencies and jitter) of the system, including the synchronization between tasks and messages, the interplay that different tasks can have at the RTOS level and the synchronization and queuing policies of the middleware. Ultimately, the timing of end-to-end (control) computations depends on the deployment of the tasks and messages on the target architecture and on the resource management policies.

The identification of precisely defined abstraction layers (platforms), where the refinement processes take place, is the essence of PBD. On one side, exposing too much specialized information from lower-level abstraction layers can result in complex platform models that are difficult to understand and cumbersome to use. On the other side, oversimplified representations of the design space may lead to predictions/abstractions so inaccurate that refinements are misguided and the final implementation is, at the very least, ineffective.

This chapter is entirely focused on the definition of the right models and abstractions for the description of the functional platform specifications, the execution architecture options and the SW architecture implementation.

The rest of the chapter is organized as follows. Section 3.2 defines the process flow, that leverages open standards and technologies. Being conformant to standards, it enables the use of modeling tools with graphical editors, model checkers and processors, and possibly a number of additional tools for supporting the management of models. Sections 3.3, 3.4 and 3.5 cover the core content of the chapter and provide, respectively, the characterization of functional, execution architecture and SW architecture platforms for timing analysis and code-generation. Finally, Section 3.6 summarizes and closes the chapter.

3.2. Process Flow Based on Standard Technologies

For the development of complex CPS, we propose an approach that follows the principles of PBD and benefits from the complementary strenghts of different model-driven approaches such as domain-specific modeling languages, MDE and MBD.

A natural candidate for the functional modeling is the MBD-oriented Simulink/Stateflow synchronous language. However, functionality could also be implemented directly as code. In both cases, an abstract view of the functional model is required. This abstract view accounts for all the information related to the timed events that are relevant for the system, including rate constraints, partial order of execution constraints and any other synchronization constraints.

The MDE-oriented open-source environment Papyrus [**CEA**] is currently the best option for modeling the execution platform. Papyrus is integrated into the Eclipse Modeling Framework (EMF) and provides complete support for OMG's UML and related modeling languages such as SysML and MARTE. However, the standard MARTE profile is not completely adequate for the description of (networked) systems considered in this work (Section 2.7). Therefore, on top of the modeling features offered by the standard SysML meta-model, we provide a domain-specific profile for cyber-physical applications that leverages (and extends) MARTE. The profile defines a concise taxonomy of stereotypes to represent common execution HW in use in CPS and the BSW (including device drivers, middleware classes and RTOS modules) that runs on top of it. The properties of stereotypes are selected carefully. They enable the representation of concepts for timing analysis and code-generation at a sufficient level of abstraction, without requiring deep digging into many lower-level details.

In addition to a dedicated profile for the model of the execution architecture, another profile is created to represent the model of the SW implementation. This model consists of the set of all tasks and messages implementing the system functions and the communication signals. When a functional model is put in correspondence with an execution architecture, the task and message model is produced as the result (either by hand by the designer or as the result of the operation of synthesis tools). The profile extensions allow to define the mapping and evaluate the computation and communication delays, but are also used for the automatic generation of a concurrent task implementation on top of Orocos-RTT. Code generation is performed using the open-source, standard MOFM2T transformation tool Acceleo [**Obe**].

Figure 3.3 visualizes the complete design flow. The Simulink functional model is the starting point, and is partitioned (at some level in the design hierarchy) in a set of subsystems. Subsystems are the unit of execution for the code generation process. Once the simulation results are satisfactory, code is generated for each subsystem, and the designer uses a purposely written model exporter to generate an abstract view of functional model. The abstract view conforms to an Ecore meta-model for SR systems. The Ecore view preserves all the structural properties of the Simulink model, such as the types and interfaces of the subsystems and the connections among them, and also accounts for the information related to the timed execution events, including rate and partial order of execution constraints. Next, a M2M standard-compliant QVTo [**Eclb**] transformation translate the Ecore view of the functional Simulink model into a SysML model

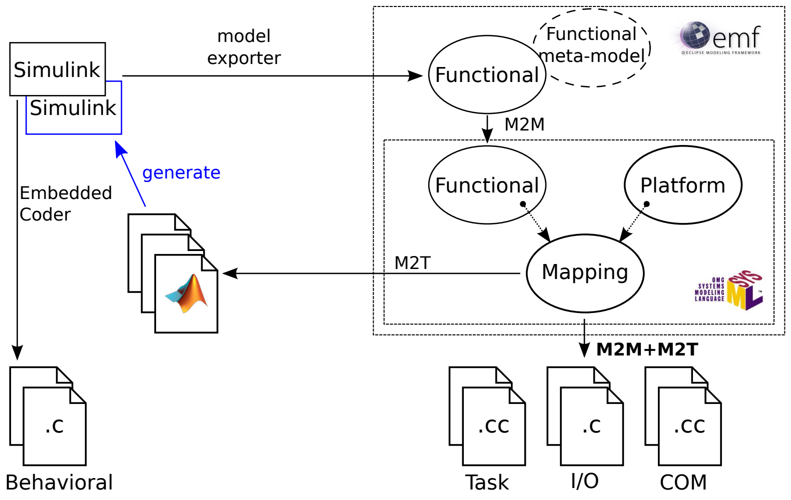


FIGURE 3.3. Heterogeneous model integration and code generation by the framework tools.

in Papyrus (leveraging a profile definition). Here, the SysML functional model is extended with the platform and mapping models.

The mapping model represents the software tasks and messages (local or on the network) that realize the functions. The task model may be constrained in such a way that only flow-preserving implementation of the functional model are allowed.

Finally, M2M and MOFM2T transformations process the mapping model to achieve the goals of (*i*) generating semantics-preserving executables from a functional model of controls; or, (*ii*) exploring the tradeoffs between time delays and control performances. In case a semantics-preserving implementation of the functionality is achievable, an implementation of mapping model is generated that executes the C code from Simulink on top of Orocos-RTT. Otherwise, a set of Matlab scripts containing back-annotation commands are generated that operate on the original Simulink model and adds to it a set of custom blocks (with connections), representing the implementation of the Simulink subsystems of the controller in tasks, executing under the control of a scheduler and exchanging messages onto a network medium (as specified by the mapping model). Designers then start a sequence of refinements of the control logic and/or the HW/SW implementation based on the estimated impact of time delays on the control performances.

3.3. Functional Modeling

3.3.1. Functional Modeling in Simulink and EMF

The functional model is created by importing in EMF a Simulink model that includes the controller part and the model of the plant.

The Simulink model must comply with the restriction that there is a decomposition level in which the controller part consists of a collection of subsystems, in which each subsystem only contains periodic blocks with the same period (each subsystem has a single rate).

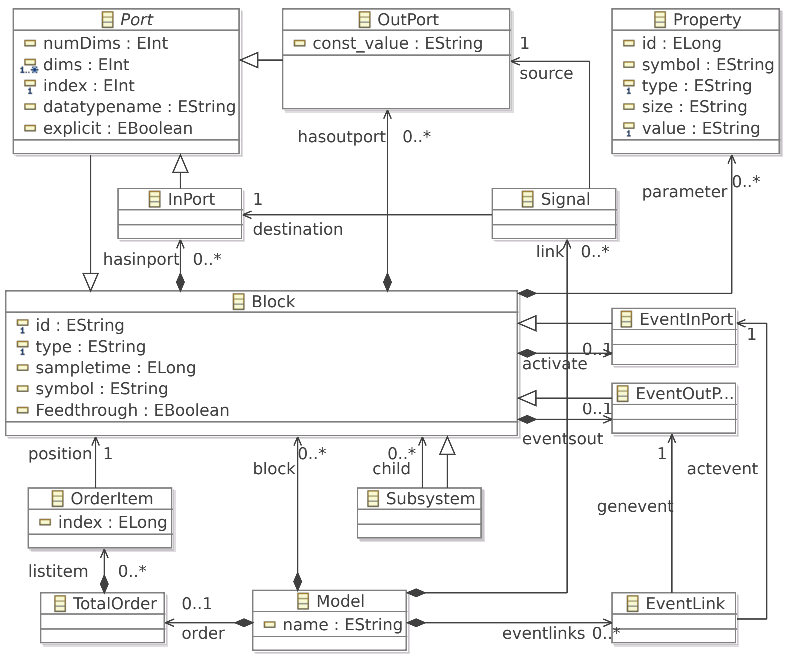


Figure 3.4. The Ecore meta-model for the functional part.

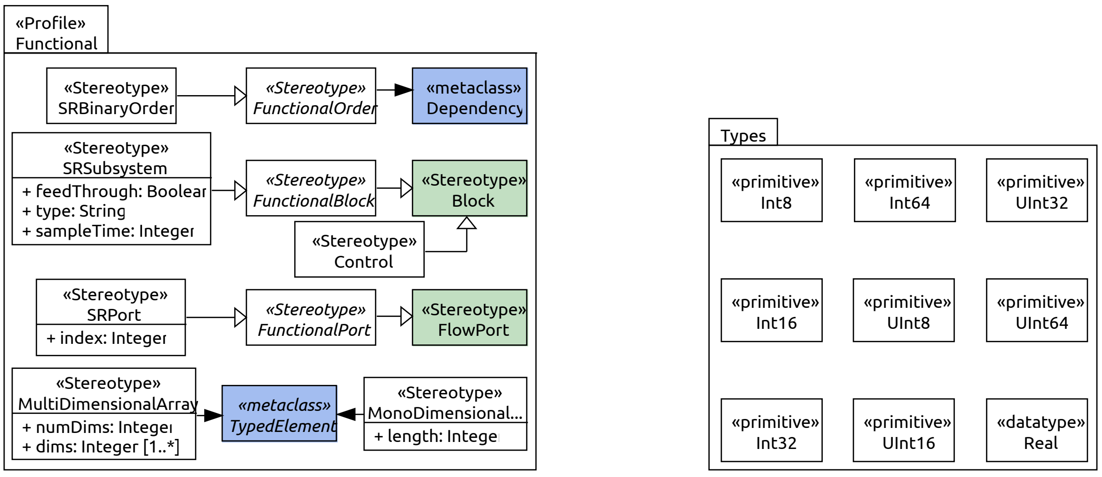
A Matlab script uses the Simulink modeling API (programming interface) to parse the model structure and export an XML view of the controller subsystems. The XML conforms to a schema created in accordance with an Eclipse Ecore metamodel, defined for representing the execution constraints that apply to the Simulink subsystems (Figure 3.4). This metamodel is not too dissimilar from the one proposed in the GeneAuto project (actually, a simplified version of it), but contrary to GeneAuto, it is formally available as an Ecore definition.

3.3.2. M2M Transformation to SysML

After the functional model is imported in the EMF framework, a QVTo M2M transformation translates the model in SysML, according to purposely created profile and type library for SR systems (Figures 3.5a and 3.5b, respectively). The QVTo transformation is structured as a set of mapping operations that are invoked in sequence within the transformation entry point. Figure 3.6 shows the sequence of mapping operation calls, that are discussed in the next subsections.

Mapping operation `Subsystem::toSRSubsystem()`

The transformation rules of this mapping operation are summarized in Table 3.1. All `Subsystem` entities are mapped one-to-one to `SRSubsystem` instances, which extend the SysML concept of `Blocks`. `EAttributes` `type`, `sampletime` and `Feedthrough` are mapped directly to their counterpart properties in `SRSubsystem`. An empty `type` identifies a special kind of



(A) SysML profile. (B) Type library.

FIGURE 3.5. The SysML meta-model for the functional part.

SRSubsystem, as one that carries out some control-related functionality. Therefore, the stereotype Control is added to the SRSubsystem instance. The EAttribute id is mapped to the name property of SRSubsystem's base stereotype SysML::Block.

This mapping performs a nested call to a map operation that manages the transformation of *Port* objects.

Ecore		SysML	
EClass	EAttribute	Stereotype	Property
Subsystem		SRSubsystem	
	type		type
	type (size(type) = 0)	Control	
	sampletime	(SRSubsystem)	sampleTime
	Feedthrough		feedThrough
	id	SysML::Block	name

TABLE 3.1. Transformation rules for the Subsystem EClass and its EAttributes.

Mapping operation Port::toSRPort()

Table 3.2 summarizes the transformation rules of this mapping operation. All the (concrete) instances of *Port* are mapped to SRPorts, which specialize the SysML concept of FlowPorts. The *Port* type determines the direction of the SysML::FlowPort instance in the destination model. InPorts are mapped to SRPorts with direction equal to FlowDirection::_in. The mapped flow direction is FlowDirection::_out for OutPorts.

```
-- transform entry point
main() {

  -- Generate SRSubsystem and SRPort elements
  var setSRSubsystems := src_mdl.block[Subsystem]
                            ->map toSRSubsystem()->asSet();
  dst_mdl.packagedElement += setSRSubsystems;

  -- Add the Control stereotype (SRSubsystems with empty types)
  setSRSubsystems->select(s | s.getValue(sub_stp, "type")
                            .oclAsType(String).size()=0)
                                ->map toControlSubsystem();

  -- Add them to the FunctionalSystem block
  var ost_ptySubsys := setOfSubsystems->map toProperty()
                            ->asOrderedSet();
  blkFunctSystem.ownedAttribute += ost_ptySubsys;

  -- Generate connections
  var setSignals := src_mdl.link
    ->select(s | s.source.container().oclIsKindOf(Subsystem) and
      s.destination.container().oclIsKindOf(Subsystem))
        ->oclAsType(Signal)->asSet();
  var setConnections := setSignals->map toConnector()->asSet();
  blkFunctSystem.ownedConnector += setConnections;

  -- Generating partial order constraints (for SRSubsystems)
  var setAllBinaryOrders : Set(Dependency) := Set{};
  setOfSubsystems->map toSetOfSRBinaryOrders(setSignals)
        ->forEach(setBinaryOrder)
        {
          setAllBinaryOrders += setBinaryOrder
        };
  dst_mdl.packagedElement += setAllBinaryOrders;
}
```

FIGURE 3.6. Sequence of mapping operation calls within the QVTo transformation entry point.

The value of the `EAttribute numDims` determines an additional stereotype for the corresponding `SRPort` in the destination model. If `numDims` > 1, the `SRPort` is further stereotyped as `MultiDimensionalArray`, to indicate that it can handle multi-dimensional signals. In this case, `numDims` and `dims` are directly mapped to the counterpart properties in `MultiDimensionalArray`. If `numDims` $= 1$, the target `SRPort` is stereotyped as `MonoDimensionalArray`, and `dims` is mapped to the signal `length` (the signal size along its single dimension).

A *Port*'s `datatypename` determines the type of the corresponding `SRPort` in the destination model. The mapping operation compares the `datatypename` string value with the names of types in the SysML type library for SR systems. If a correspondence is found, the type is assigned to the `SRPort`; otherwise, the `Real` data type from the `SysMLPrimitiveTypes` package is assigned to the `SRPort`.

Finally, the *Port* `index` is mapped to the `index` property of `SRPort`.

Ecore		SysML	
`EClass`	`EAttribute`	Stereotype	Property
`Port`		`SRPort`	
	`index`		`index`
	`datatypename`		`type`
	`numDims` (`numDims` > 1)	`MultiDimensionalArray`	`numDims`
	`dims` (`numDims` > 1)	`MultiDimensionalArray`	`dims`
	`numDims` (`numDims` $= 1$)	`MonoDimensionalArray`	
	`dims` (`numDims` $= 1$)	`MonoDimensionalArray`	`length`
`InPort`		`SysML::FlowPort`	`direction`
`OutPort`		`SysML::FlowPort`	`direction`

TABLE 3.2. Transformation rules for the *`Port`* `EClass` and its `EAttributes`.

Mapping operation `Signal::toConnector()`

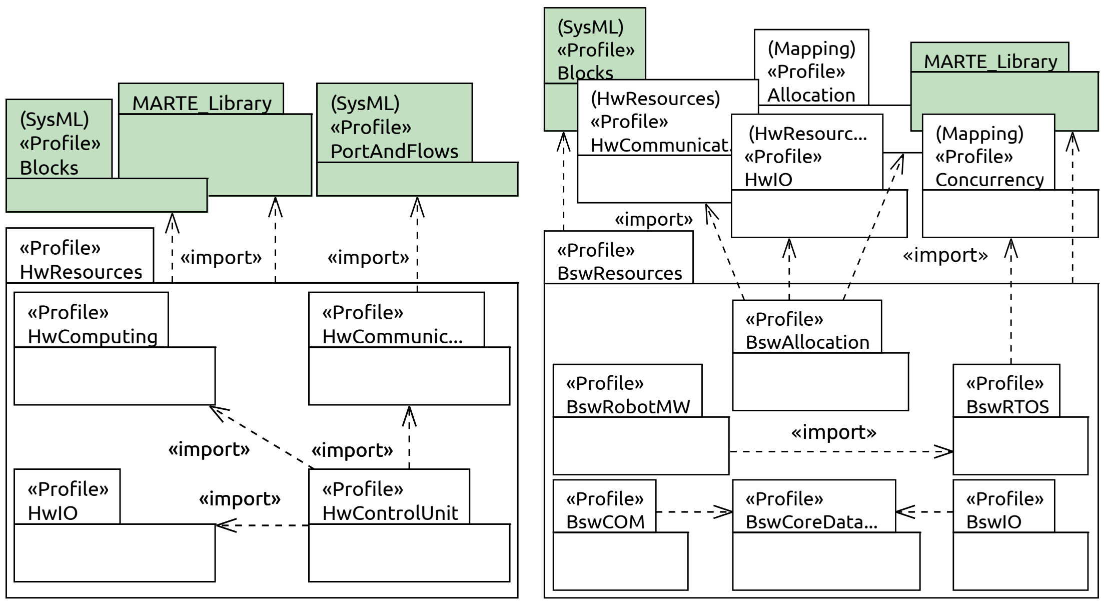
`Signals` in the source model are mapped one-to-one to standard `UML::Connectors` in the destination model. `EAttributes` `source` and `destination` are used to recover the connector's endpoints specifications (`SRPort` and its `SRSubsystem`) from early mapping operations. Table 3.3 summarizes the transformation rules of this mapping operation.

Ecore		SysML	
`EClass`	`EAttribute`	Stereotype	Property
`Signal`		`UML::Connector`	
	`source`	`UML::ConnectorEnd`	
	`destination`	`UML::ConnectorEnd`	

TABLE 3.3. Transformation rules for the `Signal` `EClass` and its `EAttributes`.

Mapping operation `Class::toSetOfSRBinaryOrders()`

The Simulink modeling API does not allow the extraction of the precedence orders among blocks in a convenient way. For this reason, the information on precedence orders of execution between pairs of blocks is determined by the M2M transformation on the basis of the values of `Feedthrough` ports attributes.



(A) Physical platform profile. (B) BSW resources profile.

FIGURE 3.7. SysML profiles for the execution platform with their dependencies.

3.4. Platform Modeling

The execution-platform meta-model defines profiles and stereotypes for the embedded system domain on top of OMG's MARTE profile. Execution-platform profiles are organized in two packages, namely `HwResources` and `BswResources`, respectively shown in Figures 3.7a and 3.7b.

For the sake of better readability, the implementation details of packages are omitted and we keep the discussion to a more abstract higher-level. An exception is made for the `BswRTOS` package (`BswResources`). The package `BswRobotMW` (`BswResources`) defines the target robotic middleware for the semantics-preserving code-generation process and is described in the corresponding chapter (Chapter 4).

3.4.1. Physical Platform Modeling

The `HwResources` package introduces model elements representing embedded HW commonly used in robotic systems, including computing, communication and device resources.

The meta-model concepts are organized in models (packages). The `HwControlUnit` model defines the structure of the HW execution architecture, as shown in Figure 3.8. A control unit (`HwCU`) is composed of one or more HW boards (`HwBoards`), eventually connected together through expansion slots. A HW board may contain multiple CPUs, both single- and multi-core; at least one concrete kind of HW communication interface (*`HwCOMInterface`*) connected to the corresponding bus (*`HwBus`*); and, finally, any number of auxiliary device resources for the interaction with the environment (*`HwDevice`*s).

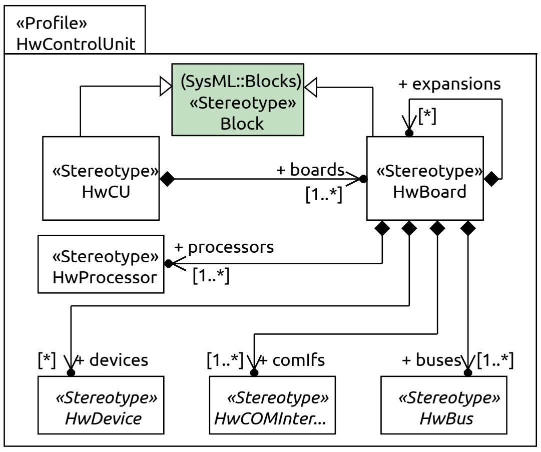


FIGURE 3.8. `HwControlUnit` model's structure.

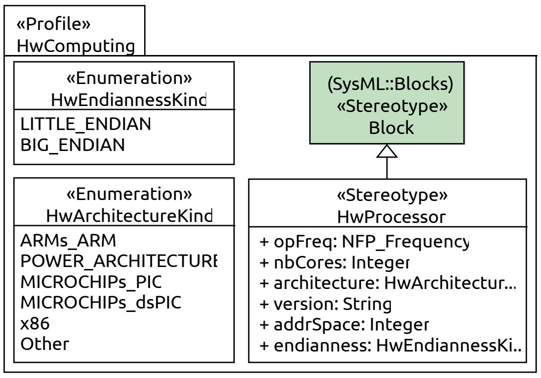


FIGURE 3.9. `HwComputing` model's structure.

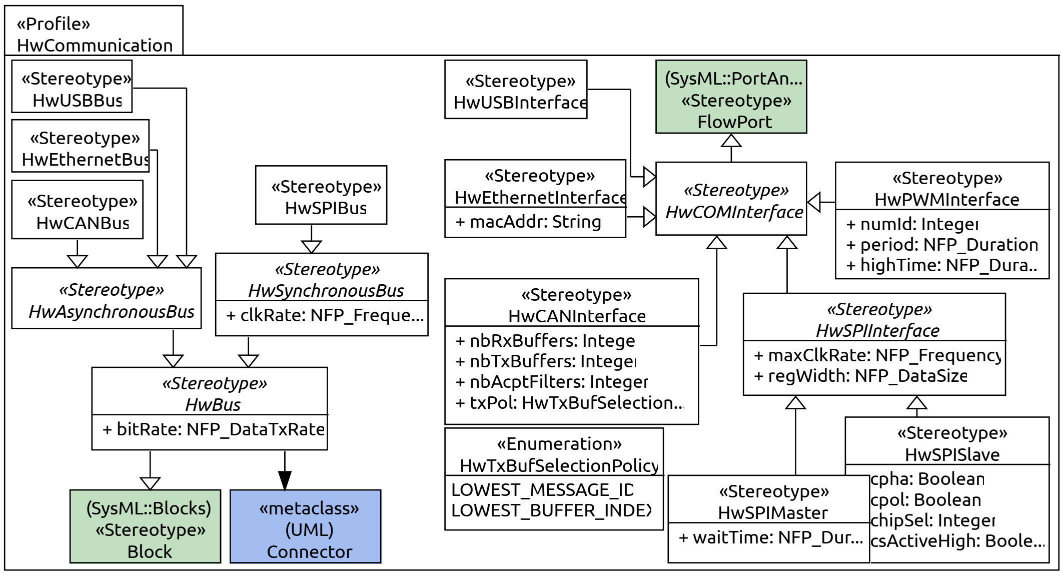

«Profile»
HwCommunication
«Stereotype» HwUSBBus
«Stereotype» HwEthernetBus
«Stereotype» HwCANBus
«Stereotype» HwAsynchronousBus
«Stereotype» HwSPIBus
«Stereotype» HwSynchronousBus
+ clkRate: NFP_Freque...
«Stereotype» HwBus
+ bitRate: NFP_DataTxRate
(SysML::Blocks) «Stereotype» Block
«metaclass» (UML) Connector
«Stereotype» HwUSBInterface
«Stereotype» HwEthernetInterface
+ macAddr: String
(SysML::PortAn... «Stereotype» FlowPort
«Stereotype» HwCOMInterface
«Stereotype» HwPWMInterface
+ numId: Integer
+ period: NFP_Duration
+ highTime: NFP_Dura..
«Stereotype» HwCANInterface
+ nbRxBuffers: Intege
+ nbTxBuffers: Integer
+ nbAcptFilters: Integer
+ txPol: HwTxBufSelection...
«Stereotype» HwSPIInterface
+ maxClkRate: NFP_Frequency
+ regWidth: NFP_DataSize
«Enumeration» HwTxBufSelectionPolicy
LOWEST_MESSAGE_ID
LOWEST_BUFFER_INDEX
«Stereotype» HwSPIMaster
+ waitTime: NFP_Dur...
«Stereotype» HwSPISlave
cpha: Boolean
cpol: Boolean
chipSel: Integer
csActiveHigh: Boole...

FIGURE 3.10. `HwCommunication` model's structure.

The concept of CPU is part of the `HwComputing` model (Figure 3.9), and is defined by the stereotype `HwProcessor`. Like in MARTE, the property `nbCores` defines the CPU's number of cores. Other properties describe the operating frequency, the instruction set architecture, the endianness and the address space size.

The *`HwCOMInterface`* and *`HwBus`* specifications are part of the `HwCommunication` model (Figure 3.10). The model defines concrete specializations of the two stereotypes to represent standard HW interfaces for network communication such as CAN, Ethernet, USB, SPI and PWM.

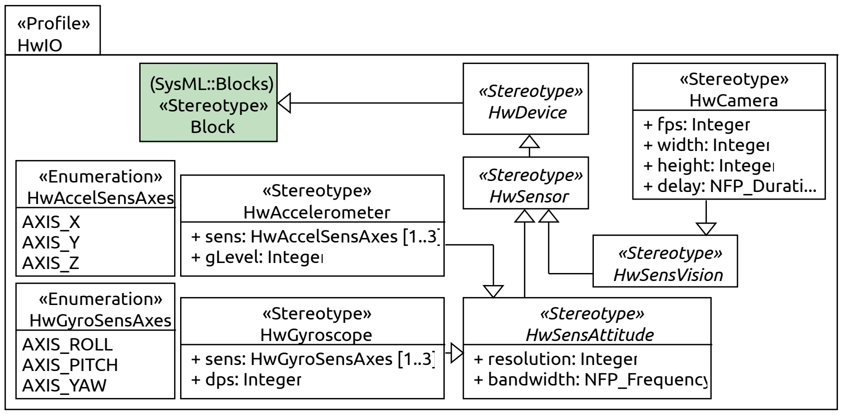


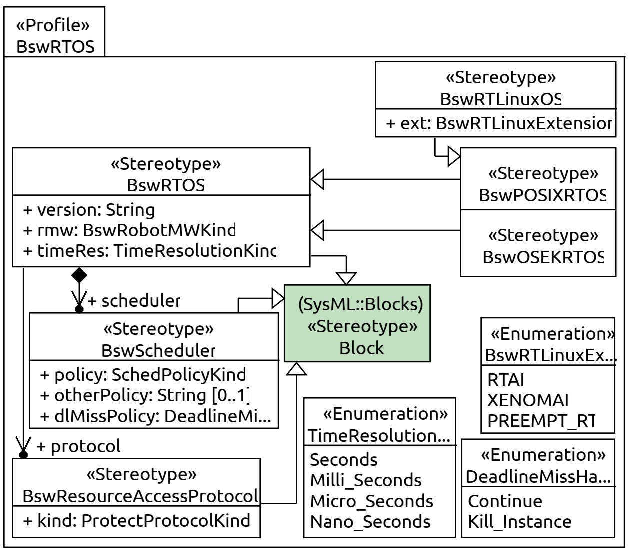
FIGURE 3.11. Portion of `HwIO` model's structure.

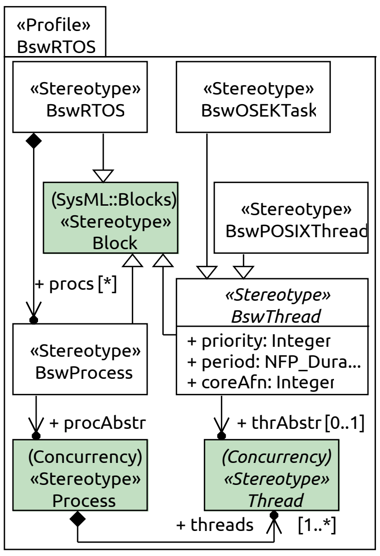
Finally, device resources are defined in the `HwIO` model. The model provides two categories of *`HwDevice`*s: sensors (*`HwSensor`*s) and actuators (*`HwActuator`*s). These are further classified into sub-categories according to their logical functionality. Figure 3.11 shows a subset of the `HwIO` profile providing a model of two attitude sensors, a gyroscope (`HwGyroscope`) and an accelerometer (`HwAccelerometer`), and a vision sensor (`HwCamera`). The properties `bandwidth` (attitude) and `fps` and `delay` (camera) describe sensors' attributes that are related to time. The `bandwidth` indicates how often a reliable sensor reading can be taken (i.e., how many measurements can be made per second); the `fps` and `delay` indicate the frequency rate at which the camera works and the processing delay, respectively. The other properties complete the specification of sensor models and are used for code-generation purposes. The `HwIO` profile also includes a model of relative encoder, potentiometer and servo-motor.

3.4.2. BSW Resources Modeling

BSW components and deployments of BSW modules onto the HW are modeled according to the taxonomy of stereotypes defined in `BswResources`, which is composed of several packages. BSW components include the I/O drivers and their queuing policies (`BswIO`), the communication stacks, with models of the transport and network protocols (`BswCOM`), possible middleware tasks and their activation policies (`BswRobotMW`), and the (real-time) operating system, with the supported scheduling policies (`BswRTOS`). Model elements representing the deployment of BSW modules onto the execution architecture are specified in `BswAllocation`.

The package `BswRTOS` is composed of two views: one defines the profile elements used by system designers to represent RTOS components that are part of the BSW modules of control units, and the other defines the elements used by the M2M transformations to *automatically* generate model artifacts representing the BSW implementation of the real-time (control) tasks and their interactions (BSW model). Figure 3.12 shows the model elements for the two views, respectively.


«Profile» BswRTOS
«Stereotype» BswRTLinuxOS
+ ext: BswRTLinuxExtension
«Stereotype» BswRTOS
+ version: String
+ rmw: BswRobotMWKind
+ timeRes: TimeResolutionKind
«Stereotype» BswPOSIXRTOS
«Stereotype» BswOSEKRTOS
+ scheduler
«Stereotype» BswScheduler
+ policy: SchedPolicyKind
+ otherPolicy: String [0..1]
+ dlMissPolicy: DeadlineMi…
(SysML::Blocks) «Stereotype» Block
«Enumeration» BswRTLinuxEx…
RTAI
XENOMAI
PREEMPT_RT
«Enumeration» TimeResolution…
Seconds
Milli_Seconds
Micro_Seconds
Nano_Seconds
«Enumeration» DeadlineMissHa…
Continue
Kill_Instance
+ protocol
«Stereotype» BswResourceAccessProtocol
+ kind: ProtectProtocolKind
(A) Designer view.



(B) M2M tool view.

FIGURE 3.12. BswRTOS model's structure.

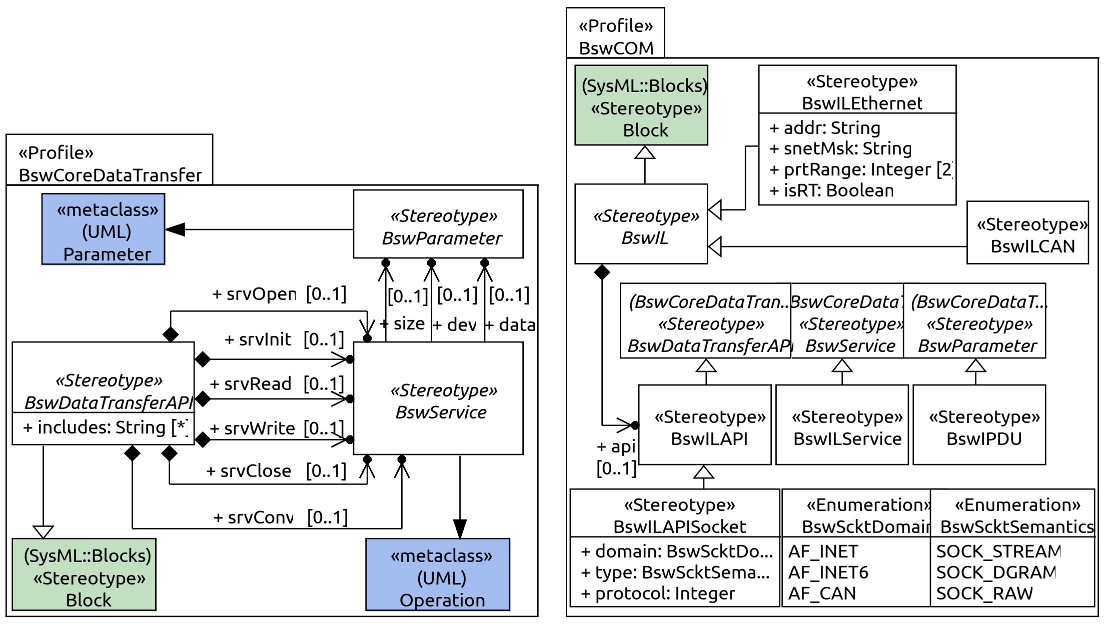
The stereotype BswRTOS (Figure 3.12a) denotes an RTOS. Some specializations of BswRTOS are defined to represent, e.g., kinds of RTOS that comply with the OSEK/VDX and POSIX standards, including real-time Linux.

The RTOS contains a scheduler (BswScheduler), which executes tasks according to a given scheduling policy (SchedPolicyKind), selected among one of the kinds defined in the MARTE model library. When concurrent tasks need to access shared resources, the RTOS uses a resource access protocol to ensure consistency of the data and time determinism.

In the envisioned development flow, system designers do not represent the application execution model (processes, threads and their interactions) as BSW model, but rather an abstract mapping model (Section 3.5). M2M transformations then process the mapping model and generate a corresponding model with OS threads and middleware components. BswProcess and *BswThread* (and its specializations) are the BSW model artifacts that map to the generic concepts of process and thread (Figure 3.12b).

Basic concepts representing data transfers in execution platforms are defined in package BswCoreDataTransfer and refined in BswCOM and BswIO. Figure 3.13 shows a partial view of BswCoreDataTransfer and BswCOM profiles.

BswCOM enables the modeling of implementation/configuration of BSW modules that support local communication and network communication among distributed control units. The specification of model elements follows to a large extent the OSEK COM standard [**OSE04**]. The package defines an interaction layer, or IL (*BswIL*), that provides the API (BswILAPI) with services for the transfer of messages (BswILServices). A number of stereotypes represent ILs operating according to specific network and/or data link layers, e.g., IP/Ethernet (BswILEthernet) and CAN


«Profile» BswCoreDataTransfer
«metaclass» (UML) Parameter
«Stereotype» BswParameter
+ srvOpen [0..1]
+ srvInit [0..1]
+ srvRead [0..1]
+ srvWrite [0..1]
+ srvClose [0..1]
+ srvConv [0..1]
[0..1] + size
[0..1] + dev
[0..1] + data
«Stereotype» BswDataTransferAPI
+ includes: String [*]
«Stereotype» BswService
(SysML::Blocks) «Stereotype» Block
«metaclass» (UML) Operation
«Profile» BswCOM
(SysML::Blocks) «Stereotype» Block
«Stereotype» BswILEthernet
+ addr: String
+ snetMsk: String
+ prtRange: Integer [2]
+ isRT: Boolean
«Stereotype» BswIL
«Stereotype» BswILCAN
(BswCoreDataTran... «Stereotype» BswDataTransferAPI
(BswCoreData... «Stereotype» BswService
(BswCoreDataT... «Stereotype» BswParameter
+ api [0..1]
«Stereotype» BswILAPI
«Stereotype» BswILService
«Stereotype» BswIPDU
«Stereotype» BswILAPISocket
+ domain: BswScktDo...
+ type: BswScktSema...
+ protocol: Integer
«Enumeration» BswScktDomain
AF_INET
AF_INET6
AF_CAN
«Enumeration» BswScktSemantics
SOCK_STREAM
SOCK_DGRAM
SOCK_RAW

(a) . (b) .

Figure 3.13. Portion of `BswCoreDataTransfer` and `BswCOM` models structures.

(`BswILCAN`). The set of functions the IL offers to handle messages is defined by specializations of `BswILAPI`, e.g., the socket API to configure and use network sockets.

Finally, the `BswIO` package specializes the definitions provided in `BswCoreDataTransfer` to allow the modeling of the API of common analog and digital I/O device drivers.

3.5. Software Architecture Modeling

The mapping model associates functional elements to tasks, and tasks to processing (HW) resources. Accordingly, communication signals of the functional view are mapped to signal variables for intra-task, inter-task, remote and I/O communications. Typically, the association is performed manually by the system designer. However, the mapping model may also be obtained as result of several optimization problems, including, but not limited to, how to map functions into tasks, how to assign the execution order of functions inside tasks and how to assign the task parameters (priority, deadline, offset) to guarantee semantics preservation and schedulability.

Figure 3.14 visualizes the structure of the mapping profile, organized in four packages. The `Concurrency` package classifies concurrent execution contexts in terms of processes and threads. The `Synchronization` package includes profile elements to represent precedence relations on tasks' executions; it also represents concepts for the code-generation tools to describe the synchronous activation of tasks (these will be introduced later, in Chapter 4). The `Interaction` package defines the signal variables, implementing functional communication signals. Finally,

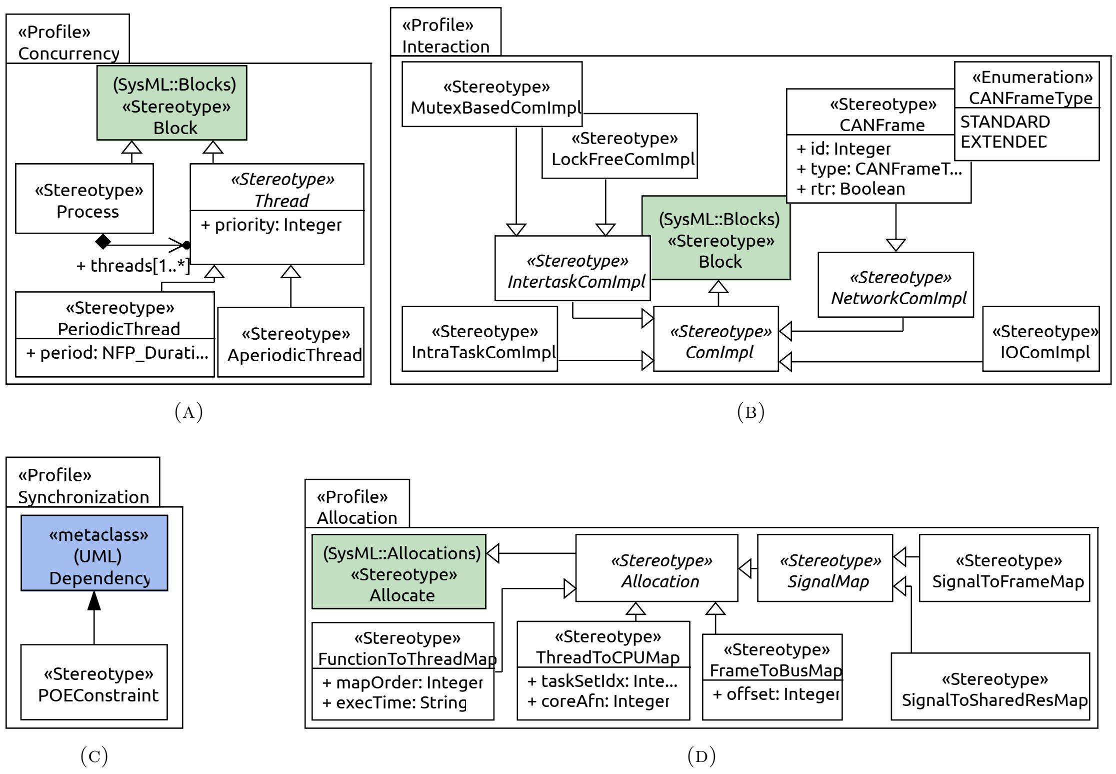

«Profile» Concurrency
(SysML::Blocks) «Stereotype» Block
«Stereotype» Process
«Stereotype» Thread
+ priority: Integer
+ threads[1..*]
«Stereotype» PeriodicThread
+ period: NFP_Durati...
«Stereotype» AperiodicThread
(A)
«Profile» Interaction
«Stereotype» MutexBasedComImpl
«Stereotype» LockFreeComImpl
«Stereotype» CANFrame
+ id: Integer
+ type: CANFrameT...
+ rtr: Boolean
«Enumeration» CANFrameType
STANDARD
EXTENDED
(SysML::Blocks) «Stereotype» Block
«Stereotype» IntertaskComImpl
«Stereotype» NetworkComImpl
«Stereotype» IntraTaskComImpl
«Stereotype» ComImpl
«Stereotype» IOComImpl
(B)
«Profile» Synchronization
«metaclass» (UML) Dependency
«Stereotype» POEConstraint
(C)
«Profile» Allocation
(SysML::Allocations) «Stereotype» Allocate
«Stereotype» Allocation
«Stereotype» SignalMap
«Stereotype» SignalToFrameMap
«Stereotype» FunctionToThreadMap
+ mapOrder: Integer
+ execTime: String
«Stereotype» ThreadToCPUMap
+ taskSetIdx: Inte...
+ coreAfn: Integer
«Stereotype» FrameToBusMap
+ offset: Integer
«Stereotype» SignalToSharedResMap
(D)

FIGURE 3.14. Structure of mapping model's SysML profiles.

the `Allocation` package specifies a set of dependencies that define mappings/deployments as extensions of the standard `SysML::Allocate` concept.

Two concepts are central to the definition of a mapping model: *threads*, represented by the stereotype ***`Thread`***, and *signal variables*, denoted as ***`ComImpl`***. Both stereotypes apply to the `SysML::Block`.

A ***`Thread`*** is a unit of concurrent execution that runs on one of the system cores under the control of an RTOS (Figure 3.14a). Each ***`Thread`*** is contained in a `Process` and is characterized by a `priority` value. Concrete specializations of ***`Thread`*** are `AperiodicThread` and `PeriodicThread` (with its `period`). Precedence relations among threads induce partial order of execution constraints on the task set, and are modeled as `POEConstraints` (which extends the UML meta-class `Dependency`, as shown in Figure 3.14c).

Each signal variable is an implementation of the communication link between functional subsystems mapped to tasks and allocated to processing resources. Four specializations of the stereotype ***`ComImpl`*** are instantiable (Figure 3.14b). The `IntraTaskComImpl` describes a communication that takes place when two communicating functional subsystems are mapped into the same ***`Thread`***.

The *InterTaskComImpl* represents a communication between two *Thread* elements that execute on the same CPU. In this case, a protection mechanisms for shared resources is used (lock and lock-free synchronization).

The stereotype `NetworkComImpl` describes a communication between two *Thread*s executing on different HW boards connected by a network, e.g., a field bus.

Finally, the `IOComImpl` denotes a communication that takes place between a control subsystem and the functional subsystem representing the plant model (a communication between a thread and an I/O driver).

The concept of allocation completes the specification of mapping profile (Figure 3.14d). Allocations are used to associate individual application elements to individual execution platform elements. Four classes of allocation are defined. First, the `FunctionToThreadMap` denotes the mapping of a functional subsystem into a *Thread*. Each mapped subsystem refers its own `step()` method, realizing the output update and the state update functions. This information is used in the code-generation phase to produce the implementation of thread code. When multiple subsystems are mapped into the same *Thread*, the `mapOrder` defines how the execution of their `step()` methods will be serialized in the generated thread code. The *mapping order* must be consistent with the partial order of execution imposed by the model semantics.

Second, the `ThreadToCPUMap` models the deployment of a *Thread* to an `HwProcessor`. The attribute `coreAfn` enables the binding of the thread to a physical processor core (affinity). Note that, for every thread, once the target CPU is known, the `execTime`s of `FunctionToThreadMap`s are filled with textual values; these represent the (measured or estimated) times the `step()`s of subsystems mapped into the thread take to execute on the target processor.

Third, the *SignalMap* represents the mapping of a functional communication link to a signal variable. Concrete kinds of *SignalMap* denote mapping to a shared resource for inter-task communication and to a resource for network communication. The stereotype `FrameToBusMap` describes the allocation of a network frame onto a physical link connecting control units. Its attribute `offset` is used when signals are multiplexed in message frames.

And finally, fourth, the `FrameToBusMap` describes the allocation of a network frame onto a physical link connecting HW control units. Its attribute `offset` is used when signals are multiplexed in message frames.

3.6. Summary

The Chapter starts by defining the application scenarios for the proposed design flow. We make a distinction between safety-critical and performance-sensitive applications and systems, and describe the role of PBD flow in the two scenarios. Next, we define the process flow based on *standards* (Simulink and OMG's SysML/MARTE, MOFM2T and M2M) and open (EMF-based) tools (Papyrus, Acceleo and QVTo).

We introduce models and abstractions for the description of the functional platform specifications, the execution architecture options and the SW architecture implementation. Models are heterogeneous and fit different purposes—SR models for control simulation, testing and behavioral code generation; SysML-/MARTE-based models (more precisely profiles and stereotypes) for representing architectural aspects and timing-related properties. We define model transformation rules that realize the integration between SR and SysML/MARTE models.

We argue that proposed stereotypes expose the adequate level of information to enable timing analysis and code-generation, without resulting in convoluted models difficult to understand and cumbersome to use.

CHAPTER 4

Generation of Semantics-Preserving Robot Controls from Simulink Models

4.1. Context and Positioning

Safety-critical CPS (cfr. Section 3.1) include numerous robotic systems, most notably those for pHRI (e.g, future robotic co-workers and advanced domestic robots), disaster recovery and space exploration.

In robotics, component-based software engineering is the most common design paradigm for application development. Application designers program the robot functionality directly as C/C++ code into a set of (interacting) SW components and test it against a virtual model of the robot in a simulation environment providing features such as physics engine(s) and complex indoor/outdoor scene rendering. Middleware frameworks provide the infrastructure to execute the component-based applications (and the simulator) and abstract the functionality from the computing platform. The simulator accepts control inputs from the SW components (e.g., desired joint torques and desired joint position) and outputs sensory feedback from the simulated world (e.g., cameras and joint positions). In most cases, the virtual HW interfaces of simulated sensors/actuators match those of the real robot, thus, at least in principle, the same code used in simulation can be re-used on the real robotic platform.

While this Software-in-the-Loop approach eases the development of high-level, non-time critical robotic tasks (perception, planning, reasoning, learning, etc.) it suffers of severe drawbacks when facing the problem of safety-critical application design. Robotic component-based SW frameworks *lack of a formal model of computation* and the system-level behavior emerges from the cooperation of SW components. This makes the realization of a system-level semantics (e.g., synchronous-reactive) difficult, because, in general, causal dependencies between producers and consumers (defining partial orders of execution) are not trivial to express by using event signals. Furthermore, current middlewares are hardly integrated in a MBD flow with automatic code generation. Control algorithms are mostly handwritten, and when they are designed using synchronous models (Simulink) the generated code can only be executed in a single core. Multicore and distributed platforms, or platforms based on a domain specific OS and middleware are not supported by generation tools that can guarantee the preservation of the original model semantics. In a sound MBD flow, the software implementation of controller should be generated automatically, so as to improve the efficiency and guarantee that no errors are introduced in a manual coding stage.

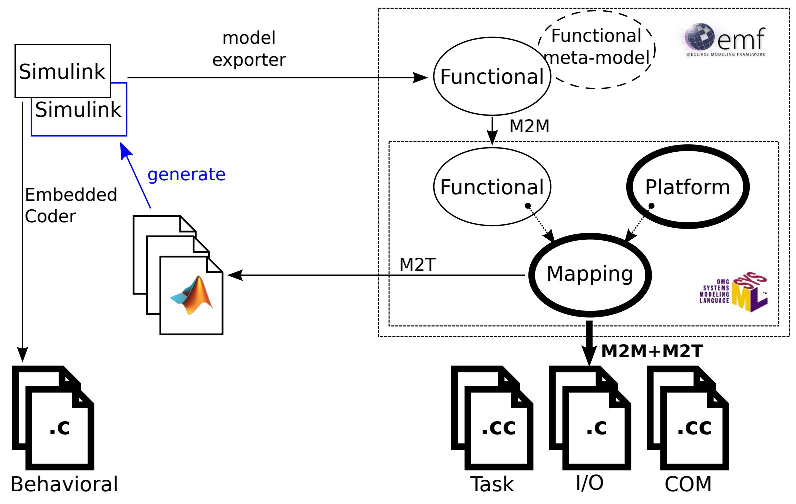


FIGURE 4.1. Heterogeneous model integration and code generation by the framework tools.

The tools supporting the proposed system-level design framework target multi-core code-generation of robotics applications with preservation of the communication flows from synchronous models. The framework enables

(1) the specification of the software and message implementation of SR models, and
(2) a semantics-preserving deployment on top of the Orocos-RTT [**Bru**] robotics middleware with the automatic generation of glue code.

Figure 4.1 represents the heterogeneous model integration and code generation enabled by the framework tools. The Simulink model of the controls enable the advance analysis and verification of the system properties by simulation. Once the simulation results are satisfactory, code is generated for each control subsystem. Then, a structural view of the functional model is imported in SysML, where it is extended with the platform and mapping models, that represent the deployment of the control functions onto the execution architecture. Finally, the mapping model is processed for the automatic generation of a multi-task implementation of control functions, which preserves the execution assumptions part of the simulated and verified Simulink model.

This chapter focuses on the description of the underlying concepts and the machinery to implement the code generation process (bold items in Figure 4.1) for single-/multi-core robotics architectures.

The rest of the chapter is organized as follows. In Section 4.2 we review the requirements and constraints that apply to our process, i.e., the need to preserve the synchronous model semantics, for which we provide a formal description. Section 4.3 gives an overview of the Orocos-RTT execution middleware, that represents the target run-time of the generation process. Section 4.4 defines the code generation process using synthetic examples. Finally, Section 4.5 summarizes and closes the chapter.

4.2. Constraints in the Implementation of Synchronous (Simulink) Models

4.2.1. Model Assumptions and Basic Formalization of the Synchronous Semantics

Simulink implements a Synchronous-Reactive (SR) model of computation (MoC). SR models are networks of Mealy-type blocks, possibly clustered into subsystems and blocks that can be continuous, discrete or triggered. Continuous blocks process continuous-time signals and produce as output other continuous-signal functions according to the block description, typically a set of differential equations. Discrete-time blocks are activated at periodic-time instants and process input signals, sampled at periodic instants, producing a set of periodic-output signals and the state updates. Finally, triggered blocks are only executed on the occurrence of a given event (a signal transition or a function call).

We are interested in the automatic generation of an implementation of the controller model, and we assume that (*i*) its design only uses discrete-time blocks, (*ii*) each block b_i processes a set of input signals at times that are multiples of a period T_i, and (*iii*) all block periods are integer multiples of the *base period* T_b.

We denote inputs of block b_i by $\overline{i_i}$ and outputs by $\overline{o_i}$ (to indicate vectors). At all times kT_i the block reads the signal values on its inputs and computes two functions: an *output update function* $\overline{o_i} = f_o(\overline{i_i}, S_i)$ and a *state update function* $S_i^{\text{New}} = f_s(\overline{i_i}, S_i)$, where S_i (S_i^{New}) is the current (next) state of b_i. Often, the two update functions can be considered as one:

$$(\overline{o_j}, S_j^{\text{New}}) = f_u(\overline{i_j}, S_j).$$

Signal values are persistent until updated. Therefore, each input and output is a right-continuous function, sampled at periodic time instants by a reading block.

A fundamental part of the model executable semantics are the rules dictating the evaluation order of the blocks. A block has *direct feedthrough* when the output is controlled directly by the value of an input port signal. Any block with direct feedthrough cannot execute until the block(s) driving its input has (have) executed. Some blocks set their outputs based on values acquired in a previous time step or from initial conditions specified as a block parameter. The set of topological dependencies implied by the direct feedthrough behavior of blocks input/output pairs defines a partial order of execution among the blocks.

Let b_i and b_j two blocks in an input-output relationship. Let $b_i(k)$ represent the k-th occurrence of block b_i (belonging to the set of time instants kT_i), then a sequence of activation times $a_i(k)$ is associated to b_i. Given $t \geq 0$, we define $n_i(t)$ to be the number of times that b_i has been activated before or at t. Finally, let $i_j(k)$ denote the input of the k-th occurrence of b_j. In case b_j has direct feedthrough, then the SR semantics specify that $i_j(k)$ is equal to the output of the last occurrence of b_i that is no later than the k-th occurrence of b_j, i.e.,

$$i_j(k) = o_i(m), \text{ where } m = n_i(a_j(k)). \qquad (1)$$

The timeline on the bottom of Figure 4.2 illustrates the execution of a pair of connected blocks with SR semantics, where b_j has direct feedthrough. The horizontal axis represents time. The vertical arrows capture the time instants when the blocks are activated and compute their outputs from the input values. If the communication link between b_i and b_j has a delay (i.e., b_j is not of

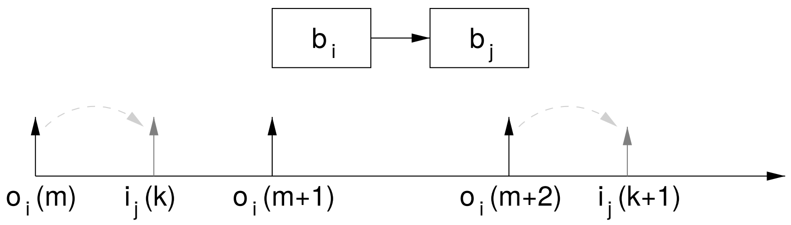


FIGURE 4.2. Execution of a pair of connected blocks with direct feedthrough according to the SR semantics ($i_j(k) = o_i(m)$).

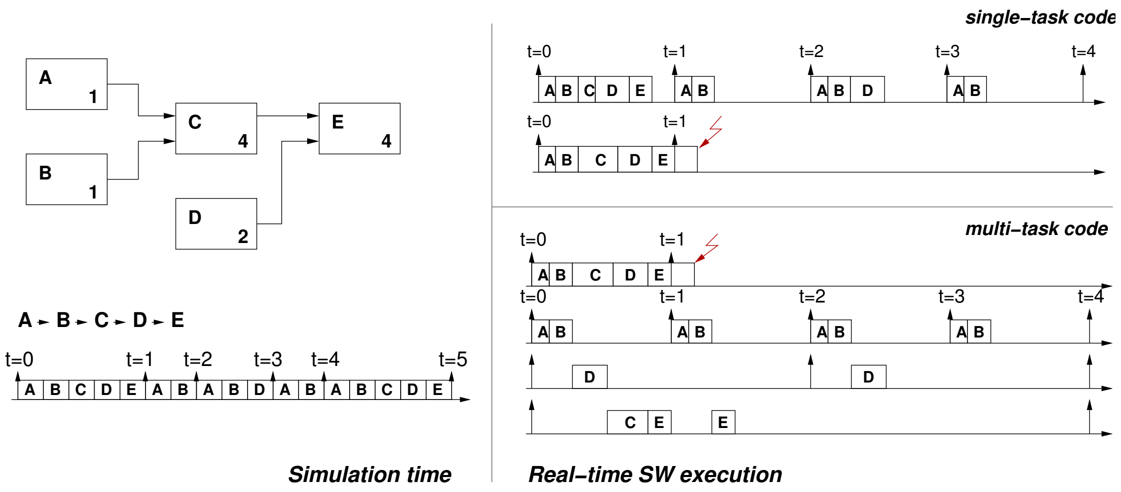


FIGURE 4.3. Evaluation of blocks at simulation time and at runtime (single- and multi-task implementations).

type feedthrough), then the previous output value is read, that is,

$$i_j(k) = o_i(m-1). \tag{2}$$

When the simulation starts, blocks are ordered, and a total order compatible with the partial order of execution is determined. When a block is activated, inputs are sampled and output update and state update functions computed in sequence to produce the system outputs. The reaction time of the system is istantaneous, meaning that it takes zero computation and communication time. An example of behavior at simulation time is shown on the left side of Figure 4.3, where five communicating blocks are identified with letters and their execution period. Simulation time in general has no direct relationship with real-time.

4.2.2. Software Implementation and Preservation of Data-Flows

In the *software implementation of functions*, the update functions of blocks and their action extensions are executed by program functions (or lines of code), executed by a task, under the control of a priority-based RTOS. The implementation consists of two functions or sequences of statements, one for the state update, the other for the output update (the output update parts must be executed before the state update). The two functions are often merged into a single update function, typically called `step()`. The function-to-task mapping consists of a static scheduling (execution order) of the function code inside the task.

The implementation must preserve the simulation semantics, so to retain the validation and verification results. In many cases, what is required from a software implementation is not the preservation of the synchronous assumption, i.e., that the reaction of the system is computed before the next event in the system, but a looser property, called *flow preservation*. It amounts at guaranteeing that all the data-flows in the system are preserved in their implementation, even if the times at which the results are produced are different. Formally, it is to guarantee that every source block is executed without instance skips, that Equation (1) or (2) hold for any signal exchanged between two blocks and the correct untimed behavior of all blocks. The right side of Figure 4.3 shows the execution instance of a multi-task implementation that does not satisfy the synchronous assumption (the output of block E is produced after time 1) but is flow-preserving.

Because of preemption and scheduling, in a multi-rate system, the signal flows of the implementation can differ from the model flows.

Flow preservations in all scenarios requires appropriate communication mechanisms. A formal description of the requirements for flow-preserving communication mechanisms can be found in [**BCDN$^+$07**], and the implementation of a general type of such wait-free buffer is discussed in [**WDNSV09**].

4.3. The Orocos-RTT Run-Time Environment

The Real-Time Toolkit of the Orocos-toolchain (Orocos-RTT) is a middleware layer which provides the infrastructure and the functionalities to program and execute component-based robotics applications. The run-time environment allows components to run on top of Real-Time Operating Systems (RTOSs). The C++ API[1] is flexible and allows many different components configuration options and operation modes to be selected. We restrict to the specific subset of settings and object semantics summarized below.

The class `RTT::TaskContext` represents a component and is the basic unit to execute application code in a single thread. Activities are RTT objects that map to threads and can be periodic, non periodic or event driven, and are all concrete implementations of an interface class which provides control methods for starting, stopping and querying them for their state. Instances of class `RTT::Activity` *run* RTT components. An `RTT::Activity` object allocates a thread which executes the execution engine of the corresponding `RTT::TaskContext` instance; the execution engine calls the `updateHook()` method of the component and so executes application code. RTT Components can be connected to each other so that they can access their control methods. Connected components are called *peers*.

In order to send or receive streams of data, components can define data-flow ports. Data input and output ports are instances of classes `RTT::InputPort<T>` and `RTT::OutputPort<T>`, respectively. A connection policy object describes how a connection between data ports should

[1]`http://www.orocos.org/stable/documentation/rtt/v2.x/api/html/index.html`

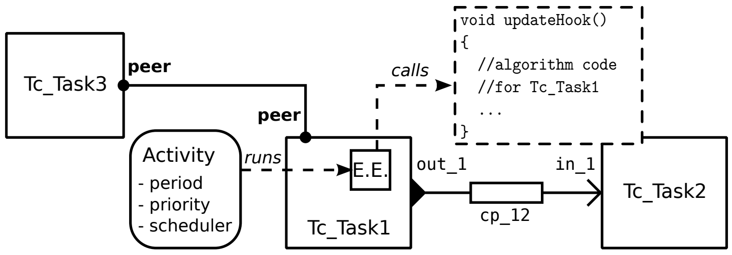


FIGURE 4.4. Schematic representation of deployed Orocos-RTT components.

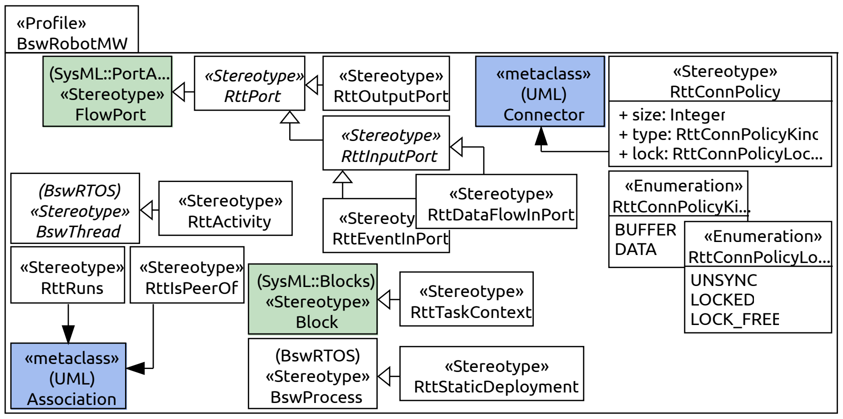

«Profile» BswRobotMW
(SysML::PortA... «Stereotype» FlowPort
«Stereotype» RttPort
«Stereotype» RttOutputPort
«metaclass» (UML) Connector
«Stereotype» RttConnPolicy
+ size: Integer
+ type: RttConnPolicyKinc
+ lock: RttConnPolicyLoc...
«Stereotype» RttInputPort
(BswRTOS) «Stereotype» BswThread
«Stereotype» RttActivity
«Stereotype» RttDataFlowInPort
«Stereoty RttEventInPort
«Enumeration» RttConnPolicyKi...
BUFFER
DATA
«Enumeration» RttConnPolicyLo...
UNSYNC
LOCKED
LOCK_FREE
«Stereotype» RttRuns
«Stereotype» RttIsPeerOf
(SysML::Blocks) «Stereotype» Block
«Stereotype» RttTaskContext
«metaclass» (UML) Association
(BswRTOS) «Stereotype» BswProcess
«Stereotype» RttStaticDeployment

FIGURE 4.5. `BswRobotMW` model's structure.

behave (`RTT::ConnPolicy`). Various configuration parameters can be set, such as the connection type (data or buffered), the locking policy (none, mutex- or lock-free-based) and the data size. Lock-free connections are the basis to build the communication mechanisms that guarantee the preservation of communication flow. *Event (input) ports* can also be added to the interface of a component. The reception of an event signal awakes the component and invokes `updateHook()`. It is possible to register a callback function that gets executed when a signal arrives on the event port: in this case, both the callback (executed first) and `updateHook()` are called.

Components are created in a C++ program and defined at compilation time (*static deployment* in a process).

Figure 4.4 shows a schematic representation of a deployment, with connected peer components, the threaded execution of a component's execution engine, and components data-flow ports connected with a connection policy. The `BswRobotMW` package in the `BswResources` profile (cfr. Section 3.4) provides a taxonomy of stereotypes to represent and configure all these concepts (Figure 4.5).

4.4. The Code-Generation Process

The code-generation process is two-step. The mapping model is the starting point for a set of M2M transformations that generate a model of BSW implementation of tasks and their

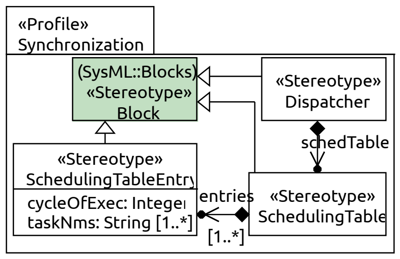


FIGURE 4.6. Structure of `Mapping::Synchronization` package (extends that of Figure 3.14c).

interactions. In addition, the M2M transformations generate a model of the scheduling and event infrastructure that guarantees the execution order among tasks.

In the BSW model, the abstract concept of task is represented by a pair of model element instances stereotyped, respectively, as `RttActivity` and `RttTaskContext`. The element `RttActivity` generated for each `RttTaskContext` instance contains the definition of the activation mode and the thread period. This follows directly from the observation that an `RttActivity` is a ***BswThread*** (Figure 4.5), and hence it can access the corresponding ***Thread*** in the mapping model through the attribute `thAbstr` (Figure 3.12b).

Next, a set of M2T transformation templates process the BSW model to generate the C/C++ code for (*i*) threads (the implementation of their `updateHook()` methods), (*ii*) threads activations and synchronization, and (*iii*) communication among threads.

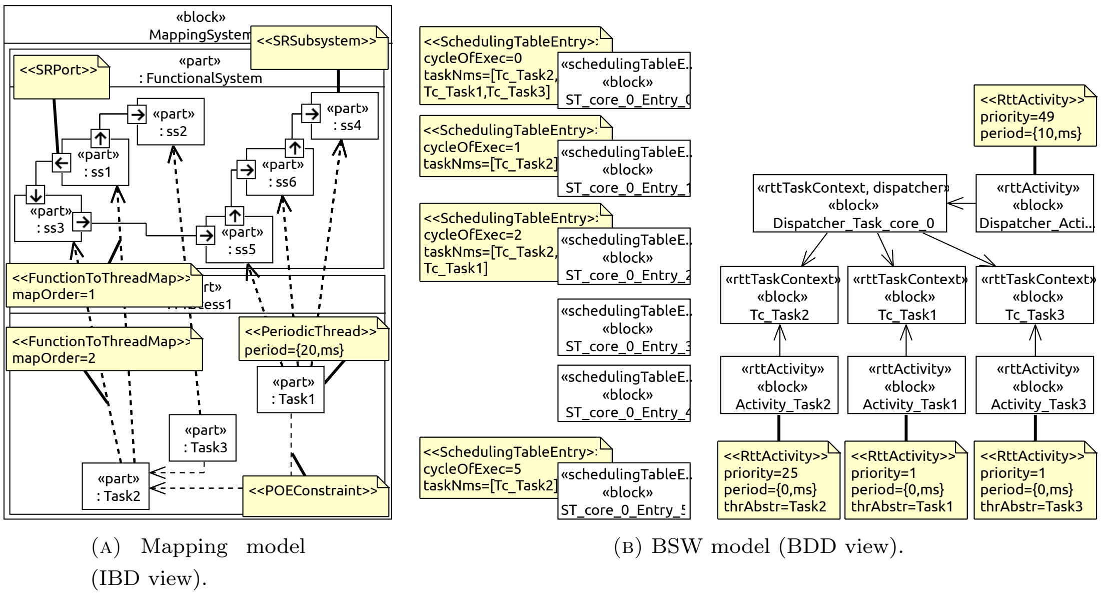
The following subsections describe the code-generation process through synthetic application examples.

4.4.1. Generation of Task Synchronization Infrastructure

Single-Core Computing Node

Consider a mapping model where three tasks are mapped onto a single-core platform, and assume the function-to-task mapping of Figure 4.7a. All tasks are periodic: `Task2` executes every 10ms, `Task3` every 30ms and `Task1` every 20ms. Input/output relationships among the subsystems (`SRSubsystems` on top) define a partial order of execution that implies precedence constraints (`POEConstraints`) among the tasks where the subsystems are mapped into. Hence, `Task2` executes before `Task1` and `Task3`. The attribute `mapOrder` expresses precedence relations for subsystems mapped into the same task.

Figure 4.7b shows the BSW model generated by the M2M transformations. `RttTaskContext` model elements corresponding to ***Thread***s are run by *non-periodic* activities, and the synchronous activation of tasks is modeled using the stereotypes in Figure 4.6, that extend the `Mapping::Synchronization` package (cfr. Section 3.5). One additional `RttTaskContext` element is generated and marked as `Dispatcher`; it executes *periodically* at the base period (the greatest common divisor of all task periods) and with highest priority. All the `RttTaskContexts` corresponding to ***Thread***s are added as *peer components* to the `Dispatcher`, which is also provided with a scheduling table, that specifyies which tasks must be activated, and in which


«block» MappingSystem
<<SRSubsystem>>
<<SRPort>>
«part» : FunctionalSystem
«part» : ss1
«part» : ss2
«part» : ss3
«part» : ss4
«part» : ss5
«part» : ss6
<<FunctionToThreadMap>> mapOrder=1
<<FunctionToThreadMap>> mapOrder=2
<<PeriodicThread>> period={20,ms}
«part» : Task1
«part» : Task2
«part» : Task3
<<POEConstraint>>
<<SchedulingTableEntry>: cycleOfExec=0 taskNms=[Tc_Task2, Tc_Task1,Tc_Task3]
<<SchedulingTableEntry>: cycleOfExec=1 taskNms=[Tc_Task2]
<<SchedulingTableEntry>: cycleOfExec=2 taskNms=[Tc_Task2, Tc_Task1]
<<SchedulingTableEntry>: cycleOfExec=5 taskNms=[Tc_Task2]
«schedulingTableE.. «block» ST_core_0_Entry_0
«schedulingTableE.. «block» ST_core_0_Entry_1
«schedulingTableE.. «block» ST_core_0_Entry_2
«schedulingTableE.. «block» ST_core_0_Entry_3
«schedulingTableE.. «block» ST_core_0_Entry_4
«schedulingTableE.. «block» ST_core_0_Entry_5
<<RttActivity>> priority=49 period={10,ms}
«rttTaskContext, dispatcher» «block» Dispatcher_Task_core_0
«rttActivity» «block» Dispatcher_Acti..
«rttTaskContext» «block» Tc_Task2
«rttTaskContext» «block» Tc_Task1
«rttTaskContext» «block» Tc_Task3
«rttActivity» «block» Activity_Task2
«rttActivity» «block» Activity_Task1
«rttActivity» «block» Activity_Task3
<<RttActivity>> priority=25 period={0,ms} thrAbstr=Task2
<<RttActivity>> priority=1 period={0,ms} thrAbstr=Task1
<<RttActivity>> priority=1 period={0,ms} thrAbstr=Task3

(A) Mapping model (IBD view). (B) BSW model (BDD view).

FIGURE 4.7. Mapping model and automatically-generated BSW model representing the deployment onto a single-core processor.

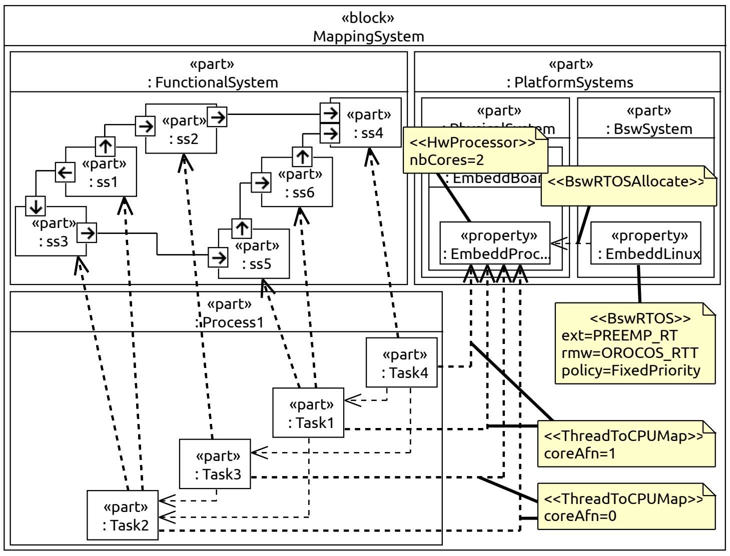
order, in the hyperperiod (the least common multiple of the periods of all tasks). Instances of `SchedulingTableEntry` are generated according to values of period, priority and scheduling policy specified by the `PeriodicThread` and `BswRTOS` (`BswScheduler`) stereotypes.

In the generated C++ code, at every cycle of execution, the dispatcher triggers the activation of its peers according to the directives in the scheduling table.

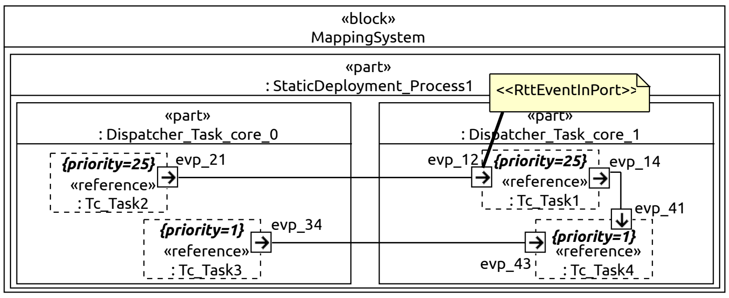
Multi-Core CPU (Static Task Partitioning)

Consider the mapping model in Figure 4.8a, where tasks are allocated onto a dual-core CPU running real-time Linux. Subsystem `ss4` is now mapped into a fourth task, `Task4`. An input/output relationship is added between subsystems `ss2` and `ss4`. These modifications imply that `Task4` executes after `Task1` and `Task3`. Assume that `Task1`, `Task2` and `Task3` have periods of 20ms, 10ms and 30ms, respectively, whereas `Task4` executes every 10ms (it oversamples the output of `Task1` and `Task3`).

Figure 4.8b shows the BSW model generated by the M2M transformations. For each core, one `Dispatcher` element is generated. Similarly to single-core platforms, all the `RttTaskContext` instances that correspond to *`Thread`*s are run by non periodic activities, and are added as peer components to the `Dispatcher` on their respective core. In the generated code, dispatchers execute at the highest priority in the system with a base period equal to the greatest common divisor of the periods of all the tasks executing on its core. Each dispatcher has a scheduling table and, at every cycle of execution, triggers the activation of peer components according to the order specified by the scheduling table itself.



(A) Mapping model (IBD view).



(B) BSW model (IBD view).

FIGURE 4.8. Mapping model and automatically-generated BSW model describing the deployment of subsystems and threads onto a dual-core processor.

However, in multi-core platforms, local priorities of tasks are not sufficient to guarantee the order of execution between components. As shown in Figure 4.8b, precedence constraints between components executing onto different cores are enforced by means of suitable event signals from predecessor to successor components on input event ports (`RttEventInPort`s). An event signal is also used to enforce precedence constraints between a low-rate producer and a high-rate consumer that run onto the same core, when the consumer must also wait for the completion of a task executing onto a different core.

Figure 4.9 shows activations and scheduling constraints (deadlines) of tasks. In the example, `Task1` needs to execute after the output update following the first instance of `Task2` and before the third instance of `Task2` completes. At the generated-code level, this is achieved by binding the input event port of component `Tc_Task1` to a callback function to handle the inter-core activation signal sent by `Tc_Task2` every two instances. `Tc_Task1` then accesses its inputs (in

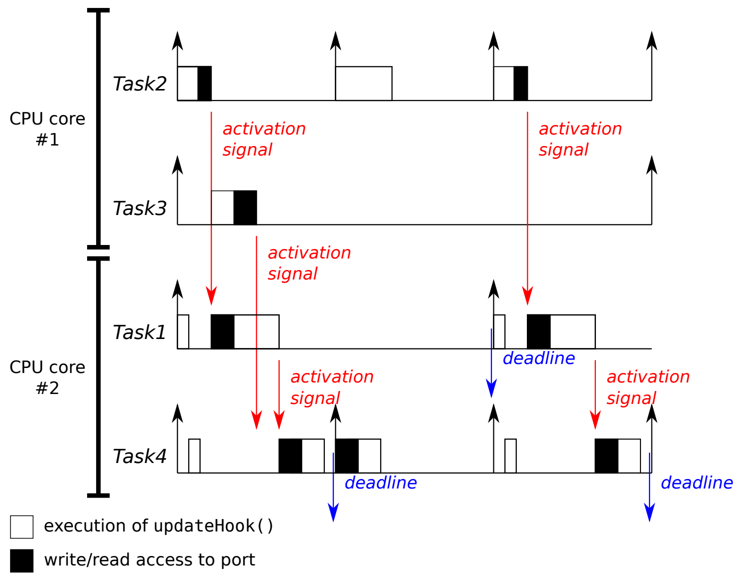


FIGURE 4.9. Threads activation constraints in the dual-core architecture.

```
// Callback function attached to the
// input event port of Tc_Task1
void reactTo_Tc_Task2(PortInterface* evp_12)
{
  Tc_Task2_ready = true;
}



// Task code of Tc_Task1
void updateHook()
{
  if (Tc_Task2_ready)
  {
    // do job
    generatedDoJob_Tc_Task1();

    // reset synch variable
    Tc_Task2_ready = false;
  }
}
```

(A) One producer on the same core.

```
// Callback functions attached to the
// input event ports of Tc_Task4
void reactTo_Tc_Task3(PortInterface* evp_43)
{
  Tc_Task3_ready = true;
}
...
// Task code of Tc_Task4
void updateHook()
{
  if((Tc_Task1_ready) || counter%2 != 0) &&
     (Tc_Task3_ready) || counter%3 != 0))
  {
    // do job
    generatedDoJob_Tc_Task4();
    // reset synch variables
    if(counter%2==0) Tc_Task1_ready = false;
    if(counter%3==0) Tc_Task3_ready = false;
    counter++;
  }
}
```

(B) Two producers on different cores.

FIGURE 4.10. Synchronization between producer/consumer Orocos-RTT components on a dual-core architecture.

`updateHook()`) only when the value of a synchronization variable is `true`, since this means that the activation has been triggered by `Tc_Task2` and not by the dispatcher, and that the signal data value is ready to be consumed. The callback function is executed first and sets the synchronization variable, as in Figure 4.10a. `Tc_Task1` then resets the variable in `updateHook()`.

Task4 must synchronize with Task1 and Task3, in order to use the correct data items when they are available. As in the previous case, activation constraints are handled at code level by binding callback functions to the input event ports of the reader component. Figure 4.10b shows the codes of callbacks and Tc_Task4's updateHook(). Tc_Task4 is triggered for the execution by the dispatcher component at a high rate; in order to synchronize with Tc_Task3, once every three activations, Tc_Task4 delays the access to its inputs until the synchronization variable Tc_Task3_ready is true, meaning that the correct data items produced by Tc_Task3 are available. The same approach is used to synchronize with Task1; in this case, Task4 reads the incoming data once every two activations triggered by the dispatcher. Task4 uses a counter variable (counter) to keep track of the number of activations, and resets the synchronization variables in updateHook().

4.4.2. Generation of Task Code

For each instance of (generated) RTT::TaskContext, M2T templates also generate the implementation of function generatedDoJob_Tc_*() in updateHook() (Figure 4.10) as a sequence of calls to the step() methods of the functional subsystems mapped into the component, serialized according to the mapping order.

As an example of generated code, with reference to the mapping model of Figure 4.8a and the updateHook() code in Figure 4.10a, the generatedDoJob_Tc_Task1() code is

```
// Generated 'do job' function for Tc_Task1
// via M2T
void generatedDoJob_Tc_Task1()
{
  step_ss5();
  step_ss6();
}
```

Note that the consistency of the partial order of execution among subsystems mapped onto the same task is guaranteed by the static order of the calls to the step() methods inside generatedDoJob_Tc_*().

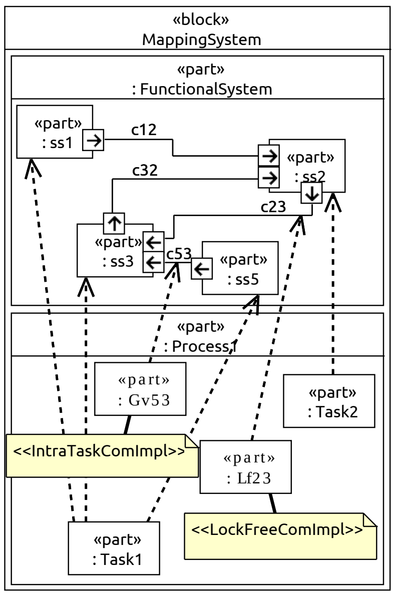
4.4.3. Implementation of the Functional Communication Links

The generation of the subsystems' behavioral code from Simulink Coder (bottom-left side in Figure 4.1) is redefined (using custom storage classes) in such a way that ports are not implemented and accessed inside the init() and step() methods as global variables, but using a simple and uniform MW-level API:

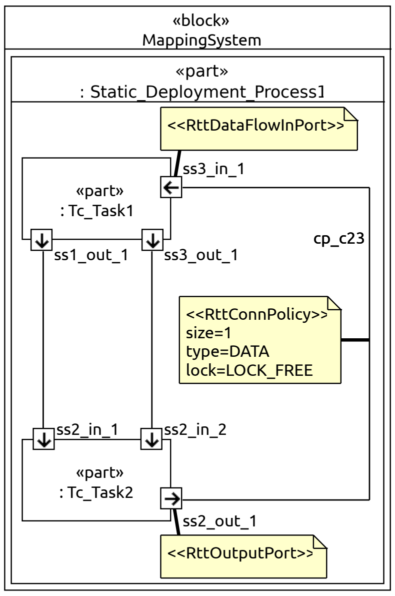
```
mw_read(block_id, port_id, value)
mw_write(block_id, port_id, value)
```

Acceleo M2T templates generate a concrete implementation for each macro invocation according to the nature of the communication.

Figure 4.11a shows a mapping model in which subsystems ss1, ss5 and ss3 are mapped to Task1, and subsystem ss2 is mapped to Task2. Assume that ss2 has an internal state and that the output does not depend on the inputs, so that the algebraic loop is eliminated. Tasks are deployed onto the same CPU (not shown). Inter-task communication signals (e.g., c23) are



(A) Signal mapping model (IBD view).



(B) BSW model (IBD view).

FIGURE 4.11. Mapping model of functional communication links into signal variables (*ComImpl*s), and automatically-generated BSW model.

mapped to `LockFreeComImpl` variables; the signal from `ss5` to `ss3` is mapped to one instance of `IntraTaskComImpl`.

M2M transformations generate the model of BSW implementation of Figure 4.11b. Lock-free inter-task communications are translated into streams of data (described by `RttConnPolicy` objects opportunely configured) that flow through the components' ports.

M2T templates generate the code of Figure 4.12 for the intra-task and inter-task communications. Acceleo scripts resolve the intra-task communication generating accesses to simple global data variables (of the same type as the connected data flow ports). Inter-task communications are realized by write/read (lock-free) accesses to the ports of connected `RTT::TaskContext` components. Port connections are realized through objects that specialize the class `RTT::ConnPolicy` and implement the semantics of RT blocks (cfr. Subsection 4.2.2). Each connection object is opportunely configured depending on the periods of sender and receiver `RTT::TaskContexts` (high-to-low or low-to-high rate transitions).

4.5. Summary

This Chapter presents a design methodology for complex robotics systems and the supporting tools for the realization of robot applications on single-/multi-core platforms, from the system-level modeling to the generated code. The development process integrates MDA and MBD paradigms. It enables the generation of a semantics-preserving implementation of robotics

```
// Middleware-level API
#define mw_read(block_id, port_id, value) \
            mw_read_##block_id##_##port_id(&value)
#define mw_write(block_id, port_id, value) \
            mw_write_##block_id##_##port_id(value)

// Implementation of intra-task communication
port_type ss5_out_1;

void mw_write_ss5_1(const port_type value)
{ss5_out_1 = value;}

void mw_read_ss3_2(port_type *value)
{*value = ss5_out_1;}

// Implementation of inter-task communication c23
extern RTT::TaskContext *pTc_Task1, *pTc_Task2;
#define mw_read_ss3_1(value)  tctask1_read_1(pTc_Task1, value)
#define mw_write_ss2_1(value) tctask2_write_1(pTc_Task2, value)

void tctask1_read_1(RTT::TaskContext *t, port_type *value)
{t->ports()->getPort("ss3_in_1")->read(*value);}

void tctask2_write_1(RTT::TaskContext *t, const port_type value)
{t->ports()->getPort("ss2_out_1")->write(value);}
```

FIGURE 4.12. Automatically-generated code for the platform-dependent realization of functional links.

controls based on Orocos-RTT, from a merger of Simulink and SysML/MARTE models defining the execution platform and the mapping, using open standards and transformation tools.

CHAPTER 5

Platform-Aware Control Simulations in Simulink Through Co-Simulation

5.1. Context and Positioning

Control applications that do not have strict safety requirements but are anyway subject to demanding timing constraints are said *performance-sensitive* (cfr. Section 3.1). Performance-sensitive applications are traditionally designed in Simulink, where control engineers define the controls functionality and the model of the controlled system, and verified according to the synchronous-reactive (SR) paradigm, in which all the computations and communications are assumed to complete within the interval between two events in logical time. When the controls are implemented in software and execute on a real architecture of CPUs and communication links, computation, scheduling and communication delays may exceed what is prescribed by the synchronous assumption and the jitters and latencies may affect the performance that were validated in simulation. The impact of these delays is often evaluated late, at testing time, with significant costs, additional development cycles and possible changes to the hardware architecture.

An early evaluation of the impact of the hardware and software implementation is desirable and requires the co-simulation of the controller functionality, the plant model, and the computation, scheduling and communication hardware and software platform, together with a model of the software tasks and the messages exchanged over the networks. The T-Res open framework [**CMDN15**] enables such a co-simulation in the Simulink environment.

T-Res has a modular and extensible architecture and offers the following unique features.

- The addition of task, scheduler, network and message implementation models to an existing Simulink model with limited and localized changes; the modeling structure clearly separates the controller model from the model of task, scheduler, communication mechanisms and other attributes of the execution platform.
- The modular integration of third-party real-time scheduling and network simulators, through simple and generic interfaces; T-Res includes bindings to the open-source simulators RTSim [**RTS**] and OMNeT++ [**OMNb**].
- The automatic back-annotation in Simulink of task, scheduler, network and message models from external formal specifications in SysML.

Figure 5.1 represents the portion of proposed system-level design flow where T-Res plays a key role. Simulink models are used to define the functionality of the controls and SysML models define the hardware execution platform and the task model of the controls implementation. After the functionality is mapped for execution on the platform model, defining the structure

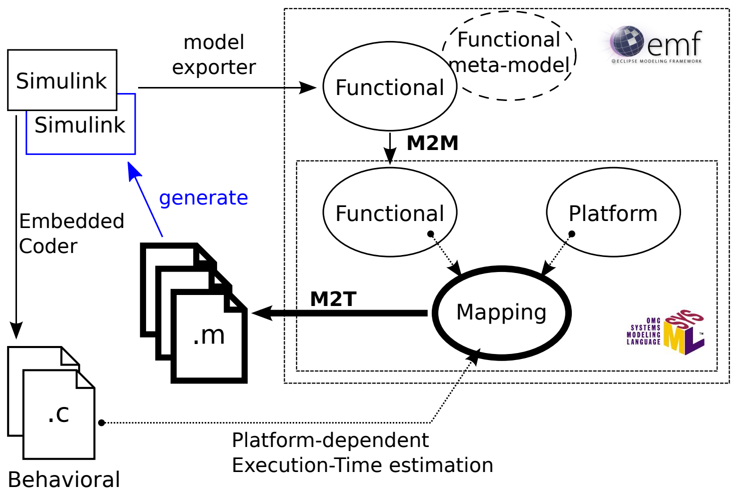


FIGURE 5.1. T-Res in a Simulink-based PBD-like flow for real-time distributed embedded control systems development.

of the tasks and messages, the execution and transmission times are estimated (or measured). The Simulink model can be annotated with blocks that allow the simulation of scheduling, computation and communication latencies, allowing to fine tune the control logic, to actively compensate these timing effects, or the task and message model (possibly with their priorities), to evaluate different implementation options.

The purpose of this chapter is twofold. First, it presents T-Res and explains how platform-aware controls can be simulated in Simulink (blue item in Figure 5.1). Second, it explains how Simulink models are annotated with T-Res models of task, scheduler, network and message implementations, automatically generated from the mapping model (bold items in Figure 5.1).

The rest of the chapter is organized as follows. Section 5.2 complements the description of the SR semantics presented in Section 4.2 and introduces concepts that are of paramount importance for the implementation of a co-simulation framework on top of Simulink, such as the time points at which Simulink computes the states and outputs of the system, and the mechanism for extending the capabilities of the Simulink environment. Section 5.3 introduces platform's discrete-events simulators and formalizes the execution models of real-time tasks and network communication. Section 5.4 presents T-Res, its software architecture as well as the designed Simulink blockset and the adopted design patterns enabling the easy integration with other third-party real-time scheduling and network simulators. Section 5.5 describes the application of T-Res onto two cases studies, the real-time control of three networked DC servo motors and of an aerial robot. Section 5.6 explains the back-annotation process. Finally, Section 5.7 summarizes and closes the chapter.

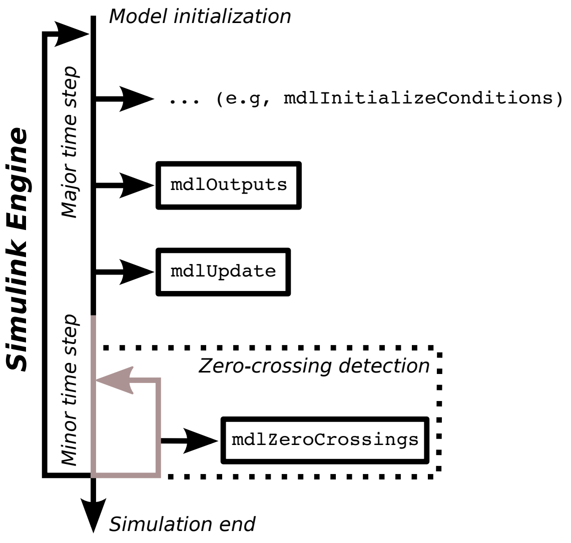


FIGURE 5.2. Simplified view of S-function callback methods invoked by the Simulink engine during the simulation loop[1].

5.2. How the Simulink Engine Simulates a Dynamic System

The Simulink engine computes the states and outputs of the system at time points (*time steps*) from the simulation start time to the finish time, using information provided by the model. This process is called solving a model. The length of time between steps is called step size. Numerical solvers solve a model at fixed or variable time steps. Fixed-step solvers do it at regular time intervals from the beginning to the end of the simulation. Variable-step solvers vary the step size during the simulation, and are invoked at those points in time that are relevant for the dynamics of the system they solve. Variable-step solvers divide the simulation time span in *major* and *minor* time steps. The solver produces a result at each major time step.

Any point in time that is relevant for the dynamic of controller or controlled system corresponds to a major step. For example, all the triggering instants of discrete-time (controller) subsystems correspond to major steps. A minor time step is a subdivision of the major time step used to improve the accuracy in the computation of the continuous-time system dynamics. Minor time steps are also used to find the point in time where continuous-time system have a *zero-crossing* point, that is a point when some of the state variables cross a zero threshold (indicating a significant change of state for the system dynamics). At any point in time corresponding to a major step, blocks are evaluated.

Simulink system-functions (S-functions) are a mechanism for extending the set of predefined Simulink blocks. An S-function is a computer language description of a Simulink block behavior written in Matlab, C, C++ or Fortran. Interactions between the Simulink simulation engine and custom blocks occurs through a predefined set of API functions. Figure 5.2 shows a simplified

[1]Reproduced from `http://www.mathworks.com/help/simulink/sfg/how-the-simulink-engine-interacts-with-c-s-functions.html`

view of the simulation cycle at run-time with the major and minor steps, and the points in the cycle in which the simulation engine invokes the API functions specified for the S-function custom block. Among those, the `mdlOutputs` is used to update the outputs of the custom block, the `mdlUpdate` to update the internal state of the custom block and `mdlZeroCrossing` to define the signals that determine the zero-crossing points and possibly use them to set time instants for future major steps.

5.3. Platform Simulators and Execution Models

5.3.1. Discrete-Event Platform Simulators

Real-time (RT) scheduling and network simulators are Discrete-Event Systems (DESs). They implement an event handling mechanism, typically with a queue. Events can arrive asynchronously or periodically and are ordered in the event queue in ascending order, following (*i*) the event occurrence time, and (*ii*) a causality order for those with equal occurrence time. Events are processed sequentially at every simulation step. Processing an event may generate another bunch of events to be executed at the current time or in the future.

A RT scheduling simulator reacts to tasks arrival events and dispatches the currently active tasks from the ready queue according to a fixed or dynamic priority-based scheduling algorithm. To preserve causality among events, a task is dispatched only when all the events at the current time have been processed. At any point in time, the next scheduling event can be the termination of the task currently in execution, or the arrival event of a task, that can possibly cause a preemption (if the new task has higher priority) and a context switch.

A network simulator simulates nodes exchanging messages over a network infrastructure with a given communication protocol. Similarly to what happens to tasks in a RT scheduling simulator, a message is dispatched only when all the events at the current time have been processed, so that the causality among events is preserved. The network simulator defines the timed events related to the transmission and arrival of messages by the networked nodes. The communication protocol is the core attribute of the communication network. It defines the set of rules according to which messages are selected for transmission on shared physical links and ultimately determines the latency of messages together with the attributes that define the network speed and reliability. It is therefore important that a network simulator supports a large set of protocols and can be easily extended to include new protocols.

5.3.2. Execution Model of Real-Time Tasks

In an RT simulator, tasks execute according to a model of time-consuming computation. We assume a model suited to the typical code generation process for Simulink models (which is also the same as in TrueTime). The execution of a task is split in preemptable units called *segments*, informally corresponding to the execution of a function called by the task main code. Each segment is identified by an execution time (possibly randomly generated according to a given distribution) and all segments in a task are executed in a sequence.

When the RT simulator is integrated with Simulink, segments map one-to-one to subsystems. Their execution order in a task must match the order of execution imposed by the model

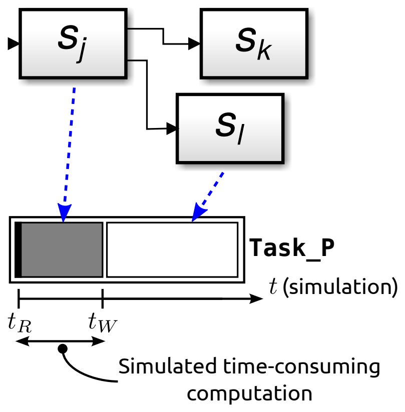


FIGURE 5.3. Execution model of a simulated real-time task in Simulink.

semantics, as in Figure 5.3, where two control subsystems (S_j and S_l) in a producer-consumer relationship are mapped into the same task (`Task_P`).

A segment interacts with other segments and/or the controlled system at the beginning and at the end of its simulated execution. With reference to Figure 5.3, t_R and t_W represent the time instants at which the subsystem S_j reads the values on its input ports and writes the results of computation on its output ports, respectively.

The (simulated) time duration of each task segment, corresponds to the execution time of the corresponding code function implementing the subsystem (and possibly generated from it in an automatic code generation flow). Figure 5.3 visualizes the time duration of segment S_j, as the dark gray box in between time instants t_R and t_W. Note that the input-output latency of S_j is *at least* $t_W - t_R$. In a simulated task set where other higher-priority tasks preempt `Task_P` while it is executing S_j, the actual input-output latency will be longer.

The following points describe formally the assumptions on the functional model, considering a strict subset of the possible Simulink semantics.

- $\mathcal{V} = \{S_1, ..., S_{|\mathcal{V}|}\}$ is the set of *functional subsystems* in the Simulink model. Subsystems can be continuous time, modeling the controlled physical system (or plant), or discrete-time, representing the controller logic. Subsystems may have input and output ports. They read or sample the inputs when start executing and generate the outputs instantaneously, according to the logical zero-execution time assumptions prescribed by the SR semantics.
- The discrete-time controller model is partitioned into single rate subsystems (different subsystems may have different rates). The sampling period t_i denotes the activation period of the control subsystem S_i. All input ports carry signals with a uniform sampling period t_i. The result of the block computation is a set of signals with the same rate, produced on the output ports.
- $\mathcal{E} = \{l_1, ..., l_{|\mathcal{E}|}\}$ is the set of *links*. A link $l_i = (S_h, S_k)$ connects an output port of subsystem S_h (source) to an input port of S_k (sink).

- A precedence relation may exist between a pair of subsystems S_i and S_j. The notation used is $S_i \prec S_j$.

The model of task execution, supported by the RT simulator, is formally described as follows.

- $\mathcal{T} = \{\tau_1, ..., \tau_{|\mathcal{T}|}\}$ is the set of tasks. Each task τ_i has an activation period T_i or activation event e_i and an optional priority π_i (or other scheduling attributes). Periodic tasks may have an activation offset; when the offset is zero, they start at the same time instant $t = 0$.
- $\mathcal{C} = \{c_1, \cdots, c_{|\mathcal{C}|}\}$ is the set of single-/multi-core computer nodes that make up the real-time computing platform.
- A mapping relation $mt(\tau_i, c_j)$ is defined between tasks and computer nodes, meaning that task τ_i executes on computer c_j according to a defined scheduling policy.
- A mapping relation $mS(S_i, \tau_j, k)$ is defined between a controller subsystem S_i and a task τ_j meaning that the code implementing S_i is executed in the context of task τ_j with order index k. We assume that the code implementing S_i is characterized by a worst case execution time γ_i on the computer node where τ_j runs. A mapping relation is only possible if the execution rate of S_i and τ_j are the same (the constraint could be relaxed allowing for integer divisors).

5.3.3. Execution Model of Network Communication

The network model is inspired by the OSEK COM Interaction Layer (IL) [**OSE04**]. We consider a limited subset of the IL services, most notably non-blocking (remote) transmission of periodic data streams, and support for multiple channels in transmission and reception (with multiplexing and demultiplexing of information).

In OSEK COM, the IL uses the concept of message objects to represent the communication signals at application level (for local as well as remote communications). In the case of remote communication the IL packs one or more message objects into assigned *Interaction Layer Protocol Data Units* (I-PDUs), and passes them to the underlying layer. I-PDUs may contain data from one or more sending objects and can be received by zero or more CPUs. A receiving CPU reads the I-PDU data content and forwards it to the destination receiving objects, where the data values become available to the application software.

We assume a ***periodic transmission mode***, where I-PDUs are transmitted periodically by the IL without the need of an explicit request. Figure 5.4 shows an example of periodic transmission mode for I-PDUs. The I-PDU is transmitted periodically to the lower layers, independently from the time at which the contents of the sending objects are updated. Some values may be transmitted twice (e.g., v_1 in the figure), or may be overwritten (v_2) and never transmitted over the network. The periodic transmissions of I-PDUs is assumed to be realized by code called periodically at a fixed time interval and executed as part of a task in a multi-tasking OS. The task is called `TxTask`. Finally, we assume ***unqueued receiving message objects***, that can be read multiple times (and are not consumed) and can be overwritten by newly arrived messages.

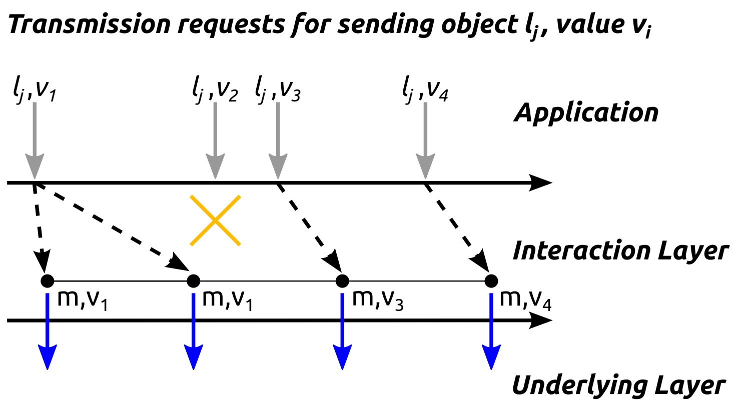


FIGURE 5.4. Timing for the transmission of periodic mode messages objects.

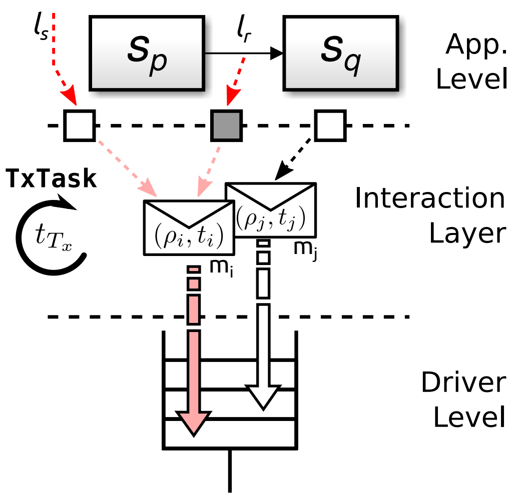


FIGURE 5.5. The middleware task `TxTask` executes with period t_{T_x}, reads message objects and enqueues messages at the driver level.

When the described network communication model is integrated with Simulink, message objects map one-to-one to communication signals between subsystems that model a distributed functionality. More precisely, these subsystems represent functions in application tasks executed onto different nodes of a distributed computing platform and exchanging data.

Figure 5.5 shows the `TxTask` reading application signals and enqueuing I-PDUs into messages at the driver level. Simulink subsystems are implemented as functions executed within application tasks, that run onto different nodes. Application tasks copy the data values for all the signals that need to be transmitted in variables shared with the `TxTask`. Inside the IL, the `TxTask` is activated periodically and, at the end of its execution, calls the underlying layer function for the transmission of the I-PDU. I-PDUs map one-to-one to messages at the driver level, when the I-PDU size is less than the maximum size allowed for messages.

The network communication model, supported by the network simulator, is formally described as follows.

- $\mathcal{M} = \{m_1, \cdots, m_{|\mathcal{M}|}\}$ is the set of messages.
- $\mathcal{N} = \{n_1, \cdots, n_{|\mathcal{N}|}\}$ is the set of networks.
- A mapping relation $mm(m_i, n_j)$ is defined between messages and networks, meaning that message m_i is transmitted over the network n_j.
- Each message m_i has an associated transmission period t_i, expressed as an integer multiple of the `TxTask` period t_{T_x}, i.e., $t_i = k_i t_{T_x}$, and a priority ρ_i.
- A mapping relation $ml(l_i, m_j)$ is defined between links and messages, meaning that the data of the signal exchanged over l_i are mapped onto message m_j. Each link l_i can be mapped onto at most one message. Many-to-one mappings, i.e., signal multiplexing, are allowed (see Figure 5.5). If l_i is not mapped to any message, then its implementation consists of local communication (typically a shared variable).

5.4. The T-Res Co-Simulation Environment

5.4.1. Architecture

Co-simulation enables the execution of several simulators concurrently. T-Res adopts a master-slave model of co-simulation, and its architecture is represented in Figure 5.6.

Simulink is the master simulation engine and instances of the slave simulators (the discrete-event platform simulators) are encapsulated into two dedicated S-functions, `Kernel` and `Network`. `Kernel` models a single-/multi-core computer node, $c_h \in \mathcal{C}$, running a real-time kernel and executing tasks and interrupt handlers according to a given scheduling policy. `Network` models a physical communication medium, $n_i \in \mathcal{N}$, in which messages between connected computer nodes are exchanged according to a network protocol. Two other S-functions, `Task` and `Message`, complete the T-Res Simulink blockset (light-gray blocks). `Task` models a task $\tau_j \in \mathcal{T}$ executed in a computer node specified by the mapping mt. `Message` models a message $m_k \in \mathcal{M}$ exchanged by computer nodes over a network specified by the mapping mm.

A software architecture of tasks and messages onto a distributed computing platform is modeled in Simulink by multiple instances of `Kernel`, `Network`, `Task` and `Message` S-functions, as in Figure 5.6.

`Kernel` and `Network` blocks execute at all major steps (time points at which the Simulink solver produces a result, cfr. Section 5.2). At every invocation, they force the slave simulators to process all the events at the current time, and update their internal structures to reflect the computed scheduling of tasks and messages (bi-directional blue arrows). Then, they command the execution of the scheduled `Task` and `Message` blocks (dashed-black arrows). Finally, they determine the times of next events to be simulated, and use the zero-crossing API call of Simulink to define new major steps (green arrows).

For the execution of the `Kernel` S-function, the major steps of the Simulink simulation must include all the periodic activation times of tasks as well as the aperiodic events that lead to the activation of other tasks. Major time steps are also defined in correspondence of execution

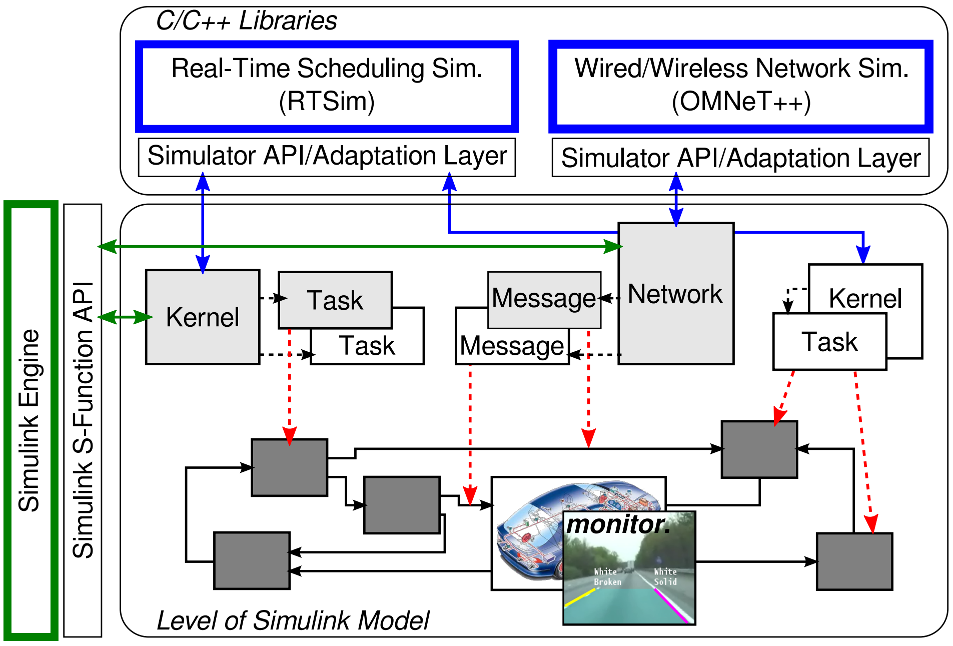


FIGURE 5.6. Co-simulation of the functional controls (e.g., ADAS, dark-gray blocks), the plant (car) and the task scheduling and network communication parts (light-gray/white blocks), in a simplified representation.

completions, of tasks and/or segments. For the execution of the `Network` S-function, the major steps must include all the periodic activation times of messages, the events that lead to aperiodic transmissions of messages and message arrival events.

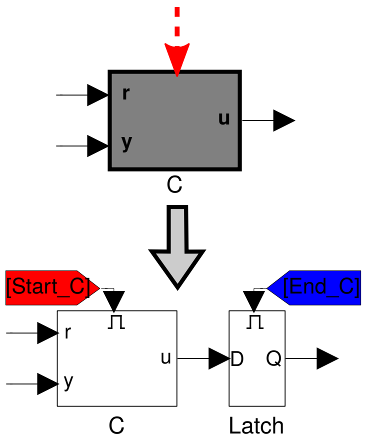
To guarantee the execution models of real-time tasks and network communications (cfr. Section 5.3), the *zero*-execution/communication time semantics of Simulink must be replaced with the *finite*-execution/communication time assumption. Dashed-red arrows in Figure 5.6 represent the realization of this concept in a simplified view. Their meaning is that `Task` and `Message` blocks, respectively, command the execution of the mapped segments (specified by the mapping mS) and enable the data-flow onto the communication signals (specified by the mapping ml) to add the delay effects due to finite computation times, scheduling and network protocol and traffic to the Simulink simulation.

There are a number of possible options to implement finite-execution/communication time in Simulink through the external activations of block executions and signal data-flows. The solution adopted here, described at high level in the next subsection, achieves a good trade-off between simplicity, effectiveness and efficiency.

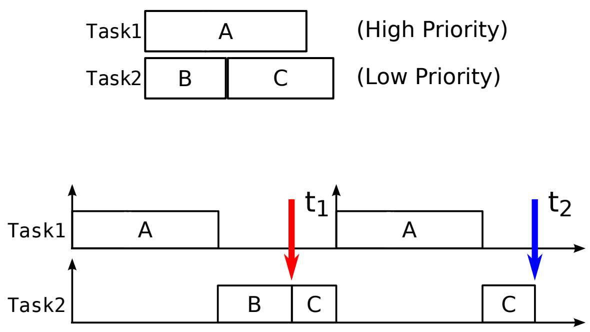
5.4.2. Simulink Implementation of Platform Execution Models

Implementation of Real-Time Tasks Execution Model

The start and completion times of the segments correspond, respectively, to the times in which the corresponding subsystems read (sample) their inputs and produce their outputs. To



(A) Segment in Simulink.



(B) Conceptual view of the scheduling trace for a simple task set.

FIGURE 5.7. Simulink implementation of time-consuming task computations execution model.

force inputs sampling at precise time steps, the activation of the Simulink subsystems is changed from periodic to triggered[2]. To synchronize in Simulink the completion of the segments with the production of the output values, a latch barrier is added on all their outputs. The bottom side of Figure 5.7a represents the actual implementation of activation/termination mechanisms of a Simulink subsystem turned into a RT task segment, with respect to the simplified view in the top side (introduced first in Figure 5.6).

These mechanisms enable the implementation of finite-execution time semantics on top of Simulink. With reference to Figure 5.7b, the top side visualizes conceptually a simple task set and the computational activities of its two task, namely `Task1` and `Task2`. The subsystem `C` executes as second segment of `Task2`. Tasks are scheduled according to a Fixed-Priority (FP) policy, and `Task1` has the highest priority. The execution trace of tasks is depicted on the bottom side.

When $t = t_1$, `C` starts executing and the subsystem is activated. Inputs are sampled and the output value $u = C(r, y)$ is produced *instantaneously* (according to the synchronous semantics). However, u is not available to other blocks until the activation of the latch barrier. The simulation continues until $t = t_2$, when `C` terminates and the latch barrier is activated. The value $Q = C(r, y)$ is then available to consumer blocks after a *finite* time, that accounts for the execution times of computations (segments) and scheduling delays (`Task1` preempts `Task2` and delays the production of the output value of `C`).

[2]This is actually realized through an enable signal converted into a function-call inside the subsystem. Full in-depth technical description is at `https://github.com/m-morelli/tres_bundle/wiki/Triggered-Activation-of-a-Simulink-Subsystem`.

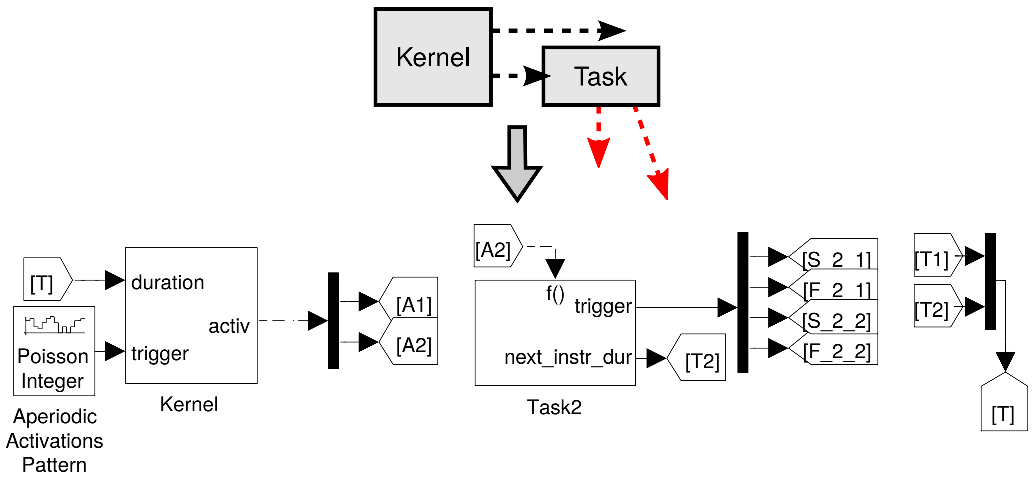


FIGURE 5.8. Instances of T-Res blocks for the representation of kernel and tasks (bottom side), with respect to the simplified view in Figure 5.6 (top side).

Technically, it is possible to simulate very fine-grained details, provided that the slave RT simulator supports them, such as the delay effects due to the use of caches, context switches and migration of tasks among CPUs.

The signals activating a subsystem (and its input sampling) and its output latch are generated by the corresponding `Task` block (i.e., the block representing the task where the subsystem is mapped into) upon the beginning of the execution and the completion of the segment. `Task`s are triggered subsystems, executed on the occurrence of a function-call event issued by the corresponding `Kernel` (i.e., the block representing the computer node where the task is deployed onto). Figure 5.8 shows the interface of `Kernel` and `Task` blocks, and the interaction among them. These particular block instances implement (a portion of) the scenario in Figure 5.7b.

`Task`'s output interface consists of two ports: `trigger` and `next_instr_dur`. The first one is an array of data-flow signals with size equal to twice the number of segments mapped into the `Task`. This port is used to command the activation of subsystems and the output latches. The second port outputs a scalar signal representing the duration of the next segment executed by the `Task`. This information is transmitted to `Kernel` each time `Task` is triggered. Whenever there is no other segment to be executed, `Task` outputs a special code on the port `next_instr_dur`, which is interpreted by the `Kernel` as a *task completion* signal. The duration of segments executed by `Task` is set through a variable in the Matlab workspace, specified to the `Task` block through its mask dialog.

The block `Kernel` has two input ports: `duration` and `trigger`. On the `duration` port it receives the indication of the durations of the next segments to be executed, one for each `Task` block. On the second port, it receives the array of activation signals of aperiodic tasks (from external sources). The block has one output port, named `activ`, which is used to signal to each task the execution of the current segment. The block `Kernel` uses a zero-crossing function to require future activations (from the Simulink engine) in correspondence of scheduled events, but it is also activated synchronously with the arrival of aperiodic tasks.

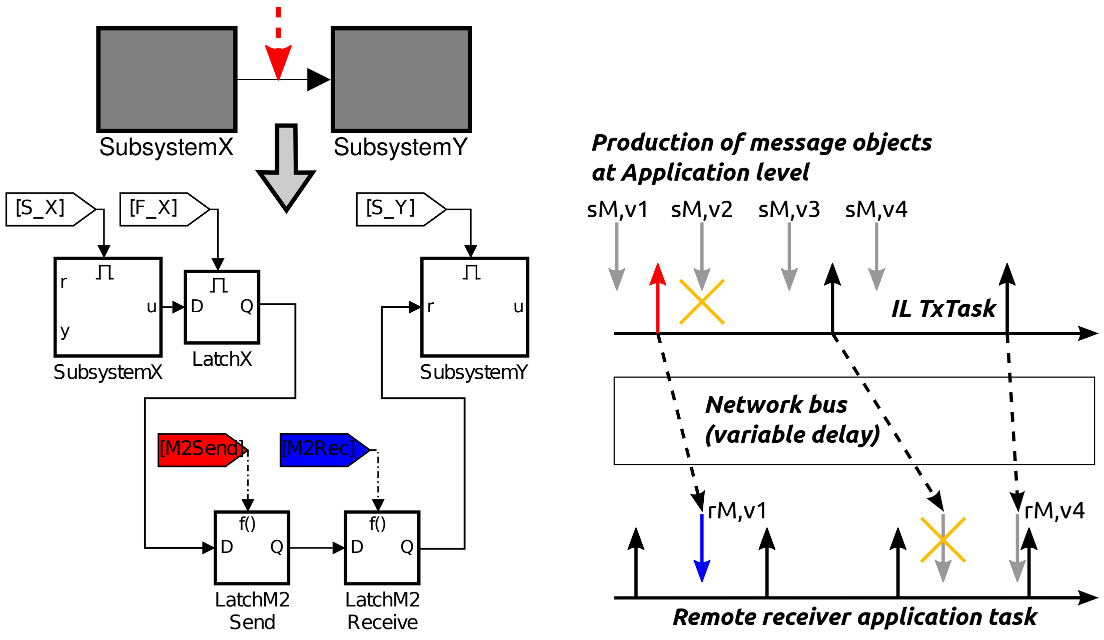
At each activation, `Kernel` checks its `trigger` port for any aperiodic requests. If there is any, the corresponding aperiodic tasks are activated in the slave RT simulator at the current time (in synch with the Simulink time). Next, `Kernel` looks for the next event in the slave simulator's event queue. Two types of events are relevant: the *segment completion* and *task completion*. In case an event of the first type occurs, `Kernel` reads the input signal on the port `duration` at the index corresponding to the task that completed the segment, and dynamically creates a new instruction and insert it in the corresponding task. Finally, once it detects which task has generated the event, it sends an activation signal to the `activ` port to trigger the corresponding `Task`. If the event *task completion* is detected, `Kernel` simply resets the internal state of the corresponding task clearing the past history of the executed segments.

A number of parameters configure the (simulated) kernel and are set through the `Kernel` mask dialog, such as the scheduling policy, the number of cores on which the task execution is simulated, and the type and the timing properties of the (heterogeneous) task set. Tasks can be periodic or aperiodic and timing properties include interarrival time, relative deadline and initial offset. Optionally, task priorities (for tasks scheduled according to FP) and core affinities can be specified for each task.

Implementation of Network Communication Execution Model

The times when a transmission IL task packs a message object into an I-PDU correspond to the times in which the corresponding functional signal starts being transmitted to a remote node. On the receive side, the `Rx-Interrupt` handler directly unpacks a message object from the received message at the driver level, and the times when the application task reads the message object correspond to those in which the reception of the corresponding functional signal sent by a remote node is completed. To enforce in Simulink the consistency of information flow with respect to the time instants when packing/unpacking of message objects occur, a double latch barrier is added on functional signals that correspond to network communications (bottom side of Figure 5.9a). This mechanism enables the implementation of finite-communication time semantics on top of Simulink.

Figure 5.9 show an example of network transmission of a functional signal from `SubsystemX` to `SubsystemY`, that execute as segments in two application tasks running on different computer nodes. At a certain instant, the sender application task updates the content of the sending message object sM with the value v_1. At its next activation, the IL `TxTask` packs sM into an I-PDU and passes it to the underlying layer. In the Simulink model, this translates into the activation of the first latch barrier (`LatchM2Send`). The corresponding link is sampled, but the signal is not instantaneously propagated to `SubsystemY`. The signal is gated by the second latch barrier that blocks it, until the message reaches the receiver node. The time required to deliver the message can vary, and depends on the underlying protocol and the network traffic. When the message is received, the data is copied to the receiving message object rM. In Simulink, the second latch barrier (`LatchM2Receive`) is activated and the value v_1 becomes available to `SubsystemY`, that will read it at the next activation. Information can be lost on the transmit



(A) Message transmission in Simulink (cfr. Figure 5.6).

(B) Transmission of information signals between remote application tasks.

FIGURE 5.9. Simulink implementation of finite-communication time network execution model.

side, when the content of the sending message object sM is updated before it is read by the IL `TxTask` (v_2). Information can also be lost on the receive side, when the message object rM is overwritten before it is read by the receiver application task (v_3). The simulation enables the investigation of the effects that message losts have on the control performances.

Technically, it is possible to simulate very fine-grained details, provided that the slave network simulator supports them, like finite copy-times and queuing policy at the adapter level.

For each functional signal transmitted over the network, the activations of the 2-latch barrier are generated by the corresponding `Message` block (i.e., the block representing the message where the functional signal is mapped into). `Message`s are triggered subsystems. A `Message` executes upon detection of a rising edge signal on its standard input port, issued by the corresponding `Network` (i.e., the block representing the network over which the message is transmitted). Figure 5.10 shows the interface of `Network` and `Message` blocks, and the interaction among them.

`Network` receives a specification of all the messages in the system (transmitted by IL `TxTask`s) through its mask dialog. The network topology and a number of configurations of computer nodes (.e.g,) are also specified through the mask dialog. In the model initialization phase (cfr. Figure 5.2), `Network` collects all this information and initializes the slave simulator accordingly.

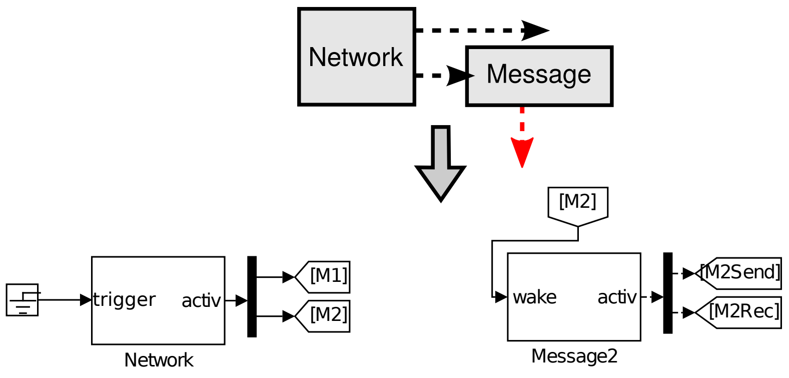


FIGURE 5.10. Instances of T-Res blocks for the representation of network and messages (bottom side), with respect to the simplified view in Figure 5.6 (top side).

`Network` uses the `mdlZeroCrossings()` function to specify future activations in correspondence of scheduled events, but it is also activated synchronously with the arrival of aperiodic messages.

During the simulation loop, at every activation, `Network` checks for any aperiodic requests and (eventually) activates the corresponding aperiodic messages in the slave simulator. Next, it looks for the next event in the slave simulator's event queue. When a start of message transmission event is active, the `Network` block triggers the corresponding `Message` block for sampling the signal values that are transmitted with the message. When the message arrives, another signal is sent to `Message` to indicate that the signal values are now available at their destination and ready to be used by the reading subsystems.

5.4.3. Interface to Other Platform Simulators

`Kernel` and `Network` blocks are designed according to principles of object-oriented programming to provide an easy integration with, potentially, any external real-time scheduling and network simulator, respectively. Figure 5.11 provides a detailed view of the organization of the C++ T-Res software architecture represented in Figure 5.6 (top).

`Kernel` and `Network` S-functions access the external simulators through dedicated abstraction layers (API). The design of the abstraction layers is based on the observation that real-time scheduling and network simulation frameworks basically consist of two high level components: an event handling system and, on top of it, the actual scheduling or network simulator. The first component is the DES that represents and manages events and event queues, and provides an API that enables the creation and deletion of events and their insertion in and extraction from the event queue. The second component uses the definitions of events (typically after specialization) and event queues and realizes the actual real-time scheduling or network simulation functionality. The real-time scheduling simulator implements the concepts of task (e.g., periodic, aperiodic), scheduling policy (e.g, DM, FP, EDF), and kernel (e.g., single- or multi-core, with

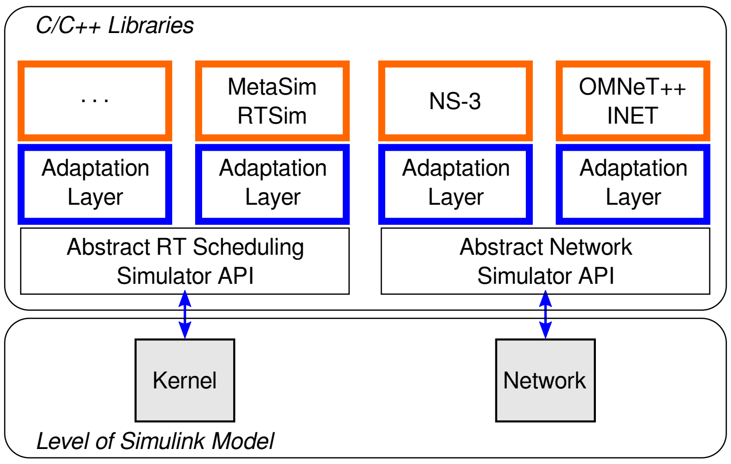


FIGURE 5.11. Extensible architecture for interfacing T-Res with other platform DES simulators (closer view of top side of Figure 5.6).

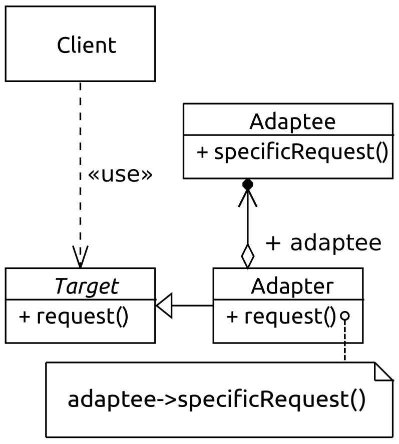
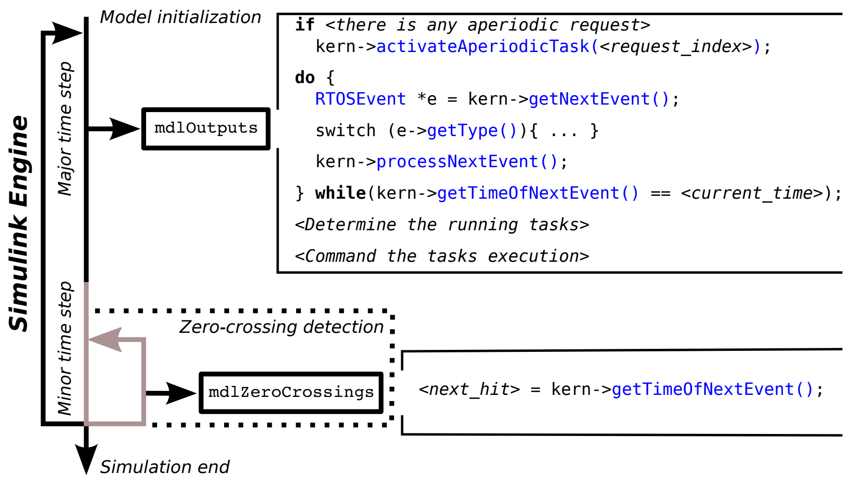


FIGURE 5.12. Implementation of software architecture of Figure 5.11 using the object adapter design pattern.

a scheduler and a resource manager). The network simulator implements the concepts of message (e.g., periodic, aperiodic), transmission protocol (e.g., Ethernet frame, CAN message, UDP socket) and network-connection topology.

The designed *RT Scheduling Simulator API* and *Network Simulator API* layers (Figure 5.11) abstract the high level simulator concepts and enable the development of the `Kernel` and `Network` S-functions so that they depend upon a set of interfaces classes, rather than upon their concrete implementations. The RT Simulator API layer defines three interface classes, *`tres::Kernel`*, *`tres::SimTask`* and *`tres::RTOSEvent`*. Similarly, the Network Simulator API layer defines *`tres::Network`*, *`tres::SimMessage`* and *`tres::NetworkEvent`*.



```
if <there is any aperiodic request>
  kern->activateAperiodicTask(<request_index>);
do {
  RTOSEvent *e = kern->getNextEvent();
  switch (e->getType()){ ... }
  kern->processNextEvent();
} while(kern->getTimeOfNextEvent() == <current_time>);
<Determine the running tasks>
<Command the tasks execution>
```

```
<next_hit> = kern->getTimeOfNextEvent();
```

FIGURE 5.13. Simulation loop of `Kernel` S-function (pseudo-code).

The *object adapter pattern* [**GHJV95**] is used to bind the interface classes, used by the S-functions, to the third-party simulators, as shown in Figure 5.12. `Clients` are the S-functions `Kernel` and `Network`, which access the *`Target`*s (the interface classes). Each *`Target`* plays in fact the role of an `Adapter`, that redirects the requests of the client to the real `Adaptee` object instance in the target simulator.

The following is the set of virtual methods to be specialized by the refinement of the *`tres::Kernel`* class, the main class realizing the adaptation to the real-time scheduling simulator. A similar set of virtual methods are provided for the *`tres::Network`* class (that are not described for conciseness).

```
// List of virtual methods of tres::Kernel
virtual void initializeSimulation(const double, const double* const*) = 0;
virtual void processNextEvent() = 0;
virtual tres::RTOSEvent* getNextEvent() = 0;
virtual int getTimeOfNextEvent() = 0;
virtual int getNextWakeUpTime() = 0;
virtual void getRunningTasks() = 0;
virtual void activateAperiodicTasks(std::vector<int>&) = 0;
```

The pseudo-code in Figure 5.13 shows how the `Kernel` S-function uses its interface classes during the interaction with the Simulink simulation loop. The actual implementation of methods of each interface class is demanded to the object adapter, which calls adaptee operations to actually carry out the requests. Conceptually similar interaction with the Simulink simulation loop is performed by the `Network` S-function, and is omitted for brevity.

In order to create an instance of a interface class without making the S-Functions depend upon its concrete implementation, the *factory method pattern* [**GHJV95**] is used. Each adapter defines a method called `createInstance()`, which takes a `std::vector` of `std::string` objects as input argument. The `std::string` objects describe a specific configuration for the adapter to be instantiated and are provided by the user through the S-function mask. Adding adapters for new external simulators is easy and just requires to register the factory method of the new

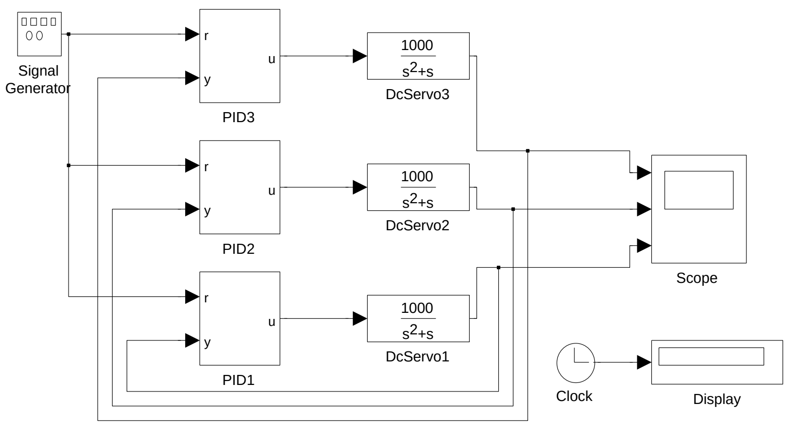


FIGURE 5.14. The application example from TrueTime [**CHL+03**], PID control of three DC-servo systems.

adapter class to a generic factory class. It does not require any modification to the code of the factory.

5.5. Application Examples

5.5.1. PID Control of Three Networked DC-servo Systems

Figure 5.14 shows a Simulink model (adapted from the TrueTime example library) in which three DC-servo systems are controlled with *Proportional*, *Integral*, *Derivative* (PID) control. Each DC-servo is described by a continuous-time SISO transfer function (a `TransferFcn` block), and is controlled by a dedicated discrete-time PID regulator. The three PID controllers have the same loop-gain coefficients K_p, K_d, K_i, and are modeled as (masked) subsystem blocks. A standard `SignalGenerator` block produces a square-wave shaped reference signal for the controllers.

The example considers the case of three periodic control tasks, namely, `Task1`, `Task2`, `Task3`, running concurrently on a single CPU. Tasks have different periods, respectively equal to $6ms$, $5ms$ and $4ms$. Each task executes the PID control logics of one regulator subsystem (the i-th task, `Taski`, executes the i-th regulator subsystem `PIDi`). We assume that each segment of control code takes a *fixed* amount of time equal to $2ms$ to execute, giving a total CPU load higher than 100% (an overload condition).

In addition, the position of the motors is read by sensors and sent to the PIDs using a periodic CAN message `Message2`. Another periodic CAN message `Message1` collects all the command signals from the controls and forwards them to the motors. The bit-rate of CAN bus is $1Mbps$. The period of the two messages is $4ms$. `Message2` (the message with the sensor data) has higher priority.

Figure 5.15 shows the original model back-annotated with T-Res blocks. One instance of `Kernel` and three instances of `Task` blocks are added to the functional model (blue blocks on the left). Since the example does not consider aperiodic tasks, a `Ground` block is connected to the `trigger` port of `Kernel1`. Each PID subsystem is transformed to a triggered subsystem and

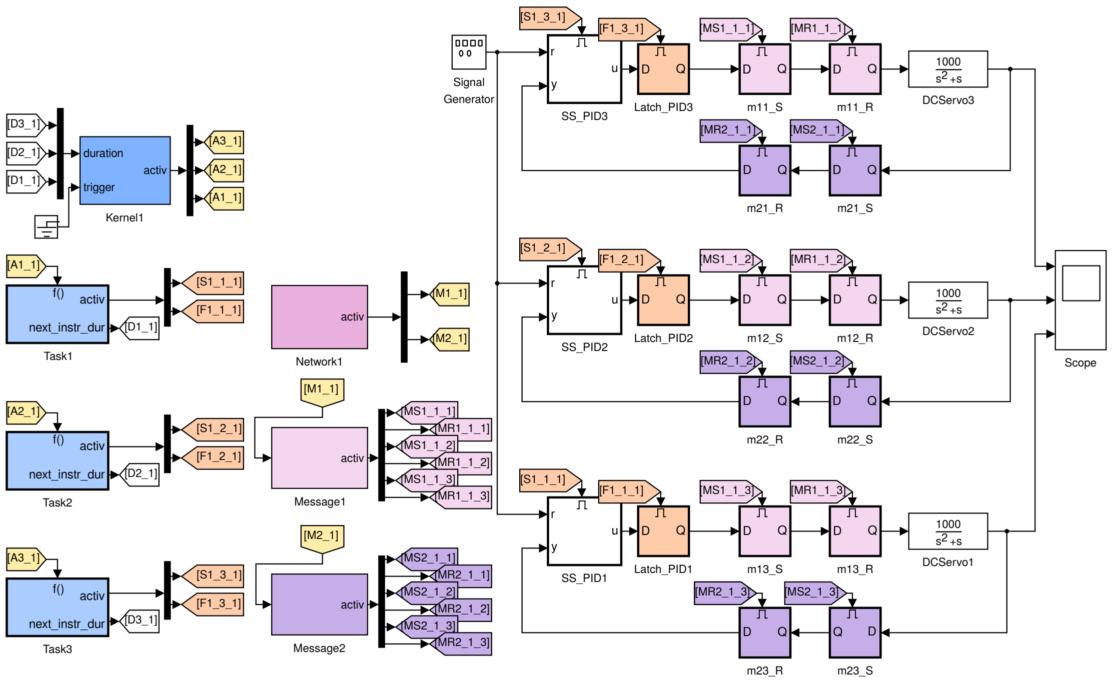


FIGURE 5.15. Simulink model of DC-servo control system with back-annotations.

a latch barrier is added on all its outputs. `Task` blocks use `Goto-From` connections to manage the activation and termination signals of the PID subsystems executing in the segments and to communicate the duration of the next segment to the `Kernel` block. Type and timing properties of tasks in the task-set and the execution times of tasks activities are described by Matlab variables in the workspace:

```
% task set description
%                   % type            %name    %iat   %rdl      %ph
task_set_descr = {'PeriodicTask', 'Task3', 0.004, 0.004,    0; ...
                  'PeriodicTask', 'Task2', 0.005, 0.005,    0; ...
                  'PeriodicTask', 'Task1', 0.006, 0.006,    0};

% sequences of pseudo instructions (task codes)
t1_descr = {'fixed(0.002)'};
t2_descr = {'fixed(0.002)'};
t3_descr = {'fixed(0.002)'};
```

One instance of `Network` and two instances of `Message` blocks (purple blocks on the left) enable the simulation of message exchanges over the CAN bus. Signals to/from servos are multiplexed into `Message1` and `Message2`, respectively. Each of these signals is replaced with a pair of `Send`/`Receive` barrier blocks, activated by the corresponding `Message`. Activation signals flow to barrier blocks through `Goto-From` connections.

The back-annotated Simulink model enables the verification of the impact that task scheduling, execution times and message transmission delays have on the performance of the controls. Figure 5.16 shows the output of the DC-servos with respect to the reference signal (black), when

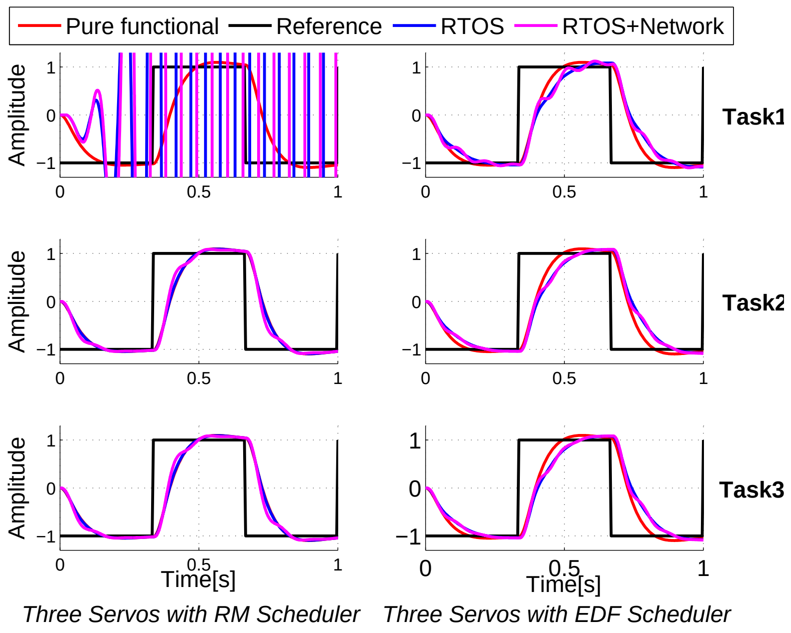


FIGURE 5.16. Verification of DC-servo control system back-annotated model.

a Rate Monotonic (RM, on the left) or EDF scheduling policy (on the right) is used. The control outputs are those of the pure-functional model (red), those of the model that considers only task scheduling and computation delays (blue) and, finally, those obtained also considering the message transmission delays (magenta).

The overload condition induces some performance degradation of controls with respect to the simulation results obtained from the Simulink model without back-annotations. In the case of RM, the task with the lowest priority (`Task1`) cannot guarantee a stable control, because of too many deadline misses. In the case of EDF, the delay due to scheduling and tasks' execution times tends to be spread among the three tasks, and after an initial transient all tasks miss their deadlines. However, the motion of the DC-servos is still controlled with a reasonable error, and the overall control performance is still satisfactory.

5.5.2. Scheduling-Aware Design of Attitude Control for a Simulated Quadrotor

In this example, we use T-Res to estimate the influence of tasks execution times and RTOS scheduling delays on the control performances of a simulated rotorcraft UAV. We perform a simple exploration of the SW design space and evaluate three different SW implementations. The robot has four rotors (quadrotor configuration) and on-board electronics for sensing and control. Figure 5.17 shows two reference platform implementations, one quadrotor robot from *3DRobotics*[3], and a Flight Management Unit (FMU) from the open-source, open-hardware project *PX4*[4].

[3]`https://store.3drobotics.com/products/iris`

[4]`https://pixhawk.org/modules/px4fmu`

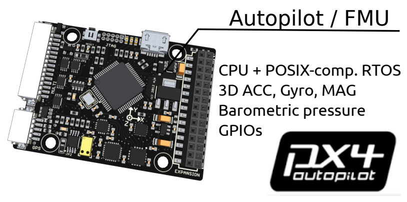
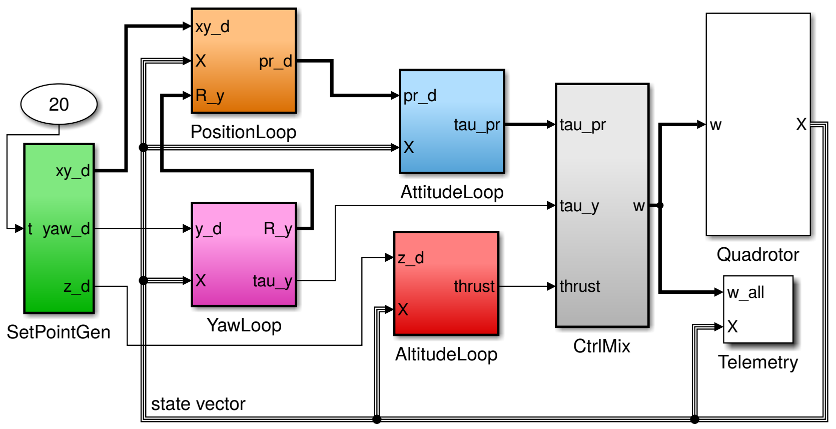
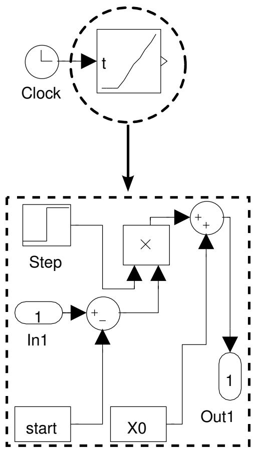


FIGURE 5.17. The IRIS quadrotor (left) and the PX4FMU Autopilot (right)



(A) Control scheme from [Cor11], organized in subsystems.



(B) Ramp

FIGURE 5.18. Models used for the quadrotor flight-control scheme.

The quadrotor is required to lift off and fly in a circle at constant altitude, while spinning slowly around its Z-axis. The adopted control scheme (shown in Figure 5.18a) is taken from [**Cor11**] with minor changes introduced to comply with our design restrictions (cfr. Section 3.3). The original model in [**Cor11**] contains multiple functional loops at the top level of the model hierarchy dedicated to set-point generation and flight control. Each loop has been included in a Simulink subsystem. The constantly increasing signal for the desired yaw angle, originally generated by a `Ramp` block in [**Cor11**], is now obtained from the set of blocks of Figure 5.18b that use the output of an external `Clock` block as time source. In Figure 5.18b, `start` represents the time at which the block begins generating the signal, `X0` is the initial value of the output and the the rate of change of the generated signal is influenced by the parameters of the block `Step`. This is because subsystems mapped into segments cannot contain continuous time blocks (such as `Ramp`).

The set-points of the desired circular path and the desired yaw and altitude are generated by the subsystem `SetPointGen`. `Quadrotor` implements the motion of vehicle. The inputs are the speeds of the four rotors; the output is the 12-element state vector with the position, velocity, orientation and orientation rate of the quadrotor. The actual vehicle velocity is assumed to be estimated by an inertial navigation system or GPS receiver (i.e., there is no velocity estimator in the Simulink model).

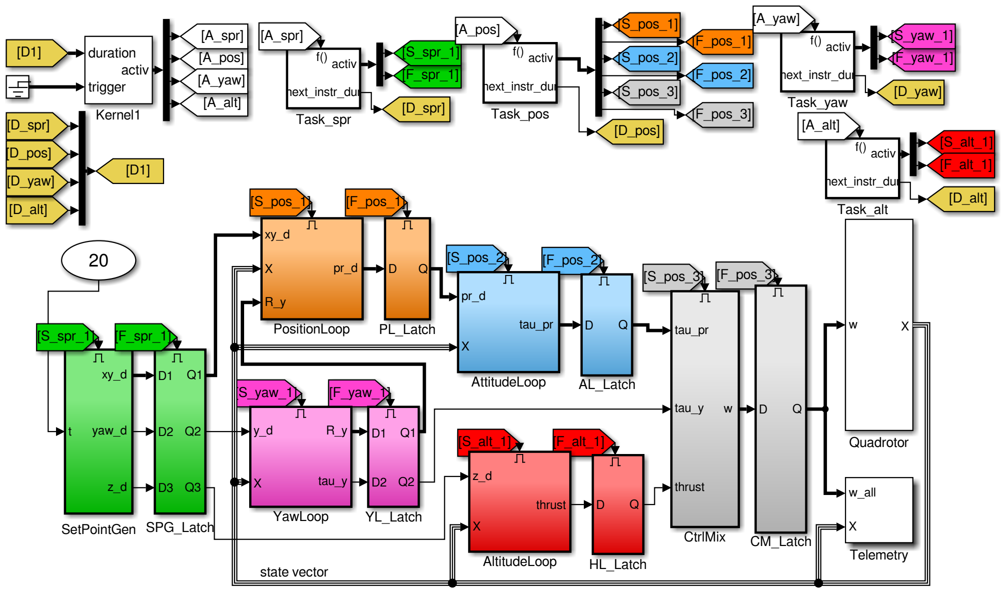


FIGURE 5.19. Attitude control with models of RT kernel and tasks from T-Res.

The control strategy involves multiple nested loops that compute the required thrust and torques so that the quadrotor moves to set-points. Position control has a two-level hierarchical structure: the subsystem `AttitudeLoop` implements the inner loop, which uses the current and desired roll and pitch angles and angular rates to control the vehicle's attitude and to provide damping (to slow down the dynamics). The subsystem `PositionLoop` realizes the outer loop, which controls the XY-position of the flyer by generating changes in roll and pitch angles so as to provide a component of thrust in the direction of the desired motion. Finally, yaw angle and altitude are controlled by proportional-derivative (PD) controllers, respectively implemented by the subsystems `YawLoop` and `AltitudeLoop`.

In practice, control loops are implemented as real time tasks, with finite execution times, running at different rates under the control of a scheduler. Typical execution rates range from $10Hz$ for reading (generating) set-points to $50Hz$ (or more) for controlling the vehicle attitude. We select a SW implementation of controls consisting of four periodic tasks. `Task_spr` runs every $100ms$ and reads the set-points. `Task_pos` uses the set-points and the current state of the vehicle to perform the position control. Every $20ms$, it executes the position loop, the attitude loop and the control mixer, in sequence. Finally, `Task_yaw` and `Task_alt` use the same information to perform yaw and altitude control with a period of $50ms$ and $25ms$, respectively. Subsystems are modeled as executing with execution times randomly generated according to uniform distributions. The execution platform is a single-core FMU board running a FP real-time scheduler. Initially, `Task_spr` is given the highest priority; the other tasks' priorities are assigned according to their period, so that the shorter the period the higher the priority (Rate Monotonic rule). We refer to this candidate design solution as `FP#1`.

```
% task set description
%                 % type           %name        %iat    %rdl    %ph  % prio
task_set_descr = {'PeriodicTask', 'Task_spr', 0.100, 0.100, 0,   0; ...
                  'PeriodicTask', 'Task_pos', 0.020, 0.020, 0,   5; ...
                  'PeriodicTask', 'Task_yaw', 0.050, 0.050, 0,   15; ...
                  'PeriodicTask', 'Task_alt', 0.025, 0.025, 0,   10};


% Sequences of pseudo instructions (task codes)
spr_instrs = {'delay(unif(0.001,0.002))'};
pos_instrs = {...
 'delay(unif(0.005,0.008))'; ... % PositionLoop
 'delay(unif(0.003,0.007))'; ... % AttitudeLoop
 'delay(unif(0.002,0.004))'; ... };% CtrlMix
yaw_instrs = {'delay(unif(0.004,0.006))'};
alt_instrs = {'delay(unif(0.008,0.009))'};
```

FIGURE 5.20. Definition of type and timing properties of tasks.

Figure 5.19 shows the original model back-annotated with T-Res blocks. One instance of `Kernel` block and four instances of `Task` blocks are added to the functional model. Each control subsystem is transformed to a segment, activated by the corresponding `Task`. The type and timing properties of tasks in the task-set and the execution times of tasks activities are specified by means of variables in the Matlab workspace, as in Figure 5.20.

Figure 5.21 visualizes a portion of execution trace of `FP#1` (two hyper-periods). The task-set is non-schedulable. Figure 5.22 shows that computation times and scheduling delays induce deadline misses of tasks `Task_yaw` and `Task_alt`, that do not affect much the altitude control but degrade the performances of circular path-following significantly. This fact is easily explained if one considers that the low-priority task `Task_yaw`, which drives the high-priority task `Task_pos` (that controls the XY-position of the flyer), is repeatedly subject to preemption from the mid-priority task `Task_alt`, and that this prevents the preservation of SR communication flows between `Task_yaw` and `Task_pos`, with respect to the pure functional control model of Figure 5.18a.

The analysis indicates that the response time of task `Task_yaw` has a significant impact on the effectiveness of the control action, and suggests to raise its priority to a value greater than the one of `Task_alt`. This can be easily done by changing the priority levels of the two tasks in the `task_set_descr` variable. We refer to the new candidate design solution as `FP#2`. Figures 5.23 and 5.24 show, respectively, the $200ms$-portion of execution trace and the simulation results of the refined design. ***The task-set is again non-schedulable, but the quadrotor can perform the flight task quite satisfactorily!*** `Task_yaw` has now a priority level greater than `Task_alt` and meets all its deadlines; consequently, the control behavior is closer to the pure functional one. On the other hand, `Task_alt` misses more deadlines than in the initial design and the altitude control performs slightly worse, as seen in Figure 5.24, where the norm of the altitude error is shown (blue line vs red line). However, it is still controlled with a reasonable error, which makes the candidate design `FP#2` preferable.

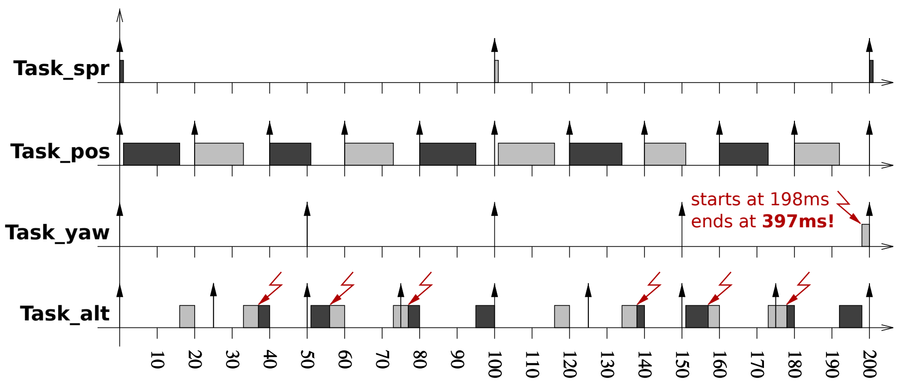
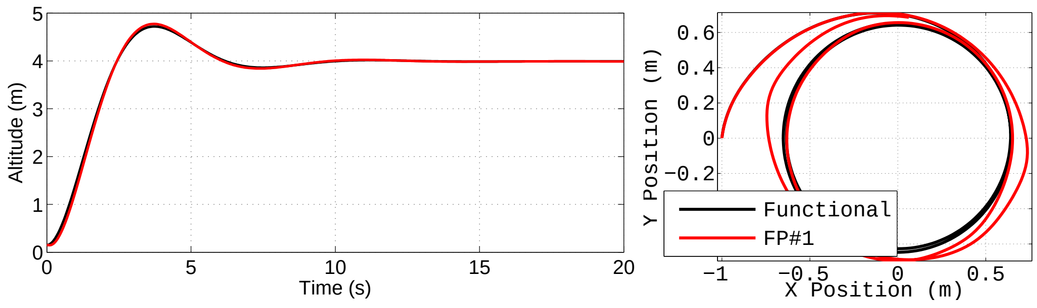


FIGURE 5.21. Execution trace of `FP#1` (200ms), clearly showing that the task-set is non-schedulable.



(A) Altitude control. (B) XY-path-following.

FIGURE 5.22. Simulation results of first candidate design solution (`FP#1`), with respect to the control performance of model of Figure 5.18a (`Functional`).

The third candidate design solution is referred to as `EDF`, because tasks are scheduled according to the Earliest Deadline First (EDF) dynamic scheduling policy. It results in a slightly worse performance of the altitude control (green line of Figure 5.24) and path following performance similar to that of the refined priority model, which is therefore still preferable.

5.6. Integration in the Proposed System-Level Design Flow

T-Res is a key component of the proposed design flow, where Simulink models annotated with platform-specific informations (i.e., T-Res blocks) are *automatically* generated from SysML specifications. The mapping model in Figure 5.1 represents the software tasks and messages (local or on the network) that realize the control functions on top of a distributed execution platform.

The mapping model includes all the information needed to automatically generate and add as back-annotations the `Kernel`, `Task`, `Network` and `Message` blocks to the Simulink functional

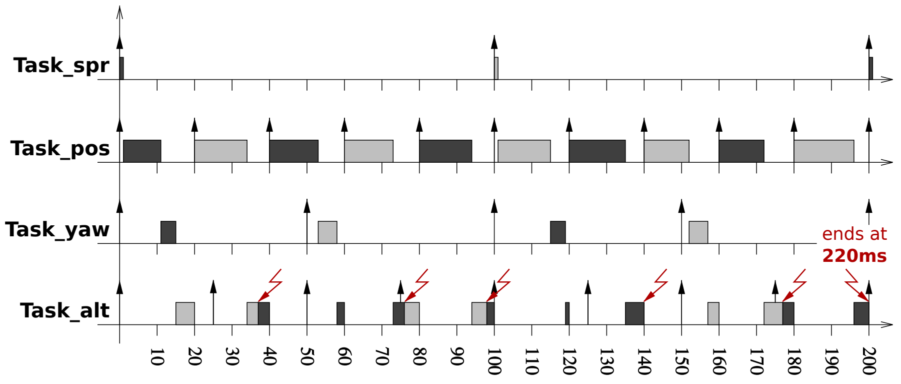
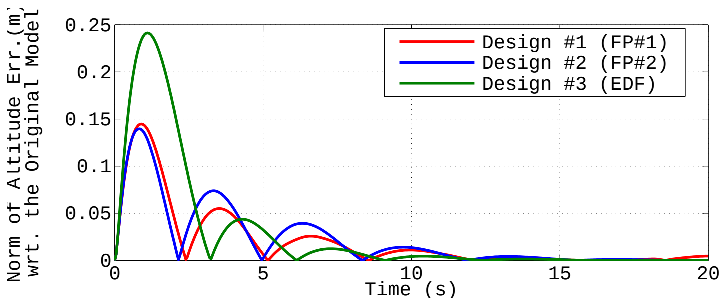
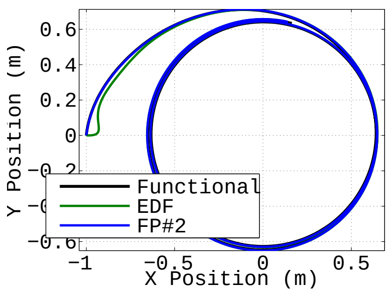


FIGURE 5.23. Execution trace of FP#2 (200ms): task-set is ***again non-schedulable***.



(A) Altitude control.



(B) XY-path-following.

FIGURE 5.24. Simulation results of refined candidate design solutions (FP#2 and EDF), with respect to the control performance of model of Figure 5.18a.

model of controls. Acceleo M2T transformation templates process the mapping model and generate a collection of Matlab scripts that contain the back-annotation commands. The execution of the Matlab scripts produces the annotated Simulink model, which can be simulated to verify that the latencies and jitter added by the scheduling and communication delays do not exceedingly deteriorate the performance of the controls.

We detail the process with reference to the quadrotor attitude-control example described in Section 5.5.2. The current set of Acceleo transformation scripts handles the back-annotation of only `Kernel` and `Task` blocks in a Simulink model.

Figure 5.25 shows a (partial) view of the Papyrus SysML model representing the described four-task implementation. The functional model (on the left) is automatically generated, and preserves all the structural properties of the Simulink model of Figure 5.18a, including the connections among the blocks (not shown). Platform and mapping models are developed in SysML by the system designers.

The Acceleo scripts are invoked from a common main template that performs the following sequence of operations:

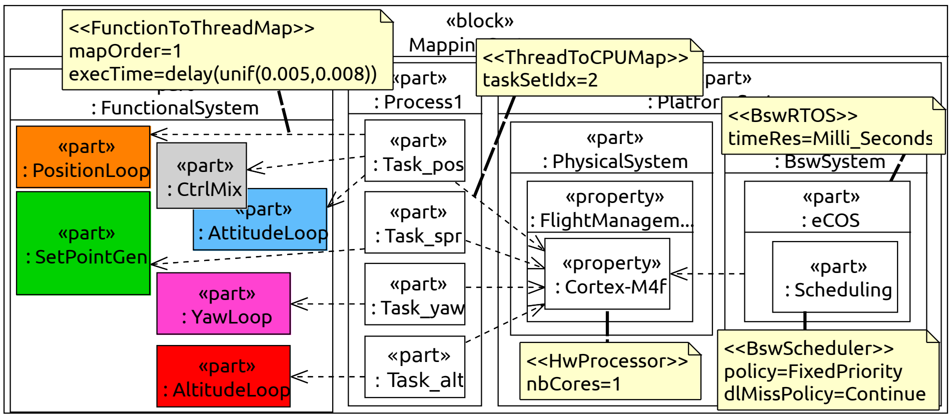


Figure 5.25. SysML IBD describing the deployment of quadrotor's control functions and threads onto the single-core FMU board running a FP real-time scheduler.

(1) the T-Res blockset is opened;
(2) the functional model is saved and a new model is created for its back-annotated version;
(3) a Matlab script is generated, that creates the initialization variables for the `Kernel` and the `Task` blocks attributes;
(4) another Matlab script is generated for the generation of the `Kernel` and the `Task` blocks instances;
(5) finally, another set of `.m` files is created to modify the input model by changing the subsystem blocks to triggered, adding latches on the output links and rerouting the connections (removing the old links and adding new ones that go through the latches.

Figures 5.26a and 5.26b show the most relevant part of the template files that generate the `Kernel` block.

The following is snapshot of the generated Matlab code that adds and configures an instance of `Kernel` block to the Simulink model of quadrotor control.

```
% - Add and configure the Kernel block
kern1 = 'quadcopter_bn/Kernel1';
add_block('t_res/Kernel', kern1);
set_param(kern1, 'taskset_descr_name','task_set_descr');
set_param(kern1, 'scheduling_policy', 'FIXED_PRIORITY');
```

All parameters are available from the platform model and their values are set through the Matlab function `set_param()`. The `Kernel` block outputs task-activation signals in the order specified by the `taskSetIdx` attributes of the `ThreadToCPUMap` allocations in the mapping model.

Tasks types and periods (or interarrival times) are available from the mapping model. Relative deadlines coincide with periods and activation offsets are set to zero. This information is used to properly initialize the Matlab variable `task_set_descr` (Figure 5.20), that describes the timing properties and the type of tasks in the task-set.

```
[template public generate_kernel(mdl : Model) post(trim())]
...
[file ('kernel_gen_commands.m', false, 'UTF-8')]
[** - Adding the Kernel block */]
[generateKernelBlock(mdl_name, cpu, rtos)/]

[** - Infrastructure for the activation of tasks and signals with task duration */]
[generateTasksManagementInfrastructMulti(mdl_name, t2c_set)/]

[**  - Adding the Duration of next task instruction */]
[generateBlocksOfNextDuration(mdl_name)/]
[/file]
[/template]
```

(A) Main Acceleo template.

```
[template public generateKernelBlock(mdl_name:String, cpu:Class, rtos:Class) post(trim())]
[** Add and configure the Kernel block */]
...
add_block('yaks/Kernel', '[mdl_name/]_bn/Kernel1', 'Position', kern1_pp);
[** Configure 'taskset_descr_name'  */]
set_param('[mdl_name/]_bn/Kernel1', 'taskset_descr_name', 'task_set_descr');
[** Compute the other mask parameters by using the Class instances cpu and rtos */]
[** Set the scheduling policy 'scheduling_policy' */]
[let sched : Class = rtos.getSchedulerFromRtos()]
[setKernelMaskParamSchedPolicy(mdl_name, sched)/]
[** Set the Deadline miss rule 'dead_miss_rule' */]
set_param('[mdl_name/]_bn/Kernel1', 'dead_miss_rule',
'[sched.getValueOfStereotypePropertyEnumLit('BswResources::BswRTOS::BswScheduler',
'dlMissPolicy')/]');
[/let]
[** Set the Time resolution 'time_res' */]
set_param('[mdl_name/]_bn/Kernel1', 'time_res',
'[rtos.getValueOfStereotypePropertyEnumLit('BswResources::BswRTOS::BswRTOS', 'timeRes')/]');
[** Set the Number of cores 'core_num' */]
set_param('[mdl_name/]_bn/Kernel1', 'core_num',
'[cpu.getValueOfStereotypeProperty('HwResources::HwComputing::HwProcessor','nbCores')/]');
[** Set the Underlying engine 'under_engine' */]
set_param('[mdl_name/]_bn/Kernel1', 'under_engine', 'RTSIM');
...
[/template]
```

(B) Generation of Matlab construction commands.

FIGURE 5.26. Acceleo instructions for the generation of the `Kernel` block.

The duration of segments executed by each task is described by a Matlab cell array of strings (Figure 5.20). Each string that describes the computation time of a segment is available from the `execTime` attributes of the `FunctionToThreadMap` allocation instances.

According to the proposed methodology, all design is realized at SysML level. Automatic back-annotation helps system integrators to keep the control model in synch with platform and mapping models, and enables design refinements (e.g., changing the scheduling policy or mapping functional subsystems to a different task-set) in a systematic way.

5.7. Summary

The Chapter presents T-Res, a co-simulation framework which integrates external simulation engines for real-time scheduling and network communication in Simulink. T-Res enables the simulation of timing delays dependent on code execution, scheduling of tasks, and network communication latencies (messages), and the verification of their impact on the performance of controls.

We first use T-Res to the study the real-time control of three networked DC servo motors. Then, we apply it to the study of the SW implementation of a simulated rotorcraft UAV. We explore three candidate SW implementations, all leading to non-schedulable task sets. However, one implementation results in satisfactory control performance for the flight task. This enforces the claim that not all control loops/tasks are of type hard real-time and may in fact miss deadlines without losing stability. T-Res allows to quantify the errors for different implementation options.

T-Res is integrated in the proposed system-level design flow. An automatic generation process allows to obtain a new Simulink model back-annotated with platform information (represented by T-Res blocks) from the SysML/MARTE mapping model.

CHAPTER 6

Simulation-Driven Process for Automated Software Synthesis

6.1. Context and Positioning

The design of real-time control systems is typically performed in two steps. First, the control system is designed as a graph of functional blocks activated at a given rate (sampling period). Simulink is often used in the industry to model the (continuous-time) dynamics of the controlled system and the (discrete-time) model of the controller functionality, to be implemented in software. Second, the software implementation is designed as a set of real-time tasks in charge of executing the functional code. The sampling periods of the functions, determined in the control design phase, become timing constraints in the software implementation phase, and deadlines are often assumed as implicit, meaning that each task instance must complete before the next activation. The software designer must define a feasible task set where each task meets its deadline.

In real-time control systems, however, the interplay of the control performance, timing constraints and scheduling effects can be somewhat subtle, and the traditional design flow may be ineffective and result in deadlines tighter than necessary.

Another problem is that the simulation results retain their validity upon condition that the software implementation preserves the simulation execution semantics. When this is not possible because the resulting task set would not be schedulable, the designer is forced to explore other options, including relaxing deadlines and allowing for additional delays in the control functions.

Exploring the interplay between control performance and real-time behavior is desiderable for a better design. We define a simulation-driven process intended to improve the design flow of real-time control systems. The traditional design flows based on the definition of implicit tasks deadlines on control functions are extended to include the exploration of relaxed deadlines and order of execution constraints. Relaxed deadlines, coupled with an ***optimization*** approach to find feasible task sets, allow the exploration and evaluation of different task implementations. The definition of relaxed deadlines and the evaluation of task implementations is performed in ***simulation***, using Simulink and T-Res (cfr. Chapter 5).

The design process is in three stages. The stages are highlighted in Figure 6.1 as bold items. In the first stage, relaxed deadlines are obtained on an estimate of control performance with respect to latency. Simulation is used to estimate the maximum delay that can be applied to each Simulink subsystem in isolation before the control performance deteriorates in an unacceptable way.

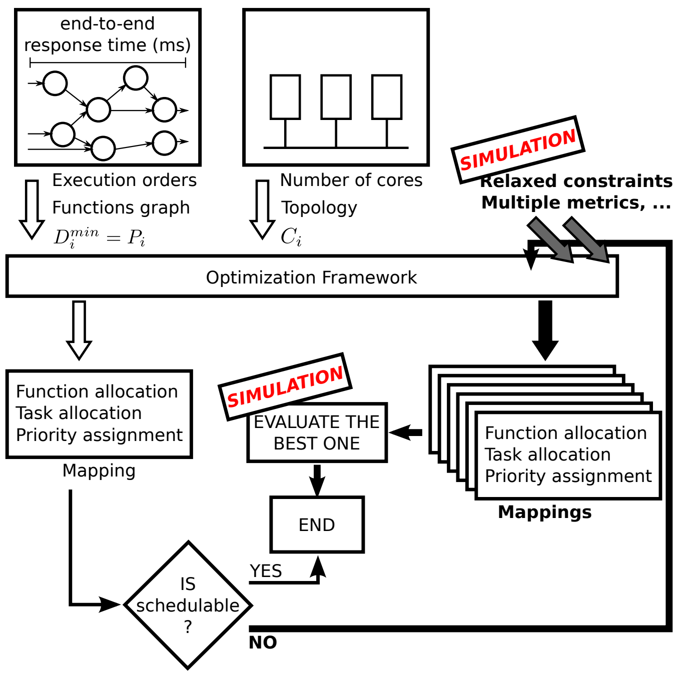


FIGURE 6.1. Flow of the simulation-driven automated software synthesis process.

In the second stage, the mapping (or implementation) of subsystems onto periodic tasks scheduled by priority is computed using an optimization formulation. To find an effective mapping (functions into tasks and priority assignment to tasks) we encode the problem as a mathematical optimization process. The problem is quadratic, but formulated as a Mixed-Integer Linear Programming (MILP), encoding the response time formulation that is obtained from schedulability analysis theory as a set of linear constraints. Different metrics are tried and evaluated according to the simulated performance results. Different mappings considering both minimal deadlines and relaxed deadlines are considered.

In the third stage, the task mappings obtained as optimal solutions by the MILP solver are evaluated by simulation, to estimate the control performance and compare the effectiveness of different metric functions and approaches.

The proposed approach couples MILP with simulation. In general, simulation allows to capture effects like cache faults, preemption of message transmission attempts, task migration delays, finite copy time of messages between driver and adapter levels, etc., that are difficult to model analytically. This chapter focuses on the definition of the simulation-driven software synthesis process for single-CPU systems.

The rest of the chapter is organized as follows. Section 6.2 describes the system model. It gives a formal representation of functions and signals dependencies as a directed acyclic graph and defines the mapping model. Section 6.3 formalizes the optimization model, i.e., the optimization variables, the considered constraints and the optimized metrics. Section 6.4 describes how the simulation with T-Res is used to compute relaxed deadlines and constraints. Section 6.5

illustrates the application of the proposed approach on the real-time control of the quadrotor UAV (see Section 5.5.2). Finally, Section 6.6 summarizes and closes the chapter.

6.2. System Model

We consider systems with a single CPU, on which a set of n functions $\mathcal{F} = \{f_1, f_2, \ldots f_n\}$ obtained as the code implementation of Simulink models must be executed. Each function f_i is the code implementation of a subsystem s_i (for convenience we assign the same index to a subsystem and its function) and has a worst-case execution time (WCET) $C_i > 0$ and a period $P_i > 0$ (matching the period of the subsystem they realize), $i = 1, \ldots, n$.

Simulink subsystems communicate by exchanging signals. In the code implementation these signals are realized as (possibly shared) communication variables. Each signal has a sender and a destination subsystem/function. In the model simulation, it is transmitted in zero logical time. Signals dependencies correspond to order of execution constraints when the outputs of the receiver subsystem are computed as a function of its input values (as opposed to its state only). Functions and signals can thus be represented as a directed acyclic graph in which nodes are functions and edges are signals. We use the notation $f_i \rightarrow f_j$ to indicate that f_j must execute after f_i according to the transitive closure of the order of execution constraints. Sink functions are those functions that do not have successors in the graph, and source functions have at least one signal that is not received from any predecessor (meaning that they process information coming from sensors, or external input.)

We define the set of all graph paths $\mathcal{P} = \{p_1, p_2, \ldots p_q\}$ from a source to a sink.

6.2.1. Definition of Mapping

A mapping is determined by:

- A partition of functions $f_1, \ldots, f_n$ on tasks $T_1, \ldots, T_h$, $h \leq n$: each function is assigned to exactly one task. The functions are called by the task in order.
- An order of execution of functions within a task.
- The priority level π_i that is assigned to each task τ_i. Priorities define a total order on tasks. By extension, the priority level of any function executed by τ_i is also π_i.

Inter-task communications are performed through finite length buffers with suitable size [**WDNSV09**]. A function samples its inputs (resp. produces its output) at execution start (resp. end).

6.2.2. Response-Time Analysis

The performance of the code implementation of the control algorithms depends on their latencies and jitter, which are in turn dependent on the response times of the functions. In order to estimate these response times, we make use of established and recent results on schedulability analysis.

Deadlines Within the Interarrival Times

When response times are guaranteed (by construction or by adding constraints) to be less than or equal to periods, the worst case response time of each function can be computed in correspondence to its critical instant, when the task in which it executes is activated at the same time with all other higher priority tasks. Analytically, the worst-case response time r_i of f_i can be computed as (from [**DNP08**], a straightforward extension of the task-based formulation for periodic tasks in [**JP86**]):

$$r_i = C_i + \sum_{j \in prec(i)} C_j + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil C_j \tag{3}$$

where $prec(i)$ is the set of functions that are in the same task as f_i but invoked before it, and $hp(i)$ indicates the set of all functions executed by tasks with priority higher than the task implementing f_i.

Deadlines Larger Than the Interarrival Times

When the system also allows for functions response times that are larger than periods, that is, when a task may be activated again when it is still awaiting its completion, the previous formula (3) may be optimistic and the exact formulation (as in [**Leh90**]) requires considering all the task activations in the *busy period* of level π_i. The exact formulation becomes very difficult to encode in a formal linear or convex optimization formulation and it is therefore discarded in favor of a recent upper bound $\overline{r}_i \geq r_i$, as defined in [**BPD15**].

$$\overline{r}_i = \frac{C_i + \sum_{j \in prec(i)} C_j + \sum_{j \in hp(i)} C_j (1 - U_j) - \gamma_i}{1 - \sum_{j \in hp(i)} U_j} \tag{4}$$

where $U_j = C_j / P_j$ is the utilization of function f_j, and γ_i is defined as:

$$\gamma_i = \sum_{j,k \in hp(i), j<k} \min\{P_j, P_k\} U_j U_k \tag{5}$$

6.3. Optimization Model

In order to compute optimal mappings with respect to a given optimization metric, we model the problem as a Mixed Integer Linear Program (MILP).

A MILP formulation is defined by a set of constraints delimiting the set of feasible solutions, and an objective function to optimize. Constraints and objective function are defined in terms of optimization variables (the design parameters to be determined) and parameters (the known values).

For our function allocation and task configuration problem we extend the MILP formulations in [**MH06**] [**ZZZ$^+$13**] [**MWTP$^+$13**]. All these papers considered deadlines lower than or equal to periods. We generalize the model to arbitrary deadlines.

6.3.1. Optimization Variables

To determine a mapping, a task must be assigned to each function, a priority must be assigned to each task, and an *execution order* must be defined for functions executed in the same task. The task mapping and task priority assignment are defined by a single set of *priority* values assigned to functions. Each priority value is implicitly assigned and identifies a single task.

The function priority defines at the same time the task into which it executes and its priority level. Given that the priority assignment defines a total order, we do not assign absolute priority values, but rather a priority order.

Priorities are defined by variables $\pi_{i,j}$, $i, j = 1, \ldots, n$:

$$\pi_{i,j} = \begin{cases} 1 & \text{if } \pi_i > \pi_j \\ 0 & \text{otherwise} \end{cases}$$

A sequence order on functions assigned to the same task (i.e. with the same priority) is defined by variables $\sigma_{i,j}$, $i, j = 1, \ldots, n$:

$$\sigma_{i,j} = \begin{cases} 1 & \text{if } \pi_i = \pi_j \text{ and } f_i \to f_j \\ 0 & \text{otherwise} \end{cases}$$

The $\pi_{i,j}$ and $\sigma_{i,j}$ assignments must be constrained in such a way that the transitive and antireflexive properties hold for priority and order assignments (omitted here, see [**ZZZ$^+$13**] for a description).

6.3.2. Constraints

A necessary requirement for a mapping is to ensure the schedulability of all the functions, i.e. the following constraints must be satisfied:

$$r_i \leq D_i, i = 1, \ldots, n \tag{6}$$

where r_i denotes the response time of function f_i, $i = 1, \ldots, n$. Response times are computed as described in Section 6.2.2.

Deadlines Less Than or Equal to Periods

When the response times are less than the periods, Equation (3) applies. The MILP encoding of (3) is (as in [**ZDN13**])

$$r_i = C_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} \sigma_{j,i}\, C_j + \sum_{\substack{j=1 \\ j \neq i}}^{n} \pi_{j,i}\, C_j\, I_{j,i} \tag{7}$$

where integer variable $I_{j,i}$ represents the possible number of interferences of (possibly higher priority) function f_j on f_i. The variable $I_{j,i}$ is defined by the bounds

$$r_i / P_j \leq I_{j,i} < r_i / P_j + 1 \tag{8}$$

Deadlines Possibly Larger Than Periods

In this case, Equation (4) is expressed as:

$$\overline{r}_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \overline{r}_i\, \pi_{j,i}\, U_j + C_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} \sigma_{j,i}\, C_j + \sum_{\substack{j=1 \\ j \neq i}}^{n} \pi_{j,i}\, C_j\, (1 - U_j) - \gamma_i \tag{9}$$

where:

$$\gamma_i = \sum_{\substack{j=1 \\ j \neq i}}^{n-1} \sum_{\substack{q=j+1 \\ q \neq i}}^{n} \pi_{j,i}\, \pi_{q,i}\, \min(P_j, P_q)\, U_j\, U_q \tag{10}$$

Equations (9) and (10) are not linear (quadratic) due to the products of the optimization variables. To linearize Equation (9), we introduce a real variable $\rho_{j,i}$ that accounts for the product $\overline{r}_i\, \pi_{j,i}$ and is defined using the *big M* formulation that is typically used to encode conditional constraints.

$$\rho_{j,i} = \begin{cases} \overline{r}_i & \text{if } \pi_{j,i} = 1 \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

The value of $\rho_{j,i}$ is determined by the following constraints:

$$\rho_{j,i} \geq 0 \tag{12}$$

$$\rho_{j,i} \leq \overline{r}_i \tag{13}$$

$$\rho_{j,i} \leq M\, \pi_{j,i} \tag{14}$$

$$\rho_{j,i} \geq \overline{r}_i - M\,(1 - \pi_{j,i}) \tag{15}$$

where M is any constant greater than $\overline{r}_i$. A suitable value for M is (from an upper bound on Equation (4)).

$$M = \sum_{j=1}^{n} C_j (2 - U_j) \tag{16}$$

In a similar way, to linearize Equation (10), we introduce the variable $\mu_{j,q,i}$:

$$\mu_{j,q,i} = \begin{cases} 1 & \text{if } \pi_{j,i} = 1 \wedge \pi_{q,i} = 1 \\ 0 & \text{otherwise} \end{cases}$$

and the following constraints:

$$\mu_{j,q,i} \leq \pi_{j,i} \tag{17}$$

$$\mu_{j,q,i} \leq \pi_{q,i} \tag{18}$$

$$\pi_{j,i} + \pi_{q,i} \leq \mu_{j,q,i} + 1 \tag{19}$$

Finally, Equations (9) and (10) are replaced by:

$$\overline{r}_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \rho_{j,i}\, U_j + C_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} \sigma_{j,i}\, C_j + \sum_{\substack{j=1 \\ j \neq i}}^{n} \pi_{j,i}\, C_j\, (1 - U_j) - \gamma_i \tag{20}$$

$$\gamma_i = \sum_{\substack{j=1 \\ j \neq i}}^{n-1} \sum_{\substack{q=j+1 \\ q \neq i}}^{n} \mu_{j,q,i}\, \min(P_j, P_q)\, U_j\, U_q \tag{21}$$

In addition, the mapping must be performed in such a way that all functions in a task have the same period. For any pair of functions $\{f_i, f_j\}$ it must be $\pi_i = \pi_j \Rightarrow P_i = P_j$, encoded in MILP form by the following constraint:

$$\pi_{i,j} + \pi_{j,i} = 1 \quad \text{for all } i,j = 1,\ldots,n, \quad \text{such that } P_i \neq P_j \tag{22}$$

Finally, the last set of constraints deals with the need of preserving the order of execution of functions. For any pair of functions $\{f_i, f_j\}$ such that $f_i \to f_j$ in some path, we have:

$$\pi_i \geq \pi_j \vee (\pi_i = \pi_j \wedge \sigma_{i,j} = 1), \tag{23}$$

represented by the following constraints:

For all $i,j = 1,\ldots,n,$ such that $f_i \to f_j$ in some path:

$$\pi_{j,i} = 0 \tag{24}$$

$$\sigma_{j,i} = 0 \tag{25}$$

$$\sigma_{i,j} = 1 - \pi_{i,j} \tag{26}$$

6.3.3. Optimization Metrics

We consider three optimization metrics based on path latency, intuitively corresponding to the worst case end-to-end response on a given path. The latency of path p_i is denoted as $\mathcal{L}_i$ and computed as [**ZZZ⁺13**]:

$$\mathcal{L}_i = \sum_{f_j \in p_i} P_j + r_j \tag{27}$$

We consider three metrics for minimization.

Average latency (AL):

$$\frac{1}{q}\sum_{i=1}^{q} \mathcal{L}_i.$$

Maximum latency (ML):

$$\max_{i=1,\ldots,q} \mathcal{L}_i.$$

Maximizing the minimum fractional slack (FS):

$$\max \min_{i=1,\ldots,q} \frac{D(p_i) - \mathcal{L}_i}{D(p_i)},$$

which is equivalent (through simple math) to minimize the maximum relative latency:

$$\min \max_{i=1,\ldots,q} \frac{\mathcal{L}_i}{D(p_i)}.$$

The deadline of path p_i is denoted as $D(p_i)$, and is computed as:

$$D(p_i) = \sum_{f_j \in p_i} P_j + D_j. \tag{28}$$

FS attempts at easing future extensibility. We compute it relative to the deadlines of paths, in order to have a normalized objective function.

6.4. Simulation

Simulation plays a key role in two of the three stages of the proposed design flow (Figure 6.1). Initially, simulation is used to obtain relaxed deadlines for functions executing the control code. A relaxed deadline D_i^{max} is computed for each function f_i by estimating the maximum delay that it can experience in isolation, i.e., assuming an ideal execution of the other functions, before the control performance deteriorates in an unacceptable way.

Next, after candidate software implementations are synthesized using MILP, the simulation is used again to evaluate the influence of their execution on the control performances, and finally, the implementation having the smallest impact is selected.

Simulations are performed using the T-Res toolkit under Simulink. In the first stage, each estimate D_i^{max} is computed by considering the corresponding function f_i as the only time-consuming computation activity (T-Res segment) in the system. Its delayed output is incremented across multiple simulation runs, until the control becomes unstable or the deviation from the pure functional control (i.e., the control where *all* functions execute in zero time, *including* f_i) is so large that the performance is considered unacceptable.

Note that, the estimated relaxed deadlines are only first order approximations of the actual maximum delays that can be tolerated, because functions (i.e., control subsystems) are coupled and not independent. Relaxed deadline values are not used to validate the system configuration, but only as guidelines for the selection of the best task implementation.

6.5. Application Example

6.5.1. Case Study Definition

We apply the synthesis process to the design of the attitude control for the simulated quadrotor of Section 5.5.2. The correspondence between the names of subsystems in Figure 5.18a (control loops) and the code implementations $f_i \in \mathcal{F}$ is summarized in Table 6.1. In the following, for sake of simplicity, we will use the numeric indices of subsystems to refer to the corresponding functions in the SW task implementation.

The periods of the subsystems and functions are: P_1 = 100ms, P_2 = 20ms, P_3 = 20ms, P_4 = 50ms, P_5 = 25ms and P_6 = 20ms.

We consider five configurations, with different execution times, each represented in Table 6.2. All times are expressed in *milliseconds*. The last column shows the total system utilization U_t, ranging from 84% to 99% (overload conditions are not considered).

6.5.2. Computation of Deadline Approximations

Figure 6.2 shows the model configuration to compute the relaxed deadline of f_2, by using T-res. A single periodic task (`Task1`), executes f_2 with a variable computation delay/execution

Original name (Figure 5.18a)	Renamed to (s_i)	Code implementation name (f_i)
`SetPointGen`	`SetPointGen_s1`	f_1
`PositionLoop`	`PositionLoop_s2`	f_2
`AttitudeLoop`	`AttitudeLoop_s3`	f_3
`YawLoop`	`YawLoop_s4`	f_4
`AltitudeLoop`	`AltitudeLoop_s5`	f_5
`CtrlMix`	`CtrlMix_s6`	f_6

Table 6.1. Subsystem names and corresponding function names for the quadrotor example.

	C_1	C_2	C_3	C_4	C_5	C_6	U_t
I_{84}	3	4	5	5	4	2	0.84
I_{92}	2	5	5	5	5	2	0.92
I_{94}	3	5	5	8	5	1	0.94
I_{94b}	2	5	5	4	6	2	0.94
I_{99}	2	6	5	4	6	2	0.99
D_i^{min}	100	20	20	50	25	20	

Table 6.2. Case study configurations for the design exploration.

time, defined as a variable in the Matlab workspace and incremented across multiple simulations, until an approximation of D_2 is found.

Figure 6.3 shows the control performance when varying the response time of functions f_6 and f_2 (in isolation), respectively indicated by r_6 and r_2. The top side shows the impact of r_6 on the altitude (Z) control. The control performs well until r_6 is incremented to 41ms, when it suddenly becomes unstable and makes the quadrotor fall down on the ground. The bottom side of Figure 6.3 shows the effects of increasing r_2 with respect to the XY path following. The graph shows the norm of the difference of the controlled position with respect to the pure functional control, indicated as $\|XY_{err}\|$, versus time. For $r_2 = 30$ms the difference is small at steady-state (black dashed line), and the control performance is acceptable. It becomes larger as r_2 increases, and reaches a significant steady-state value for $r_2 = 90$ms (continuous-thin black line). There are also small peaks in the range 0–5 sec., that indicate a further deviation from the original simulation results in the early phases of the application of the control action. For $r_2 = 120$ms, $\|XY_{err}\| \simeq 5$cm at steady state and peaks are even larger. The control performance is considered unacceptable for $r_2 > 120$ms.

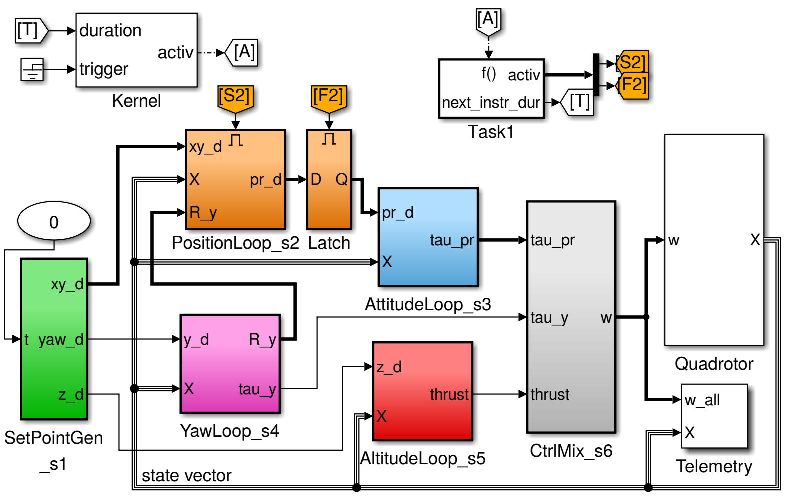


FIGURE 6.2. The T-Res setup to evaluate the maximum acceptable delay for f_2.

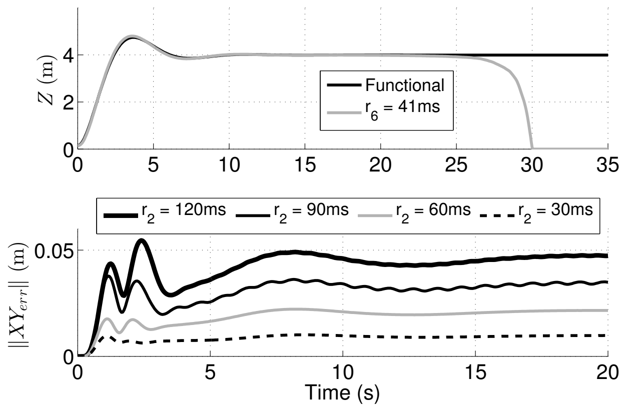


FIGURE 6.3. Values of delays for f_6 (top) and f_2 (bottom) for which the performance is significantly compromised.

The procedure is repeated for the other functions implementing the flight-control logic. As a result, the relaxed deadlines are: $D_1^{max} = 500$ms, $D_2^{max} = 120$ms, $D_3^{max} = 40$ms, $D_4^{max} = 301$ms, $D_5^{max} = 82$ms and $D_6^{max} = 40$ms.

6.5.3. Exploration Strategies and Application of MILP

The first step to the software synthesis is to compute an optimal mapping for each possible configuration, that preserves constraints (6) with $D_i = D_i^{min}$ and (24)–(26) (preservation of execution order). Unfortunately, the MILP solver does not return any feasible mapping for the five configurations. Hence, to obtain a feasible mapping, we explore two possible relaxations of the model:

- an execution that violates the order of execution among subsystems, denoted as R_o and,

- an implementation R_d that allows $D_i = D_i^{max}$, $i = 1, \ldots, n$.

In the exploration of the possible solutions, we allow executions that violate the order prescribed by the Simulink semantics (R_o). Order violations may results in data (control samples) loss by overwriting or skipping. The effect is similar to a disturbance that may be tolerated by the control systems.

We therefore solve three optimization problems by combining the two model relaxations:

- Model R_o is obtained by relaxing the execution order preservation constraints (Equations 24, 25, 26);
- Model R_d is obtained by setting $D_i = D_i^{max}$, $i = 1, \ldots, n$;
- Model R_{od} is obtained by relaxing the execution order preservation constraints (Equations 24, 25, 26) and setting $D_i = D_i^{max}$, $i = 1, \ldots, n$.

For each of these three problems we try the three optimization metrics **AL**, **ML** and **FS** (cfr. Section 6.3.3). The nine resulting MILP formulations are solved for each of the five execution time configurations in Table 6.2. The MILP problems are solved with an IBM ILOG Cplex 12.6 solver, which returns the mappings summarized in Table 6.3. Tasks are listed from higher to lower priority; for each task, the function indexes mapped onto it are shown. For example, [6],[1],[2,3],[5],[4] indicates that the highest priority task executes function f_6, the next task executes function f_1, then another lower priority task executes f_2 and f_3 in sequence, then a task executes f_5 and finally the lowest priority task executes f_4. When multiple optimal solutions are found, they are all listed in the corresponding cell.

The highest utilization configuration is only feasible when $D_i = D_i^{max}$, $i = 1, \ldots, n$. Also, different relaxation methods and different metrics functions can bring to substantially different configurations, even for our relatively simple case study. The task of the performance evaluation stage is to understand which relaxation strategy and which metric function works best.

6.5.4. Evaluation of Candidate Designs

Representing the Mappings in Simulink

We use T-Res to evaluate the quality of the candidate implementations (the task mappings returned as optimal solutions by the MILP solver) with respect to the control performances.

The model in Figure 5.19 can represent all the mappings in the *last column* of Table 6.3 with a suitable definition of task priorities and functions computation times. The priorities of the tasks and the execution times of the functions are specified in configuration variables in the Matlab workspace and set as parameters of the `Kernel` and `Task` blocks (as in Figure 5.20). Similar Simulink models and Matlab code configurations of the blocks `Kernel` and `Task` enable the representation of *all* the mappings in Table 6.3.

Performance Evaluation

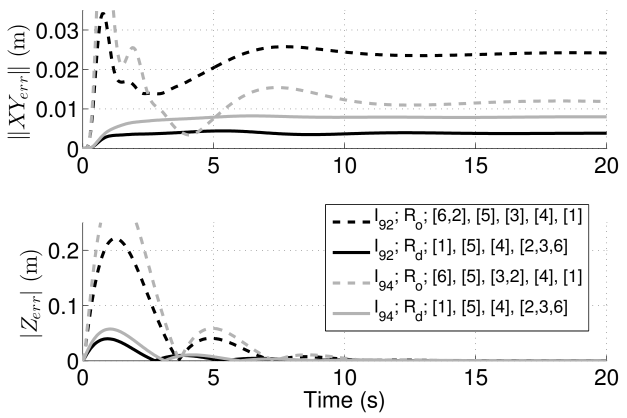
Figures 6.4–6.5 show the measure of the absolute difference of the controlled variables values between the task implementations (with execution and scheduling delays) and the pure functional design. On top, the figures show the norm of the difference $\|XY_{err}\|$ in the controlled XY

		R_o	R_{od}	R_d
I_{84}	AL	[6],[1],[2,3],[5],[4]	[6],[1],[4],[2,3],[5]	[1],[4],[5],[2,3,6]
	ML	[6],[1],[2,3],[5],[4]	[1],[4],[6,2,3],[5]	[1],[4],[5],[2,3,6]
	FS	[6,2,3],[5],[4],[1]	[6,3],[5],[2],[4],[1]	[1],[4],[5],[2,3,6]
			[6,3],[5],[1],[2],[4]	[1],[5],[4],[2,3,6]
			[6,3],[1],[5],[2],[4]	
			[6,3],[5],[2],[1],[4]	
I_{92}	AL	[6,3,2],[5],[4],[1]	[6],[1],[4],[3,2],[5]	[1],[4],[5],[2,3,6]
	ML	[6,2,3],[5],[4],[1]	[1],[4],[6,2,3],[5]	[1],[4],[5],[2,3,6]
		[6,3,2],[5],[4],[1]		
	FS	[6,2],[5],[3],[4],[1]	[6,3],[5],[2],[1],[4]	[1],[5],[4],[2,3,6]
		[6,2,3],[5],[4],[1]	[6,3],[1],[5],[2],[4]	[1],[4],[5],[2,3,6]
			[6,3],[5],[1],[2],[4]	
			[6,3],[5],[2],[4],[1]	
I_{94}	AL	[6,3,2],[5],[4],[1]	[6],[1],[4],[3,2],[5]	[1],[4],[5],[2,3,6]
	ML	[6,2,3],[5],[4],[1]	[6],[1],[4],[2,3],[5]	[1],[5],[4],[2,3,6]
		[6,3,2],[5],[4],[1]	[6],[1],[4],[3,2],[5]	[1],[4],[5],[2,3,6]
	FS	[6,2],[5],[3],[4],[1]	[6,3],[5],[2],[1],[4]	[1],[4],[5],[2,3,6]
		[6,3],[5],[2],[4],[1]	[6,3],[1],[5],[2],[4]	
		[6],[5],[3,2],[4],[1]	[6,3],[5],[1],[2],[4]	
		[6,3,2],[5],[4],[1]		
I_{94b}	AL	[6,3,2],[5],[4],[1]	[6],[1],[4],[3,2],[5]	[1],[4],[5],[2,3,6]
	ML	[6,2,3],[5],[4],[1]	[1],[4],[6,3,2],[5]	[1],[4],[5],[2,3,6]
		[6,3,2],[5],[4],[1]	[1],[4],[6,2,3],[5]	
	FS	[6,3,2],[5],[4],[1]	[6,3],[5],[2],[1],[4]	[1],[5],[4],[2,3,6]
		[6,2,3],[5],[4],[1]	[6,3],[5],[1],[2],[4]	
			[6,3],[1],[5],[2],[4]	
I_{99}	AL	Infeasible	[6],[1],[4],[3,2],[5]	[1],[4],[5],[2,3,6]
	ML		[1],[4],[6,3,2],[5]	[1],[4],[5],[2,3,6]
				[1],[5],[4],[2,3,6]
	FS		[6,3],[4],[1],[5],[2]	[1],[4],[5],[2,3,6]
			[6,3],[5],[1],[4],[2]	

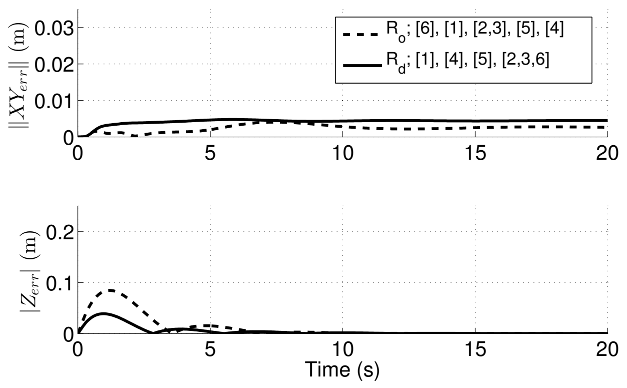
TABLE 6.3. Table of all computed mappings.

variables (XY-path following). At the bottom, they show the absolute value of the difference of the controlled altitude $|Z_{err}|$. Lower errors indicate better results.

The use of different metrics in R_o and R_d has no significant impact on the control performance. This is probably due to the size of the case study and the restrictions of such models.



(A) Difference with respect to ideal case: $U \geq 92\%$..



(B) Difference with respect to ideal case: $U = 84\%$.

FIGURE 6.4. Difference with respect to ideal case.

Figure 6.4a shows the results for the mapping obtained by optimizing the **FS** metric. The dashed lines represent the performance when the order of execution is relaxed (R_o), whereas continuous lines correspond to the case of relaxed deadlines (R_d). Only instances I_{92} and I_{94} are shown in the figure, respectively as dark and light lines, since they are representative of the behaviour of all cases R_o and R_d and high processor utilizations ($U \geq 92\%$). R_o generates mappings with worse performances. This is clearly visible for the altitude control. The trajectory control exhibit two kinds of behaviours: either the dashed line is strictly higher than the continuous one ($U = 92\%$), or the two lines nearly converge at steady-state but the dashed one has an initial peak ($U = 94\%$). This is due to f_1 having the lowest priority, which occurs in most R_o mappings (see Table 6.3). In those cases the vehicle is forced to follow a wrong reference trajectory at the beginning of the control application, and this causes a large deviation (peak) from the simulation results obtained with the pure functional design. Function f_1 has the lowest priority in most R_o mappings because of the tight deadlines of the other functions. The low priority of f_1 corresponds to a violation of the execution order. The solutions found with both

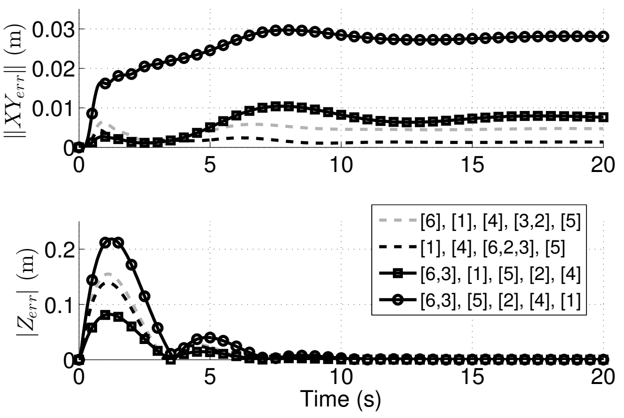


FIGURE 6.5. R_{od} model, comparison of task configurations and optimization metrics ($U = 92\%$).

relaxations enabled (R_{od}) (not shown in Figure 6.4a) yield a performance in between the two R_d and R_o cases.

A lower processor utilization ($U = 84\%$) may increase the solution space, and the model R_o yields to a mapping with a very good performance in terms of trajectory following (dashed line in Figure 6.4b).

In order to evaluate the performances of the different metrics, we focus on the problem definition R_{od}, which yields a larger solution space. Figure 6.5 shows the performance of the solutions for case I_{92} (other instances have a similar behaviour). The mapping with the best overall performance (i.e., on both trajectory following and altitude control) is [6,3], [1], [5], [2], [4], obtained by minimizing **FS**. This mapping produces a good performance for the trajectory following, because the most critical functions f_1 and f_6 have high priorities. It also has a good performance in altitude control, because f_5 is not too much delayed. In case the task of trajectory following is considered to be more critical than altitude control, optimizing the **ML** metric yields the best performance (dark dashed line in Figure 6.5). Minimizing the **AL** metric seems to yield the least performant mappings (light dashed line). This result is somewhat expected, since this metric is quite coarse: it does not target individual paths and allows some responses to be quite large while others can be very small.

The continuous lines in Figure 6.5 also show how different mappings that are equivalent from the optimality point of view for a given objective function (**FS** in this case) may exhibit different control performances.

6.6. Summary

This Chapter presents a simulation-driven optimization process for the implementation of real-time control software. The process enables the exploration of the interplay between the control performance and the real-time behavior under relaxed constraints, such as task deadlines and order-of-execution constraints. It couples MILP with simulation using T-Res. Simulation allows to capture effects difficult to model analytically, such as cache faults, preemption of message transmission attempts, task migration delays, finite copy time of messages between driver and adapter levels.

The experiments conducted on a simulated quadrotor case study show that simultaneous consideration of control performance and real-time concerns leads to SW implementations providing better results than classical decoupled approaches, where the controls are designed first and the SW implementation is synthetized next (dealing with the real-time concerns separately).

CHAPTER 7

Conclusions

7.1. Contributions

The goal of this thesis is the definition of a design process, supported by tools, for the development, verification and deployment of time-sensitive complex CPS control applications. We formulate it as a system-level design process, encompassing the tight integration of control, hardware and (real-time) software architectures. We follow the tenets of PBD approach and assume models based on a Synchronous-Reactive (SR) execution semantics for defining the functional platform, SysML-/MARTE for representing the execution platform, and timing issues (and their influence on control performance) as primary design concern.

SysML/MARTE are suited at representing architectural aspects (including timing). Vice versa, SR languages allows for control simulation, testing and behavioral code generation. The definition of an integrated design flow exploiting their complementarities has attracted significant interest over years ([**VD06**, **DN12**]). This work makes the following contribution on this topic. ***We present a PBD framework that supports the transition from the functional model to the code implementation (with the preservation of communication flows), and enables designers to explore tradeoffs between delays and control performances when a semantics-preserving implementation of functionality is not achievable***.

The following summarizes the related contributions and categorizes them with respect to the research issues defined in 1.4. All the implemented software and (meta-)models are available at `https://github.com/m-morelli` (in dedicated repositories, indicated below) as ***open-source***.

Contributions to RI.1

Chapter 3 defines the process flow, including the definition of a common semantic domain and rules for the integration of SR and SysML/MARTE platform models. The common semantic domain enables the formal representation of platform models and mapping process; formal modeling, in turn, enables the implementation of model transformations that concretely realize the integration.

Platform models incorporate and expose appropriate information to allow accurate prediction of control performance properties (e.g., stability, steady and transient state performance) and timing properties (as in classical real-time systems) of a candidate implementation. The design cycle is a sequence of tool-assisted stepwise refinements which is iterated until the implementation satisfies the initial specification.

This work brings the following contributions.

- An EMF Ecore definition of a custom meta-model, expressing the structure of SR models (subsystems, ports, connections, etc.) and all the information related to the timed events (including rate constraints, partial order of execution constraints and any other synchronization constraints).
- A Simulink-to-EMF model exporter written in Matlab language, that generates an XML view of the control model (the XML conforms to a schema in accordance with the EMF Ecore meta-model)—the dedicated repository is `see_simple`.
- A QVTo M2M transformation to translate the (exported) EMF model into a SysML model in Papyrus, by leveraging profile and type library definitions for SR systems—available as `cypbd_functional` (profile) and `cypbd_fum2m` (transformation).
- A set of SysML-/MARTE-based profiles for the modeling of execution architectures and SW architecture implementations (task and message models) and for the definition of functionality-to-architecture mapping—repositories are `cypbd_architecture` (execution architectures) and `cypbd_mapping` (SW architectures and mapping).

Contributions to RI.2

Chapter 4 defines models and rules to generate a SW implementation of robot control applications that is guaranteed to preserve the SR execution semantics of functional model and, in turn, the system properties formally verified during the design phase. The target of the generation process is Orocos-RTT, a component-based middleware very popular among robotics practitioners (researchers and industry). The generation process enables semantic-preserving deployment of applications on single-/multi-core execution architectures.

We present a set of model transformations (`orocosrtt_gen`) that improve the state-of-practice of code-generation from synchronous models. The presented process overcomes the limitations of current commercial code generation solutions, that (*i*) produce code only for the execution on single-core, and (*ii*) require interspersing the functional model with platform-specific blocks (for, e.g., the OSEK RTOS, dSPACE RTI, etc.), violating the desirable separation of concerns between the functional and the platform designs.

Contributions to RI.3

Chapter 5 presents T-Res (`tres_bundle`), a Simulink-based co-simulation framework to estimate the impact that time delays, dependent on code execution, scheduling of real-time tasks and communication latencies, have on the performance of controls. T-Res overcomes a number of limitations of TrueTime [**CHL$^+$03**], probably the best known Simulink-based tool targeting the exploration of the interplay between functional (control performance) and non-functional (timing constraints and scheduling effects) system properties. Specifically, T-Res brings the following improvements: an easy integration with Simulink control toolboxes and models; an explicit separation between controllers, tasks, messages, schedulers; a modular architecture enabling easy integration with third-party real-time scheduling and network simulators.

T-Res is integrated into the proposed PBD work-flow. An Acceleo MOFM2T transformation (`tres_gen`) processes the mapping model and generates a set of Matlab files, that back-annotate the Simulink model with blocks representing the execution of the controller model into a set of real time tasks under the control of a scheduler and exchanging messages onto a network medium. This is a key contribution with respect to the state-of-art model-driven design tools for complex CPS, e.g., robots [**Bru15**]. These tools assume that all control loops are hard real-time. In reality, many systems/applications are tolerant to delays and deadline misses, and a flow like the one we propose is a crucial advantage for designers.

The presented PBD flow enables the prediction of system's timing behavior (latencies and jitter) at model-level (i.e., *in the early phases of the development*). In Section 5.5.2, we apply it to the study of the SW implementation of a simulated rotorcraft UAV. We explore three candidate SW implementations, all leading to non-schedulable task sets. Despite this, we find that one can perform the flight task quite satisfactorily. Note that current design tools would have discarded it, because they rely on the output of schedulability analyzers.

Contributions to RI.4

Exploring the interplay between control performance and real-time behavior is desiderable for a better design. Chapter 6 defines a simulation-driven process intended to improve the design flow of real-time control systems in those situations where there are no feasible task-sets for the deadlines determined in the control design phase.

We extend the traditional design flows based on the definition of implicit tasks deadlines on control functions (generally coincident with their sampling periods) to include the exploration of relaxed deadlines and order-of-execution constraints. Relaxed deadlines, coupled with a MILP-based optimization approach to find feasible task sets, allow the exploration and evaluation of different candidate SW implementations. The definition of relaxed deadlines and the evaluation of implementation options is performed in simulation using T-Res. Simulation allows to capture effects difficult to model analytically, such as cache faults, preemption of message transmission attempts, task migration delays, finite copy time of messages between driver and adapter levels. This is another advantage with respect to a number of works in the literature, that define design processes based on an analytical formulation of control and real-time scheduling co-design problem.

7.2. Limitations (Ongoing Work)

Tool Support

The tool support for the presented PBD flow is currently a proof-of-concept. *Extensive testing and robustness improvements are currently ongoing.*

Some parts of the presented toolchain are (currently) incomplete. The M2T transformation templates for semantics-preserving application deployment are only in prototype form (not included in the online repository). The M2T templates for the automatic generation of back-annotations of network and message models in Simulink are still missing. *All these parts are under development.*

The work on automated synthesis involved several manual operations. In order to fluidize the design and optimization process, we aim to setup a toolchain integrating the design and simulation tools with optimization engines. *This work based on model-driven solutions is ongoing. The resulting automated toolchain will be used to perform a large number of experiments.*

Distributed Architectures

Apart from providing appropriate tool support, *research effort is currently being carried out to extend the contributions to RI.2 and RI.4 to distributed architectures.* With reference to the problem of automated synthesis that accounts for relaxed deadlines and order-of-execution constraints (RI.4), we are setting up a new case study for the simulated quadrotor. We add visual servoing and obstacle detection to the existing altitude and attitude control tasks. We explore the space of possible implementation solutions by comparing the control performance of the best SW realizations for multiple candidate HW architectures.

Experimental Validation

A current limitation of this work is that the validity of proposed design framework has been extensively tested in simulation, but no experimental validation was conducted. The main impediment has been setting-up a case study of sufficient complexity to justify the adoption of PBD. Ready-to-use laboratory kits are typically made up of a single powerful electronic board connected to a multitude of sensors/motors. Their architecture is highly integrated and difficult to extend. All the computations related to control run on the board, and sensors/actuators have dedicated wired buses (that rarely include CAN). In such a configuration, communication delays do not impact much the control performances, hence the effects of interplay between control and timing become significant when both a complex functionality is realized (that requires advanced skills in control engineering) and a careful evaluation of functions' execution times is available (that may be difficult to obtain). On the other hand, system architectures in use in the industry (e.g., automotive) of course do perfectly fit the scope, but (Simulink) models of functional applications are mostly unavailable, due to intellectual property concerns.

During this work, in collaboration with the research group lead by Prof. Luigi Palopoli at DISI Lab. of the University of Trento, we set up a robotic car testbench to show the application of the methodology. A preliminary description of the functional and SW/HW system architecture is described in [**MMR$^+$13**]. The system has sufficient complexity to justify the development of significant functionality using a PBD flow, including image processing, supervisory controls and low-level control loops. The execution architecture is distributed and leverages computation boards and communication technologies that are widely available. Because of its complexity and the distributed execution platform, the software and messaging architecture is not trivial and justifies timing analysis (schedulability and communication). The HW platform is available, and the functional and architecture models are currently being completed. *In the short term, we will start experimental validation of PBD flow.*

7.3. Opportunities for Future Research

Additional Execution Models

This work describes a formalized process flow for system-level modeling and timing analysis of complex CPS control applications. We consider primarily a time-triggered model of activation of control logics. Future research may explore control activation models based on events rather than periodic triggers and the possible improvements with respect to the CPU resource utilization, as discussed in [**ABEP13**] and [**LMV$^+$13**].

For what concerns the problem of control under variable computational demand, the Adaptive Variable Rate (AVR) task model described by Biondi *et al.* [**BDNB15**] represents an viable solutions. The integration of the AVR model in the process flow may enable the definition of variation points (execution modes) for the task, verified at design-time and selected at run-time to accommodate the control accuracy under different load conditions.

Fault-Tolerance

Time predictability is one of the most important design concerns for today's and future advanced CPS. However, other design aspects are critical in the system development process, e.g., *fault-tolerance*. The controller execution platform is made of heterogeneous components (generally multiple electronic control units connected with buses) that may incur transient or permanent faults. Redundancy, i.e., the replication of system HW and embedded SW processes, is one possible solution to make the controller fault-tolerant. But it has an impact on costs, especially for high-volume products like the ones from the automotive and (in future) robotics industry.

Future work may address the specification of models, abstractions and synthesis solutions to enable the exploration of fault-coverage/cost tradeoffs (the work of Pinello et al. [**PCSV08**] may provide cues on this topic) conjointly with those of control and scheduling.

Cyber-Security

Security concerns also take on great importance in the design of CPS. (Cyber-)Security mechanisms are needed to protect systems against attacks and meet requirements such as integrity, authenticity, confidentiality, or availability. In many of such systems, the tight resource constraints and strict timing requirements make it very difficult or even impossible to add security mechanisms after the initial design stages without violating other system constraints. To produce secure and safe systems with desired performance, security must be considered together with other objectives at the system level and at early design stages.

Ideas to extend the presented work in this direction may come from several sources, e.g. [**Lin15**], where an approach in line with principles of PBD is presented that enables design-time architecture exploration and selection based on security mechanisms. Future work may aim at extending that exploration strategy by also considering control-performance concerns expressed using a SR model of computation.

List of Publications

[CMDN15] F. Cremona, M. Morelli, and M. Di Natale. TRES: A Modular Representation of Schedulers, Tasks, and Messages to Control Simulations in Simulink. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC, pages 1940–1947, 2015.

[DNBM$^+$12] M. Di Natale, M. Bambagini, M. Morelli, A. Passaro, D. Di Stefano, and G. Arturi. Enabling Model-Based Development of Distributed Embedded Systems on Open Source and Free Tools. In *3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) at Euromicro*, 2012.

[MCDN14] M. Morelli, F. Cremona, and M. Di Natale. A System-Level Framework for the Evaluation of the Performance Cost of Scheduling and Communication Delays in Control Systems. *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) at Euromicro*, 2014.

[MDN13] M. Morelli and M. Di Natale. Generation of Flow-Preserving Orocos Implementations of Simulink/Scicos Models. In *8th International Workshop on Software Development and Integration in Robotics (SDIR) at IEEE-ICRA*, 2013.

[MDN14a] M. Morelli and M. Di Natale. An MDE Approach for the Design of Platform-Aware Controls in Performance-Sensitive Applications. In *Emerging Technology and Factory Automation (ETFA), IEEE 19th Conference on*, pages 1–8, Sept 2014.

[MDN14b] M. Morelli and M. Di Natale. Control and Scheduling Co-design for a Simulated Quadcopter Robot: A Model-Driven Approach. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 49–61. Springer, 2014.

[MMR$^+$13] M. Morelli, F. Moro, T. Rizano, D. Fontanelli, L. Palopoli, and M. Di Natale. A Robotic Vehicle Testbench for the Application of MBD-MDE Development Technologies. In *Emerging Technologies Factory Automation (ETFA), IEEE 18th Conference on*, pages 1–4, Sept 2013.

[Mor15] M. Morelli. Automated Generation of Robotics Applications from Simulink and SysML Models. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC, pages 1948–1954, 2015.

[MSDN$^+$15] M. Morelli, Y. Seddik, M. Di Natale, C. Mraidha, and S. Tucci-Piergiovanni. Simulation-Driven Optimization of Real-time Control Tasks. In *International Conference on Embedded Software and Systems (ICESS), IEEE*, pages 1–8, August 2015.

Bibliography

[AAD] Architecture Analysis & Design Language (AADL). `http://standards.sae.org/as5506b/`.

[ABEP13] A. Aminifar, E. Bini, P. Eles, and Z. Peng. Designing Bandwidth-Efficient Stabilizing Control Servers. In *34th IEEE Real-Time Systems Symposium*, pages 298–307, December 2013.

[ABRW94] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Stress: A simulator for hard real-time systems. *Software: Practice and Experience*, 24(6):543–564, 1994.

[AIS] AIST. OpenRTM-aist. `http://openrtm.org/`.

[AUT] AUTomotive Open System ARchitecture (AUTOSAR v4.0). `http://www.autosar.org/`.

[AVCO$^+$10] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez. V3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development. *Journal of Software Engineering for Robotics*, 1(1):3–17, 2010.

[ÅW11] K.J. Åström and B. Wittenmark. *Computer-Controlled Systems: Theory and Design*. Courier Dover Publications, 2011.

[B$^+$06] A. Benveniste et al. Hard Real-Time Development Environments. In B. Bouyssounouse and J. Sifakis, editors, *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Springer Berlin Heidelberg, New York, 2006.

[BCDN$^+$07] A. Benveniste, P. Caspi, M. Di Natale, C. Pinello, A. Sangiovanni-Vincentelli, and S. Tripakis. Loosely Time-Triggered Architectures Based on Communication-by-Sampling. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT, pages 231–239, New York, NY, USA, 2007. ACM.

[BDNB15] A. Biondi, M. Di Natale, and G. Buttazzo. Response-Time Analysis for Real-Time Tasks in Engine Control Applications. In *Proceedings of the ACM/IEEE 6th International Conference on Cyber-Physical Systems*, pages 120–129. ACM, 2015.

[BKH$^+$13] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali. The BRICS Component Model: A Model-based Development Paradigm for Complex Robotics Software Systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC, pages 1758–1764, New York, NY, USA, 2013. ACM.

[BN12] J.F. Broenink and Yunyun N. Model-driven robot-software design using integrated models and co-simulation. In *Embedded Computer Systems (SAMOS), International Conference on*, pages 339–344, July 2012.

[BPD15] E. Bini, A. Parri, and G. Dossena. A Quadratic-Time Response Time Upper Bound with a Tightness Property. In *36th IEEE Real-Time Systems Symposium*, December 2015.

[Bru] H. Bruyninckx. Open RObot COntrol Software (OROCOS). `http://www.orocos.org/`.

[Bru15] D. Brugali. Model-Driven Software Engineering in Robotics: Models Are Designed to Use the Relevant Things, Thereby Reducing the Complexity and Cost in the Field of Robotics. *IEEE Robot. Automat. Mag.*, 22(3):155–166, 2015.

[But11] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[But12] D. Buttle. Real-Time in the Prime-Time. *Keynote presentation, Euromicro ECRTS Conference*, July 2012.

[CEA] CEA List. Papyrus. `http://www.eclipse.org/papyrus/`.

[CFM+12] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh, et al. Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms. In *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 21–26, 2012.

[CHL+03] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.E. Årzén. How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and TrueTime. 23(3):16–30, June 2003.

[CMDN15] F. Cremona, M. Morelli, and M. Di Natale. TRES: A Modular Representation of Schedulers, Tasks, and Messages to Control Simulations in Simulink. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1940–1947. ACM, 2015.

[Cor11] P.I. Corke. *Robotics, Vision & Control: Fundamental Algorithms in Matlab.* Springer, 2011.

[CVMC11] A. Cervin, M. Velasco, P. Martí, and A. Camacho. Optimal Online Sampling Period Assignment: Theory and Experiments. *Control Systems Technology, IEEE Transactions on*, 19(4):902–910, 2011.

[DDM+07] A. Davare, D. Densmore, T. Meyerowitz, A. Pinto, A. Sangiovanni-Vincentelli, G. Yang, H. Zeng, and Q. Zhu. A Next-Generation Design Framework for Platform-based Design. In *Conference on Using Hardware Design and Verification Languages (DVCon)*, February 2007.

[DKS+12] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane. RobotML, a Domain-Specific Language to Design, Simulate and Deploy Robotic Applications. In *Proceedings of the Third international conference on Simulation, Modeling, and Programming for Autonomous Robots*, SIMPAR, pages 149–160, Berlin, Heidelberg, 2012. Springer-Verlag.

[DN12] M. Di Natale. Specification and Simulation of Automotive Functionality Using AUTOSAR. In K. Popovici and P.J. Mosterman, editors, *Real-Time Simulation Technologies: Principles, Methodologies, and Applications.* CRC Press, Boca Raton, FL, 2012.

[DNP08] M. Di Natale and V. Pappalardo. Buffer Optimization in Multitask Implementations of Simulink Models. *ACM Trans. Embed. Comput. Syst.*, 7(3):23:1–23:32, May 2008.

[DNSV10] M. Di Natale and A. Sangiovanni-Vincentelli. Moving from Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603–620, 2010.

[DP02] D. Decotigny and I. Puaut. Artisst: An extensible and modular simulation tool for real-time systems. In *5th IEEE ISORC Symposium*, pages 365–372, 2002.

[dSP] dSPACE GmbH. SystemDesk. `http://www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm`.

[Ecla] Eclipse Foundation. Eclipse modeling framework (emf). `http://www.eclipse.org/modeling/emf/`.

[Eclb] Eclipse Modeling Project. QVT Operational. `http://www.eclipse.org/mmt/?project=qvto`.

[EJL+03] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, et al. Taming Heterogeneity-The Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[Est] Esterel Technologies. SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite/`.

[Gen] Automatic Software Generation for Real-Time Embedded Systems (Gene-Auto). `http://gforge.enseeiht.fr/projects/geneauto`.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HA04] U. Hatnik and S. Altmann. Using ModelSim, Matlab/Simulink and NS for Simulation of Distributed Systems. In *Parallel Computing in Electrical Engineering. International Conference on*, pages 114–119. IEEE, 2004.

[HGS+13] N. Hochgeschwender, L. Gherardi, A. Shakhirmardanov, G.K. Kraetzschmar, D. Brugali, and H. Bruyninckx. A Model-Based Approach to Software Deployment in Robotics. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 3907–3914, Nov 2013.

[HMTN+08] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. In *Proceedings of the IEEE International Symposium on Industrial Embedded Systems, SIES*, 2008.

[HZPK08] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL tool Suite. *ACM Trans. Embed. Comput. Syst.*, 7(4):42:1–42:25, August 2008.

[INR] INRIA. Scicos: Block diagram modeler and simulator. `http://www.scicos.org/`.

[Int] International Organization for Standardization / Technical Committee 22 (ISO/TC 22). ISO 26262 Road vehicles – Functional safety. `http://www.iso.org/iso/home/news_index/news_archive/news.htm?refid=Ref1499`.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[Kac10] Dmitry Kachan. Integration of NS-3 with MATLAB/Simulink. Master's thesis, Lulea University of Technology, Sweden, 2010.

[KML+04] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and Cloning in Modeling and Meta-Modeling. *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling*, 12:263–278, 2004.

[L+] E.A. Lee et al. Cyber-physical systems. `http://cyberphysicalsystems.org/`.

[Leh90] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *11th IEEE Real-Time Systems Symposium*, pages 201–213, December 1990.

[Lin15] C.-W. Lin. *Security Mechanisms and Security-Aware Mapping for Real-Time Distributed Embedded Systems*. PhD thesis, University of California, Berkeley, August 2015.

[LMV+13] C. Lozoya, P. Martí, M. Velasco, J.M. Fuertes, and E.X. Martin. Resource and Performance Trade-Offs in Real-Time Embedded Control Systems. *Real-Time Systems*, 49(3):267–307, 2013.

[LSV+98] E. Lee, A. Sangiovanni-Vincentelli, et al. A Framework for Comparing Models of Computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217–1229, 1998.

[MH06] A. Metzner and C. Herde. RTSAT– An Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures. In *27th IEEE Real-Time Systems Symposium*, pages 147–158, December 2006.

[MMT+] J. Matsumura, Y. Matsubara, H. Takada, M. Oi, M. Toyoshima, and A. Iwai. A Simulation Environment based on OMNeT++ for Automotive CAN–Ethernet Networks. *Analysis Tools and Methodologies for Embedded and Real-time Systems*.

[MSHT12] Y. Matsubara, Y. Sano, S. Honda, and H. Takada. An open-source flexible scheduling simulator for real-time applications. In *15th IEEE ISORC Symposium*, 2012.

[MWTP+13] A. Mehiaoui, E. Wozniak, S. Tucci-Piergiovanni, C. Mraidha, M. Di Natale, H. Zeng, J-P. Babau, L. Lemarchand, and S. Gerard. A Two-step Optimization Technique for Functions Placement, Partitioning, and Priority Assignment in Distributed Systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES, pages 121–132, New York, NY, USA, 2013. ACM.

[Nat] National Instruments. LabVIEW System Design Software. `www.ni.com/labview/`.

[NJH09] M. Nickl, S. Jörg, and G. Hirzinger. The virtual path: The domain model for the design of the MIRO surgical robotic system. In *9th International IFAC Symposium on Robot Control, IFAC. Gifu, Japan*, pages 97–103, 2009.

[NS-] NS-3. Discrete-Event Network Simulator for Internet Systems. `https://www.nsnam.org`.

[Obe] Obeo. Acceleo. `http://www.eclipse.org/acceleo/`.

[Obj12] Object Management Group (OMG). The Systems Modeling Language (SysML). `http://www.omg.org/spec/SysML/1.3/`, 2012.

[(OMa] Object Management Group (OMG). Action Language for Foundational UML (ALF). http://www.omg.org/spec/ALF/.

[(OMb] Object Management Group (OMG). Model Driven Architecture (MDA). http://www.omg.org/mda/specs.htm.

[(OMc] Object Management Group (OMG). MOF Model to Text Transformation Language (MOFM2T). http://www.omg.org/spec/MOFM2T/.

[(OMd] Object Management Group (OMG). Object Constraint Language (OCL). http://www.omg.org/spec/OCL/.

[(OMe] Object Management Group (OMG). Query/View/Transformation (QVT). http://www.omg.org/spec/QVT/.

[(OMf] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models (fUML). http://www.omg.org/spec/FUML/.

[(OMg] Object Management Group (OMG). The Meta-Object Facility (MOF). http://www.omg.org/mof/.

[(OMh] Object Management Group (OMG). Unified Modeling Language (UML). http://www.omg.org/spec/UML/.

[(OM11] Object Management Group (OMG). A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems. http://www.omg.org/spec/MARTEs, 2011. OMG Document Number formal/2011-06-02.

[OMNa] OMNeT. An Extensible, Modular, Component-Based C++ Simulation Library and Framework for Building Network Simulators. http://www.omnetpp.org.

[OMNb] OMNeT++. An Extensible, Modular, Component-Based C++ Simulation Library and Framework for Building Network Simulators. http://www.omnetpp.org.

[Ope] Open Source Robotics Foundation. Robot Operating System (ROS). http://www.ros.org/.

[OSE04] OSEK. OSEK/VDX Communication Specification. http://portal.osek-vdx.org/files/pdf/specs/osekcom303.pdf, 2004. Version 3.0.3. [Online; accessed 01-October-2013].

[PCSV08] C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli. Fault-Tolerant Distributed Deployment of Embedded Control Software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 906–919, 2008.

[Pro] Project P. http://www.open-do.org/projects/p/.

[RGR$^+$10] G. Raghav, S. Gopalswamy, K. Radhakrishnan, J. Delange, and J. Hugues. Model Based Code Generation for Distributed Embedded Systems. In *Proceedings of the 2010 on Embedded Real Time Software and Systems*, 05 2010.

[RLSS10] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-Physical Systems: the Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference*, DAC, pages 731–736, New York, NY, USA, 2010. ACM.

[RTC] RTCA, Inc. DO-178B: Software Considerations in Airborne Systems and Equipment Certification. http://www.rtca.org/.

[RTS] RTSim. A Simulator for Real-Time Systems. http://rtsim.sssup.it.

[Sci] Scilab Enterprises. Xcos. http://www.scilab.org/scilab/features/xcos.

[SG14] B. Selić and S. Gérard. Chapter 12 - Extending MARTE [Advanced]. In B. Selić and S. Gérard, editors, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*, pages 237 – 249. Morgan Kaufmann, Boston, 2014.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM, 2004.

[SSBK10] C. Schlegel, A. Steck, D. Brugali, and A. Knoll. Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering. In *Proceedings of the 2nd international conference on Simulation, Modeling, and Programming for Autonomous Robots*, SIMPAR, pages 324–335. Springer, 2010.

[SSVDB+05] M. Sgroi, A. Sangiovanni-Vincentelli, F. De Bernardinis, C. Pinello, and L. Carloni. Platform-Based Design for Embedded Systems. In *Embedded Systems Handbook*. CRC Press, 2005.

[SV02] A. Sangiovanni-Vincentelli. Defining Platform-Based Design. *EEDesign of EETimes*, February 2002.

[SV07] A. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[SVDN07] A. Sangiovanni-Vincentelli and M. Di Natale. Embedded System Design for Automotive Applications. *Computer*, 40(10):42–51, October 2007.

[SVS14] A. Sangiovanni-Vincentelli and D. Sciuto. Looking into the Crystal Ball: From Transistors to the Smart Earth. *Design Test, IEEE*, 31(2):47–55, April 2014.

[Thea] The MathWorks, Inc. SimEvents. A Discrete-Event Simulation Engine and Component Library for Simulink. `http://www.mathworks.it/products/simevents/`.

[Theb] The MathWorks, Inc. Simulink. `http://www.mathworks.com/products/simulink/`.

[UDT10] R. Urunuela, A. Deplanche, and Y. Trinquet. STORM a Simulation Tool for Real-Time Multiprocessor Scheduling Evaluation. In *IEEE ETFA Conference*, 2010.

[VD06] Y. Vanderperren and W. Dehaene. From UML/SysML to Matlab/Simulink: Current State and Future Perspectives. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 93–93. European Design and Automation Association, 2006.

[WDNSV09] G. Wang, M. Di Natale, and A. Sangiovanni-Vincentelli. Improving the Size of Communication Buffers in Synchronous Models with Time Constraints. *Industrial Informatics, IEEE Transactions on*, 5(3):229–240, 2009.

[web] BRICS - Best practice in robotics. `http://www.best-of-robotics.org/`.

[WNBG12] S. Wätzoldt, S. Neumann, F. Benke, and H. Giese. Integrated Software Development for Embedded Robotic Systems. In *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, SIMPAR, pages 335–348, Berlin, Heidelberg, 2012. Springer-Verlag.

[ZDN13] H. Zeng and M. Di Natale. An Efficient Formulation of the Real-Time Feasibility Region for Design Optimization. *Computers, IEEE Transactions on*, 62(4):644–661, April 2013.

[ZZZ+13] Q. Zhu, H. Zeng, W. Zheng, M. Di Natale, and A. Sangiovanni-Vincentelli. Optimization of Task Allocation and Priority Assignment in Hard Real-time Distributed Systems. *ACM Trans. Embed. Comput. Syst.*, 11(4):85:1–85:30, January 2013.

INSTITUTE
OF COMMUNICATION,
INFORMATION
AND PERCEPTION
TECHNOLOGIES

Scuola Superiore
Sant'Anna