UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Electrical Engineering, Computer Science, and Mathematics
Department of Computer Science
Warburger Straße 100
D-33098 Paderborn

An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems

by

Marco Aurélio Wehrmeister

A thesis submitted to the
Graduate Program in Computer Science (PPGC)
Federal University of Rio Grande do Sul
and to the
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Ph.D.) and **Doctor of Natural Science (Dr. rer. nat.)**

Paderborn, September 2009

Acknowledgments

I have a noticeable improvement in my professional/personal life after these six years working in the Embedded Systems Lab at the Federal University of Rio Grande do Sul. I have been with many people that contributed directly or indirectly to this improvement. I would like to express gratitude to all of them.

First of all, I would like to thank Dr. Carlos Eduardo Pereira. He is not only my advisor but also a friend. His help along two years of master and four years of Ph.D. did strongly contribute to several aspects of my professional and personal life in Porto Alegre. Our discussions have been a fundamental piece for the accomplishment of this work. I tried to learn the maximum I could from his example.

Other important part of my Ph.D. was my "sandwich" stage at the University of Paderborn, Germany. For this, I would like to thank Dr. Franz Josef Rammig for accepting me as member in his working group, for the discussions, critics and suggestions on my work, and more specially, for the opportunity to do the bi-national Ph.D. I do not have words to describe how this stay in Germany opened my horizons concerning personal and professional aspects.

From home, I would like to thank my wife, Josi, for her support, encouraging, and comprehension in the last six years. She was and still is a fundamental piece during all phases of my life. I am also thankful to my parents, Nelson and Berta, my brothers Fernando and Leonardo, and my sister-in-law Thaize for the encouragement and support. Specially, I would like to thank my uncle Vendelino, aunt Janete, and cousins Rudolf, Priscilla, Bárbara, and Júlia for their support in Porto Alegre, decreasing the yearning for the family of Blumenau. In special, I would like to thank Bárbara for the English review of some chapters of this thesis.

I would also like to acknowledge all professors and administrative staff of the Informatics Institute, which somehow contributed for the conclusion of my Ph.D. Specially, I would like to thank Dr. Flávio Rech Wagner, Dr. Luigi Carro, and Dr. Marcelo Soares Pimenta for the valuable discussions, critics, and suggestions during all this time.

I am thankful to all colleagues and friends from the Embedded Systems Lab who directly or indirectly contributed to this work. I cannot name all of them because I will certainly forget many names. However, I should mention those that provide remarkable contributions to this work: Edison Pignaton Freitas, Marcio Oliveira, and Elias Teodoro da Silva Jr. Additionally, I would like to thank two undergrad students: William Silva for the help with some case studies in ORCOS platform; and Ronaldo Rodrigues Ferreira (a.k.a. Bixo) for the English review of

this text's first draft.

Contents

List of Figures

List of Tables

Abstract

Currently, the design of distributed embedded real-time systems is growing in complexity due to the increasing amount of distinct functionalities that a single system must perform, and also to concerns related to designing different kinds of components. Industrial automation systems, embedded electronics systems in automobiles or aerial vehicles, medical equipments and others are examples of such systems, which includes distinct components (e.g. hardware and software ones) that are usually designed concurrently using distinct models, tools, specification, and implementation languages. Moreover, these systems have domain specific and important requirements, which do not represent by themselves the expected functionalities, but can affect both the way that the system performs its functionalities as well as the overall design success. The so-called non-functional requirements are difficult to deal with during the whole design because usually a single non-functional requirement affects several distinct components.

This thesis proposes an automated integration of distributed embedded real-time systems design phases focusing on automation systems. The proposed approach uses Model-Driven Engineering (MDE) techniques together with Aspect-Oriented Design (AOD) and previously developed (or third party) hardware and software platforms to design the components of distributed embedded real-time systems. Additionally, AOD concepts allow a separate handling of requirement with distinct natures (i.e. functional and non-functional requirements), improving the produced artifacts modularization (e.g. specification model, source code, etc.). In addition, this thesis proposes a code generation tool, which supports an automatic transition from the initial specification phases to the following implementation phases. This tool uses a set of mapping rules, describing how elements at higher abstraction levels are mapped (or transformed) into lower abstraction level elements. In other words, such mapping rules allow an automatic transformation of the initial specification, which is closer to the application domain, in source code for software and hardware components that can be compiled or synthesized by other tools, obtaining the realization/implementation of the distributed embedded real-time system.

Keywords: Model-Driven Engineering (MDE), Aspect Oriented Development (AOD), UML, Code Generation, Aspects Weaving, Real-Time Embedded Systems

Chapter 1

Introduction

1.1. Motivation

The use of specialized electronic devices to assist in daily activities is increasing rapidly. The so-called embedded systems are hardly perceived as computing systems. Currently, at least 20-30 embedded systems can be found in a common household, e.g. cell phones, digital cameras, DVD players, microwave ovens, car's electronic systems and others. On the other hand, the same household has only 1 or 2 desktop computers or laptops [128]. Moreover, many of these systems have several tasks distributed in multiple processing units (deployed either locally or physically distant from each other) that must cooperate to accomplish a common goal, while respecting stringent application's real-time requirements. For example, in a modern middle-range car, it is possible to find over 50 embedded systems controlling several functions ranging from anti-lock braking (ABS) and fuel injection systems to infotainment systems such as GPS navigator or a music/video player [128]. To meet this high demand from industry, many researchers propose/develop methodologies, standards, architectures and tools to assist the systematic development of such special kind of distributed, cooperative and real-time embedded systems.

As technology advances faster, there is an increasing demand for new embedded systems capable of performing a large amount of complex functionalities, which impact strongly in their design time and complexity. Such growing complexity is partially caused due to the distinct nature of elements involved in the design of these systems, i.e. designers must produce, usually concurrently, hardware (HW) and software (SW) components. However, as one can see in figure 1.1, there is a productivity gap between the software and hardware teams: the first one needs 5 years to increase productivity twice, while the later improves it a little faster but still not in the same rate as the increase in technology capabilities. In addition, the non-functional nature of some important requirements have a great influence in design complexity. The embedded systems domain has characteristics that constrain system design, such as fewer availability of computational resources (e.g. memory and processing power), restrictions on low energy consumption without performance degradation, and also a tight time-to-market [25].

To deal with the above mentioned problems, researchers and designers propose to raise the
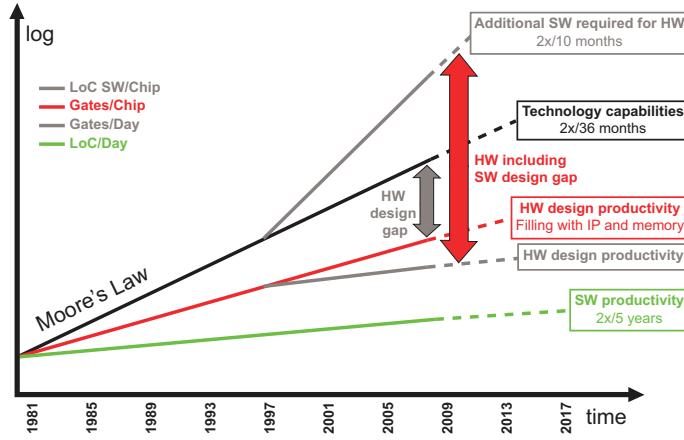
Figure 1.1.: Hardware and software design gap [66]

abstraction level used during system design. Figure 1.2 shows a chart from a recent embedded systems market survey [81], in which it can be seen that approximately 43% of embedded systems designers use higher levels of abstraction, such as UML, Simulink or SystemC, in their projects. In this context, the Object-Oriented (OO) paradigm appears as an interesting choice due to some characteristics, such as abstraction and hierarchy, which are pointed since the 70's as key concepts to manage complexity growth and the increasing design effort [53]. Over the last years, the use of OO in the design of distributed embedded real-time system is the focus of several works, as can be seen in important conferences and publications, as for example the IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), [78], or [27]. However, not all issues involved in the design of distributed embedded real-time systems are well handled only by using OO concepts. The crosscutting nature of some important requirements impacts in different parts of a system (i.e. non-functional requirements crosscut functional requirements), hindering the reuse of produced artifacts (e.g. models, source code, IPs, etc.) [42].
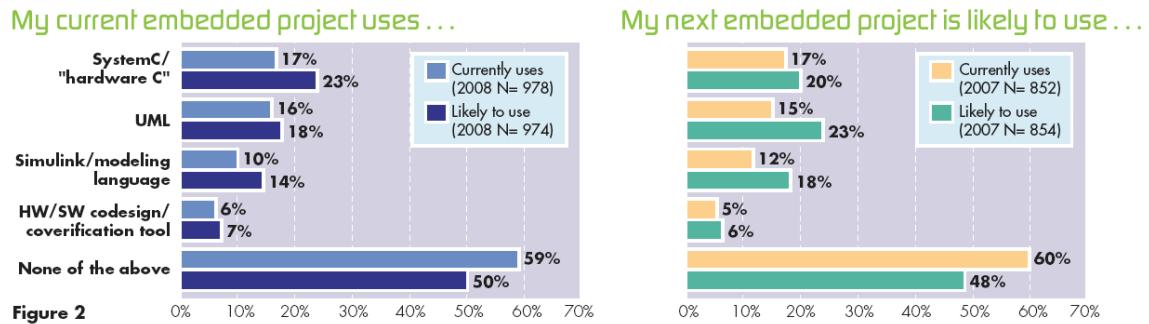
Figure 1.2.: Using higher abstraction levels in embedded system design [81]

In the literature there are some proposals, e.g. [119] and [124], that suggest the use of aspects to deal with the problem of crosscutting non-functional requirements in embedded systems design. The Aspect-Oriented (AO) paradigm [42] allows a separated specification of system's functional and non-functional requirements. Additionally, it allows designers to concentrate efforts on important concerns, such as the handling of real-time, performance and energy consumption constraints. Additional AO helps in decoupling the produced artifact allowing their reuse in the same or further projects. Thus, the achieved separation of concerns in requirements handling can improve the design of distributed embedded real-time systems, opening room for reusing the produced artifacts. Usually, non-functional requirements affect (i.e. crosscut) functional requirements in different ways, in different design phases and/or in different system modules [42]. Traditional OO approaches do not handle these requirements in a satisfactory form. To illustrate the this situation, let's consider the control of concurrent access to a shared resource. The code responsible for handling this requirement must be added each time some task needs to use a shared resource, and hence, it is scattered within different modules.

Another way to decrease the gap between hardware and software designs, and also the design time, is the adoption of a common language to specify both the structure and behavior of a distributed embedded real-time system [128]. Thus, the information exchange between design teams (i.e. teams that develop hardware and software components) is facilitated, minimizing possible misunderstandings in the specification [27]. In the last years, it can be observed the increasing use of the Unified Modeling Language (UML) [92] in the design of embedded systems. Such claim can be confirmed in the book "UML for SoC design" [78], which describes different research works that proposed the use of UML to design Systems-on-Chip (SoC).

The idea of using models to design complex systems is becoming stronger because models help in the understanding of complex problems and their potential solution through higher levels of abstract in the specification [113]. Based on the fact that models are essential for traditional engineering projects (e.g. the construction of buildings, the aerodynamic design of an aircraft or the construction of an electromechanical engine), several researchers and industry professionals advocate that models produced during the design of computational systems must play the main role during the whole design cycle [114].

The so-called Model-Driven Engineering (MDE) [113, 110] defines that the design should mainly focus on the creation of graphical models instead of writing source code for computer programs. Hence, models are the most important artifacts in the design of computational systems because they are easier to specify, understand and maintain. Besides, they are less sensitive to changes in the implementation technology, in other words, models are intended to be platform and technology independent. To support this idea, a fundamental premise of MDE is that system implementation (e.g. source code) must be automatically generated from models, avoiding discrepancies among models and the actual system implementation. One example of standardization for MDE is the Model-Driven Architecture (MDA) [87], which offers a conceptual framework and a set of standards to be used in the development of general-purpose systems, and proposes the use of UML for the specification modeling language. The transition from a Platform Independent Model (PIM) to a Platform Specific Model (PSM) is performed through standardized models transformations specified using the MOF Query/View/ Transfor-

mation (QVT) language [93]. However, in spite of this infrastructure, UML and MDA do not have elements to deal with functional and non-functional requirements in a separated manner because they only use OO-based concepts.

Despite all work done in academy, the use of high-level models/languages is not a common practice in current industry projects as can be seen in figure 1.3 also from [81]. Analyzing together figures 1.2 and 1.3, one can infer that designers want to use higher abstractions levels during design, however the most important used tools are those that deal with low-level artifacts, such as compilers or debuggers. One possible reason for this situation is due to the fact that low level tools, such as compilers and debuggers, are much more mature. Other relevant reason for this situation might also be that current available tools and methodologies do not fulfill the (critical) needs of actual designs. Therefore, it is clear that there is a need for improved tools supporting high-level techniques. Tool support is key to allow the use of MDE to cope with the complexity of the embedded systems design [114].

1.2. Goals and Scope Delimitation

Considering the mentioned shortcomings in the design of distributed embedded real-time systems, this research work has looked for solutions for the following problems: *(i)* manage the complexity to handle requirement of distributed embedded real-time system; *(ii)* support for separation of concerns in the handling of functional and non-functional requirements; *(iii)* the use of a common language to describe the initial specification (i.e. model) of system structure and behavior; and *(iv)* productivity increasing through an automatic transition from initial design phase, e.g. modeling, to further phases, e.g. implementation.

As embedded systems are used in very distinct application domains, applying systems with a broad range of different characteristics and capabilities, this work restricts itself to distributed embedded real-time systems applied to automation systems, such as industrial and home automation, or electronic control systems of vehicles and aircrafts. Thus, to overcome the mentioned problems in the design of such applications, this work advocate the increase of the abstraction level by using models as the main artifact used during the whole design. As a result, the produced high-level models can be successively refined until a system implementation is obtained.

To specify models for distributed embedded real-time systems, this work recommends the use of a standard modeling language, such as UML. Its diagrams are used to describe functional requirements, as well as those requirements related to the Quality of Service (QoS) required/offered from/by system elements, which are specified using stereotypes from the recently approved UML Profile for *Modeling and Analysis of Real-time and Embedded Systems* (MARTE) [94]. Furthermore, it is proposed to handle non-functional requirements already in earlier phases, separating these requirements handling from functional ones. Thus, to handle non-functional requirements, this work proposes that AO concepts must be applied combined with UML models. To achieve such goal, aspects must deal with real-time, performance and distribution requirements, as well as energy consumption, memory and area usage. It is impor-

Which of the following are your favorite/most important software/hardware tools?
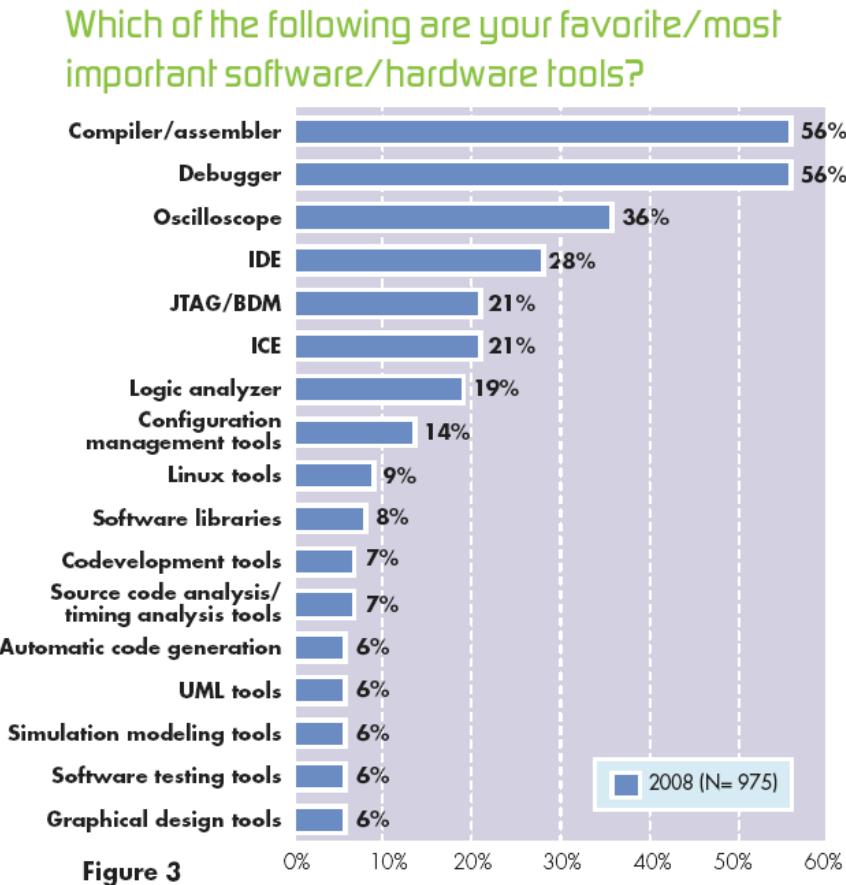
Figure 1.3.: Most important tools in embedded system design [81]

tant to highlight that there are other equally important requirements in the domain of distributed embedded real-time systems, e.g. fault tolerance, which are complex enough (by themselves) to be dealt within the scope of other thesis. Consequently, for scope delimitation, this work does not consider them.

According to Selic [113], models can be considered only project's documentation for requirements (functional and non-functional) of distributed embedded real-time systems. In such situation, designers might consider their real value too small because models may easily diverge from the system real implementation. To overcome this problem, a fundamental premise of MDE is to have adequate tool support to allow automatic generation of system implementation from their high level models. Hence, other goal of this work is to provide a tool capable of generating code from UML's structural and behavioral diagrams. Moreover, it must be aware of AO concepts specified within the model, i.e. it must identify the used aspects, as well as the functional elements affected by them. The adaptation performed by the aspect must be woven in the generated source code. Additionally, this code generation tool must be flexible, i.e. it must not constrain the generated source code to a specific target language. To achieve these

goals, the tool can use scripts to generate code fragments for each element in the UML model.

According to the motivations described until here, this work has the following goals:

- To propose a design flow, which allows the use of MDE and AOD techniques, improving and increasing the reuse of previously developed and tested artifacts (e.g. models, libraries, mapping rules, code generation scripts, etc.);

- To advocate for the use of UML diagrams decorated with MARTE profile stereotypes in combination with aspects (from a high-level aspects framework) for the specification of the structure, behavior and non-functional requirements handling in the design of distributed embedded real-time systems. This will put together initial system specification using a well-know and accepted standard, which helps in information exchange about system characteristics and expected functionalities among design teams (i.e. hardware and software teams);

- To improve separation of concerns in the handling of requirements, i.e. functional requirements are handled apart from the non-functional ones;

- To propose modeling guidelines, as well as UML diagram interpretation semantics to eliminate or at least decrease the ambiguity in diagrams interpretation. This allows the transformation from the UML meta-model to a defined meta-model, whose semantics is more suitable for code generation due to its accuracy in the specification of system structure and behavior;

- To create a tool for code generation to support the automatic transition from specification to implementation phases. The tool must support means for specifying mapping rules to transform model elements into source code constructions in the chosen target language. The generated code must be as complete as possible, meaning that the code should not contain only class skeletons;

- Mapping rules must allow their further reuse. In other words, they must be described in such a way that it might be possible to create a repository of created mapping rules. However, it is important to highlight that the definition of such repository is out of the scope of this Ph.D. thesis;

- To evaluate - using software engineering metrics - if the proposed approach and, in particular, the use of AO positively impacts in the system specification, and also in the automatic generation of source code.

1.3. Thesis Contributions

This work was developed within the context of the SEEP project (SEEP stands for *Platform-based Embedded Systems* or "*Sistemas Elerônicos Embarcados baseados em Plataformas*"). Following SEEP ideas, the main goal is to provide mechanisms to manage the increasing design complexity by using MDE techniques and separation of concerns in the handling of functional

and non-functional requirements. Therefore, this work's contributions are as follow:

Use of MDE techniques in embedded systems design: The use of models to assist in the development of software for general purpose computes is not a new research topic. In addition, there are already some works on the "model-driven engineering" topic proposing solutions to some problems. However, the employment of MDE in the design of distributed embedded real-time systems can be considered a recent research topic, which still has several gaps to be fulfilled. Thus, it can be stated that the study and assessment of model-driven methodologies and techniques applied to the design of distributed embedded real-time systems is a relevant contribution.

Platform independent modeling of embedded systems: This work suggests the use of UML and the recently approved MARTE profile together with concepts of AOD. This can also be considered a contribution in the design of distributed embedded real-time systems because they allow initial system description without considering its implementation. In other words, it is possible to specify structure, behavior, as well as crosscutting non-functional requirements without concerning if an element will be implemented as software or hardware, allowing a unified system specification that can be understood by both software and hardware design teams.

Handling of crosscutting non-functional requirements in embedded systems design: Other remarkable contribution is the aspect framework created to handle non-functional requirements of distributed embedded real-time systems domain. To the best of our knowledge, up to now there is no work in the literature reporting the creation/ development of a high-level aspects framework that can be used in both modeling and implementation levels.

Tool support for the proposed design flow: The code generation tool proposed in this work is also an important contribution because it provides an automatic transition from the modeling phase to the implementation of distribution embedded real-time systems. Again, to the best of our knowledge, there is no other tool providing the same flexibility allowed in the specification of the mapping rules scripts. These small scripts concentrate only on one or few model elements, which facilitates the specification of mapping rules. It is important to highlight that it is expected that the tool could also generate HDL code from the UML model. Other functionality provided by this tool is the generation of platform configuration, in term of either configuration files, or platform source code tailoring. In other words, besides generating configuration files, the target platform can be configured by means of removing source code lines (related to unused platform services) from its source code files.

Tool support for aspects weaving: A very important contribution of the code generation tool is the ability to weave aspects adaptations. It is possible to modify the generated code

fragment using aspects (i.e. aspects weaving in the source code), as well as modify the high-level model (i.e. aspects model weaving). Such feature was not found in any tool available in industry or academy. Moreover, aspects specified within the UML model steer the platform customization, meaning that platform services are included depending on which aspect have been specified in the model.

1.4. Text Organization

The remainder of this text is structured as follows: *Chapter 2* presents an overview on the basic concepts used in this text. It includes key concepts related to embedded real-time systems, as well as requirements present in this domain; concepts of OO and AO paradigm; MDE and platform-based approaches, and also a short overview of the MARTE UML profile.

In *Chapter 3*, the state of the art is discussed. Following topics are covered: design and modeling approaches for embedded systems; handling of embedded systems requirements; and code generation techniques.

Chapter 4 presents the design flow proposed in this work, named *Aspect-oriented Model-Driven Engineering for Real-Time systems* (AMoDE-RT), which supports activities from requirements analysis to system realization using a target platform.

Chapter 5 discusses guidelines for using UML to specify system structure and behavior. It also introduces an aspects framework, named *Distributed Embedded Real-time Aspects Framework* (DERAF), which provides aspects with high-level semantics to specify the handling of crosscutting non-functional requirements within UML models.

Chapter 6 introduces the code generation tool named GenERTiCA (*Generation of Embedded Real-Time Code based on Aspects*) created to support the AMoDE-RT design flow. Further, this chapter presents an intermediate PIM named *Distributed Embedded Real-time Compact Specification* (DERCS), discussing how to transform UML models into DERCS models. The code generation and aspects weaving approaches used by GenERTiCA, as well as the specification of mapping rules to produce source code from the UML model, are also discussed in this chapter.

Three case studies, that illustrate the proposed approach and GenERTiCA usage, are presented in *Chapter 7*. The case studies are: the movement control of an Unmanned Aerial Vehicle (UAV), the movement control of a wheelchair, and the control systems for an automated packing system. Additionally, this chapter provides an evaluation of the AMoDE-RT approach based on a set of software engineering metrics.

Finalizing, *Chapter 8* presents the conclusions of this work, and also draws directions for future work.

Chapter 2

Theoretical Background

2.1. Introduction

This chapter presents some concepts used within the context of this text. The goal here is to provide basic understanding and some references for relevant concepts addressed in this text. For a more detailed discussion on them, interested readers are referred to text books and tutorials such as: [24], [72], [25], [132], [115], [14], [42], [70], [122] and [117].

2.2. Distributed Embedded Real-Time Systems

2.2.1. Introduction

Distributed Embedded Real-Time Systems can be defined as systems that must precisely meet time requirements in spite of their running tasks be distributed in different processing units, having few available physical resources [122]. They must provide temporal predictability while performing multiple concurrent and communicating tasks, which are deployed on different resource constrained processing units (sometimes physically distributed over different locations), e.g. processing power, amount of available memory, or energy consumption restrictions. In the sequence, details on each characteristic that defines a distributed embedded real-time system are presented.

2.2.2. Real-Time Systems

Real-time systems are a special kind of computational systems on which the correct processing of an algorithm is not enough to ensure correct system behavior, i.e. the algorithm worst case execution time must be predictable, as well as algorithm results must be delivered in predefined time instants, meeting the application's time requirements. Thus, real-time systems are

considered deterministic systems [72]. The ability to process data in milliseconds or even in nanoseconds does not define a computational system as a real-time system; what really matters is that system response times are limited and predictable. Stankovic [118] presents several misconceptions and misunderstandings on real-time systems and their definition.

When considering the accomplishment of real-time requirements, real-time systems can be classified in two categories: *Hard Real-Time System* and *Soft Real-Time System*. The former represents systems that will have critical failures, which can cause catastrophic losses, if any time constraint is not fulfilled [24]. On the other hand, the later represents systems that can continue their execution, in a degraded operation mode, even when some time requirements are missed. Hard real-time systems are commonly found interacting with the environment, such as embedded control systems. For example, a car's engine supervisory system is an embedded hard real-time system because a late response can damage the engine and the passenger. Consequently, such situation can lead to losses of car's occupants' lives. Other examples are medical devices or industrial control systems, whose malfunctioning can cause, respectively, life and monetary losses. Soft real-time systems are commonly applied in systems that receive data streams that need to be processed. The processed result is than delivered to other components or systems connected to the soft-real time system. As an example, entertainment audio/video broadcast systems can be mentioned. In this system, the not fulfillment of time requirements decreases the video and audio quality but the system remains operating.

Furthermore, real-time systems domain has some important concepts that should be highlighted [24]:

- **Deadline** is the maximal time instant at which a task must provide its results, i.e. the system has to finish execution of a given algorithm within a maximal time limit. Deadlines are key issues in hard real-time systems.

- **Worst Case Execution Time** (WCET) is the maximal time spent for an algorithm to finish its execution and deliver the computed results.

- **Period** is the time interval between two consecutive executions of an activity.

- **Predictability** is a key characteristic for real-time systems because their behavior must be know. Latency and jitter must be guaranteed within a known maximal time interval. *Latency* indicates the time spent from the stimulus detection until the execution of the code responsible to handle such stimulus. *Jitter* is a random variation in the timing of a signal, especially a clock.

- **Exception handling** can be performed to overcome the problems caused by deadlines misses, or unexpected latency or jitter. Hence, corrective actions are performed in order to alleviate or even to eliminate the effects of a temporal failure.

2.2.3. Embedded Systems

There are many definitions for embedded systems, some are contradictory while others are complementary [128]. However, there is an important characteristic that is shared among all definitions and allows separating embedded systems from general-purpose systems: the ability to perform specialized tasks for specific purposes within the context of a larger system. Usually, these specific purpose systems have less processing power than general-purpose systems [132].

The current processors market share indicates that more than 90% of the sold processors are used in embedded systems [128]. Almost all modern electronic devices, from toys and cell phones to vehicles or industrial embedded control systems, use microprocessors or microcontrollers to deliver their expected functionality. As can be noted, the proportion between the usage of general-purpose processors and embedded processors is huge.

Usually, embedded systems should use processors with lower energy consumption, given that in many application they impact in the processing power delivered to the software application. Embedded systems are often built with limited memory resources due to other constraints such as components cost, physical size, or energy consumption, requiring very optimized operating systems or even their elimination. Hence such operating system must provide only the amount of services required by the application software. In spite of all these constraints, the requirements of the target application lead the decision on which processor or memory amount to use, or if the systems will use an operating system [25].

Many embedded systems are developed assuming they must be used for a long period of time without maintenance. The fact is that the intention is to produce an embedded system for a given application domain, letting it operate autonomously for its entire expected lifetime [132]. For that reason, many embedded systems do not own mechanical parts, e.g. fans, magnetic disks, etc. These sorts of components are affected by natural harm caused by their use, thus these components need to be replaced or fixed. Besides, there are alternative components that provide the same functionality, e.g. ROM and flash memory components can store both the operating system and application software.

2.2.4. Distributed Systems

Distributed systems are systems composed by a collection of processors with their own local memory, i.e. they do not share memory. These processors are usually spatially distributed and are connected through a communication infrastructure. In a distributed system, the goal is to decentralize processing among the processors in a transparent way without given indications of this split to the final user, for which the system does not appear to be distributed [122].

Besides providing cooperation among multiple processors aiming at increasing the whole processing power, distributed systems are also applied in applications requiring decentralization due to special needs, such as steer-by-wire systems, which have interconnected sensors and actuators deployed in each wheel and also in the steering wheel in order to improve the overall performance of the system.

The most remarkable characteristics of distributed systems are related to their technical issues [111]:

- The architecture can adopt the following approaches: *(i)* client-server; *(ii)* publisher-subscriber; *(iii)* peer-to-peer;

- There is a mechanism to control how, when and where concurrent processes should execute;

- There is a mechanism to control concurrent access to shared resources. In other words, concurrent processes should synchronize their access to such shared resources in order to guarantee data integrity; and

- As processes communicate with each other, there is a communication control mechanism. It should allow correct messages delivery to their destinations.

2.3. Requirements in Embedded Systems Domain

In software industry, there is no common definition on what the term *requirement* really means [115]. There are two extremes: on one hand, it represents high level and abstract statements of services provided by the system or constraints that it must fulfill; on the other hand, it represents detailed, mathematically formal definition of system functions. According to Sommerville [115], there is different level of system specification, which are intended to different types of readers:

- **User requirements** are statements, usually in natural language, for client and contract managers that do not have a detailed technical knowledge;

- **System requirements** are detailed statements on system services and constraints. The system requirements document is intended to senior technical staff and project managers;

- **Software design specification** is an abstract definition of the software design, which is the base for the following design and implementation phases. Thus, it is intended to software engineers who will, in fact, develop the system.

In this text, the term *requirements* is used to refer to system requirements. An important sub-classification is the separation of system requirements in:

- **Functional requirements** specify services provided by a system, along with how it should react to certain inputs, and how it should behave in particular situations. Functional requirements specification must be complete (i.e. all services required by users should be provided) and consistent (i.e. requirements should not have contradictions). For large and complex systems, it is almost unfeasible to achieve functional requirements consistency and completeness [115];

- **Non-functional requirements**, as the name suggests, are not concerned with functions delivered by the system. Rather, they are constraints on the services or functions, or

supporting elements that assist the execution of such services and functions [32];

- **Domain requirements** are obtained from characteristics of the target domain rather than user needs. They can be functional or non-functional, representing the fundamentals of the application domain, e.g. a requirement for the deceleration of a train in an automated train protection system.

Concerning the design of distributed embedded real-time systems, non-functional requirements are as important as functional requirements. In embedded systems domain, it is not uncommon to have non-functional requirements that are (in some sense) contradictory, such as for instance performance and energy consumption. Thus, non-functional requirements must be classified in order to help in their handling during design. Even though not particularly intended to distributed embedded real-time system, a good example of non-functional requirements classification is the one presented by Bertagnolli [10], which describes, in details, a classification for non-functional requirements related to fault-tolerant systems. As one can suppose, some of these requirements can be found in the distributed embedded real-time system domain. However, according to Freitas [37], there are other important non-functional requirements that are commonly found in this domain, as follows: *(i)* time; *(ii)* performance; *(iii)* distribution; and *(iv)* embedded issues. The classification of such requirements is shown in figure 2.1.

Time issues, such as real-time constraints and characteristics, are depicted under the ***time*** branch, which was also divided in two sub-branches:

- *Timing:* in this branch, it can be seen typical elements of a real-time system such as deadlines, activation period and cost (i.e. WCET); these were discussed in section 2.2.2. However, there are other important requirements:

  - *Release time* represents the moment at which an activity is ready to be executed;
  - *Activation latency* is the delay to start an activity execution, i.e. difference between the instant at which an activity became ready to execute and the instant of the beginning of its real execution;
  - *Start* is the time instant at which an activity begins its execution; and
  - *End* is the time instant at which an activity finishes its execution.

- *Precision:* under this sub-branch, one can see requirements related to QoS in the accomplishment of real-time constraints, such as jitter that was also discussed in section 2.2.2. Following, there are other requirements:

  - *Tolerated delay* represents the maximum latency the can be admitted;
  - *Laxity* is obtained by calculating the deadline minus the WCET of an activity, representing this activity's maximum idleness.
  - *Freshness* is the time interval on which the associated data is considered valid;
  - *Resolution* defines the lowest time granularity (e.g. nanoseconds, milliseconds, etc.) in which the system can operate.
  - *Drift* represents deviation of system's logical time from physical time.

Non-functional requirements under the ***performance*** branch represent constraints related to both time and distribution non-functional requirements. For this reason, they received a
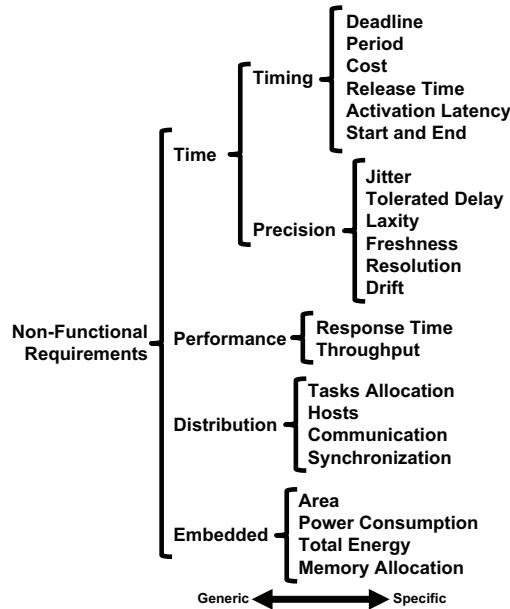
Figure 2.1.: Non-Functional Requirements Classification [37]

separated classification [37]. Basically, *throughput* refers to the rate an element can deliver its results, be them results from an algorithm execution or messages sent/received. *Response time* represents the delay after which the system delivers a result, which depends on the execution of both local and remote activities.

The classification related to distribution non-functional requirements is not complete. Figure 2.1 show only the most relevant ones. As can be seen in the **distribution** branch, there are four most common non-functional requirements:

- *Task allocation* refers to deployment of activities on different processing units that compose the distributed embedded real-time system. Associated with other non-functional requirements, it is also related with allocation such activities in nodes with different capabilities, aiming at meeting real-time constraints;

- *Hosts* is related to node monitoring. The status of all nodes, which participate in the accomplishment of system activities, need to be regularly checked, in oder to evaluate if they are working as expected. Usually, it is associated with the task allocation requirement;

- *Communication* is associated with communication features, such as network topology, connection type among nodes (e.g. connection-oriented, connectionless), if communication should use an acknowledgment mechanism or not, if messages should be encrypted or not, among other communication characteristics;

- *Synchronization* defines policies for concurrent access to shared resources. This affects the form concurrent activities perform their actions, which, depending on the adopted policy, can affect the overall system performance.

The last branch is related to ***embedded*** non-functional requirements. These requirements are closer related to design constraints, i.e. they represent constraints that can influence directly in the performance, and hence, the fulfillment of other constraints. They were divided in three features:

- *Area* constrains the system physical size and/or the amount of hardware.This requirement can demand monitoring and management activities in order to optimize the usage of system hardware, or even migration of activities from software to hardware, and vice-versa;

- *Energy* requirements constrain system runtime in terms of energy and power consumption. Such constraints have more impact in distributed embedded real-time systems that use batteries as power supply, due to the fact that the system stops if batteries run out of charge. Additionally, power dissipation can also be a problem in portable systems due to devices overheating. Such issues must be carefully considered during design;

- *Memory* non-functional requirements, similar to the previous ones, constrain the memory usage during system runtime. They can also demand monitoring and management activities in order to improve their usage.

It is important to highlight that requirements in this classification are not independent from each other, meaning they have conflicting aims, e.g. task migration *vs.* processing power or remaining energy. Moreover, some of them are related to system-wide characteristics and constraints, while others have a more limited scope. A detailed discussion on requirements analysis is out of the scope of this text. Interested readers can refer to [37], in order to obtain a detailed discussion.

2.4. Embedded Systems Design Approaches

2.4.1. Introduction

There are several approaches to design embedded systems. They vary from ad-hoc design flows to more formal and rigorous methodologies. In the same way, there are several different abstractions used to specify system architecture and expected behavior. Looking at the literature, there are approaches that see the system as a set of data structures, operations and functions, while others try to encapsulate them in single elements. Some of them use rigorous mathematical formulations while others use more informal specifications, which are most commonly found in current industry practices, specially in initial design phases [27].

Current practices for designing distributed embedded real-time systems deal, in an acceptable form, with some problems that appear during design. However, the increasing number of stringent requirements (e.g. energy consumption, performance, portability, dependability, and time-to-market) demands new methodologies, tools and abstractions to assist designer to cope with the growing design complexity. According to Carro and Wagner [25], embedded systems are becoming more software intensive, thus innovation depends more on software than on hardware. The design of embedded software should essentially follow some of the principles of

hardware design, i.e. reuse of previously developed and validated/certified source code.

A design flow consists in capturing requirements at a well defined abstraction level that allows several refinements towards an efficient realization of the specified system [7]. Sometimes, these steps from requirements to implementation are not as smooth as designers expected. Requirements must be translated to a system level architecture, which represents the conceptual structure and expected system functional behavior. Following, this architecture is translated into an implementation, defining the system logical organization. At the last step, implementation is realized as the system physical structure [128].

Therefore, it is important to use a suitable abstraction when designing distributed embedded real-time systems. As previously stated, software is becoming more important in such design. Hence, it makes sense to use approaches from the software engineering domain, even though the project of embedded systems comprehends hardware and software designs, in order to close the gap presented in figure 1.1. In this context, approaches such as Object-Orientation (OO) and Aspect-Orientation (AO) appear as interesting options. The following sections will present more details on each paradigm.

2.4.2. Object-Oriented Paradigm

The object-oriented paradigm allows designers to reason on the problem in term of entities instead of operations and functions. In fact, these entities in OO are called *objects*, which have their own local state, and operations that can change this state. In other words, objects encapsulate data and behavior to manipulate these data [14]. Consequently, a system is composed of several interacting objects that maintain their own local state, while providing operations on this information. The direct access to object's data is not allowed to other objects, i.e. there is no external access to such information, only object's operations can access it.

Using OO based system analysis, classes and objects are extracted from functional requirements and non-functional ones. Further, in OO design, these objects and classes are refined by including additional details into them. If needed, new classes can be created. During modeling design phase, designers must identify data types to represent object's state as well as operations that make objects behave as expected [115].

According to Armstrong [4], despite the fact that OO concepts were introduce in late 1960s with Simula programming language [35], there is no thoroughly understanding on the fundamental concepts that define the OO approach. In that work, she has identified the following concepts as the *quarks*[1] of object-oriented paradigm:

- **Class** is a description of structural characteristics (attributes) and behavior (methods) shared by one or more similar objects;

- **Object** is an individual or identifiable element in a OO system. It can represent either a real or abstract system element. As mentioned it contains data representing its state in a

---

[1]A quark is a fundamental particle that represents the smallest known unit of matter. Hence they are the basic building blocks for everything in the universe [49].

given time instant;

- **Inheritance** is the mechanism that allows characteristics to be reused among classes, i.e. attributes and behavior (methods) of one class can be included in other classes;

- **Encapsulation** is a technique to restrict the access to data and behavior of classes and objects through a pre-defined set of messages that objects of a given class can receive;

- **Abstraction** is the act of creating classes to simplify the problem(s) by means of using different levels of details;

- **Attribute** is an remarkable characteristic of an elements class. The set of attributes represent a class' structure;

- **Method** represents object's behavior. It is a way to access, set or manipulate object's information;

- **Message passing** is the process through which objects can exchange information or trigger the execution of a behavior of the message's receiver object;

- **Polymorphism** is the ability of different classes (from the same hierarchy) to respond to the same message through a different behavior, which is more appropriate to each class;

- **Instantiation** is the act of creating objects from a given class;

- **Relationship** are associations[2] among classes or objects. There are the following types of relationships:

    - *Plain associations* indicate that classes or objects are related through message passing, i.e. they do not represent any structural characteristic, even though the implementation could require an attribute of the same type as the other association end, in order to respect the *encapsulation*
    - *Aggregations* indicate that other classes or objects make part of the structure of the aggregator class/object. Parts can exist without the aggregator element;
    - *Compositions* represents a stronger aggregation relationship, where the involved classes or objects are dependent from each other, i.e. there is no composite without its parts and vice-versa.

2.4.3. Aspect-Oriented Paradigm

Before starting the description of concepts important in the aspect-oriented paradigm, it is important to highlight some more fundamental concepts from software engineering. Such concepts were extracted mainly from [30], [64] and [115].

- **Concerns**, according to [64], are "...interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or mode stakeholders...". Concerns are related to both functional and non-functional

---

[2]It is important to mention that *inheritance* is also considered a relationship among classes.

requirements;

- **Separation of concerns** means to deal with each concern in isolation, in order to allow the creation of modular artifacts that handle them. However, in the literature of embedded systems, it is usual to find the term *separation of concerns* meaning the separation of functional and architectural concerns, as well as the separation of computation and communication. This text uses this term as the separation of functional concerns from non-functional ones;

- **Modularization** means the ability to group or partition artifacts into entities called *Modules* (i.e. an abstraction unit in the adopted language) that ideally must be loosely coupled and highly cohesive;

- **Composition** is the ability of integrating several modular artifacts into a coherent whole;

- **Decomposition** is the division of a larger problem into smaller ones, which may be handled apart from each other;

- **Tangling** indicates that multiple concerns are mixed together in one module;

- **Scattering** indicates that one concern is spread over multiple modules;

- **Crosscutting** represents the occurrence of *tangling* and *scattering* that happens when the selected decomposition is unable to modularize concerns effectively;

- **Crosscutting concerns** are concerns that cannot be mapped to unique modules, thus leading to tangling and scattering. Non-functional requirements can be viewed as crosscutting concerns, because they are usually intermixed with functional requirements inside several modules. Figure 2.2 depict the crosscutting concerns related to transaction management presented in [30].
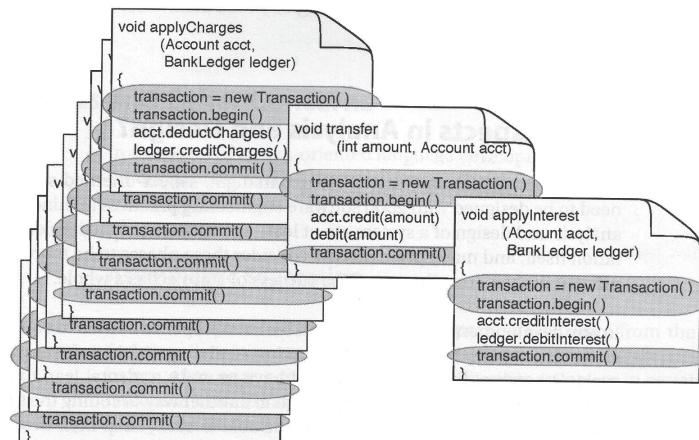
Figure 2.2.: Scattering: same code in multiple places [30]

Some authors, such as [30], state that AO is the natural evolution of OO. Traditional approaches like OO do not deal with crosscutting concerns in a suitable way. In other words, OO decomposition is unable to encapsulate crosscutting non-functional requirements, leading to tangling and scattering in the handling of these requirements. AO analysis and design have emerged from the *aspect-oriented programming* [67]. According to Clarke and Baniassad [30] there are two different approaches in AO, which follow the software composition presented in [56]: *(i) asymmetric*, which separates aspects from the core functionality; and *(ii) symmetric*, which treats separated concerns at the same hierarchy level, i.e. aspects and base concerns have the same importance. This work follows the asymmetric approach for AOD.

Following, the basic AO concepts, which are based in [109] and [126], are presented. These concepts have a broader scope compared with those presented in [67], which are closer to programming languages than to general concepts. This work is based on the following AO concepts:

- **Aspects** represent units of modularization for crosscutting concerns, i.e. they can encapsulate into a single entity all structural and/or behavioral element of a crosscutting concern;

- **Adaptations** specify how concerns are adapted (i.e. enhanced, replaced, or even deleted) when an aspect affects them. There are two kinds of adaptations:

  - *Structural adaptations* represent modifications in the structure of a concern, e.g. adding a new attribute or method to a class, or modifying the formal parameters list or the return type of a method;
  - *Behavioral adaptations* specify changes in the behavior of a concern, e.g. inserting a specific behavior before or after a message passing, or replacing an entire behavior for another one;

- **Aspects weaving** is the composition process that spreads aspects adaptations in affected concerns. In other words, aspect adaptations are applied at specific join points of the affected concerns;

- **Join points** are well-defined places in the structure or behavior of concerns where an aspect can perform adaptations;

- **Pointcuts** are links between aspects adaptations and join points, i.e. they are specified within an aspect to indicate the places where the aspect must perform a given adaptation. Usually, this relationship between *adaptation* and *join points* is one-to-many, that means, one *adaptation* to one or many *join points*. In addition, pointcuts also specify a *relative position* that indicates if the adaptation should be applied *before*, *after* or *around* the join point.

2.4.4. Evaluating the Design with Metrics

2.4.4.1. Introduction

A high quality systems is the goal of all designs. Hence, it is important to have mechanisms to allow the assessment of a design in order to verify its quality in terms of a given set of characteristics. Such mechanisms should provide quantitative information to permit a more precise evaluation [115]. Particularly, considering distributed embedded real-time systems design, measurement mechanisms must derive numeric values for some attributes of both hardware and software designs. As previously stated, the design of distributed embedded real-time systems is becoming software dominated, shifting the costs in development, validation and test from hardware to software. For this reason, despite the importance of metrics extraction for hardware designs, this section will only discuss software metrics.

In the software engineering literature, there are several metrics and evaluation frameworks to extract quantitative information from software. Each of these works aims at the evaluation of different system characteristics. This text presents a brief description of two of these works: *(i)* the C&K metrics suite; and *(ii)* the assessment framework for AO systems from Sant'anna et al. [106].

2.4.4.2. C&K Metrics Suite

The C&K metrics suite [28] was proposed to measure the main factors affecting OO software quality, i.e. abstraction, encapsulation, and inheritance. These metrics have been used in many works, including the evaluation of software for NASA's aerospace systems [101]. C&K metrics are composed from six measurements:

- **Weighted Methods per Class (WMC)** counts the number of methods implemented within a class;

- **Depth of Inheritance Tree (DIT)** indicates the maximum depth in the classes hierarchy tree, i.e. the number of levels from a class to the inheritance tree top;

- **Number of Children (NOC)** represents the number of immediate sub-classes that have the same parent class;

- **Coupling Between Object Classes (CBO)** counts the number of other classes associated to a given class;

- **Response for a Class (RFC)** indicates the number of methods that can be potentially invoked in response to a message received by an object of a given class;

- **Lack of Cohesion in Methods (LCOM)** uses the degree of similarity among method pairs of a class. It uses the set of attributes, which are shared between two methods, to calculate class cohesion. It counts the number of empty sets (i.e. the number of method pairs that do not share the same attributes set) minus the number of non-empty sets (i.e.

number of method pairs that share at least one attribute).

Only the numbers provided by the measurement of system characteristics are not enough to assess the quality of a design. These metrics should be related with each other in order to allow their analysis, and hence, to determine design quality. Table 2.1 represents the relationship among C&K metrics and quality attributes that are being evaluated. Marked cells indicate that a metric influences the quality attribute.

Although the goal is usually to minimize metrics values, it should be highlighted that DIT and NOC metrics do not follow this goal. A higher DIT increases complexity, however it improves reuse. Likewise, a higher NOC leads to an increase in the effort for testing (because more classes should be tested) but also improves reuse. Therefore, it is not useful to read metric values or quality attributes in isolation. They should be analyzed along with other metrics or quality attributes in order to assess which are more important to design goals, and consequently, to make trade-offs to achieve the desired quality.

2.4.4.3. Assessment Framework for AO systems

Sant'anna et al. [106] have proposed an extension for C&K metrics to allow the evaluation of OO and AO systems. Additionally, an assessment framework was proposed to assist in the analysis of metrics values extracted from the system. To allow the use of the same metrics set to evaluate systems developed using different paradigms, it is necessary to homogenize the way to obtain these metrics values in order to take into account abstractions provided by such paradigms. Thus, Sant'anna et al. [106] treat aspects, classes and interfaces as *components*, while methods and aspects adaptations are called *operations*. Following, the metrics set is presented:

- **Separation of Concerns metrics** measure the ability to encapsulate the handling of a concern. They are divided in the following metrics:

  - *Concern Diffusion over Components (CDC)* counts the number of components (i.e. aspects or classes) engaged in the handling of a certain concern;
  - *Concern Diffusion over Operations (CDO)* counts the number of operations (i.e. methods or aspect adaptations) related to the handling of a concern;
  - *Concern Diffusion over LOC (CDLOC)* counts the number of transition points for each concern in the source code, i.e. code lines are divided in fragments (where each

Table 2.1.: Metrics influence in quality attributes

	WMC	DIT	NOC	CBO	RFC	LCOM
Comprehension	X	X		X	X	
Maintainability	X			X	X	
Reusability	X	X	X	X		X
Testability		X	X	X	X	

fragment handles only one concern), thus transitions from one fragment to another
are counted;

- **Coupling metrics** measure how dependent an element is regarding other system's elements. Two metrics compose this group:

  – *Coupling between Components (CBC)* is an extension to CBO from C&K metrics. It counts the number of other components that are coupled with a given component. For classes, CBC is similar to CBO, however for aspects CBC counts other classes that are specified within adaptations. If a component is coupled more than once with other component, it is counted only once;

  – *Depth of Inheritance Tree (DIT)* is an extension to DIT from C&K metrics by means of including the aspects inheritance tree;

- **Cohesion metrics**. Cohesion is the closeness measure for the relationship of a component with its internal elements. It is translated by the following metric:

  – *Lack of Cohesion in Operations (LCOO)* is similar to LCOM of C&K metrics. The difference is that, in addition to methods, adaptations are also taken into account;

- **Size metrics** measure the size of the model:

  – *Vocabulary Size (VS)* counts the number of system components, i.e. the amount of classes and aspects;

  – *Lines of Code (LOC)* counts the number of lines of code;

  – *Number of Attributes (NOA)* counts the internal vocabulary of each component, i.e. the number of attributes of each class or aspect;

  – *Weighted Operations per Component (WOC)* measures the complexity of a component in terms of its operations, i.e. the sum of complexity of each method and/or adaptation. The measure for operation complexity is obtained by counting the number of parameters of the operation, assuming that an operation with more parameters than another is likely to be more complex. WOC extends C&K metrics' WMC because WMC considers the complexity for all method being equal to "1";

In addition to the presented metrics set, Sant'anna et al. [106] define

relationships among metrics to assess the quality of reusability and maintainability for a system. The assessment framework defines *qualities* that are divided in *factors*, which in turn are split into *internal attributes* associated with *metrics*. Figure 2.3 shows these relationships.

Reusability and maintainability qualities of a system can be defined by two factors: understandability and flexibility. The understandability factor is obtained through separation of concerns, coupling, cohesion and size attributes. Separation of concerns directly affects the understandability of a system, because the more localized concerns are, the easier is to find and to understand them. Cohesion and coupling indicate the level of independency of one element regarding others. The more independent an element is, the easier is to understand it. Model size impacts on understandability due to the amount of elements that should be understood. For the flexibility factor, the key attributes are coupling, cohesion, and separation of concerns. A
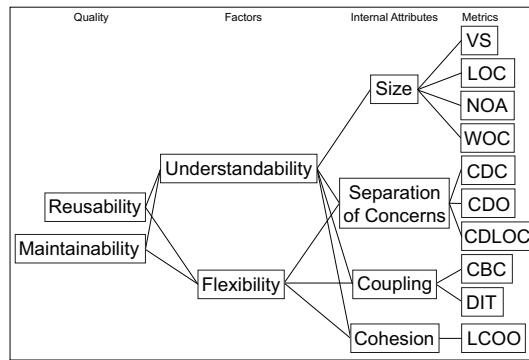
Figure 2.3.: Quality model proposed in [106]

component is flexible if it is independent or almost independent of the rest of the system, meaning that it represents a specialized part of the system with a specific and well-defined mission. These characteristics are translated into low coupling and high cohesion (i.e. it has a low dependence on other parts of the system) and a good separation of concerns (i.e. the component is responsible for a well defined mission).

2.5. Model-Driven Engineering

2.5.1. Overview

To start the discussion on Model-Driven Engineering (MDE), it should be stated what "model" means. According to Bézivin [12], there are many different, and also contradictory, definitions for the word "model", which depends on the context in which the term is used. For computing related systems, a consensual definition of model was given by Rothenberg [102] as follows:

> "...Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality..."

MDE is an approach that proposes the use of generative and transformational techniques for computing systems design where system implementations are (semi-) automatically derived from models or specifications. In such approach, *models* are used as primary engineering artifact throughout the production lifecycle [113]. According to Schmidt [110], MDE is a promising approach to deal with the complexity of platforms (which is not effectively decreased by using

third-generation languages), as well as express domain-specific concepts. Thus MDE combines:

- **Domain-Specific Modeling Languages (DSML)** formalize the application structure, behavior and requirements of a particular domain. Moreover, they define relationships among concepts of the target domain, as well as specify key constraints and semantics related to them. DSML are described in terms of *meta-models*, whose elements represent concepts of the domain. Instances of meta-models represent the use of domain concepts within a design;

- **Transformation engines and generators** whose purpose is to "understand" the information contained in the model in order to produce (semi-)automatically other types of artifacts, such as more detailed models, source code, simulation inputs, components configuration files, and others. Such tools help ensure consistency between the specification of the system and its implementation;

An already mentioned example of standard for MDE approaches is the Model-Driven Architecture (MDA) [87], which was proposed by the Object Management Group (OMG). The set of standards supporting MDA is:

- **Meta-Objects Facility (MOF)** [89], a standard for meta-models specification;

- **Unified Modeling Language (UML)** [92], a general purpose modeling language for systems specification. It was built upon MOF and represents a *de facto* standard for modeling languages;

- **MOF Query/View/Transformation (QVT)** [93], a standard defining transformation languages requirements and operational mappings to allow transformations of source models into other target models that should conform to MOF meta-models

- **XML Metadata Interchange (XMI)** [91], a standard for metadata information exchange, specified using a XML dialect, to allow the information exchange on MOF-based specifications, such as interchange of UML models among different tools;

- **Common Warehouse Meta-model (CWM)** [86], which provides standard interfaces that can be used to enable interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments.

The principle of MDA is to specify system functionality using a *Platform-Independent Model* (PIM) using an appropriate DSML. PIM provides a system specification that is suitable for deriving system implementation for different target platforms. Further, this PIM is translated to a *Platform-Specific Model* (PSM), which, on the other hand, provides a platform specific viewpoint of the system, i.e. it combines the specifications in the PIM with the details specifying how that system uses a particular type of platform. In order to enable this transformation (or mapping), a *Platform Model* (PM) must be provided. The PM provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides concepts representing the different kinds of elements to be used in the specification of how platform should be used by the application.

2.5.2. MARTE UML profile

UML was created to be a general purpose modeling language for software development. Its wide acceptance makes it an interesting option also to design distributed embedded real-time systems. However, UML lacks suitable constructions/abstractions to represent specific concepts of embedded and also real-time systems domains. The first attempt to overcome such deficiencies was the UML profile for Schedulability, Performance, and Time (SPT) [88]. SPT provides concepts to allow both model-based schedulability and performance analysis, and also a rich framework to model time and time-related mechanisms. However, according to Gérard and Selic [51], experiences in applying SPT revealed shortcomings within the profile in terms of its expressiveness for modeling real-time and embedded phenomena. The amount of issues in the SPT profile resulted in a Request for Proposals (RFP) for a new UML profile for specifying embedded and real-time systems. Consequently, a new profile named *Modeling and Analysis of Real-Time and Embedded systems* (MARTE) [94] was proposed. It was accepted by OMG in July 2007 and is in the finalization process.

The MARTE profile addresses: *(i)* new elements to UML 2.x are proposed; *(ii)* design of both software and hardware aspects of embedded system; *(iii)* broader schedulability and performance analysis capabilities; *(iv)* specification of embedded systems characteristics, such as memory capacity and energy consumption; *(v)* support to component-based architectures; *(vi)* other computational paradigms, such as asynchronous, synchronous, and timed; and *(vii)* compliance with the UML profile for Quality of Service and Fault Tolerance [95]. An overview of MARTE profile is presented in figure 2.4.

As can be seen, MARTE profile is composed by four packages: *(i)* MARTE Foundation; *(ii)* MARTE Design Model; *(iii)* Real-Time and Embedded Analysis; and *(iv)* MARTE Annexes. The *MARTE Foundation* package provides a domain-specific meta-model for core concepts MARTE, as well as their characteristics and relationships among such concepts, i.e. it defines the semantics base for the DSML provided by the profile. Elements of this package are shared among other packages.

In fact, MARTE is intended to cope with two concerns: modeling of real-time and embedded systems features, and to support analysis of system properties. *MARTE Design Model* package provides first-order language constructs to specify model expressing specific phenomena of real-time and embedded systems. It allows platform modeling in terms of software (see *Software Resource Modeling* (SRM) package) or hardware (see *H*ardware Resource Modeling (HRM) package) platforms. According to Gérard and Selic [51], MARTE sees platforms as a set of resources, possibly comprising finer-grained resources into a hierarchical manner, in which each resource offers at least one service. A resource is seen as a service provider with finite capacity, which usually comes from physical limitations of the underlying hardware (e.g. memory capacity, bandwidth, processing power, etc.). Considering software platforms, SRM package provides a model-based view for concepts provided by RTOS API, such as semaphores and concurrent tasks (or processes). On the other hand, regarding hardware platforms, HRM package provides concepts to assist software design and allocation by providing a high-level hardware description model instead of using block diagrams. Additionally, concepts provided
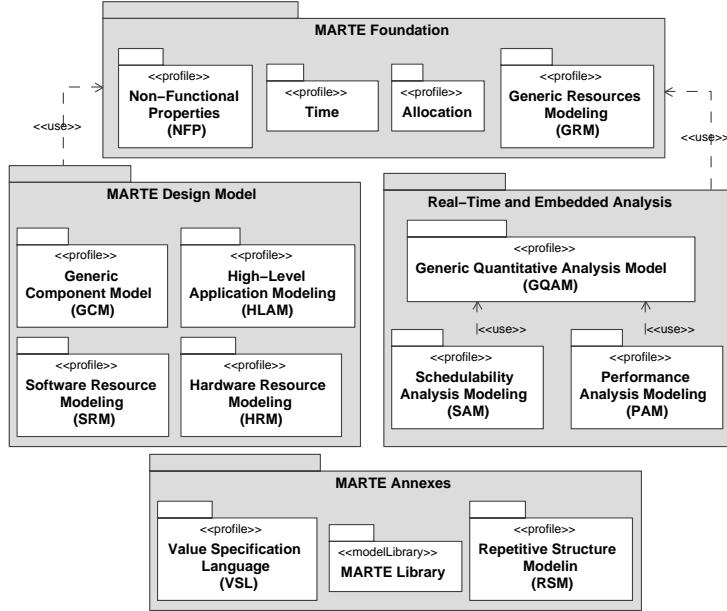
Figure 2.4.: Overall MARTE architecture [94]

by HRM assist in the analysis of real-time and embedded properties, and also in hardware models simulation, which depends on the description detail level and simulation accuracy.

Model-based analysis is supported by the *Real-Time and Embedded Analysis* package, which provides a foundation for applying transformations from UML models into a wide variety of analysis models. According to OMG [94], the *Generic Quantitative Analysis Modeling* (GQAM) defines basic UML extensions needed to decorate UML models, in order to perform any kind of analysis. Currently, two kinds of analysis packages are provided, namely *Schedulability Analysis Modeling* (SAM) and *Performance Analysis Modeling* (PAM) packages. The former provides stereotypes to allow schedulability analysis, while the later provides stereotypes for performance analysis.

Due to their importance to this work, two packages of the *MARTE Foundation* package need to be detailed. The first one is the *Time* package, which provides a general framework for representing time and time-related concepts. MARTE adopts time models that rely on partial ordering of time instants. The temporal ordering of behavior activities can be represented in many ways, depending on the level of precision required. There are three main classes of time abstraction: *(i) causal/temporal*, which concerns only about instruction precedence/dependency; *(ii) clocked/synchronous*, which adds the notion of simultaneity and divides the time scale in a discrete succession of instants; *(iii) physical/real-time*, which demands accurate modeling of real-time duration values. Stereotypes available in *Time* package are shown in figure 2.5.

A Clock exists in a TimeDomain and gives access to time at a certain resolution. Timed-Constraint represents a constraint (instant or duration value) associated with a model element bound to a Clock, while TimedEvent represents an event whose occurrence is ex-
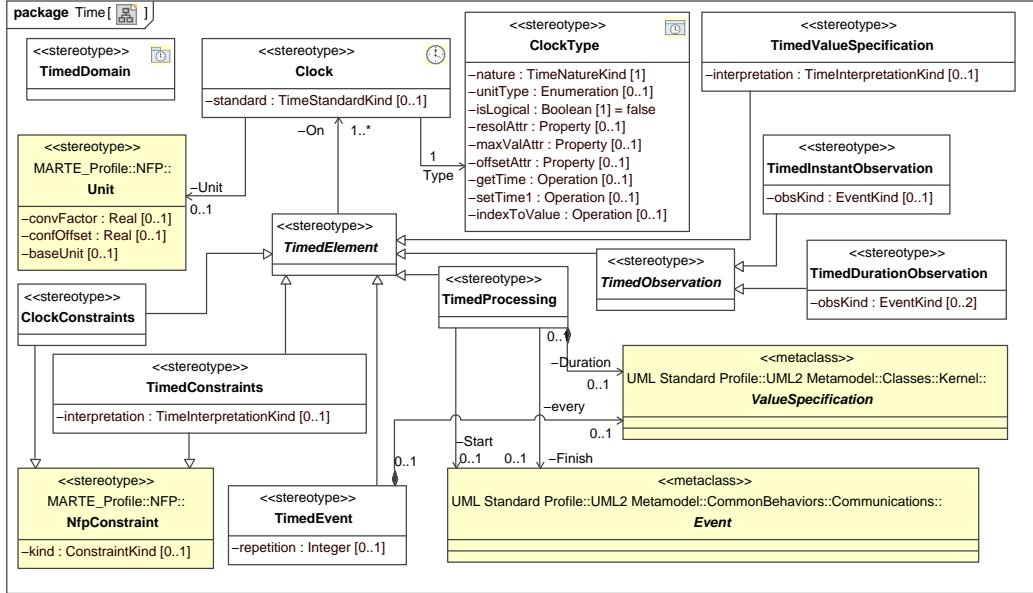
Figure 2.5.: Stereotypes of Time profile [94]

plicitly bound to a `Clock`. The `every` property specifies the duration between successive occurrences, thus indicating a periodic event. `TimedProcessing` represents activities having known start and finish times, or a known duration, which are bound to a `Clock`. For a detailed description of the other stereotypes, readers are referred to [94].

Another important package is the *Generic Resources Modeling* (GRM), which offers concepts to model a general platform for executing real-time embedded applications. According to OMG [94], this package allows the modeling of executing platforms at different levels of details. Figure 2.6 depicts the stereotypes available in GRM package.

The central concept of the GRM package is the notion of a `Resource`, which represents a physically or logically persistent entity that offers one or more services. There are many types of resources such as, `TimingResource` representing a hardware or software entity that is capable of following and evidencing the pace of time. `Concurrency Resource` and `SchedulableResource` represent protected active resources that can perform their activities concurrently with others. The former indicates resources, which take their processing capacity from a potentially different `Computing Resource` enabling physical or logical concurrency. On the other hand, the later only allow logical concurrency because it competes for processing capacity of a `Processing Resource` with other `SchedulableResource` elements. A `Scheduler` coordinates the access to the `ProcessingResource` from all `SchedulableResource` elements associated to it. A resource makes use of a service from other resource by means of `Acquire` and `Release`. The former represents the allocation of or the access to some resource, while the later represents the de-allocation or liberation of the allocated resources. The control of concurrent accesses to common resources at run-time is
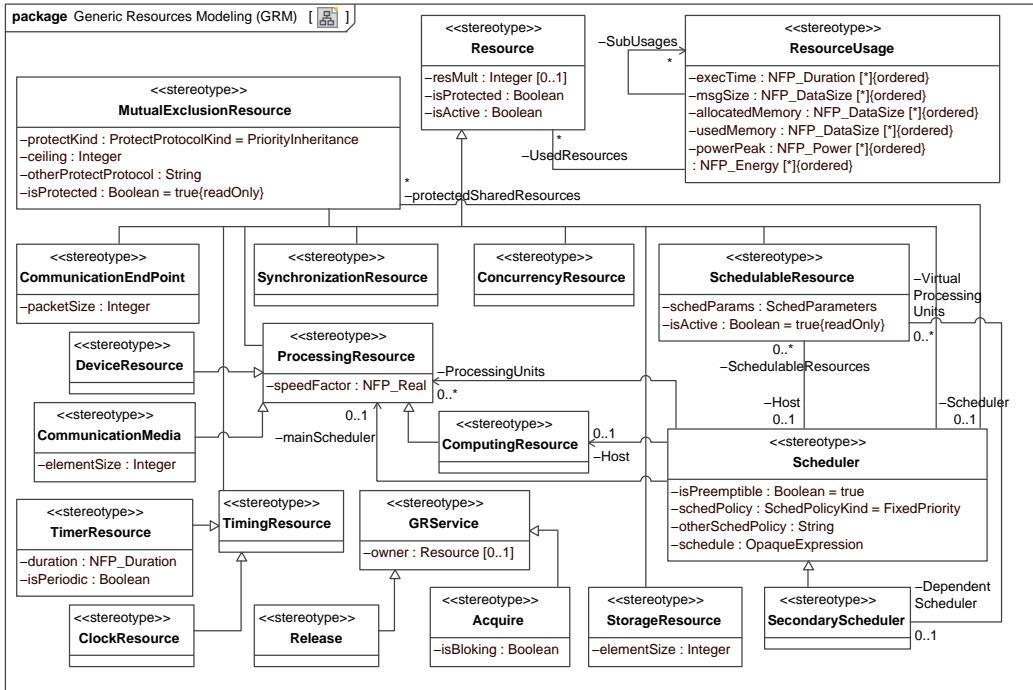
Figure 2.6.: Stereotypes of GRM profile [94]

performed by a `MutualExclusiveResource`. Other kinds of resources can be represented using the GRM package. For more details see [94]

Chapter 3

State of the Art Analysis

3.1. Introduction

According to Sangiovanni-Vicentelli [105], to raise the abstraction level used during design of digital systems is fundamental to manage the increasing design complexity, leading to costs decrease and designers productivity improvements. In embedded system projects, many languages considered as "high-level" languages (e.g. SystemC or System Verilog) cannot suitably deal with important requirements such as temporal predictability of an application. To increase designers' productivity and also decrease the amount of eventual errors caused by inconsistent specifications or requirements misunderstanding, project focus should move from intermediate levels to higher levels of abstraction, as well as to separate the handling of functional requirements from non-functional ones.

Many researchers propose to rise the abstraction level by using models as first-class elements during whole design of distributed embedded real-time systems. However, only using models does not assure an improvement on design or designers productivity. Therefore, to achieve the benefits from using model-driven techniques, a methodology is very important. Hence, some side effects, such as lack of synchrony between models and implementation, can be decreased or even avoided. Additionally, the methodology must provide a smooth transition from high level specification (i.e. model) to implementation of the distributed embedded real-time system, and also allows the reuse of artifacts created and tested in previous designs.

This chapter discusses the state of the art in the design of distributed embedded real-time systems. It presents methodologies and modeling techniques, as well as code generation approaches to produce source code from model, and the employment of separation of concerns in the handling of requirements.

3.2. Design and Modeling Approaches

3.2.1. Overview of Related-Work

This section discusses traditional methodologies (i.e. those methodologies using OO) applied to the design of distributed embedded real-time systems. The presented approaches use a higher abstraction level in terms of UML models to produce the initial specification of the structure and behavior of distributed embedded real-time systems.

Schattkowsky and Mueller [107] have proposed a MDA-based method to specify and execute embedded real-time systems. Their approach supports system specification using class diagrams, state diagrams, and sequence diagrams from UML 2.0. In the class diagram, designers specify classes, as well as their attributes and operations. Each class' operation is considered a state machine. Different sequence diagrams are associated with different states of a state machine in order to describe the behavior (i.e. actions sequence) that must be executed within a state of the state machine. The execution environment supports state machines composed of simple or composite states, however, concurrent states are not supported. Asynchronous calls to methods lead to the instantiation of a new state machine, which executes its behavior concurrently with other state machines. Another remarkable feature of that work is that interruptions and exceptions can be specified within state machines, i.e. external devices such as sensors can generate external signals that are perceived by the runtime environment. In order to execute models, that work proposed the *Abstract Execution Platform* (AEP) [108], which is a stack-based machine with instruction to manipulate OO constructions expressed in the UML model. The produced models are "compiled", generating a binary code that runs in the AEP. In fact, according to the authors, AEP is a virtual machine that can be implemented in both software and hardware, similarly to a Java Virtual Machine.

Arpinen et al. [5] present a technique to execute embedded applications specified with UML 2.0 in configurable multiprocessor systems. Application is specified using UML 2.0 diagrams, which are decorated with stereotype from the TUT-profile [69] providing concepts of embedded real-time systems to support automatic transition from UML models to the SoC implementation. The design flow starts with the application architectural description, specified with class and composite structure diagrams, defining system elements in terms of components interconnected by ports. System behavior is expressed in terms of state machines, which represent application tasks. The next step is the architectural exploration, which is responsible to allocate, map and schedule tasks into different processors. Following, the design flow is split into two branches: code generation of application software and platform synthesis. State machines are transformed into *Extended Finite State Machines* (EFSM) in order to allow C code generation. Composite structure and class diagrams are used to configure the platform, allowing the needed VHDL code generation. Arpinen et al. [5] presents a case study, which shows the implementation of a MAC protocol for wireless networks. That application has been implemented using four Altera's Nios II processors and three hardware accelerators interconnected through a HIBI communication architecture.

Other work that uses UML as modeling language is presented in [82]. That work presented

an approach to transform UML models into SystemC code, allowing system simulation. For system description, class diagrams and state diagrams decorated with stereotypes indicating SystemC constructions are used. According to Nguyen et al. [82], class diagrams represent a system in terms of components, and how these components should be interconnected with each other to provide system architecture. Thus, classes are used to describe computational entities having a runtime state and an associated behavior that modifies their state. In this sense, classes within a class diagram are decorated with stereotypes representing SystemC elements, such as modules, interfaces, ports and channels. Each state diagram describes the behavior of a single component (i.e. a class), in which composite states (with *and-state* regions) are used to model concurrency. Actions can be associated with the entry or exit of a state, as well as with state transitions. That approach follows the semantics of the UML specification for state machines, i.e. a state transition is triggered by an event only if all guard conditions are true. As a consequence, within a system UML model, all actions and guard conditions are textual descriptions using SystemC syntax.

Riccobene et al. [100] presented a proposal to modify the SoC design flow used by STMicroelectronics. The original design flow starts with requirements specification using natural language. These requirements guide the specification of executable models, which capture all expected behavior in a platform independent fashion. After this step, the design flow is split into two concurrent phases: hardware and software designs. Although their concurrent nature, these separate designs must interact in some steps to achieve system final implementation. Riccobene et al. [100] argue that, by using UML, it is possible to improve the process in the sense of standardizing executable PIMs, and hence, improving communication between designers teams, which can share the knowledge about system functionalities and requirements. Therefore, from the design flow splitting, each team can decorate the executable PIM with profiles suitable to its domain: the hardware team use a SystemC profile proposed in [100] to map UML constructs into synthesizable code; while the software team can use a profile more suitable to the programming language used to implement functionalities that will run on hardware units created by the hardware team. Adopting this approach, it is possible to use code generation tools for both designs. To model the system, the following UML diagrams are employed: *(i)* class diagram to describe components types, as well as their attributes and operations; *(ii)* composite structure diagram to specify used components, their ports and interfaces; *(iii)* sequence diagram to create testbenches; and *(iv)* state diagrams to represent the behavior of each operation. In that paper, a small case study has been presented, consisting of a FIFO-based producer/consumer, which is implemented as hardware using a UML 2.0 model decorated with stereotype of authors' SystemC profile.

Other interesting work is the Metropolis project [7], which provides an infrastructure, a toolset, and a design method to allow a uniform representation for heterogeneous components of an embedded system. In order to accomplish such approach, Balarin et al. [7] propose the separation of computation and communication specification, by means of isolating computation element from communication ones. Hence, elements reuse can be improved. The infrastructure core is a meta-model allowing the representation of several computation and communication semantics at different abstraction levels, using different computation models. In this way, a meta-model represents a set of processes interconnected by interfaces communicating through
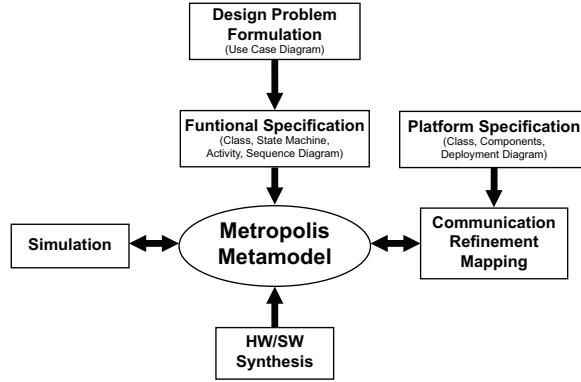
Figure 3.1.: Methodology for Multimedia Systems available in Metropolis [27]

different medias. Processes have their own properties and constraints. Their execution is controlled by a scheduling policy. Furthermore, Metropolis methodology suggests an approach that uses successive refinements, in which more details are incorporated, to depart from higher abstraction levels until arrive to system implementation. According to [52], Metropolis project has different methodologies applicable to different domains, which are concerned with special characteristics of their own domain, being very different from other domains. By December 2008, there are five domains having their own methodology: fault tolerant data flows in automotive systems; multimedia; wireless communication and sensor networks; microprocessor modeling and analog/mixed signal systems. In order to support such diversity of methodologies, some principles must be followed: *(i)* functional decomposition, i.e. in the highest abstraction level, the system is considered a single process, which is decomposed in a set of concurrent processes; *(ii)* between two communicating processes, there is always an extra process, which is responsible to transform (or adapt) the values from the output of one process to the input of the other one; *(iii)* for each communicating process, a media, which defines communication semantics, is associated; *(iv)* in addition, each communicating process is enclosed by a wrapper, connecting it to the media; *(v)* at each refinement step, a media is replaced by a set of processes and medias, adding more details on the communication; *(vi)* finally, the specified elements are mapped into architectural components (i.e. hardware or software components) of the chosen platform. Figure 3.1 shows the methodology used in the multimedia domain. In such methodology, UML diagrams are decorated with stereotypes from UML platform profile [27], which defines elements of Metropolis infrastructure and also some models of computation. Thus, model elements represent different concepts of the selected model of computation. Further, model refinement is performed to map model element into platform elements that are available in a repository. Recently, in [36], an extension named Metro II has been proposed. It involves the improvement of Metropolis framework in terms of three features: heterogeneous IP import, orthogonalization of performance from behavior, and design space exploration.

HASoC (Hardware and Software Objects on Chip) [41] is an OO methodology, which is partially based on RUP [68] and provides an incremental and iterative design flow for embedded real-time systems. It suggests the design must start with the specification of a UML model validated using an *uncommitted model*, which represents an abstract execution model where objects

are not associated with a given implementation, be it as hardware or software. Requirements are specified by means of use case diagrams, in which each use case is associated with at least one sequence diagram that indicates an execution scenario. Static and dynamic system structures are specified using, respectively, class and objects diagrams. Once system specification is finished, the produced model is partitioned into hardware and software components producing the so-called *committed model*. These components are mapped into implementation platforms, which are reused from a platforms repository previously developed and tested. Further, this model is refined in order to include additional implementation details, which must respect design constraints. In this step, the following platforms are selected: interfaces between hardware and software objects, i.e. device drivers; and available hardware components, e.g. processors, memories, communication buses. At the end, selected components integration is performed, leading to the final system implementation.

An iterative MDE method, which combines semi-formal and formal notations, for fault-tolerant distributed embedded real-time systems, called *Method C*, is presented in [96]. The aim of this method is to keep the development "continuum", whose concept is defined as "...the continuity between different software development lifecycle steps without any logic or semantic break so that they are at an effective level of automation. ..." Method C proposes that gaps between abstraction levels should be fulfilled by means of model transformations using meta-models of the adopted languages. Languages supported in this method are UML and MARTE profile, AADL [103], and +CAL [71]. UML diagrams decorated with MARTE profile stereotypes are used to specify application elements that may also be related with real-time domain concepts (e.g. tasks, timers, semaphores, etc.). On the other hand, behavior is specified using formal semantics (e.g. Petri Nets) provided by both AADL and +CAL. The former allows the description of software and hardware parts of the system, while the later is a formal action language to be used within state or activity diagrams.

The SEEP project (portuguese acronym for *Sistemas Eletrônicos Embarcados baseados em Plataformas*) [77] proposes a methodology that integrates design and test of embedded systems considering a wide range of requirements. The proposed methodology encompasses the whole design cycle, from system modeling using UML to the generation of embedded hardware and software components. Figure 3.2 shows the design flow proposed in SEEP. Design starts with requirements specification as well as description of expected functions using high abstraction level UML models. The next step is system exploration, in which designers can select different algorithms to perform the expected functionalities meeting application and design requirements. Following, an architectural space exploration phase takes place. In this phase, designed functions are mapped to different hardware components that must also respect requirements. The automatic generation of hardware and software components, which is based on the functionalities partition performed in previous phases, happens in the next step. At the end of the design cycle, the embedded real-time system, which performs the application for which it has been designed, is obtained.
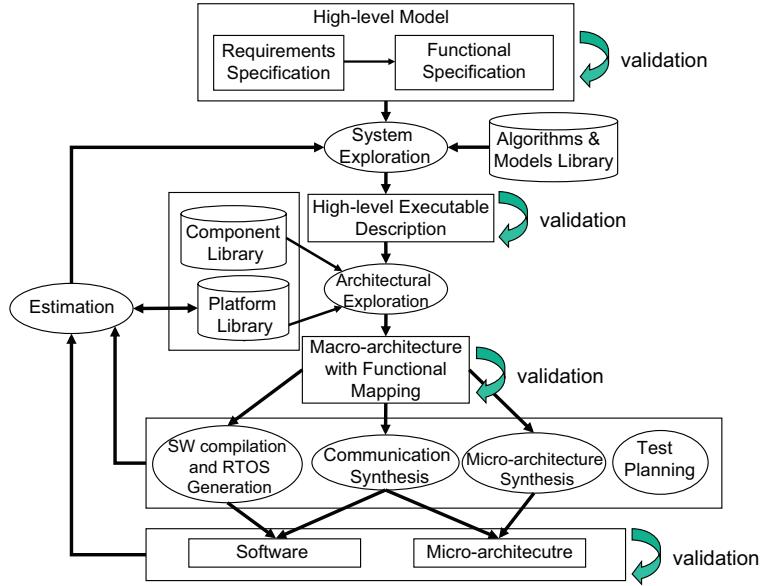
Figure 3.2.: SEEP design flow

3.2.2. Discussion

Works presented in this section advocate the use of standard languages as UML to specify the structure and behavior of embedded systems. Using class diagram for specification of system static structure is a well-established approach, as well as using state diagrams to specify behavior. Most of the presented works do use such diagrams. However, sometimes the use of state machines is not a suitable form to specify behavior because it is not that easy, for example, to understand concurrent activities or non-reactive system behavior. Furthermore, other issue, which is not suitable to behavior specification in models, is the use of textual *action languages* to describe actions performed within each state. This makes the behavior description closer to a computer program (written using a programming language) than to a graphical representation that may be easier to understand.

Each of these approaches, excluding Perseil and Pautet [96], proposes their own profile to decorate UML diagrams in order to provide specific semantics to modeled elements. Such situation hinders the exchange of information on the modeled system between design teams, mainly if one of the involved teams does not know the proprietary profile. This problem also happens during code generation. Code generation tools must be aware of the profile semantics in order to produce source code representing the profile's stereotypes semantics. Semantics standardization in terms of UML profiles is a good approach to deal with the mentioned problems.

As can be observed, most of the presented works do not deal specifically with the design of distributed embedded real-time systems or separates the handling of functional requirements from non-functional ones. Although not clearly stated, the work proposed by Balarin et al. [7] may deal with distributed functionalities due to their proposal for using higher abstraction

levels to specify communication apart from computation. Additionally, considering the other mentioned approaches, that work was the only approach separating requirements from distinct natures, i.e. computation and communication.

As this work is inserted in the context of the SEEP project, one of its goals is to support a mechanism for separation of concerns in the handling of requirements from initial design phases, while in the same time supporting the distribution of functionalities over different processing units. To accomplish this goal, this work proposes adaptations in SEEP design flow as described in the Chapter 4.

3.3. Separation of Concerns in Requirements Handling

3.3.1. Introduction

This section discusses methods and techniques for modularization in requirements handling, focusing on the separation of the handling of functional requirements from non-functional ones during the whole development cycle. At the beginning, this section presents some proposals applied to general-purpose systems, i.e. non-embedded systems. Afterwards, other approaches that apply such separation of concern in the design of embedded real-time systems are also discussed.

3.3.2. Separation of Concerns in General Systems Development

Stein et al. [120] propose the *Aspect-Oriented Design Modeling* (AODM) approach to represent concepts of AspectJ, an AO programming language proposed by Kiczales et al. [67], within UML diagrams. Aspects are represented as classes annotated with «aspect» stereotype, as shown in figure 3.3a, and can specify *advices*, *introductions* and *pointcuts*. Two kinds of adaptations are supported: structural and behavioral adaptations. Structural adaptations, which are called *introductions* in AspectJ terminology, are specified in the class diagram by means of attributes or operations specification, which will be inserted in classes whose structure is affected by the aspect. As can be seen in figure 3.3b, an *introduction* is specified, within the context of an aspect, as a dashed ellipse decorated with «introduction» stereotype, e.g. *BookCopy* inserts the method *getState()* in affected classes. On the other hand, behavioral adaptations, which are called *advices* in AspectJ terminology, are specified in sequence diagrams, which shows how a given interaction is affected by the aspect, e.g. inserting a method call into another object before or after the affected interaction. An *advice* is represented as a method decorated with the «advice» stereotype (see figure 3.3a). An important part is the specification of the elements affected by aspects, i.e. the specification of *join points*. This is done using *Join Point Designation Diagrams* (JPDD) [121], which is a sequence diagram or a class diagram that indicates model elements may be affected by aspects. Figure 3.3c depicts an example of JPDD that selects all method calls to *DbServer.addUser()* performed from any object after the user registration in the authentication server. The link between aspect adaptations and the selection

of affected elements (JPDD) is described by *pointcuts*, which are specified as an attribute decorated with the «pointcut» stereotype. In other words, a *pointcut* indicates which model elements (by means of JPDD) must be modified by which adaptation (*advice*) at which moment (before, after, or around). It is important to highlight that, as the AODM follows AspectJ semantics, structural adaptations (*introductions*) are tightly coupled with the classes it affects, hindering the reuse of such structural adaptations.

Theme/UML [31, 30] is an approach to support separation of concerns by means of conceptual constructions called *themes*. According to Clarke and Baniassad [30], Theme/UML is an AO approach that supports *symmetric* separation of concerns rather than *asymmetric* separation, which is supported by most of AO approaches (e.g. AODM, AspectJ, AspectC++, and others). In this sense, a *theme* is more general than an aspect because it can represent fragments of behavior and/or structure representing a concern. In other words, all elements related with the handling of a concern are specified within only one *theme*. One interesting characteristic of that approach is that it is common to find different views of the same element in different *themes*, i.e. certain elements and behaviors are shared among more than one *theme*. To allow the representation of *themes* within UML diagrams, the UML meta-model has been extended in [29]. That work proposes the concept of *composite patterns*, which supports the composition/decomposition ability required by symmetric separation of concerns. Making an analogy, *composite patterns* can be compared to UML templates, which allow model elements be partially defined. In Theme/UML, *themes* are specified as packages containing all concepts related to a concern
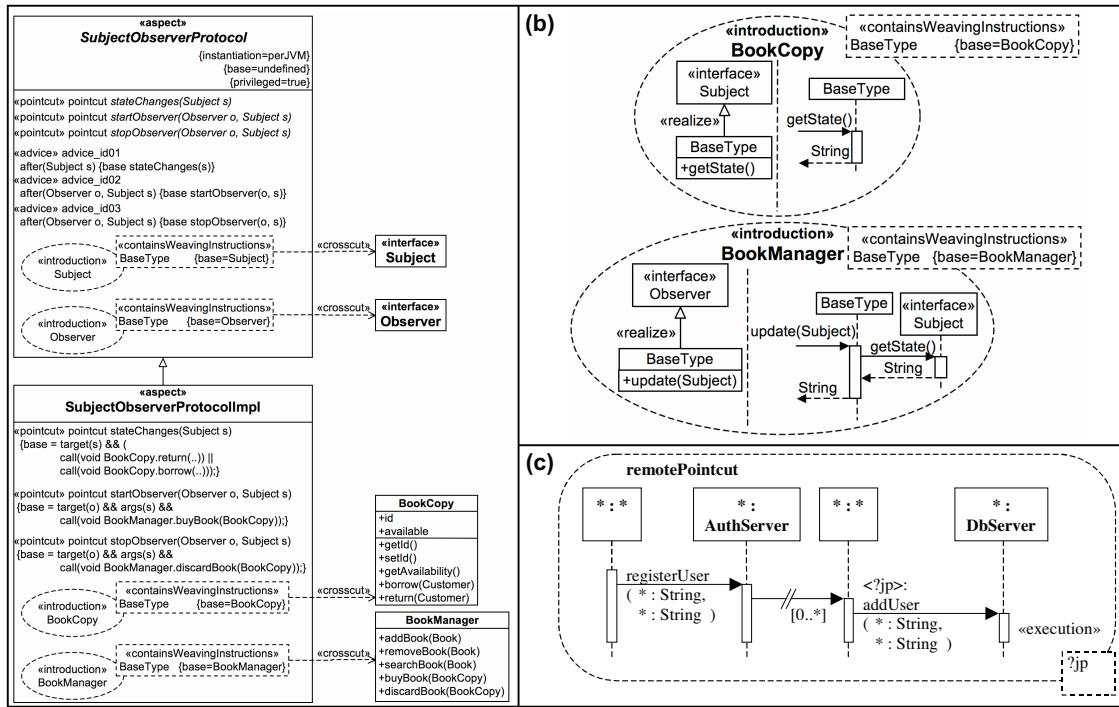
Figure 3.3.: Aspects and join points modeling in AODM [120, 121]

that are specified using class and sequence diagrams, as depicted in figure 3.4a. In addition, integration between *themes* is defined by means of a binding, which indicates which elements of a *theme* are affected by elements of other *theme*. This integration is depicted in figure 3.4b. Comparing to the asymmetric approach, this is similar to the relation between functional elements that are affected by aspects. There are two kinds of integration (i.e. aspect weaving): to override and to merge concepts. In the former approach, elements (structural and/or behavioral) passed as parameters override associated elements in the affected *theme*. On the other hand, in the later approach, concepts of the affecting *theme* are merged with elements of the affected *theme* at the points indicated as parameters.

The AO modeling approach proposed by France et al. [43] addresses concerns during modeling step, aiming exploring different design alternatives in a platform-independent fashion. Such approach produces Aspect-oriented Architecture Models (AAM), which consist in a base architecture model named *primary model* to specify the application model, and a set of aspect models. Both kinds of elements are specified as UML diagrams. In this way, aspects models describe how the primary model is affected by non-functional requirements. The composition of aspects models in the primary model (i.e. aspect weaving) may cause conflicts of interests leading to the emergence of undesired system properties. Such situation, according to France et al. [43], can be minimized (or solved) by means of adapting the aspects model. Furthermore, aspects provide structural and behavioral adaptations in the primary model specified with templates in class diagrams (for structural adaptations) and collaboration diagrams (for behavioral
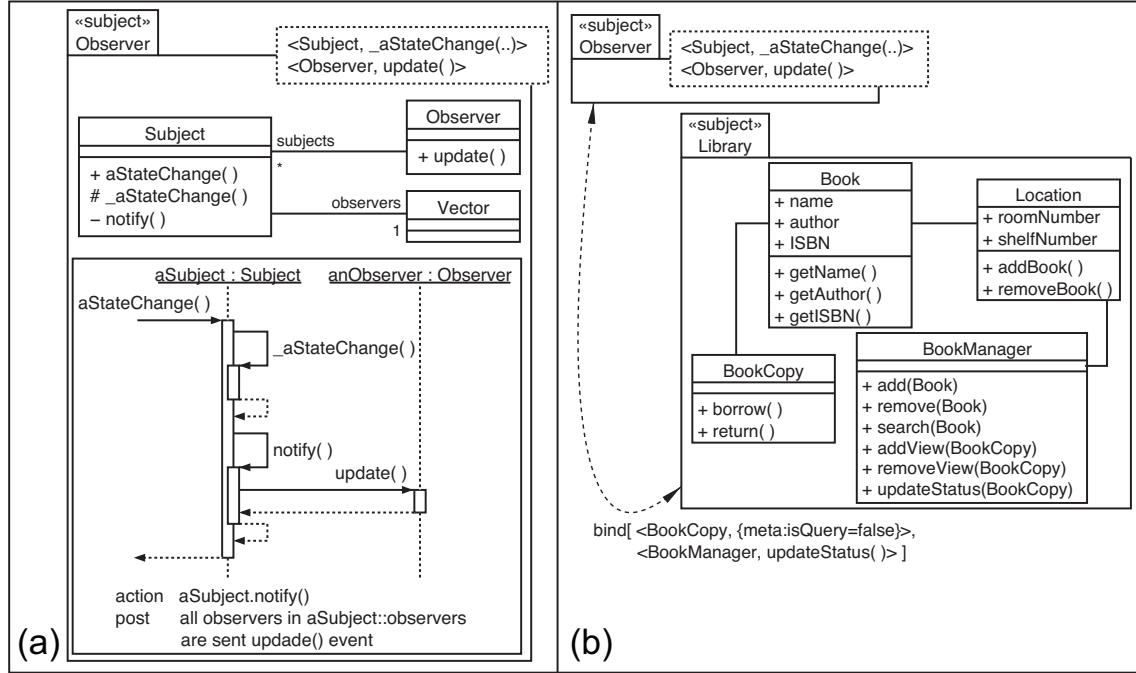
Figure 3.4.: Examples of Theme/UML models [31]

adaptations). It is important to highlight that, in that work, there is no mention about how elements of the primary model are selected (i.e. join points specification) to be adapted by aspects.

Pinto, Fuentes and Troya [98] propose an aspect- and component-based approach to separate the handling of non-functional requirements from functional ones from early specification to implementation phases during software development. This approach defines transformations from UML models, which are decorated with stereotypes from the *Com ponent-Aspect Model* (CAM) profile, to CAM models, which describes a system in terms of components, aspects, and composition rules to weave aspect into components. According to Pinto, Fuentes and Troya [98], behavior specification is realized using standard mechanisms of UML, i.e. state, activities, and/or interaction diagrams. Although important, that work does not discuss how behavior is represented in CAM models. Further, the information described in a CAM model is specified with the DAOP-ADL language [97], which uses the *eXtensible Markup Language* (XML) [129] format to describe components and aspects of a system, and also their relationships. An example of such XML file is given in figure 3.5. DAOP-ADL specifications are interpreted by a middleware platform called *Dynamic Aspect-Oriented Platform* (DAOP) [97], which provides a composition mechanism that performs aspects weaving dynamically at runtime, i.e. it performs aspects adaptation in the affected components while running the application. In this sense, during the weaving process, aspects see components as "black boxes". Such approach constraints aspects adaptations to modify component behavior by means of intercepting operation calls or event occurrences, in other words, it is not possible to define join points, and hence, modify internal behavior of a component.

An approach to specify Aspect-Oriented Executable Models (AOEM) has been proposed
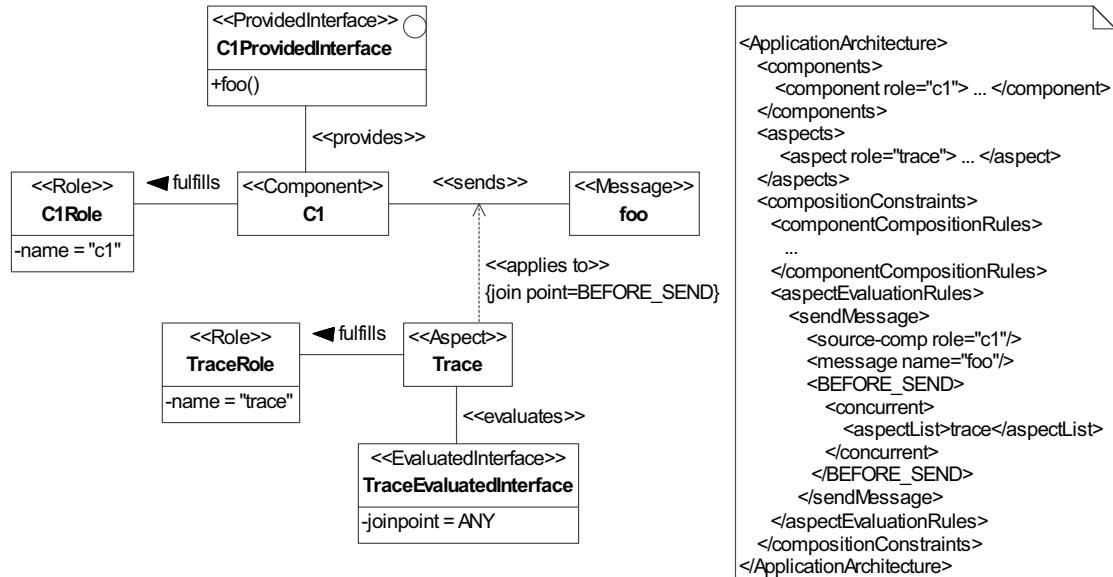
Figure 3.5.: CAM model represented as a class diagram [98]

in [47]. This work provides a UML profile to describe AO-related concept within UML models. Three different models are produced: *(i)* a *base model* specifying system functional concerns; *(ii)* an *aspects model* specifying crosscutting concerns, including their precise and complete behavior, in terms of AO elements using the AOEM profile; *(iii)* a *pointcut model* describing (using the AOEM profile) how crosscutting concerns are composed in the base model in terms of pointcuts. Further, a weaver is used to transform the produced models into a plain UML model, which can be executed using Pópulo UML virtual machine [44]. Additionally, the AOEM profile provides stereotypes to specify the action language defined in the UML 2.x specification. To allow the specification of AO-related actions, the AOEM profile extends the standard UML action language by means of allowing, for example, getting the intercepted message name, or target or source object. For more information on this AO extension for the UML actions language, readers are referred to [46]. Furthermore, aspect advices are specified as activities diagrams (see figure 3.6b), whose actions are decorated with stereotypes of the AOEM profile. These advices are related to pointcuts, which are specified with sequence diagrams (see figure 3.6c), showing the link between the join point selection and the advice. That work allows only the interception of sent messages, i.e. only message-related events can be selected as join points. However, Sánchez et al. [104] propose a modification in the specification of pointcuts and join points by means of using JPDDs [121].

3.3.3. The Use of AOD in the Design of DERTS

Zhang and Liu [133] use UML diagrams and AO concepts to separate the handling of timing requirements from other non-functional requirements in the design of real-time systems. That work proposed the use of only one aspect, in which all timing information of a system is contained. A UML profile is defined to decorate elements in a class diagram in order to represent both AO and real-time concepts. Such profile provides language level concepts of AO, e.g. aspects, advices, join points, crosscut, and control. However, it is important to highlight that the last two concepts, i.e. crosscut and control, are not defined in AO languages. *Crosscut* is used to model the weaving relationship between classes and aspects, while *control* models the weaving relationship between behavior and aspects. Although system behavior is modeled with state
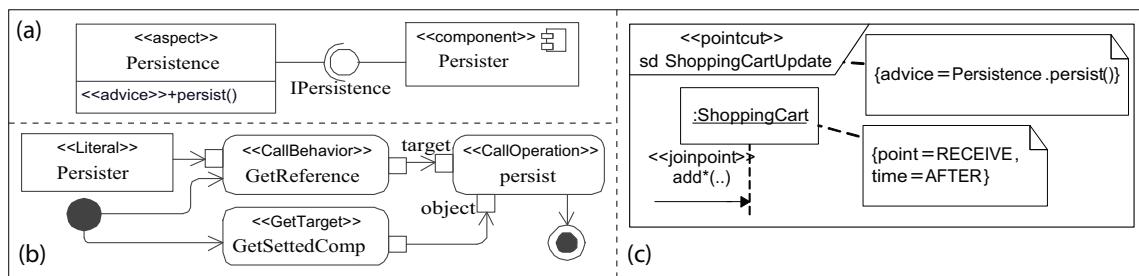
Figure 3.6.: AO modeling [47]: (a) aspects modeling; (b) advice modeling; (c) pointcut specification

diagrams and also proposing the control relationship, that work does not show how the modification in the base behavior will eventually happens, neither how to specify join point to select element in the base classes and behavior. Figure 3.7 depicts an example of timing handling specification. As can be seen, time values description is done by means of notes (i.e. UML text boxes) associated to a time aspect in the class diagram. Moreover, there are other stereotypes representing real-time domain concepts, such as clocks and timers. However, they are very similar to stereotypes from the UML SPT profile [88]. Particularly, the approach presented in [133] is not adequate to describe such key requirements as timing constraints and requirements in real-time system design.

Noda and Kishi [84] have proposed an approach for using AO concept to model embedded software. More specifically, they propose to use AO to model the context in which the embedded system operates. That work uses the symmetric approach for aspects, similarly to [30]. Functional and non-functional concerns are modeled as aspects, which are related to each other by means of two types of inter-aspect relations: *(i) trigger* and *(ii) refer*. The former indicates that one aspect triggers the behavior of other aspect, while the later means that an aspect refers to properties of another aspect to determine its behavior. Such relation can be seen in figure 3.8a. Aspects are modeled as a class diagram and one or more state diagrams, and thus, each class in the class diagram has its behavior specified in a state diagram. Figure 3.8b depicts an aspect concerning the role of front doors in the vehicle illumination system that was used as case study in [84]. In addition, to define details of inter-aspect relations, a rules-based language has been proposed. Basically, this language describes inter-aspects relations in term of events, transitions, and guard conditions for transition in state diagrams. Therefore, this can be seen as a complement to system behavior specification. Figure 3.8c shows a fragment of relation rules.

Lohmann et al. [74] propose the initial ideas for the CiAO operating system, which is the successor in the operating systems family called PURE [11] for deeply embedded systems, i.e. those embedded systems with very restricted processing power and memory availability. The
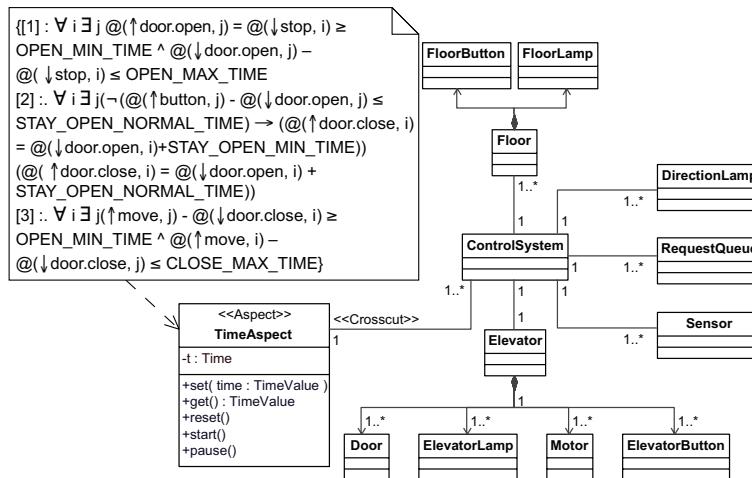
Figure 3.7.: Specification using a *time aspect* [133]
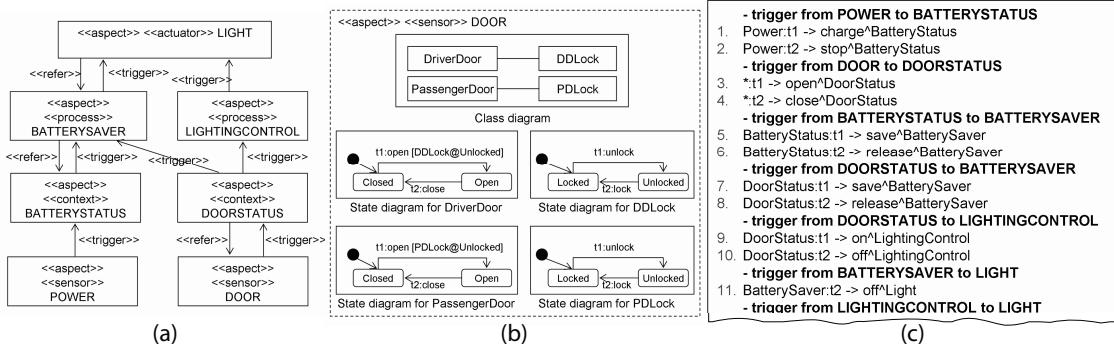
Figure 3.8.: AO modeling [84]: (a) functional and non-functional concerns; (b) aspects model; (c) inter-aspects relations rules

main goal of CiAO is to provide a very fine grain configurable operating system. Such granularity is obtained by using concepts of AO programming supported in the AspectC++ language [116]. In this sense, CiAO separates non-functional handling code from application components code by means of using aspects that are woven into the application code at the configuration phase. According to Lohmann et al. [74], such separation improves the reusability of application components. In [75], the authors reported their experience on using AO programming to design and implement the interrupt synchronization as a configurable property in the CiAO operating system.

AO concepts are used in the *Virginia Embedded Systems Toolkit* (VEST) [119] in order to separate and check non-functional properties in the analysis and composition of component-based embedded real-time systems. Two kinds of concepts were identified: *prescriptive aspects* and *aspects checks*. *Prescriptive aspects* are, in fact, advices (or adaptations) that modify the information of model elements (e.g. tasks priority or the replication level of a component). Such adaptations are described using a proprietary language called *VEST Prescriptive Aspect Language* (VPAL). On the other hand, *aspect checks* indicate the dependency check among components of a embedded real-time system. Such checking is performed using the information from the system model. Sometimes, component characteristics can influence other components behavior, e.g. tasks priorities and/or activation periods together with communication latency may influence the end-to-end scheduling of activities. Hence, VEST provides automatic check of components offered/required QoS that considers modifications in components performed by the aspects weaving process. That work has performed two case studies that lead to the conclusion that using aspects one can improve the analysis and composition of components in the design of embedded real-time systems.

Tsang, Clarke and Baniassad [124] present an interesting work comparing two implementations of a traffic simulator, which represents vehicles equipped with speed sensors as well as sensors capable to measure the distance from other surrounding vehicles, allowing vehicles be self-driven through four track streets. The first version is implemented using pure OO concepts and the Real-Time Specification for Java (RTSJ) [13], while the second version uses AO con-

cepts and AspectJ [67] in order to refactor RTSJ constructions (e.g. the creation of threads, memory management, synchronization, and others) that are encapsulated in aspects. The evaluation was made using an adapted version of the C&K metrics [28] to deal with AO constructions. That work has shown that using AO leads to the improvement of modularity because many RTSJ elements and constructions can be encapsulated in separated entities (i.e. aspects). However, according to [124], there are some metrics that are worse in AO compared to the OO version, e.g. number of methods per class, which hinders the understandability and maintainability. As conclusion, Tsang, Clarke and Baniassad [124] pointed that one can achieve more benefits from using AO if the relation aspect/application is broad and generic, meaning that maximizing encapsulation of redundant application code into aspects, one can have an overall improvement of the application code. The more redundant code a application has, better is the application code modularity provided by aspects to encapsulate such redundant code.

The *AspeCtual COmponent-based Real-time system Development* (ACCORD) approach [123] proposes the integration of component-based techniques with AO concepts for software development of real-time systems. In that work, a Real-Time COmponent Model (RT-COM) has been proposed. It supports the notion of time and temporal constraints, space and resource management constraints, and composability semantics. Additionally, the RT-COM provides the concept of gray box components that preserve some of the main features of a black box component, such as well-defined interfaces as access points to the component, and they also allow aspect weaving to change component behavior and internal state. Tesanovic et al. [123] define three kinds of aspects: *(i) application aspects* can change internal behavior of components, e.g. security, synchronization, real-time policy handling, etc.; *(ii) runtime aspects* refer to concerns related to system integration with the run time environment, e.g. resource demand, platforms to which components are compatible, WCET of components behavior in each platform, and others; *(iii) composition aspects* describe with which components a component can be combined, respecting component's version and offered and demanded QoS. Although ACCORD provides a component model that could allow the use of AO concepts at higher abstraction levels, that work specifies both components and aspects using the AspectC++ programming language [116].

The SysWeaver approach [99] uses different tools to generate code from models. The proposed approach separates functional requirements, which are modeled using Mathlab Simulink, from requirements the authors have called *para-functional* requirements, e.g. timing, replication, security, jitter, and others. In fact, *para-functional* requirements have the same meaning of non-functional requirements as used in this work. *Para-functional* requirements are modeled using the SysWeaver tool, which interacts with other complementary tools to provide toolchain integration, allowing domain-specific analysis such as schedulability or model checking of other system properties. Moreover, according to Rajkumar [99], that approach uses the concept of components, whose encapsulation mechanism combined with system properties model checking enable the construction of "systems-of-systems" that are "correct by construction". In [99], the SysWeaver approach has been used to design an ABS system for vehicles.

Balasubramanian et al. [9] present an approach to address crosscutting concerns in component-based MDE using *Aspect-Oriented Domain Modeling* (AODM). An AO model weaving technique is used to spread crosscutting concerns encapsulated in aspects. The tool called *Constraint-*

Specification Aspect Weaver (C-SAW) performs this aspects weaving in the context of *Platform-Independent Component Modeling Language* (PICML) [8], which is a DSML for developing com- ponent-based systems that has been developed using the *Generic Modeling Environment* (GME) [1]. PICML provides a proprietary modeling syntax for creating models of component-based distributed systems, which includes information on interfaces, components properties and system software building rules. C-SAW is a model transformation engine, which has been implemented as a plug-in to the GME. It takes as input the created PICML model and a text file describing aspects and transformations that must be performed in the PICML model. Such model transformations are described using the *Embedded Constraint Language* (ECL), which is an extended subset of the OMG's Object Constraint Language [90]. Particularly, ECL provides two important concepts: *(i) modeling aspects*, providing modular constructions to specify crosscutting concerns; and *(ii) strategies*, specifying transformations logic that will be applied in PICML model elements affected by modeling aspects. The approach proposed by Balasubramanian et al. [9] was intended to be applied in the development of large-scale component-based distributed system, in order to improve model scalability, and also the handling of crosscutting concerns. Thus, they presented a surveying system that uses many *Unmanned Aerial Vehicles* (UAV) to help in disaster recovery efforts stemming from floods, earthquakes, or hurricanes. UAV transmits videos from the surveyed area to a control center, where rescue teams can decide rescue actions. C-SAW has been used to perform modifications in several components of different modeled UAV.

3.3.4. Discussion

The use of AO paradigm in initial computing system design phases is recent and has not achieved the maturity level of approaches using the OO paradigm. Such claim is supported by the diversity of proposals for AO modeling that can be found in the literature, i.e. there are several approaches to specify the same concepts using their own form, repeating what happened before the UML creation. However, the same cannot be said for AO implementation, which has achieved a certain degree of maturity as can be seen by the wide use of languages, such as the AspectJ or AspectC++. Although there are proposals to apply AO in early phases, there is no standard form to separate functional requirements handling from non-functional requirements. In particular, [120], [31], and [45] are the most remarkable works. The first one is an approach being refined to support other AO languages in addition to AspectJ. The second one proposes an interesting approach, but the specification of how a theme affects other themes is not adequate due to the lack of scalability, i.e. in systems with large amount of crosscutting themes, the specification of *bind* relationship to express weaving hinders model maintainability and evolution. Finally, the third approach proposes extensions to UML in order to allow the specification of AO concepts and also to perform AO model weaving.

To the best of our knowledge, the use of AO concepts in the domain of distributed embedded systems is still low. There are few approaches suggesting their use in the implementation or configuration of embedded software, and even fewer that try to apply these concepts in design or modeling. For example, the approach proposed by Zhang and Liu [133] suggests the use of only one aspect to deal with all time related non-functional requirements. The time require-

ments specification proposed by that approach is not appropriate because time requirements can have different viewpoints (e.g. periodic activations, deadlines or WCET for algorithms execution, latency measurements, and others) that can be misunderstood by designers. In addition, the use of UML notes to specify important information is not appropriate due to the lack of representation in the UML meta-model. Besides, there are other important requirements from the domain of distributed embedded real-time systems whose handling can be improved if aspects are used. The approach proposed by Noda and Kishi [84] uses only aspects to model all concerns in embedded systems design. However, that work does not deal specifically with timing, embedded, or distribution non-functional requirements. They can be handled separately from functional requirements but this must be done specifically from design to design because of the textual specification of aspects composition. Moreover, Tesanovic et al. [123], Rajkumar [99] and Balasubramanian et al. [9] propose component-based design approaches that specify aspect in terms of textual descriptions instead of graphically modeling them.

3.4. Code Generation

3.4.1. Introduction

Code generation means to use a computer program to assist in production of source code, be it application source code, HDL source code, code for platform configuration, and others. Commonly, a code generator program takes as input a high level specification in addition to a set of templates in order to create one or more source code files as output. According to Herrington [59], code generation is not constrained to be only a quick way to produce source code. Other benefits can be achieved as follows:

- **Quality:** code generator tools use templates to produce code (for a target platform) from elements specified in high level models. The more complete a set of templates is, the better is the quality of the obtained generated source code. If the templates describe an optimized code generation based on designers quality and optimization criteria, a quality increase is reflected in the final generated source code;

- **Consistency:** the naming standardization for classes, methods, and attributes is fully consistent in the generated code. Hence, the application of naming standardization facilitates classes interfacing and use because such standards are defined within the templates;

- **Productivity:** code generation increases productivity gains due to their ability to adapt quickly to changes during design. In other words, modifications in the specification can be automatically propagated to system implementation. In addition, code generation allows the inclusion or exclusion of big portions of source code;

- **Abstraction:** advantages in terms of design abstraction level can be achieved using code generation tools that work with input specifications (e.g. models of the system structure and behavior, database schemes, or user interface designs) in a neutral form, i.e. using platform independent languages. In other words, it is possible to generate source code

for different programming languages (such as Java, Smalltalk, or C++) from the same abstract model.

This section presents some proposals to generate code from UML models, as well as commercial tools that implement such code generation. Additionally, some works that produce HDL code from UML are also presented.

3.4.2. Code Generation from UML Models

Many different approaches to generate source code from UML models can be found in the literature. Some of them use only one diagram (e.g. class diagram), while others use a combination of different diagrams (e.g. class diagrams with state, sequence and/or activities diagrams) to generate code ranging from classes skeletons to code containing system elements behavior. This subsection present the some approaches.

The work presented in [55] demonstrates the mapping from class diagrams to Java source code. This approach allows the generation of high-level class skeletons, which allows abstraction of details on attributes implementation, i.e. attributes data type, from the class implementation point of view. In other words, the implementation does not need to know the attributes existence because their stored values are accessed only through get/set methods. The code generation process only considers classes from the UML model decorated with «Entity» stereotype. Each entity is mapped to an interface and a pair of classes that implement this interface, i.e. for each entity *X*, the following Java elements are generated: *(i)* an interface named *X*; *(ii)* an abstract class named *XAbst*; and *(iii)* a concrete class named *XInst*. The created interface contains the operations defined in the UML model for the entity. The abstract class implements the interface and specifies attributes, as well as their data types (e.g. integer or string attributes) and auxiliary methods that access theses attributes, which are generated automatically by the code generator. Finally, the concrete class extends the abstract class by means of adding the methods that must be implemented to support the operations from the entity interface. In fact, the code generator produces empty methods that must be filled by the programmer in order to provide the entity behavior. The concrete class accesses class attributes by means of the auxiliary get/set methods specified in the abstract class. Additionally, associations among classes in the UML model are represented by *cursors*, which are entities encapsulating the complexity of associations navigation and updates. The concept of *cursors* has been proposed to separate the associations semantics from their real representation and implementation.

Burmester, Giese and Schäfer [21] have presented a code generation approach that uses the FUJABA (From UML to Java And Back Again) Real-Time Tool Suite [20] to generate code for RTSJ applications. System structure is modeled using the components diagram from UML, while behavior is specified with an extended version of the UML state diagram called *Real-Time Statechart*. The PIM of the system is transformed into a PSM that uses the SPT profile [88] to specify real-time concerns. Every Real-Time Statechart is transformed into, at least, one active object, which represents the main thread and is implemented as periodic *RealtimeThread*. At each period, all transitions that can be triggered are checked, and those that passed some conditions (see [21]) are executed. Orthogonal states are not implemented

as multiple concurrent periodic threads, but by exactly one periodic thread (the main thread) and multiple concurrent aperiodic threads. It is important to highlight that, depending on the deployment information of Real-Time Statecharts, a JVM can have multiple periodic threads, i.e. one for each Real-Time Statechart deployed in the JVM.

Bordin and Vardanega [16] propose a source code generation strategy for multiple target OO languages from HRT-UML models, i.e. UML models annotated with the FW profile [26] that specifies HRT-HOOD [23] concepts. In that work, the RTSJ has been assessed in order to check (regarding some requirements proposed by that authors) its potential to be used by code generation tools. RTSJ source code with Java annotations has been generated from HRT-UML models. Such annotations allow traceability of HRT-HOOD concepts (e.g. cyclic or sporadic execution of methods, or protected or unprotected method execution) between model and source code, and also, decreases the size of the generated code because it hides information from the programmer [16]. Hence, a pre-processor, which converts these annotations into plain RTSJ code, is required to be used before the generated source code compilation.

A code generation approach based on MDA concepts was presented by Hausmann and Kent [57]. In order to generate skeleton source code from class diagrams, the proposed approach uses transformations based on meta-models. For each target language, a meta-model, as well as the mapping rules from the PIM to the PSM, must be specified. The process of creating mapping rules is based on pairs of elements, their relationships, domains and constraints. A pair represents two elements, in different models, that are related through a relationship, which specifies relation constraints, and in which domain elements are linked. The mapping between PIM and PSM is specified in class diagrams, in which meta-model elements of different models are linked by means of class diagram associations. Additional OCL constraints can be included in the associations. Besides not showing the final generated source code, mapping rules from UML to Java language has been depicted in [57]. It can be seen that this graphical approach to describe mapping rules can assist in the overall visualization of the transformation, however, it can hinder the creation of more complex mappings among meta-model elements of different models.

Generation of AO source code is the focus of the work presented in [58]. The goal is to allow automatic generation of AspectJ source code from extended UML diagrams by using the Theme/UML approach. Theme/UML models are exported to XMI files which are taken as input to a code generation tool developed with the *eXtensible Stylesheet Language Transformations* (XSLT) [130]. The generated code is not complete, i.e. only skeletons of classes and aspects are provided. However, for aspects, the code includes the pointcuts that link advices with join points that were specified in the Theme/UML model. Furthermore, Hecht et al. [58] state that it is possible to generate code for the body of advices, since created Theme/UML diagrams provide enough information on the modification of system elements, which will be executing during the aspects weaving process.

Nitto et al. [83] use UML as a language to describe processes and also to validate modeled processes. To allow the intended validation, UML models are translated to *ORCHESTRA Process Support System* (OPSS) models [33], which are executable models with formal semantics. In an OPSS description, one process is divided into activities performed by agents. Elements

in class, activities and state diagrams are transformed to OPSS elements. The application structure, which is specified in the class diagram, is translated directly in Java classes (skeleton source code) of the OPSS framework. State diagrams represent the lifecycle of an object, and are translated to Java code representing objects behavior. Finally, activities diagram describes activities flow of a process, as well as associations among activities and agents. It is used to produce Java code that represents the precedence relationship of activities execution.

The formalization of class and sequence diagrams has been proposed by Long et al. [76] in order to allow code generation from UML 2.0 models. The proposed model semantics is based on the *Relational Calculus of Object Systems* (rCOS) semantics, which was devised to design OO systems. That work generates skeleton code from class diagrams, and code for methods body from the sequence diagram. The code generation algorithm interprets sequence diagrams as a composition of messages sequences, allowing its use for creation of code from separated fragments of sequence diagrams. Source code can be generated only if the model passes a consistency checking. In that work, the code generation for rCOS language is demonstrated, showing how class skeletons, containing attributes and empty methods, are created. Considering the behavior, each message in sequence diagrams is transformed to a method call in rCOS. Nested messages are mapped to method calls in the body of the parent method.

The generation of SystemC code from UML models is investigated in [2]. Initially, that works assesses constructions of UML 2 and those of SystemC 2.2, comparing them in order to create a mapping between concepts of both languages. Considering structural specification, UML packages are mapped to SystemC name spaces, and UML active classes and classes with ports are mapped to SystemC modules. However, other types of non-mentioned classes are mapped to standard C++ classes. Ports in UML have a required and a provided interface. On the other hand, in SystemC a `sc_port` must have exactly one interface, which corresponds to the required interface of UML port. Provided interface of UML ports is equivalent to `sc_export` construct of a SystemC module. Regarding the specification of communication among elements, UML communication can be modeled as signals, i.e. asynchronous messages, that are sent through ports. The destination is specified using connectors. At the receiving object, the signal is stored in a queue and will eventually be consumed. In SystemC, ports are connected through channels, whose reference is stored in the port during system initialization. Therefore, to map the mentioned semantics from UML to SystemC, UML connectors are mapped to SystemC `sc_fifo` channels that connect `sc_export` of a module to `sc_port` of another one. Furthermore, to produce SystemC source code from UML models, the mapping process is composed by three steps: *(i)* the initial UML description is manually annotated with the SystemC profile; *(ii)* the model is automatically transformed into a new UML description that includes direct representation of SystemC construction, e.g. state diagrams are translated to classes, in which each method implements the behavior performed in a state. Additionally, UML concepts without SystemC correspondence are removed from the model; and *(iii)* the UML model produced in the previous step is transformed to the corresponding SystemC code. This transformation is an one-to-one transformation. This SystemC code generation approach has been implemented as a plugin to the Telelogic Tau tool [63], using its the C++ code generation facilities.

There is an interesting on-going research in the Embedded Systems Lab of the Federal University of Rio Grande do Sul, whose initial results were published in [39]. That work proposes a meta-modeling infrastructure, called *Model-Driven Embedded System design* (MoDES), to represent distributed embedded real-time systems in higher level of abstraction. The goal of MoDES is to provide a common infrastructure to various MDE tools, as for example, high-level design space exploration or code generation tools. That approach suggests a methodology that applies successive refinements from an initial specification, which is a PIM, to an implementation model of the system using a selected target platform. The initial PIM (which can be specified using UML, Simulink, or other modeling language) is transformed into an application model that is an instance of the *Internal Application Meta-Model* (IAMM), which represents application functionalities in a uniform manner. Likewise, models of many implementation platforms (e.g. SystemC, Java, VHDL, and others) are specified using a uniform platform representation called *Internal Platform Meta-Model* (IPMM). The set of mapping rules is described using the *Mapping Meta-Model* (MMM), which is used to guide the transformation of IAMM and IPMM model in a system realization model named *Implementation Meta-Model* IMM. The IMM represents the implementation of the initial model (i.e. the one specified using UML, Simulink, etc.) using a selected target platform (e.g. Java, VHDL, SystemC, etc.). Hence, it is possible to generate code from the IMM.

3.4.3. Commercial Tools

This section presents some commercial CASE tools that allow code generation from UML diagrams. During the study of the state of the art in code generation, many tools with different automatic code generation capabilities were found: from code skeleton for classes to tools that are capable of generate configuration files for server of distributed components such as CORBA or Enterprise Java Beans.

Rational Rose [61] CASE tool has many different versions with different code generation capabilities. All of them work on the previous version of UML, i.e. the version 1.4. The tool Rational Rose Technical Developer (previously called Rational Rose Realtime) allows the automatic creation of Java, C and C++ source code. It generates code skeletons for classes. However, if any code was informed in the *Code* tab of methods specification, this text is also included in method's body. Additionally, behavioral code can be generated from state diagrams. This code generation follows the same approach, i.e. code is typed in the *Code* tab of states.

Rhapsody [62] and Tau [63] are modeling tools from Telelogic, which was recently acquired by IBM. Both tools supported the specification of UML 2.1 models. In addition, Tau also supports SysML. Rhapsody can generate code for Ada, C, C++ and Java, while Tau for C, C#, C++ and Java. The approach to produce code is similar to the Rational Rose tool, i.e. both Rhapsody and Tau generate code skeletons for classes, and the body of methods must be written in a special field in methods specification.

Borland's Together [17] CASE tool allows the generation of code skeletons for class, and also methods body. It uses the last version of UML (version 2.1) to automatically create code for Java, J2EE, C++ and C#. Code skeletons are generated from the class diagram, while methods

body from the sequence diagram. The code generation can be customized by means of changing the generation templates.

Artisan Studio [6] (previously called Artisan Real-time Studio) supports UML 2.0 and SysML modeling, and also automatic generation of C, C++, C#, Java and Ada source code using external tools. It also generates code skeletons from class diagrams. In addition, source code for classes behavior is generated from state diagrams. Actions performed in each state must be written (in the selected target language) in special fields in states specification. To allow code generation, UML elements must be decorated with stereotypes of the target language, and hence, external code generators can produce the right constructions in the selected target language. Furthermore, C/C++ code generation tool uses templates allowing some customization of the generated code.

Poseidon for UML [50] is a CASE tool that supports UML 2 modeling, and implements a script-based code generation, which uses the Velocity Template Engine [3]. There are predefined scripts for the following languages: C#, C++, CORBA IDL, Delphi, Perl, PHP4, SQL DDL, and VB.net. The designer can create its own code generation script that accesses information of the UML model to generate code for other target languages. However, only the class diagram can be accessed, and thus only code skeletons can be created.

Other tool is the ObjectiF [80], which also uses a template-based code generation approach to produce code skeletons from class diagrams. This tool uses stereotypes to assist in Java, C# and C++ source code generation. ObjectiF can generate automatically *get()* and *set()* methods (with the corresponding behavior) for attributes that are decorated with a specific stereotype. Additionally, it generates attributes and methods representing composition, aggregation, and plain associations among classes. Moreover, it can also create the implementation of unit tests for classes using NUnit or JUnit.

The CodeGenie MDD toolset [40] provides a code generation tool that takes as input XMI files from executable UML models. Three levels of code generation are supported: *(i) code skeletons for classes* representing only software static structure; *(ii) code skeletons with architectural mechanism* including architectural mechanisms (e.g. event queues, stacks, circular buffers, etc.) in addition to classes structure; and *(iii) code skeletons with architectural mechanism and behavior* complementing the previous level by adding behavioral code generated from the state diagram.

3.4.4. Discussion

Code generation from UML model is not a new topic. As one can see, generation of code skeletons from class diagrams is a well-defined approach due to the large number of tools that can generate this kind of code. Some of the presented works can generate behavioral code from state diagrams. However, a drawback can be pointed: depending on the target application, state machines are not the most suitable model of computation to describe the developed application behavior. Besides, the specification of actions performed in state diagrams is neither standardized nor a common consensus. It can be done using programming languages or more abstract

textual action languages. Thus, proposals that use other UML behavioral diagrams can be seen as an interesting option to specify actions in an UML model. Although sequence diagrams are used in some works, there is no mention on the use of new constructions available in the UML 2.x, such as those to specify "ifs", "loops" and others. In other words, only method calls are generated that is not sufficient to generate the complete code from the UML model.

Other open problem is the interpretation of UML diagrams and their combination. Different viewpoints offered by different model elements provide remarkable information, which can be combined to obtain the complete description of system structure and behavior. However, there is no defined semantics for different diagrams integration. Therefore, interpretation rules must be created to allow the extraction of a concise specification which, for code generation purposes, must be unambiguous and simple.

Finally, it was observed that most of the approaches propose the mapping 1-to-1 between model and source code, i.e. the more detailed a model is, bigger is the amount of code lines that can be generated from it. However, the specification of excessive details in the model decreases a key advantage of using models: the visualization facility of the structure and behavior of the modeled system. Including details in excess hinders model understandability, and also decreases the reuse of its elements. In order to avoid unnecessary details in the model, a code generation tool could infer missing information on model elements based on modeling guidelines. Hence, using 1-to-N mapping rules between model elements and lines of code, this tool could generate code as complete as the completeness of the mapping rules specification. Other way to keep models without unnecessary details is the use of AO concepts. Details that are not directly related to the desired functionalities can be encapsulated in aspects. Crosscutting behavior can also be represented in this way. Therefore, the code generation tool could be aware of the adaptations performed by aspects, which would modify the generated code. In other words, the code generation tool could also perform aspects weaving. Thus, using AO concepts at modeling level allows the use of non-AO target languages.

3.5. Discussion on the Open Problems

This section discusses open problem identified in the works previously cited in this chapter. UML is broadly used and well accepted in the domain of software engineering for modeling "general purpose" computing systems. Such situation has been drawing the attention of professionals of other computing domains, such as embedded systems and hardware designers. One feature that is desired by great part of designer, in all computing domains, is the capability of automatic source code generation from high-level specifications, in order to decrease design effort and avoid error prone manual coding activities.

Related work presented in section 3.2 compared some approaches that use UML in the design of systems whose functionalities are implemented in either software of hardware. Every approach uses different diagrams to structure and behavior specification. This shows that there is no consensus on which diagrams must be used to specify a distributed embedded real-time system. Additionally, it can be observed that many of these approaches, e.g. [5], [82], [100]

and [7], use proprietary profiles to extend UML semantics according to their needs. Given that UML does provide mechanisms to extend its semantics, doing that with non-standard (i.e. proprietary) profiles is not a good approach because this hinders the specification understanding by stakeholders outside the design task. The use of standardized profiles, as in [96], overcomes the mentioned problem. Thus using standard profiles provided by organizations such as OMG (the group that maintains UML standard) is a very important issue that must be approached by new modeling techniques. Besides, excluding the work presented by Balarin et al. [7], none of the presented approaches separate the handling of crosscutting concerns, which decreases the modularity of artifacts (e.g. models or source code) created in previous projects, hindering their reuse in new projects.

Approaches that separate the handling of functional and non-functional requirements are presented in section 3.3. The majority of the cited work aims at the design of software for "general purpose" computing systems, i.e. not embedded software, whose developer do not have to worry about constraints that are intrinsic to embedded real-time systems domains, such as timing constraints, restricted processing power, limited memory amount, or energy consumption. The mentioned separation of concerns is becoming popular in that domain by means of using concepts of the AO paradigm. Using aspects to handle crosscutting non-functional requirements improves the modularity and the encapsulation of concerns. There are many attempts to adapt the UML for representing AO concepts in models. The main drawback of approaches, such as [120], is that they propose changes in the UML graphical syntax instead of using the UML extensibility mechanism as [45] and [98] propose. Such heavyweight extensions hinder the language standardization. The use of lightweight extensions (i.e. UML profiles) is preferable, since they allow the use of any modeling tool that supports the standard extensibility mechanism of the UML specification. Furthermore, the use of *composite patterns*, which is proposed by Clarke and Walker [31], allows the use of UML standard graphical representation without modifications. However, the problem of this approach is that the specification of the affected elements is not scalable, i.e. it is not suitable to specify composition relationships of crosscutting concerns that affect a huge amount of other concerns, leading to problems in the specification of large systems.

Considering the design of distributed embedded real-time systems, there is little discussion on the use of AO concepts. Few works can be found in the literature. Most of them are related to the implementation phase of such systems instead of earlier design phases. [133] and [84] are exceptions. Zhang and Liu [133] propose the use of a single aspect to specify the handling of timing requirements within UML models. Other important non-functional requirements of embedded systems domain are neglected. Additionally, the specification of timing properties (as notes in the class diagram) is hard to understand and also not suitable due to the weak relation with UML meta-model elements. Noda and Kishi [84] uses the symmetric approach for modeling crosscutting concerns likewise [31]. Although it is an interesting approach, it suffers the same drawback of [31] approach, i.e. the lack of scalability. Moreover, mixed specification using graphical elements and textual descriptions is not desirable, because it is not easy to visualize aspects composition, i.e. which aspect crosscuts other aspects. [123], [9], [99], and [119] propose the use of AO in component-based MDE of embedded systems. The first three approaches propose the specification of aspects and their adaptations in terms of proprietary

text-based languages. Moreover, [123] and [9] use proprietary modeling syntax to model system components, while [99] uses Simulink syntax. Although the mentioned modeling syntaxes provide DSML to specify embedded system components, they lack standardization for specification. Going towards the approaches that use AO for implementing embedded systems, [119] use aspects to check if there is a matching of required/offered information by components that are related with each other. [74] and [124] are implementation related approaches, i.e. they use AO programming languages to deal with crosscutting non-functional requirements. Analyzing the results reported by both works, one could conclude that there are a lot of open issues that can be investigated. Other crosscutting non-functional requirements, such as access synchronization of shared resources, memory management, or communication issues, could be handled with aspects at application implementation and target platform tailoring. Moreover, the creation of a set of aspects to deal with non-functional requirements from higher abstraction levels (i.e. requirements specification and modeling) to more concrete levels (i.e. implementation and platform tailoring) is a very interesting research topic.

Analyzing the mentioned code generation approaches, it can be stated that there is no formalization or even consensus for UML diagrams interpretation or integration of different diagrams. Such problem hinders the generation of complete code for computing systems. However, one exception is the class diagram, for which there is a "well-defined interpretation". All presented works can at least generate code skeleton for specified classes. Although useful, code skeletons are a small fraction of all code that could be generated from the entire UML model. The problem is that there is no consolidated approach to generate behavioral code from elements of other diagrams. Some works propose the use of state diagrams, whose actions are specified using the target programming language or any other kind of textual action language. Others propose the use of sequence or activities diagrams but not all constructions can be translated to code in a given platform. Anyway, the generation of code containing the behavior specified in the UML model is still not well defined compared to the generation of code skeletons from class diagrams.

Some directions for MDE of distributed embedded real-time system were pointed, however there are many open problems that can be addressed from the research point of view. Those open problems go from the formalization of models interpretation semantics to the empirical use of mappings to transform models into source code. In addition, using AO concepts would allow a better modularization and handling of crosscutting concerns and non-functional requirements. Code generation approaches could consider AO concepts specified within UML models in order to allow code generation for both AO and non-AO programming languages. Hence, besides code generation, the tool could perform aspects weaving in the generated code, and also tailor the target platform based on the aspects specified in the model. Additionally, optimization could be performed while reading the UML model or generating code.

Chapter 4

MDE process for DERTS design

4.1. Introduction

One of the goals of this thesis is to propose a design flow that increases the abstraction level during design of distributed embedded real-time systems, in order to address its complexity. The proposed design flow must allow a smooth transition from initial specification phases to implementation/coding phases. For that, the *Aspect-oriented Model-Driven Engineering for Real-Time systems* (AMoDE-RT) design flow has been created. AMoDE-RT uses MDE techniques combined with AO concepts to accomplish the mentioned goals. It is important to highlight that, to be effective, AMoDE-RT needs adequate tool support (which is also provided by this work) in order to assist its use in the design of distributed embedded real-time systems. Figure 4.1 depicts an overview of the AMoDE-RT design flow.
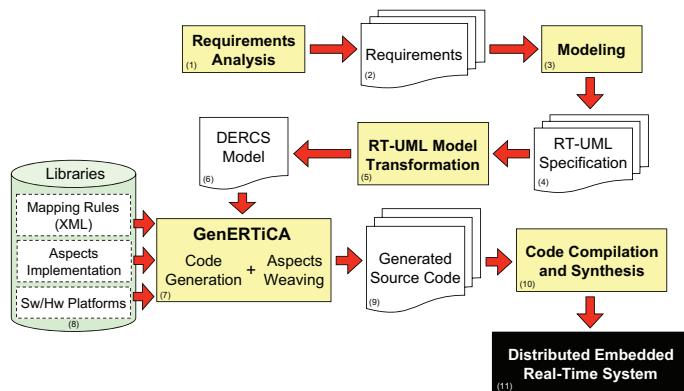
Figure 4.1.: Overview of the AMoDE-RT design approach

4.2. Aspect-Oriented Model-Driven Engineering for DERTS

The first step in AMoDE-RT is gathering requirements and constraints of the distributed embedded real-time system. This is performed using the RT-FRIDA approach, which is an extension to the FRIDA [10] requirements analysis approach aiming at applying it into the distributed embedded real-time systems domain. RT-FRIDA is the result of a cooperative work performed together with the colleague Edison Pignaton de Freitas for his M.Sc. dissertation [37]. In addition to requirements analysis, the RT-FRIDA also shares the modeling step with AMoDE-RT. A brief discussion of both steps is given in the following paragraphs, and an in depth discussion on AMoDE-RT modeling approach is presented in the next chapter. For details on RT-FRIDA, readers should refer to [37].

An overview of RT-FRIDA steps is depicted in figure 4.2. The requirements identification step is the first step and is divided in two activities that can be performed in parallel: *functional requirements* specification and *non-functional requirements* specification. Firstly, a use case diagram is created. It depicts all expected functionalities for the distributed embedded real time system, and also the external elements that interact with these functionalities. For each use case specified in this diagram, a functional requirements template (see figure 4.3a) must be filled. After that, the filled templates of functional requirements are analyzed regarding possible conflicts. Thus a conflicts resolution matrix is created, in which the first row and first column are filled with the IDs of functional requirements. If a functional requirement conflicts with other one, a "X" is marked in the cell that intersects row and column of conflicting requirements.

For non-functional requirements specification, additional steps are then performed. RT-FRIDA provides checklists (see an example in figure 4.4a) that assist in identifying the non-functional requirements that have been presented in section 2.3. Answering these checklists' questions helps in the identification on which non-functional requirements affect functional requirements. As performed for functional requirements specification, a template must be filled for each non-functional requirement (see figure 4.3b). In addition to checklists, there is also a parser that can be used to identify key words in documents written in natural language, indi-
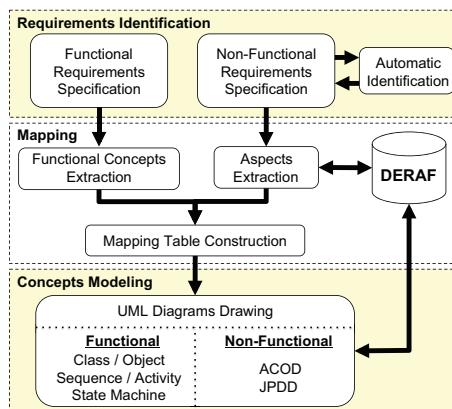
Figure 4.2.: Overview of RT-Frida

cating the presence of unspecified non-functional requirements [37]. After that, there is also a conflicts resolution step similar to the one in functional requirements specification, i.e. designers fill a conflicts resolution matrix indicating which non-functional requirements affect others.

The second step of RT-FRIDA approach is the mapping of requirements to (candidate) design elements. This is done using a mapping table as the one depicted in figure 4.4b. As it can be observed, rows indicate functional requirements, while columns non-functional requirements. If any non-functional requirement affects any functional requirement, a "X" is marked in the cell that intersects row and column of involved requirements. Furthermore, this mapping table links requirements to (candidate) design elements, allowing requirements traceability from requirements analysis to system design. Hence, the last column indicates which (candidate) classes in the design model are responsible to handle functional requirement. Similarly, the last row indicates which aspects are used to handle crosscutting non-functional requirements. Aspects are provided by a predefined aspects framework named *Distributed Embedded Real-time Aspects Framework* (DERAF), which is discussed in details in the section 5.2 of the next chapter. It is important to highlight that this table is initially filled with candidate handling element and, during the whole design phase, it can be modified/updated with new elements that will be included to the design model. Consequently, it is important to keep this table updated in order to maintain traceability of requirements to design elements and vice-versa.

At the end of these two steps, designers have produced a set of documents specifying functional and non-functional requirements that the system under development must deal with, and also the relationships among these requirements.

(a) Funtional requirements template

		Item	Description
General	Identification	ID	This identifier allows requirements traceability over the whole project.
		Name	Use case name.
		Goal	Description of the use case goals.
		Author	The person that is responsible for the use case description.
	Context	Pre-condition	A condition that must hold **before** the execution of the use case.
		Post-condition	A condition that must hold **after** the execution of the use case.
	Actors	Primary Actor	Actors that are the source of the events for the main scenario stimuli
		Secondary Actor	Passive actors that interact with the use case, but do not execute any action within its context.
	Decision and Evolution	Priority	Used to decide the relative importance among use cases. There are three levels: **Maximum, Medium, Minimum.**
		Situation	A requirement can be in one of the following situations: 0 - Identified; 2 - Specified; 4 - Canceled; 1 - Analyzed; 3 - Approved; 5 - Finished;
Paths		Main (Normal)	Describes the main flow of the use case, as well as its results, without condiser error conditions.
		Alternate	Describes the alternate flow to the use case.
		Exception	Describes a exceptional situation in the use case flow.
Scenario		Main	Describes the main steps of the use case scenario.
		Variations	Describes steps that modify one or more steps within the scenario.

(b) Non-funtional requirements template

		Item	Description
	Identification	ID	This identifier permits the requirement traceability over the whole project.
		Name	Crosscutting concern's name.
		Autor	The person that is responsible for the corsscuting concern specification.
	Specification	Classification	Classification in which the concern belongs.
		Description	Description of how the concern afect system functionalities.
		Afected Use Cases	List of the use cases affected by the concern.
		Context	Determines in which situation a use case is affected by the concern.
		Scope	(Global/Partial) The requirement is **global** if it affects the whole system, and is **partial** if afects only part(s) of the system.
	Decision and Evolution	Priority	Concern's importance regarding other non-functional concerns. Higher numbers represent higher importance.
		Status	0 - Identifyed; 3 - Approved; 1 - Analysed; 4 - Canceled; 2 - Specifyed; 5 - Finished;

Figure 4.3.: RT-FRIDA templates for requirements specification

(a) Checklist example

	Relevance	Priority	Restrictions / Conditions / Description
Time			
Timing			
Is there any periodic activity or data sampling?	**X**	8	Movement Control; Environment Sensing; Main Rotor Sensing; Back Rotor Sensing;
Is there any sporadic activities?			
Is there any aperiodic activity?			
Is there any restriction in relation to the latency to start an execution of a system activity?	**X**	9	Corrective Action
Is there any specific instant to start or finish an execution of a system activity?			
Was any WCET specified? Or at least, is there any concern about this?	**X**	10	The smapled data of both rotors must be ready at a maximum of 10 ms.

(b) Requirements mapping table

		Non-Functional Requirements				**FR handling elements**
	ID	NFR-1	NFR-2	...	NFR-n	
Functional Requirements	FR-1					Class1, Class3
	FR-2					Class2
	...					...
	FR-n		X			ClassN
NFR handing elements		Aspect1	Aspect2 Aspect3	...	AspectN	

Figure 4.4.: Other tools provided by RT-FRIDA

These documents are then used in the next phase: system modeling. UML diagrams annotated with the stereotype of the MARTE profile [94] are used to model the structure and behavior of distributed embedded real-time systems. In this phase, UML models are created and successively refined up to achieve the desired level of detail, providing sufficient information to allow system realization. In the initial UML model, elements describe concepts that are closer to the target application domain, e.g. sensors, steering devices, turbines, speed and trajectory information, robot arms, etc. These elements represent problem domain concepts, hiding details about their implementation. Higher abstraction levels are easier to understand, and allow designer to focus on applications foundations instead of concerning about implementation issues. Thus, they represent the handling of functional requirements. Application elements can be reused from previous designs, and hence, it is possible to create repositories of application domain elements. Such elements can be made up of many different UML elements and/or diagrams. For instance a robot arm can be compound of three joints and a gripper. To reuse this domain-level element, at least five classes (three for the joints, one for the gripper, and the composite class for the robot arm) are reused. Additionally, behavioral diagrams describing robot arm's behavior could also be reused.

The specification of non-functional requirements handling is done with assistance of aspects provided by DERAF. They are used in two moments: *(i)* in modeling phase (see section 5.3.2); *(ii)* in implementation phase, more specifically, in code generation/aspects weaving step (see chapter 6). During modeling phase, aspects are chosen based on their high level semantics to handle crosscutting non-functional requirements. For instance, the *ConcurrentAccessControl* aspect deals with issues on concurrent access control of shared resources. Hence, if the system has this non-functional requirement, *ConcurrentAccessControl* aspect is selected and specified in the *Aspects Crosscutting Overview Diagram* (ACOD). Moreover, based on information of

the mapping table created previously, designers must specify which UML model's elements are affected by this aspect. For that, designers create *Join Point Designation Diagrams* (JPDD), which are special diagrams that specify model elements selection. JPDD, which can be stored in a repository and reused in further designs likewise DERAF aspects, are specified using common UML modeling tools with support to profiles. Details on modeling both functional and non-functional requirements are given in the next chapter.

At the end of modeling phase, designers have created a UML model that specifies elements to deal with the functional and non-functional requirements, using, respectively, OO and AO concepts.

Although increasing abstraction level during design is good for managing complexity, the higher the abstraction level is, more are the chances of ambiguous or even erroneous interpretations of the same specification. Usually, high level specifications cannot be executed in computational devices (e.g. microprocessors, integrated circuits, or Programmable Logic Controllers (PLC)) due to their incomplete semantics and/or lack of sufficient details. To overcome these issues, specification ambiguities must be removed, and also computational elements (e.g. FIFO queues, scheduler, synchronization mechanisms, and others) must be included into these high-level specifications. A transformation of the initial model into a more concise one must happen. AMoDE-RT's third step performs the transformation of the UML model annotated with MARTE profile stereotypes into an instance of the *Distributed Embedded Real-time Compact Specification* (DERCS), which is a PIM suitable to code generation and model execution purposes. By transforming UML into DERCS, the information on system structure, behavior and non-functional requirements handling, which is spread over different UML diagrams whose information may overlap each other, is combined in fewer and concise elements in DERCS representation. For more information on UML to DERCS transformation see section 6.2.

The next step is source code generation from the DERCS model. As mentioned, one of the goals of this work is to provide a smooth transition from high-level models to the implementation of distributed embedded real-time systems. Thus, a code generation tool called *Generation of Embedded Real-Time Code based on Aspects* (GenERTiCA) has been developed. In fact, GenERTiCA performs not only code generation, but also aspects weaving. The code generation process executes a set of scripts (mapping rules) to perform model-to-text transformations from DERCS elements to constructions in the target platform.

Mapping rules are specified as small scripts that create source code fragments (representing target platform constructions) for elements in the DERCS model. Source code files are made up of these generated code fragments. Scripts are stored and organized in mapping rules files specified using the eXtensible Markup Language (XML) [129] format. Therefore, it is possible to create a repository to allow the reuse of previously created scripts and mapping rules for platforms. The code generation process iterates all elements looking for the script that defines the mapping from the element being evaluated into suitable construct(s) in the target platform.

Additionally, if the element under evaluation is affected by any DERAF aspect, the aspects weaving process is performed after the generation of the code fragment. GenERTiCA uses aspects implementations to modify code fragments, i.e. changes in generated code fragments are

performed by implementations of aspects adaptations. There is also the possibility to perform adaptations in DERCS model elements before generating code. Thus, GenERTiCA provides code and model aspects weaving. It important to highlight that implementations of aspects adaptations are scripts similar to "normal" mapping rules scripts. Hence, it is also possible to create repositories of different implementations for the same aspect adaptation, depending on the target platform. Moreover, DERAF aspects are also used to tailor platforms, in the sense of configuring the selected target platform by adding only services that are required by the application. More details on GenERTiCA, and also the code generation and aspects weaving processes are given in chapter 6.

The last step of AMoDE-RT is the use of a third party tool to compile and synthesize the generated application code. In addition, the generated platform configuration files are used to configure the final platform that will be deployed. After that, the realization of distributed embedded real-time system being designed is ready to be executed or tested.

4.3. Adaptations in the SEEP design flow

As already mentioned, this work was developed within the scope of the SEEP project. Thus it proposes adaptations to the original SEEP design flow, in order to accommodate the proposed AMoDE-RT design flow, as depicted in figure 4.5. The start of SEEP flow has been extended to incorporate steps 1, 2 and 3 of AMoDE-RT (see figure 4.5a). Thus, in the original "High-level Model" step, "Requirements Specification" and "Functional Specification" were substituted by, respectively, "Functional Requirements Specification" and "Non-Functional Requirements Specification". A "System Model Specification" step has been included after requirements specification. In this step, designers can reuse application elements and DERAF aspect in the UML model, as mentioned above. The result of this step is the "Complete System Specification", which is the created UML model whose diagrams are decorated with stereotypes of the MARTE profile. After that, a "Remove Ambiguities from Model" step was added. It represents the transformation of the UML model created in the previous step into a DERCS model, which is represented by the "Compact System Specification" box. It is important to highlight that the DERCS model can be used as input to the system exploration activity, which incorporate the platform-independent computational elements mentioned in the previous section. Thus, different implementation of these elements can be evaluated and the one that best fits system requirements can be selected.

Other proposed extension to the original SEEP design flow is depicted in figure 4.5b. After the "Architectural Exploration", a "Code Generation + Aspects Weaving" activity has been included – it appears after the "Macro-Architecture with Functional mapping" box in figure 4.5. This code generation activity represents the forth step of AMoDE-RT design flow, and is performed by the GenERTiCA code generation tool. Repositories of mapping rules and DERAF aspects implementations were included in SEEP design flow (see figure 4.5c). Both mapping rules and aspects implementation rely on the platforms available in the platforms library (see figure 4.5d). GenERTiCA reuses elements of these repositories to perform the code generation

and aspects weaving processes. As result of this activity, source code files for both software and hardware are created. Further, these source code files are used in the compilation and synthesis step, and can also be tested using the SEEP test approach. Finally, real implementation of the distributed embedded real-time system being designed is obtained.
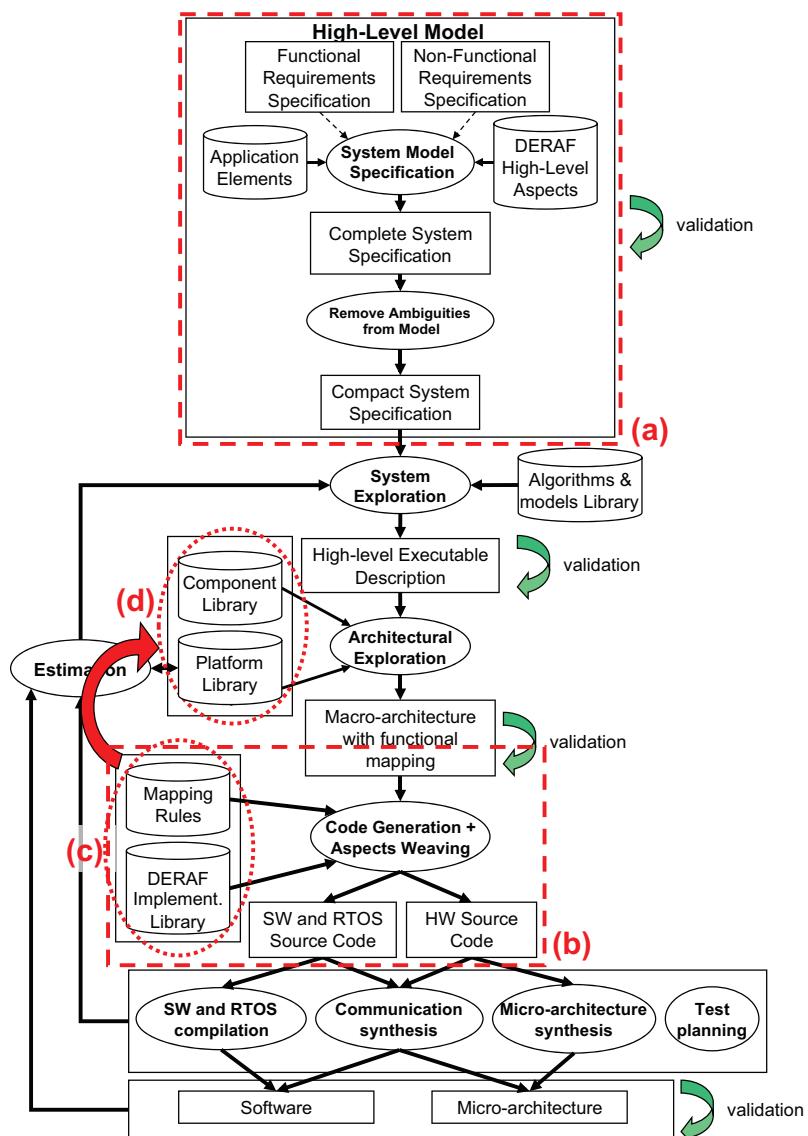
Figure 4.5.: Adaptations proposed to SEED design flow

Chapter 5

Specifying DERTS Using UML and Aspects

5.1. Introduction

This chapter discusses the distributed embedded real-time systems specification, in terms of modeling their structure, behavior, and non-functional requirements handling. The word "model" has very different meanings, which are related to context in which it is used. In the context of this work, models are simplified descriptions of computing elements that are being developed to provide the expected functionalities for a computing system, which must cope with application/domain requirements.

As stated in chapter 4, this work uses UML to specify models of distributed embedded real-time systems. However, as UML lacks specific syntax and/or sufficient semantics to describe embedded and real-time system domain concepts [127], a subset of the MARTE profile stereotypes is used to complement the system's features specification.

This chapter discusses guidelines to create UML models, which must be followed to allow the transformation of UML models into DERCS models for code generation purposes. In fact, in addition to suggestions on the diagrams selection, these modeling guidelines define some restrictions in modeling activities, allowing a correct system specification interpretation and transformation. Hence, the information on structure, behavior and non-functional requirements handling that is spread over different diagrams can be combined in the DERCS model. The discussion is divided in two parts: *(i)* specification of functional requirements handling, which approaches the use of some UML diagrams to specify the structure and behavior of systems; and *(ii)* specification of non-functional requirements handling, which explains how to specify AO concepts in UML models.

5.2. Functional Requirements Handling Elements

5.2.1. Introduction

The current version of UML specification, namely version 2.2 [92], supports 14 diagrams, whose brief description is given as follows:

- **Structural Diagrams** show a complete or partial view of system's structure. Available diagrams are:

  - *Class Diagram* shows system static structure in terms of classes and interfaces, their attributes and operations, as well as relationships among them;

  - *Composite Structure Diagram* depicts the system structure as hierarchically linked blocks. The internal structure of a structured classifier is shown as parts interconnected by ports, which are linked to interfaces;

  - *Component Diagram* provides a component view of system structure, i.e. it shows classes and their instances as components. Relationships are represented as provided/required interfaces;

  - *Deployment Diagram* describes the system architecture, by means of assigning objects onto execution platforms;

  - *Object Diagram* depicts the dynamic structure, i.e. class instances and their relationships, at a specific instant;

  - *Package Diagram* show the system as a set of packages that represent the logical grouping of classifiers; and

  - *Profile Diagram* is very similar to the class diagram, but instead of showing classes, this diagram depicts stereotypes.

- **Behavior Diagrams** depict complete or partial expected system behavior:

  - *Use Case Diagram* shows system's main functionalities in a very abstract fashion, as well as external actors that interact with the system;

  - *State Machine Diagram* displays hierarchical finite state machines, which are composed of composite states with one or more orthogonal states. These states machines are an extended version of Harel's statecharts [54];

  - *Activity Diagram* depicts system behavior in terms of activities and control flow. Activities use a Petri net-like semantic, i.e. its execution semantics is based on tokens. Additionally, there are special kinds of nodes that represent forks, joins, branches, and others;

  - *Interaction Diagrams* shows the communication among concurrent objects. There are four kinds:

∗ *Sequence Diagram* show multiple objects exchanging messages during their lifetime. Objects are represented as lifelines. Messages, which can be synchronous or asynchronous, are represented as horizontal lines from one lifeline to another one. There are also special constructions that represent loops, branches, concurrent messages exchanges, and others;

∗ *Communication Diagram* is similar to the sequence diagram, but instead of showing messages exchanged over time, it shows only the messages order without any special control flow element;

∗ *Interaction Overview Diagram* is a special kind of activity diagram, in which the nodes represent sequence diagrams instead of activities; and

∗ *Timing Diagram* represents discrete values or states changing while time passes. It is similar to continuous waveforms.

For more details on UML diagrams, interested readers are referred to [15] and [92].

According to Vanderperren, Mueller and Dehaene [127], information captured in UML models is often redundant and overlaps. Consequently, it is not necessary to use all of these diagrams to model a distributed embedded real-time system. Depending on the used design method and project goals, only some of them are useful. Moreover, some diagrams are more suitable (or clear) than others to specify system characteristics in a given target application domain. For instance, although activity, state and sequence diagrams are behavior diagrams, sequence diagrams show the behavior related to objects exchanging messages in a better way than activity or state diagrams allow, in spite of all of them could express such actions.

In this sense, AMoDE-RT modeling approach restricts UML usage to eight diagrams: *(i)* use case diagram; *(ii)* class diagram; *(iii)* sequence diagram; *(iv)* composite structure diagram; *(v)* deployment diagram; *(vi)* activity diagram; *(vii)* state diagram. However, only (i), (ii) and (iii) are mandatory, the other diagrams are optional. As mentioned, to allow information contained in these diagrams to be correctly extracted, and transformed into a DERCS model, a set of modeling guidelines for each diagram has been created, and must be followed. The following subsections discuss these guidelines, providing examples on how to create the supported diagrams.

5.2.2. Specification of System Expected Functionalities

As mentioned in chapter 4, the use case diagram is used to show the main functionalities of the distributed embedded real-time system being designed. Figure 5.1 depicts a sketch showing elements that are important in the AMoDE-RT approach. The "stick man" is the graphical representation of an actor, which represents a role played by a user, thing, or any other system that interacts with the system. Ellipses represent use cases, which indicate a set of actions performed by the system that yields an observable result to associated actors. In other words, use cases represent the main expected functionalities. It is important to highlight that crosscutting non-functional requirements, which affect system functionality, are also represented in the use

case diagram. Therefore, the information of the mapping table, which has been created in the requirements analysis (see chapter 4 and [37]), is used to decorate uses cases with stereotype indicating the first-level of non-functional requirements classification presented in figure 2.1 of section 2.3. These stereotypes are shown in figure 5.1 as «NFR_*» stereotypes. Hence, the traceability between requirements and model elements is reinforced.

Figure 5.1.: Graphical representation of system requirements

5.2.3. Specification of System Structure

5.2.3.1. Class Diagram

The main diagram to describe system structure is the class diagram. As expected, this diagram describes the static structure of the distributed embedded real-time system under design. It shows all classes that are responsible or related to the handling of functional requirements. Figure 5.2 shows an example of such diagram.

The proposed modeling approach assumes the common and wide use of this diagram, i.e. classes are depicted with their attributes and method signatures, as well as their relationships with other classes. Names of classes and attributes must be substantives to represent elements, which are relevant to the system or their characteristics. On the other hand, method names must be verbs to represent activities performed by objects of such classes. This naming convention must be also followed in interfaces specification. Furthermore, class names are used to fill the mapping table of RT-FRIDA in order to allow traceability between design elements and requirements specification.

One important feature to be observed in figure 5.2 is the use of stereotypes decorating certain elements. As mentioned in section 2.4.2, encapsulation restricts the access to class attributed by means of providing access methods. Hence, if a class needs to access an attribute of other class, the designer must include *get* and/or *set* methods for that purpose. Such methods are specified as in the class *MovementController*, i.e. a «getter» or «setter» stereotype must decorate, respectively, *get* and *set* methods. The attribute that is accessed by them is specified using the tagged value *Attribute*. Later, in the transformation of the UML model into a DERCS model, this information is used to automatically generate the corresponding behavior of such methods.

Taking into account the specification of concepts that are specific to the real-time domain, the UML model can represent active and passive objects. Active objects are resources that are able to perform actions concurrently with other active objects [24, 94]. Hence, the proposed approach assumes that active objects include their own thread of control. Classes that represent active objects are decorated with the stereotype «SchedulableResource» from the
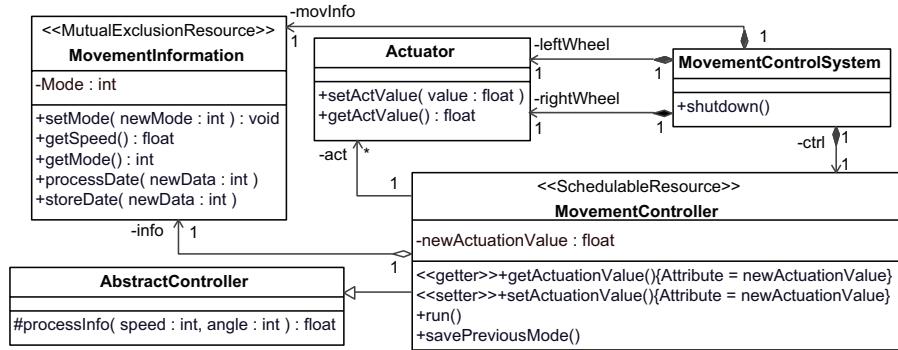
Figure 5.2.: Specification of the static structure

MARTE profile, as the class *MovementController* in figure 5.2. On the other hand, passive objects are resources that perform actions in response to stimuli of both active or passive objects, meaning that a passive object can eventually be accessed concurrently in the context of more than one active object execution flow (i.e. thread). If the concurrent access of such objects needs to be synchronized, classes that represent this kind of object must be annotated with the «MutualExclusionResource» stereotype of the MARTE profile. The *MovementInformation* class in figure 5.2 is an example of controlled shared passive object class. Classes without any stereotype or decorated with «Resource» stereotype are interpreted as passive objects with concurrent access synchronization.

Multiple inheritance is not allowed, i.e. one class can have only one parent class as specified in the generalization relationship between *MovementController* and *AbstractController*. If classes, which are children of different parent classes, need to share some features, an interface specifying these features should be created. Then, those classes should be linked to this interface by means of the interface realization relationship. Other three relationships are supported: *(i)* association; *(ii)* composition; and *(iii)* aggregation. In all of these relationship at least one association end must be *1* (one), i.e. only the following cardinalities are allowed: 1-to-1, 1-to-n, 1-to-n..*, and 1-to-* (where n is a positive natural number). Hence, many-to-many relationships cannot be specified. Additionally, at least one association end must have a name, and must be navigable, indicating that objects of the class represented by one association end can communicate with objects of the class represented by the navigable association end, e.g. in figure 5.2, *act* indicates that *MovementController*'s objects can interact with *Actuator*'s objects. During the transformation of the UML model, information of the navigable association end is used to create an attribute in the class of the other association end. These constraints were imposed in order to provide a precise interpretation of these relationships during the transformation of the UML model into the DERCS model (for details see section 6.2).

5.2.3.2. Composite Structure Diagram

Besides the specification of the static structure of the distributed embedded real-time system, designers can also specify the dynamic structure (or part of it) using the composite structure

diagram. In the context of this work, dynamic structure means the set of active and passive objects (i.e. class instances) that compose the system. As already mentioned, the use of composite structure diagram is not mandatory. The information on system objects can also be extracted from the sequence diagram by means of its lifelines. However, using this diagram is particularly interesting in the design of systems that do not create new class instances after the initialization phase, as usual in hard-real time control system, due to system constraint or application requirements. Thus, all objects required in the system execution phase could be specified in a single composite structure diagram. Figure 5.3 shows an example of composite structure diagram.

As can be seen, the whole system under design is represented as a class (*MovementControlSystem*) that encloses its set of active and passive objects, which are depicted as rectangles likewise classes in the class diagram, e.g. *ctrl* (active object), *leftWheel*, *rightWheel*, and *movInfo* (passive objects). The difference is the syntax for name specification in such classifiers: "*object_name : class_name [amount_of_objects]*". For objects that make up other objects, *object_name* must be the name of the navigable association end of the respective composite/aggregation relationship in the class diagram, e.g. instances of *Actuator* inside *MovementControlSystem* that refers to "leftWheel" and "rightWheel" compositions depicted in figure 5.2. Moreover, *amount_of_objects* defines the amount of instances of a given class, e.g. "*movInfo : MovementInformation [1]*" represents one object of the *MovementInformation* class. Two or more instances of the same class can be indicated using numbers inside brackets, or different rectangles (each one having a unique name) as demonstrated in "leftWheel" and "rightWheel" objects.

As one can see in figure 5.3, composite relationships are specified as solid lines rectangles, and aggregation relationships as dashed lines, e.g. "info" object inside *MovementController* that refers to "info" aggregation depicted in figure 5.2. On the other hand, normal associations are depicted as lines linking objects, e.g. "act[0]", "act[1]" that represent the association with the same name in figure 5.2.

Composite structure diagrams also depict MARTE stereotypes, which were used in the class diagram to refine classes' semantics according to concepts of real-time and embedded domains, facilitating the differentiation of active objects from passive ones. However, it is important to highlight that, as this information is represented as instances of UML meta-model elements, it does not need to be specified twice. Hence, there is no need to re-annotate objects with the same stereotypes used in the class diagram. During the transformation of the UML
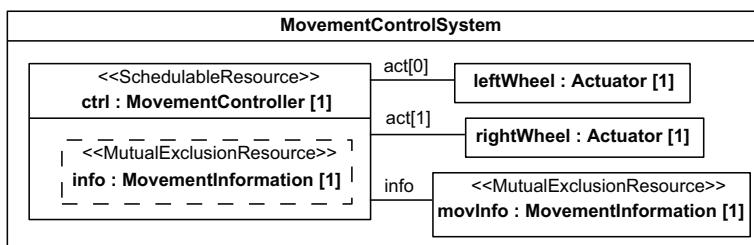
Figure 5.3.: Specification of the dynamic structure

model into the DERCS model, such information is obtained from the meta-model elements of the class diagram.

5.2.3.3. Deployment Diagram

Other structural diagram used in AMoDE-RT modeling approach is the deployment diagram, which specifies on which computing device (e.g. devices with processors and memory, ASIC or FPGA hardware devices, or hybrid devices) objects execute their behavior, as well as in which kind of platform they are implemented. Figure 5.4 shows an example of such diagram. Different computing devices are specified as *nodes* in deployment diagrams, while different platforms as *artifacts* placed inside these nodes, e.g. *Node_1* and *Node_2* are computing nodes, and *Java* and *C++* are platforms representing node's implementation. Objects are specified as instances linked with artifacts through *manifest relationships*. Therefore, objects are deployed in the node (or computing device) that owns the artifact associated to them, e.g. *ctrl* is an active object implemented as software using a Java platform; and *leftWheel* and *rightWheel* are implemented in C++.

Objects that are linked with artifacts in the same node represent *local objects*. On the other hand, objects linked with artifacts residing in different nodes are considered *remote objects*. At modeling level, the semantics of the communication among local objects is the same as remote objects, i.e. one object sends a message to another, waiting or not for a response. The same is true for objects modelled as implemented as software and/or hardware. The differentiation among messages sent to local or remote objects is done during code generation phase by Gen-ERTiCA that evaluates nodes on which source and target objects (related to sending message actions) are deployed. However, it is important to highlight that there are some non-functional requirements related to distributed objects communication. They are handled by aspects of DE-RAF, allowing designers to focus on concepts of the target application domain instead of on implementation issues, as explained in the section 5.3.
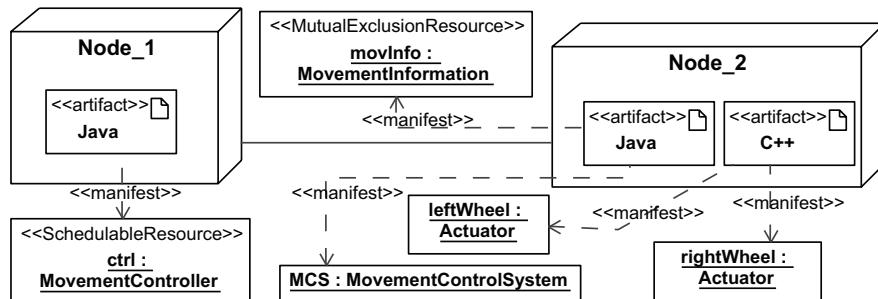
Figure 5.4.: Specification of objects deployment

5.2.4. System Behavior Specification

5.2.4.1. Sequence Diagram

System behavior is specified as a combination of different UML 2.2 behavior diagrams, i.e. different diagrams of the same type, such as different sequence diagrams, as well as different kinds of diagrams, such as a combination of different sequence diagrams with state and/or activity diagrams. In AMoDE-RT modeling approach, the sequence diagram plays the main role for describing the behavior of a distributed embedded real-time system. It was chosen due to its intuitive syntax to depict objects communicating with each other (i.e. message exchanges), as well as its capability of controlling the execution flow within the diagram. As explained in the following paragraphs, a set of reserved words has been created to represent other kinds of actions, such as value assignment to variables or object attributes, evaluation of expressions, and objects state changes. Consequently, it is possible to specify most actions a distributed embedded real-time system needs to perform as its behavior.

There are some modeling rules that must be followed in order to allow the combination of the behavior information spread into different sequence diagrams. Figure 5.5 depicts an example of a valid behavior specification using a sequence diagram. All lifelines represent active or passive objects, whose name must be either the name of an attribute or a variable.
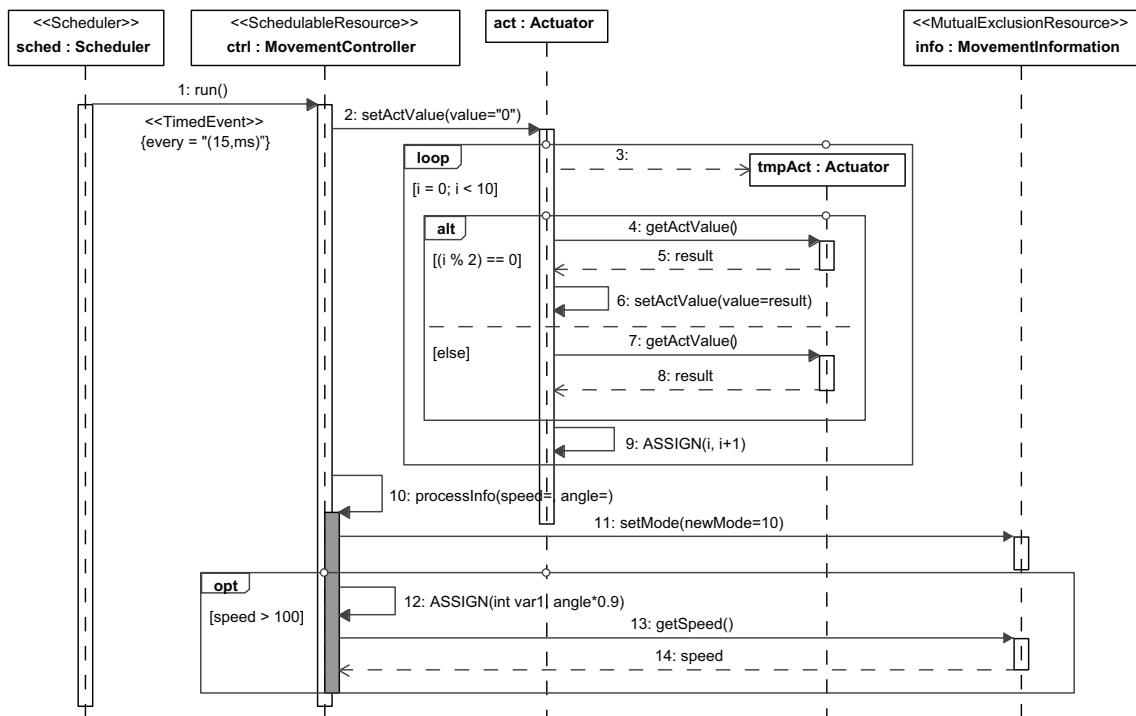
Figure 5.5.: Specification of the behavior in terms of actions performed by objects

Therefore, an object that sends a message to other object must be related to it through either a relationship between both class (specified in the class diagram), or the creation of this object (as a local variable) within the context of its methods' behavior. For example, the third lifeline (from left to right) represents the association relationship between *MovementController* and *Actuator* classes, whose association end has been named as "act" (see class diagram depicted in figure 5.2). As mentioned in the previous section, this association end represents an attribute with the same name in the *MovementController*, allowing the communication between objects of this class and objects of the *Actuator* class.

However, there is an exception of this naming rule: the first lifeline does not need to represent any specific object. Hence, it can have any name (including the "*" wildcard character, as depicted in figure 5.6), and does not require any associated object. This is possible because the transformation algorithm usually interprets messages departing from the first lifeline as the beginning of an actions' execution flow. For example, in figure 5.5, "message 1" indicates the beginning of the behavior of *MovementController.run()* method. The same is valid for messages "1" and "6" in figure 5.6 that represent start of, respectively, *MovementController.processInfo()* and *MovementInformation.setMode()* methods' behavior Although the relaxed naming rule, there is a pitfall in the specification of messages sent to the first lifeline: if the message is sent from other lifeline to the first one, e.g. "message 5" in figure 5.6, it means a sending message action, and hence, the first lifeline must represent an object to allow the correct interpretation of this action. The transformation algorithm will try to find the method associated with the message (in this case, the method *invalid()*), but will succeed due to the lack of associated object. In other words, the situation presented in figure 5.6 represents an invalid message flow due to the lack of any object and/or class associated to the first lifeline, i.e. the name "*" indicates that the lifeline can represent any object.

Furthermore, considering the execution flow depicted in sequence diagrams, messages must be specified according a nesting constraint: it is expected that the next message departs either from the lifeline that has received the previous message, or from one of the lifelines that had sent any previous message. In this sense, the diagram presented in figure 5.5 represents a valid flow, because all messages have been specified according such constraint. Considering this sequence diagram, let's assume a "imaginary" execution flow that passes a control token from the
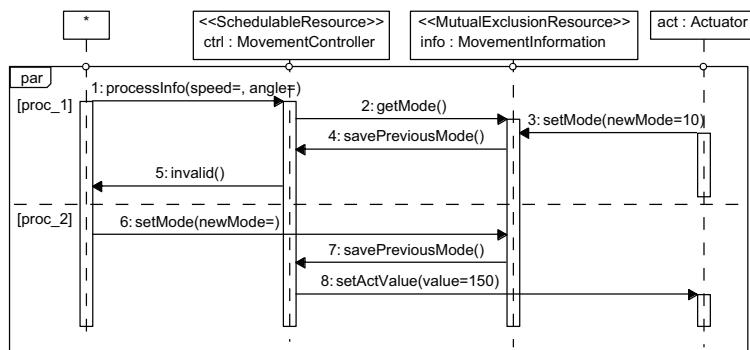
Figure 5.6.: Invalid behavior specification using sequence diagram

sched : Scheduler lifeline to *ctrl : MovementController* lifeline in "message 1". This token is forwarded from this lifeline to *act : Actuator* ("message 2"), which, in turn, passes it to the *tmpAct : Actuator* lifeline ("message 3"). After the instantiation of *Actuator* class, the token is returned to the *act : Actuator* lifeline. Following, the token is passed again to *tmpAct : Actuator* in either "message 4" or "message 7", returning back to the previous lifeline (i.e. *act : Actuator*) after the execution of the behavior specified within the corresponding alternative in the combined fragment. Finally, after the execution of the assignment action specified in "message 9" (explained in the following paragraphs), the token is returned to *ctrl : MovementController* lifeline. Further, "message 10" specifies a recursive message, which indicates the beginning of *MovementController.processInfo()* method behavior (the darker part in the lifeline). Thus, all messages sent from this lifeline part belong to *processInfo()*'s behavior. Likewise explained previously, the execution token flows among lifelines respecting the message nesting order. In this example, it is important to note that almost all messages (e.g. messages 1, 2, 4, 6, 7, 10, 11, and 14) have been specified as synchronous call operation messages, meaning that the execution of the calling method's behavior must be held until the called method returns the execution control token.

On the other hand, figure 5.6 shows an invalid behavior specification using sequence diagrams. It describes a broken execution flow due to "message 3". Once reading this diagram according to the mentioned messages nesting constraint, it is expected that "message 3" departs from *info : MovementInformation*, *ctrl : MovementController*, or "*" lifelines rather than from *act : Actuator*. Hence, the specified execution flow violates the expected messages nesting order. Nevertheless, if "message 3" and "message 5" (as explained in previous paragraphs) are removed, the sequence diagram depicted in figure 5.6 becomes a valid behavior specification, due to the compliance with the mentioned constraints.

As mentioned, sequence diagrams are key diagrams to system behavior specification. They are intended to depict objects interactions in terms of messages exchanged among them. However, behavior of distributed embedded real-time systems cannot be fully specified using only sending messages actions. There are other equally important actions: *(i)* values assignment to object attributes or variables; *(ii)* evaluation (or execution) of mathematical or boolean expression; *(iii)* explicit changes in the object state[1]; *(iv)* array-related actions, such as insert/remove elements, get/set element values, or get the array length. The problem is that there is no available construction in sequence diagrams to specify such actions. Thus a set of reserved words was created to specify these actions. Table 5.1 presents the created reserved words, which are used in the specification of message names to represent the mentioned actions. In order to allow the correct interpretation of such names during the transformation phase, the syntax depicted in table 5.1 must be followed.

Other important feature in behavior specification is the control of execution flow using constructions such as branches or loops. Since the approval of UML 2.0 superstructure specification, sequence diagrams allow the specification of control constructions, named *combined*

---

[1]The term "state" in the context of OO can interpreted as two complementary definitions: *(i)* values of object's attributes at a given instant; and/or *(ii)* a explicit state, which is generally specified in a state machine. In this work, "object state", "state of the object", or simple "state" refers to (ii).

Table 5.1.: Reserved words for actions specification

Syntax	Description
`ASSIGN([data type] target, value)`	Represent an assignment action of a value to a variable or object attribute, where: **data type** is optional, and indicates the variable data type; **target** specifies the name of the target variable or attribute, in which the value is stored. The naming constraint (i.e. lifeline naming) must be respected; **value** is the value to be assigned.
`EXPRESSION([[data type] target,] expr)`	Represent the evaluation (or execution) action of a mathematical or boolean expression, where: **data type** is optional, and indicates the variable data type; **target** is optional, and specifies the name of a variable or attribute in which the expression result is stored. The naming constraint (i.e. lifeline naming) must be respected; **expr** is the expression to be evaluated.
`MODIFY_STATE(newState)`	Represent the action that changes explicitly the object state, where: **newState** represent the new state in which the object will be after the execution of this action.
`INSERT_ELEMENT(target, [pos,] value)`	Represent the action of inserting a value in a given array, which can be a variable or attribute: **target** specifies the array name. The naming constraint (i.e. lifeline naming) must be respected; **pos** is optional, and specifies the array position after which the value is inserted. If it is omitted, the element is added at array's end; **value** is the value to be inserted.
`REMOVE_ELEMENT(target, pos)`	Represent the action of removing a value from a given array, which can be a variable or attribute. **target** specifies the array name. The naming constraint (i.e. lifeline naming) must be respected; **pos** specifies the array position that must be removed.
`ARRAY_LENGTH(target)`	Represent the action of reading the length of a given array, which can be a variable or attribute. **target** specifies the array name. The naming constraint (i.e. lifeline naming) must be respected.

fragments, which operate on an interaction fragment. Thus it is possible to specify alternative or optional execution of interaction fragments, parallel execution of interaction fragments, repetition of interaction fragments execution, and others. The proposed modeling approach allows using a subset of all combined fragments kinds:

- **Alternatives** (`alt`) designates different choices for execution of actions sequences. To use this construction, designers must specify at least two alternatives. Each alternative sequence is guarded by a boolean expression, which must hold in order to deviate the execution flow to the alternative interaction fragment. If an alternative does not have a guard expression, the actions sequence of this alternative is executed if and only if guard conditions of all other alternatives do not hold. In the case of two or more guard conditions hold, the action sequence specified within the first alternative (considering the alternatives order depicted in the sequence diagram) is executed; other action sequences are ignored. In other words, the actions specified in an `alt` fragment are not concurrent. Figure 5.5 depicts an example of such combined fragment;
- **Option** (`opt`) defines an optional sequence of action that are executed whether the guard expression holds. It is similar to `alt` combined fragment, but it specifies only a single alternative. Therefore `opt` combined fragment must always have a guard condition. Figure 5.5 shows an example of this combined fragment;
- **Parallel** (`par`) represents parallel execution of action sequences, which execute concurrently and independently from the other parallel parts. As sequences of actions could terminate in different instants, designers must not specify any action after a `par` combined fragment. Figure 5.6 also shows an example of this combined fragment;
- **Loop** combined fragment (`loop`) represents the repetition of the actions sequence execution. The actions sequence is repeated while the guard expression holds. Loops can also have a fixed number of repetitions, which is specified using the syntax "`var = minNumber; var < maxNumber`", where `var` is the name of the repetition counter; `minNumber` is the initialization value for the counter; and `maxNumber` is the number of repetitions. Figure 5.5 shows an example of a loop combine fragment that has a fixed number of repetitions.

To conclude the discussion on behavior specification using sequence diagrams, it is important to consider the specification of real-time features. Similar to the specification of the dynamic structure, stereotypes that decorate classes of active and passive objects are also depicted in sequence diagrams, due to the availability of this information in instances of UML meta-model element previously specified. In this sense, MARTE stereotypes do not need to be specified twice for the description of same system element, e.g. «SchedulableResource», «MutualExclusionResource» and «Scheduler» stereotypes have already been used in the class diagram, and could be depicted in sequence diagram elements of figure 5.5.

An important view of system behavior is the specification of active objects' concurrent behavior that need to be periodically executed at a certain frequency. This kind of active object must have only one periodic behavior, i.e. only one method can have its behavior triggered periodically. Thus, designers must create at least one sequence diagram for each periodic active object, showing the activation pattern for its periodic behavior. An example of such diagram is

presented in figure 5.5. This diagram must always start with a message sent from the scheduler object to the active object, indicating the start of the periodic behavior execution. Such message must be decorated with MARTE's «TimedEvent» stereotype. The time interval between two consecutive executions of the behavior must be specified using the `every` tag, whose value must follow MARTE's *Value Specification Language* (VSL) [94] syntax: "`(n, timeUnit)`", where `n` is a number and `timeUnit` is the time unit. For instance, in figure 5.5, "message 1" is annotated as "`every = (15, ms)`", indicating that the interval between two consecutive executions of this behavior is 15 milliseconds.

5.2.4.2. Activity Diagram

Another diagram to specify system behavior that is supported by AMoDE-RT modeling approach is the activity diagram. Although optional in the proposed modeling approach, this kind of diagram may be used in combination with sequence diagrams to specify the overall view of system behavior in terms of runtime phases.

Distributed embedded real-time system runtime can be divided in three distinct phases: *(i)* initialization; *(ii)* execution; and *(iii)* shutdown. The activity diagram is used to specify these phases as shown in figure 5.7. Each activity is associated with a sequence diagram, which details actions performed in the activity. AMoDE-RT modeling approach uses sequence diagrams rather than textual action languages [79] to specify complex behavior and/or actions sequence, due to graphical specifications are considered easier to understand than textual descriptions. Besides, diagrams are more intuitive and technology independent, facilitating the information exchange among different design teams. Additionally, textual languages are considered very similar to conventional programming languages (e.g. Java or C/C++), and hence, they are not the most suitable form to specify system behavior in high-level models.

System initialization and shutdown activities describe all actions that need to be performed, respectively, before and after the core functionalities provide by the system. As it can be seen in figure 5.7, after the initialization activity the execution flow is split in several concurrent activities. Usually, these activities indicate periodic behaviors executed by active objects. If this overview of system behavior is provided, it is expected that the amount of sequence diagrams provided in the UML model is at least equal to the amount of activities specified in the activity
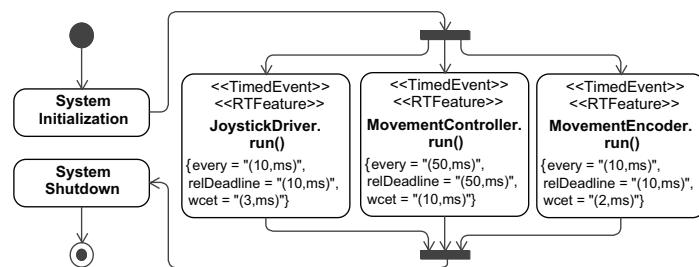
Figure 5.7.: System behavior overview specified using activity diagram

diagram.

Activities' timing information are specified by MARTE stereotypes and tagged values, indicating the activation period (`every` tag of «`TimedEvent`»), deadline, and WCET (respectively, `relDeadline` and `wcet` tags of «`RealTimeFeature`») for activities execution. Figure 5.7 depicts three activities that are annotated with the mentioned MARTE stereotypes and tagged values, e.g. *JoystickDriver.run()*, *MovementController.run()* and *MovementEncoder.run()*. As mentioned, this information is specified once, i.e. one or more instances of UML meta-model elements related to system elements, and reused in many different diagrams that depict the same elements, such as class diagram, sequence diagram, activity diagram, and/or any other diagram supported by AMoDE-RT modeling approach.

It is important to highlight that runtime phases can also be specified using state diagrams, in which each state represents a phase. Although sequence diagrams could also be linked to states (in a state diagram) to indicate the behavior executed when the system is in a given state, the activity diagram is considered more suitable for depicting such viewpoint due to its sequential execution flow semantics, and also to the clearer visualization of concurrent behavior. Furthermore, AMoDE-RT modeling approach binds state diagrams to classes to specify explicitly states in which object can be during different system runtime. Consequently, using state diagrams to specify system-level states may cause problems in the model's information interpretation and transformation.

5.2.4.3. State Diagram

Although sequence and activity diagram are considered sufficiently complete to specify the distributed embedded real-time systems behavior, there are application domains in which this view of system behavior is not the most suitable one. Usually, in reactive systems, behavior is usually specified in term of events and actions performed in response to these events. To support this specification viewpoint, AMoDE-RT modeling approach uses state diagrams, which are associated with classes specified in the class diagram. In this sense, state diagrams represent both explicit states, in which a class instance (i.e. object) can be at a given instant, and the behavior performed while it stays in these states. It is important to highlight that one state diagram is associated with only one class, and vice-versa. Figure 5.8 shows an example of state diagram constructions, which are allowed in the proposed modeling approach.

State diagrams show behavior as states, transitions, and events. Actions are associated with states indicating their execution in three moments: *(i)* on entering the state; *(ii)* during the stay on that state; and *(iii)* on exiting the state. Similarly to activity diagrams, actions must be specified using sequence diagrams, which are associated to states in the mentioned moments. Moreover, state transitions are fired by events, which may be internal or external. Internal events are logical events from the application domain, e.g. the detection of a certain kind of threat in a surveillance system. Other example of internal event is the instant in which a method call action is performed, e.g. transitions from "Reading Joystick Position" to "Reading Movement Information", or from "Processing Data" to "Storing Processed Data" as depicted in figure 5.8. External events indicate remarkable occurrences, which happened in the external environment
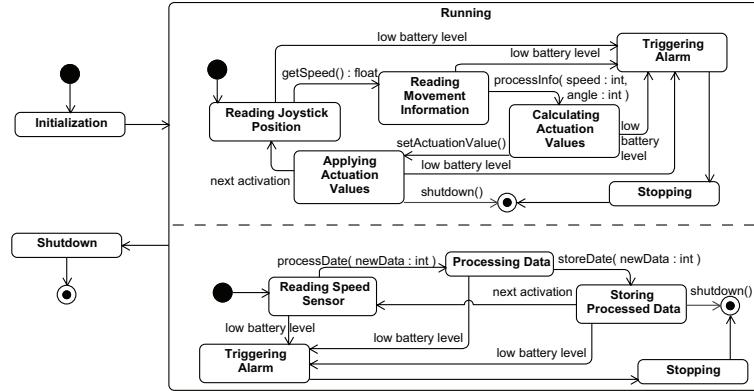
Figure 5.8.: Behavior of classes specified using state diagrams

in which the system is embedded, e.g. signals issued by presence sensors. Such events are specified as substantives in the transitions name, e.g. "low battery level" transitions in figure 5.8. Furthermore, deep and shallow history pseudo-states [92] are not supported by AMoDE-RT modeling approach. Consequently, the only way to start/finish a state machine is passing through, respectively, initial and final states.

Orthogonal states are also supported. Designers can specify one sub-state machine in each orthogonal region, meaning that once entering in a orthogonal state, objects can be in several *AND-states* [92] at the same time, e.g. "Running" state in figure 5.8. A transition from these orthogonal states to any other state is possible only if all sub-state machines arrive in their final state. Hence, if the exit event (which triggers the exit transition from an orthogonal state) happens, and one or more sub-state machine are not in their final state, the event is passed to all sub-state machines which remain active. On the other hand, if there is no active sub-state machine, the exit transition from the orthogonal state happens. For instance, let's assume that the state machine depicted in figure 5.8 is in "Running" orthogonal state. The first sub-state machine (upper orthogonal region) is in the final state, and the other one is in "Storing Processed Data". When a "shutdown" event occurs, it is passed only to the second sub-state machine, causing the transition from "Storing Processed Data" to the final state, and thus, from "Running" to "Shutdown". It is also possible to specify a transition from orthogonal states without triggering events, indicating that, once all sub-state machine arrive the final state, the exit transition from the orthogonal state is triggered automatically.

5.3. Non-Functional Requirements Handling Elements

5.3.1. Introduction

As already mentioned, AMoDE-RT modeling approach uses aspects to specify non-functional requirements handling in UML models. Given that UML does not have constructions to depict AO concepts in its diagrams, a lightweight extension to UML in terms of a UML profile has

been proposed. The concepts represented by the created profile stereotypes are based on the AO concepts presented in section 2.4.3, and also the AO conceptual model proposed by Schauerhuber et al. [109] (see figure 5.9). That model provides more general AO concepts compared to the ones discussed in section 3.3, which are simply adaptations of specific AO languages concepts. From this model, the following stereotypes have been derived:

- «Aspect»: used in ACOD (see section 5.3.3) to specify which DERAF aspects (see next section) are selected in the current design. It must be used to decorate UML's class meta-model element, extending its semantics to represent aspects;
- «BehavioralAdaptation»: used in ACOD to decorate aspects "methods" to specify behavioral adaptations performed in system functional elements. It must be used to decorate UML's operation meta-model element extending its semantics to represent behavioral adaptations;
- «StructuralAdaptation»: used in ACOD to specify structural adaptations performed in system functional elements. It is also used to decorate UML's operation meta-model element;
- «Crosscut»: used in ACOD to decorate associations between classes and aspects. In ACOD, *crosscut associations* do not represent by themselves a relationship between aspects and classes. Instead, they represent information that is inserted by aspects in affected classes. Thus, this stereotype extends UML's association relationship using the
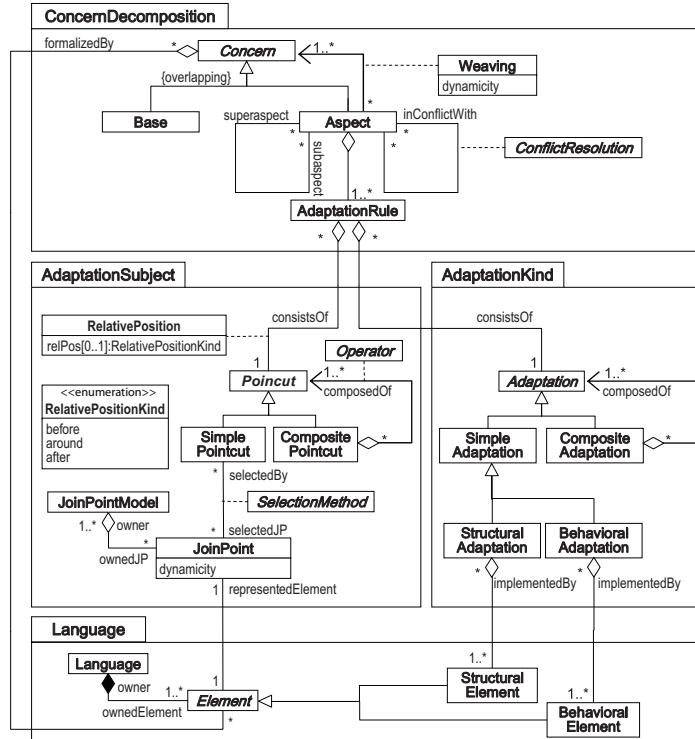
Figure 5.9.: Conceptual AO model [109]

mentioned semantics;

- «Pointcut»: used in ACOD to decorate aspects "methods" to specify the link between join points and aspect adaptations. It must be used to decorate UML's operation meta-model element extending its semantics to represent pointcuts. In the decorated methods, parameters represent pointcut information as explained in section 5.3.3;
- «JoinPoint»: used in JPPDs (see section 5.3.4) to decorate the point in which aspects can perform adaptations. It only indicates which (kind of) model element is affected by aspect adaptations, instead of modifying their semantics;
- «JPDD»: used to decorate sequence or class diagrams indicating that they represent join point selections rather than a functional diagram.

The following sections discuss how to use AO concepts to deal with non-functional requirements in the distributed embedded real-time systems design. Firstly, an aspects framework to handle the mentioned requirements is presented. These aspects are used in the *Aspects Cross-cutting Overview Diagram* (ACOD), which is proposed in this work and discussed in section 5.3.3. Finally, join point specification is presented in section 5.3.4, which discusses the use of *Join Point Designation Diagrams* (JPDD).

5.3.2. Distributed Embedded Real-time Aspects Framework

5.3.2.1. Overview

To provide modularized handling for non-functional requirements, a framework of aspects named *Distributed Embedded Real-time Aspects Framework* (DERAF) has been created. In this sense, DERAF aspects encapsulate in a single element all issues related to the handling of non-functional requirements.

Based on the AO conceptual model presented in the previous section, DERAF is an extensible high-level aspects framework to be used in earlier design, as well as implementation phases. The main idea is to provide aspects that enhance the modeled system by means of adding specific behavior and structure to specify non-functional requirements handling. These "new" behavior and structure are independent from any specific implementation technology.

More specifically, DERAF was intended to be used together with UML and MARTE profile. To achieve this goal, details about how to implement aspect adaptations have been abstracted, i.e. designers choose which aspects are used to specify the non-functional requirements handling based on aspects adaptations high-level semantics. Thus, in UML model, DERAF aspects are used as "black boxes". In addition, designers must indicate which functional elements are affected by the selected aspects using join points specification, as discussed in section 5.3.4

Considering the non-functional requirements presented in section 2.3, each requirement can be handled by one or more DERAF aspects. Figure 5.10 shows an overview of the DERAF aspects. As it can be seen, DERAF provides six packages grouping aspect based on their goals. The following sub-sections provide a brief discuss on the semantics of each available aspect. In addition, a more comprehensive description of DERAF aspects is presented in appendix A.

79
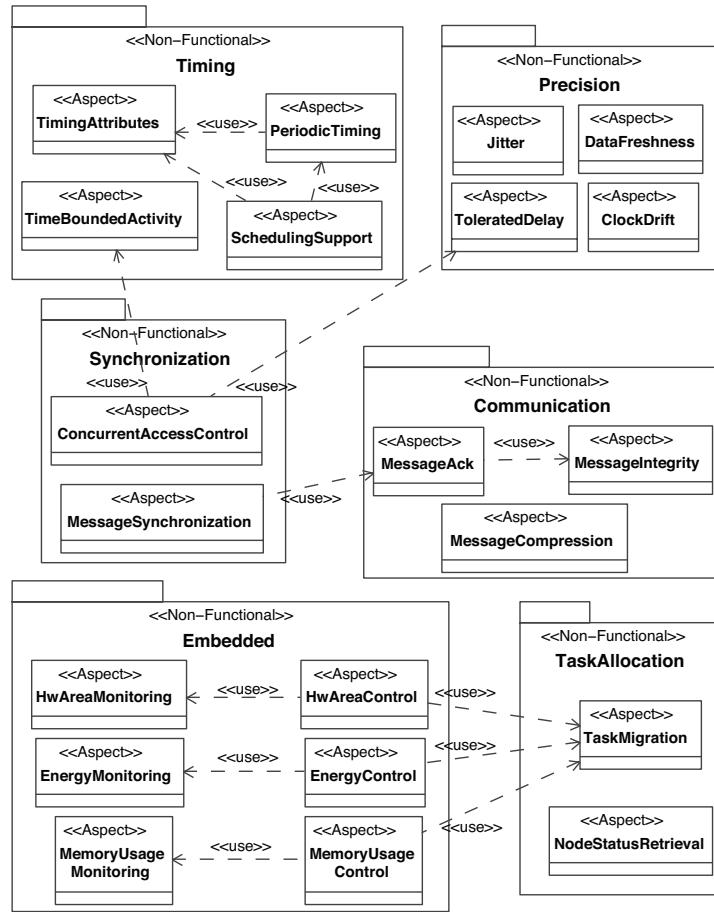
Figure 5.10.: All aspects provided by DERAF

5.3.2.2. Timing Package

This package contains aspects to handle time-related requirements, such as deadlines for activities execution, WCET information, periodic tasks activation, and others.

TimingAttributes aspect is responsible to deal with active objects characteristics such deadline, priority, WCET, and absolute time instants on which their behavior must start and finish the execution. Attributes representing the mentioned characteristics are inserted in the affected active object classes, as well as methods and behavior to initialize and handle these attributes. As mentioned, the handling of these timing issues is delegated to the target platform that must provide support to this aspect semantics.

PeriodicTiming aspect provides means to trigger periodically an active object behavior execution. Thus, besides adding an attribute indicating the execution frequency in the affected active object class, this aspect must also enclose the affected behavior with a repetition mechanism, whose execution is controlled according the information stored in the mentioned new attribute. In other words, this aspect is used to deal with the handling of periodic active objects

(or threads).

SchedulingSupport aspect inserts a scheduler object in the affected computing nodes. This object is responsible to control active objects execution, indicating instants at which they must start performing their behavior.

TimeBoundedActivity aspect controls the execution time duration of an activity or action by counting the time elapsed since the start time instant. If maximum allowed duration is surpassed, this aspect provides means to abort the affected activity/action execution. Examples of this aspect use are: to restrict the maximum time a shared resource can be in exclusive access mode, or the maximal time amount an active object can wait for the reply of a remote objects.

5.3.2.3. Precision Package

Precision in meeting time requirements are handled by the aspects of this package, which concentrates efforts in features such as the maximum tolerated delay in starting activities, variance in activities timeliness, information's validity duration, or the deviation of local clock reference compared with the global one.

Jitter aspect measures the accuracy variance in activities performed by the system. This aspect provides means to measure the time before (or after) an observed activity happen, storing this information (the history must provide information of at least one time sample) to calculate the variance among the observed time instants. This aspect can be used, for example, to calculate the jitter in an periodic active object activation or execution, or to compute the time variance of a periodic message sending.

ToleratedDelay aspect controls the maximum tolerated latency to the actual start of a given system activity. Thus, the time between the command and the execution of the observed activity must be measured and calculated. If the observed duration is greater than the maximum allowed latency, this aspect provides means to handle this exception.

ClockDrift aspect controls the clock deviation between the local time source and the global one. Assuming that the target platform provides means to allow clock synchronization, this aspect uses the global clock as reference to calculate the local clock deviation. Thus, designers must specify time instants (or system events, e.g. the starting of an behavior execution) at which the local clock must be compared with the global clock reference in order to check if there is a difference between the two measured values.

DataFreshness aspect is responsible to deal with the validity duration (or utility) of different system information [22]. For that, this aspect associates timestamps to affected data by adding new attributes to representing such information, as well as inserting behavior to control these data use. In other words, each time a controlled data needs to be read, its validity must be checked and, if it is out of validity, a corrective behavior must be performed, e.g. wait until the date to be updated, read data directly from its source, decrease the frequency at which periodic behaviors (which read the controlled data) are executed. Analogously, each time a controlled data is updated, its validity duration must also be updated.

5.3.2.4. Synchronization Package

Synchronization and the concurrent access control to shared resources requirements are dealt by this package's aspects.

ConcurrentAccessControl aspect provides means to control the concurrent access to objects, which share their attributes information with other objects. The access to object's different elements can be controlled: *(i)* the object itself; *(ii)* their attributes; and/or *(iii)* their methods. Therefore, depending on the controlled element, one or more arbiters (i.e. concurrency controller instances) are created. Every time an (active or passive) object needs to access controlled shared elements, it must request the access to them (i.e. request a lock) that are granted or not by the arbiter. Depending on the arbiter implementation (e.g. mutex, semaphore, monitors), and also to the number of objects that are accessing the shared resource at the moment, the access request can be authorized or not. Similarly, after the use of the shared resource, the object that had the access permission must notify the arbiter, indicating that it is leaving the shared resource and does not need to use it anymore.

MessageSynchronization aspect deals with holding behaviors execution until the arrival of an acknowledgement message (or a reply message) indicating that the (remote) object has received the message sent. It provides a waiting mechanism that could be implemented as either *(i)* a busy wait, i.e. a loop that waits until the acknowledgement message arrives; or *(ii)* using the system scheduler, which preempts the execution of the current active object, marking it as blocked, and thus, opening room for other active objects execution. Later, when the expected acknowledgement message arrives, the blocked active object is marked as ready to execute, and its execution is resumed following the scheduler's decision.

5.3.2.5. Communication Package

This package provides aspects to deal with objects communication in terms of messages sending. The first intention was to cover the communication between objects that are located in computing devices that are physically separated. However, depending on application requirements, this package's aspects can also be used for specifying the communication of objects located in the same computing device.

MessageAck aspect provides an acknowledgment mechanism to notify reception of a message to its sender. In this sense, this aspect affects both sides of a message exchange: sender and destination objects. On one side, the sender object sends a messages and waits for an acknowledgement of message reception. On the other side, the receiver objects needs to send an acknowledgement message after each received message. *MessageAck* is related with *MessageSynchronization* aspect.

MessageIntegrity aspect is responsible for handling messages integrity by providing checking information within a message. Similarly to *MessageAck*, this aspect also affects both message's sender and receiver objects. Sender objects must generate integrity checking information, appending it in the message to be sent, while receiver objects must generate checking informa-

tion from the received message, comparing it with the information that came with the received message. The acknowledgment mechanism must be notified whether the checking information match or not.

MessageCompression aspect is in charge to compress/decompress messages in order to improve bandwidth utilization. Like the other aspects of this package, this aspect affects both message's sender and receiver objects. At sender side, the message is compressed using a compression algorithm, while at receiver side the message is decompressed using the same algorithm.

5.3.2.6. TaskAllocation Package

Aspects provided by this package handle non-functional requirements related to objects distribution on different computing devices at runtime. These aspects are typically related to distributed system nodes that are physically separated.

NodeStatusRetrieval aspect includes a mechanism to gather information on the system dynamic characteristics, such as processing load, message sending and reception rates, and if the computing device is running.

TaskMigration aspect adds a migration mechanism to move active objects from one computing device to another one. Therefore, active objects can migrate from one node to another, as well as from software to hardware, or vice-versa [2].

5.3.2.7. Embedded Package

Non-functional requirements related to physical resources availability, which are very common concerns in embedded systems design, are handle by this package's aspects. Energy consumption, memory usage, and hardware reconfigurable area can be cited as examples of such concerns. Basically, the available aspects are concerned in monitoring and controlling the mentioned physical resources. Thus, depending on the physical resource being controlled, the control policy, and platform capabilities, different actions can be performed by these aspects as, for instance: *(i)* depending on the system requirements and runtime state, to remove objects related to non-critical activities; *(ii)* active objects migration; *(iii)* to loosen timing constraints; *(iv)* to decrease processor operation frequency; *(v)* to turn off unnecessary hardware components; It is important to highlight that this aspects are dependent on target platform capabilities, meaning that the platform must provide means to monitor and control system physical resources.

HwAreaMonitoring aspect is related to systems that use reconfigurable hardware devices, such as FPGAs. It provides a mechanism to monitor the reconfigurable area by which the

---

[2]Objects migration between software and hardware (at runtime) is usually known as "reconfiguration". However, in embedded systems domain, "reconfiguration" usually means to upload a bitstream into a FPGA device. Thus, in order to avoid misunderstandings, this text uses the term "reconfiguration" to refer to the later, while reconfiguration between software and hardware are called "migration"

remaining reconfigurable area (in terms of configurable logic blocks) is (re)calculated at each reconfiguration command.

HwAreaControl aspect controls the hardware reconfigurable device usage by adding an arbiter to allow or deny every reconfiguration based on the information of this package monitoring aspects.

EnergyMonitoring aspect relies on the target platform to provide a mechanism to monitor energy consumed by system activities. This mechanism must measure the remaining energy level before the observed activities start, and after their completion. Further, it calculates the amount of energy that was consumed by these activities.

EnergyControl aspect provides an object that uses information provided by the monitoring aspects to control the energy consumption. To accomplish such goal, this object could perform the actions mentioned in the beginning of this subsection.

MemoryUsageMonitoring aspect is similar to the other two monitoring aspects but it is related to software rather than to hardware. It provides a mechanism that must calculate the overall memory usage of a computing device at every object allocation/deallocation.

MemoryControl aspect uses the information provided by *MemoryUsageMonitoring* and *HwAreaMonitoring* aspects to control the memory allocation requests for objects allocation following an adopted memory control policy.

5.3.2.8. Discussion

As one can conclude from the DERAF aspects description, some aspects deal with the same non-functional requirements, such as *MessageSynchronization*, *MessageAck*, *MessageIntegrity*, and *MessageCompression* that handle objects message sending; or *MemoryUsageMonitoring* and *MemoryUsageControl* aspects that handle memory non-functional requirements. There are aspects that access resources provided by other aspects, such as the *SchedulingSupport* aspect that uses resources provided by *TimingAttributes* and *PeriodicTiming* aspects adaptations; or control-related aspects that use information provided by monitoring aspects to control the use of embedded system physical resources.

However, it is important to note the conflicting nature of some aspects adaptations. The behavior that handles a non-functional requirement can affect the handling behavior of other non-functional requirements, e.g. the energy controller (inserted by the *EnergyControl* aspect) decides to migrate an active object from software to hardware to save energy, but the hardware area controller (inserted by the *HwAreaControl*) aborts this migration activity due to insufficient available reconfigurable area. These conflicts must be solved at design time. RT-FRIDA provides tools to enable requirements conflicts resolution by assigning an importance value to each requirement [37]. Hence, this information must be taken into account in aspects implementation, so that, problems related to requirements conflicts can be minimized.

The key factor that motivated DERAF creation was to provide a set of high-level aspects,

which offer well-defined adaptations semantics, to be used in UML models. However, to allow its practical use, it is also necessary to provide the realization of these aspects in terms of application or platform source code, or platform configuration code. Additionally, aspects implementation must follow the pre-defined semantics on "how" and "where" aspects adaptation can be applied. Keeping the coherency between high-level adaptations semantics and their implementation, it is possible to increase the reuse of aspects implementation previously created, reducing the effort necessary to handle non-functional requirements in further projects.

In this sense, it is important to highlight that, although timing information is specified by *Timing Package*'s aspects within ACOD context, specific details on how to handle timing are delegated to the target platform, in order to keep DERAF aspects platform-independent. In other words, the implementation of aspects adaptations defines exactly how to deal with each timing feature using constructions available in the target platform, and respecting DERAF's high-level semantics as specified in UML model. An example of such aspect implementation can be seen in figure 6.9 of the Chapter 6 that shows how periodic timing is handled using constructions available in an RT-Java based platform. Other timing issues (e.g. deadline, WCET, etc.) are handled in a similar form: DERAF defines high-level semantics to these non-functional requirements handling, which are further implemented using constructions and services available in the target platform. Consequently, the exact handling of timing features depends on the target platform.

It should be stated that there are two kinds of possible implementations to aspect adaptations: *(i)* those that adapt application code; and *(ii)* adaptations that tailor platform source code, or produce platform configuration files. The former represents modification in the application code itself, e.g. *PeriodicTiming* aspect's *LoopMechanism* adaptation, *DataFreshness* aspect's *VerifyFreshness* adaptation, or *ConcurrentAccessControl* aspect's *AcquireAccess* adaptation. On the other hand, the other kind enables or disables a feature in the target implementation platform, e.g. *MessageAck*, *MessageIntegrity*, and *MessageCompression* aspects adaptations. The most important thing is to note that, to provide the expected aspects adaptations according to the pre-defined semantics, the target platform must offer the required services. It is not the intention of the described DERAF semantics to provide a definitive solution for the handling of each non-functional requirements addressed by its aspects, they are suggestions to address with these requirements handling.

Finally, it is worth mentioning that the aspects set provided by DERAF does not cover all non-functional requirements present in the distributed embedded systems domain. Currently, non-functional requirements such as fault tolerance are not addressed by DERAF aspects. However, it is an extensible framework, meaning that it is not difficult to include other aspects. It is only necessary to follow two rules:

1. High-level semantics, indicating "how" and "where" aspects adaptations are applied, must be pre-defined; and
2. To respect aspects' pre-defined semantics in their implementation using services and constructions of a given target platform.

5.3.3. Aspects Crosscutting Overview Diagram

As UML does not provide any meta-model element or graphical construction to represent aspects, this works has proposed the *Aspects Crosscutting Overview Diagram* (ACOD), which is an extended version of the standard class diagram. ACOD is based on the concepts presented by Stein et al. [120] and Schauerhuber et al. [109], and shows DERAF aspects affecting or not functional elements. There are two ACOD versions with different levels of detail: *(i)* overview ACOD presents all aspects affecting classes without depicting details about aspects' information; and *(ii)* detailed ACOD depicts all aspects specified in the UML model along with their adaptations and pointcuts, and all classes that receive new information from aspects. Detailed ACOD is the main information source for aspects specification. Thus, designers must always create this diagram to specify AO-related elements. Overview ACOD can be generated automatically by evaluating all pointcuts specified in the detailed ACOD (using the join points indicated in these pointcuts) to discover which aspects affect which classes. Hence, overview ACOD is considered an informative diagram rather than an aspects specification.

AO-related stereotypes proposed in this work are used to annotate UML meta-model elements depicted in ACOD to represent AO concepts as presented in figure 5.11. *Aspects* are represented as classes decorated with the «Aspect» stereotype. Aspect's *behavioral adaptations* are specified as methods decorated with the «BehavioralAdaptation» stereotype, while *structural adaptations* as methods decorated with the «Structural- Adaptation» stereotype. Similarly, *pointcuts* are specified as methods decorated with «Pointcut» stereotype. As pointcuts specify the link between join points selection and aspect adaptations, this information is specified as method parameters as follows:

- The first parameter represents the join point name, and indicates which model elements are selected by these JPDD, e.g. *JPDD_InfoAttributeRead* in *DataFreshness*;
- The second parameter indicates which adaptations are performed in selected model elements, e.g. *Deadline* and *SetTimingAttributes* in *TimingAttributes*. If more than one adaptation of the same aspect modify the same join point, adaptations names can be com-
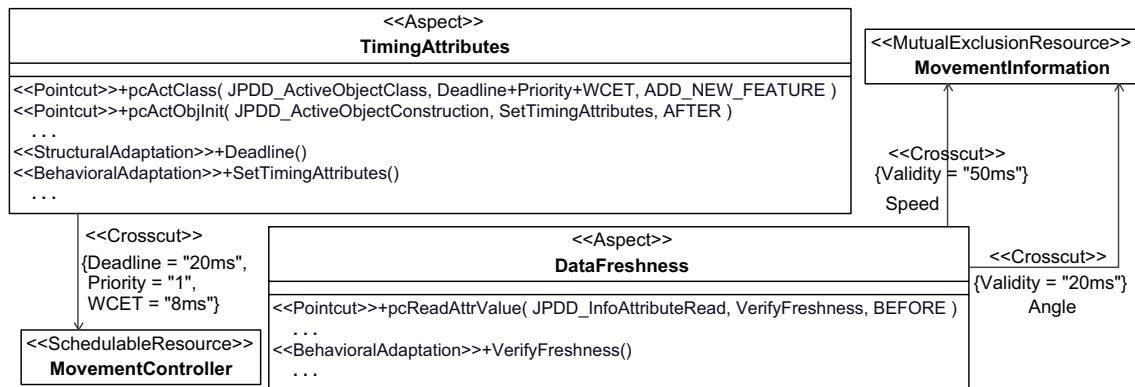
Figure 5.11.: Aspects specification using ACOD

bined in the same pointcut, using the "+" character to separate each adaptation name, e.g. "Deadline+Priority+WCET";

- The third parameter specifies position (related to the join point) at which associated adaptations are applied. For structural adaptations, this parameter is optional. The following relative positions are supported:

  – `BEFORE`: used for behavioral adaptations to indicate that they are applied *before* join point occurrences, e.g. *pcReadAttrValue* in *DataFreshness*;
  – `AFTER`: used for behavioral adaptations to indicate that they are applied *after* join point occurrences, e.g. *pcActObjInit* in *TimingAttributes*;
  – `AROUND`: used for behavioral adaptations to indicate that they *enclose* (i.e. adaptations are done before and after) join point occurrences;
  – `REPLACE`: used for behavioral or structural adaptations to indicate that join point occurrences *are replaced* by these adaptations;
  – `MODIFY_STRUCTURE`: used for structural adaptations to indicate that they *modify* elements selected by join points;
  – `ADD_NEW_FEATURE`: used for structural adaptations to indicate that new features (e.g. attributes) are added in affected elements, e.g. *pcActClass* in *TimingAttributes*.

An important ACOD feature must be highlighted: associations between aspects and classes, which are decorated with the «`Crosscut`» stereotype. If an aspect structural adaptation inserts new attributes in classes, the affected classes must be included in ACOD specification. For each affected class, an unilateral one-to-one association decorated with the «`Crosscut`» stereotype must be created from the aspect to the affected class. Values for the new attributes are specified as tagged values in the *crosscut association* as depicted in figure 5.11. As one can see, *TimingAttributes* inserts three attributes (i.e. *Deadline*, *Priority*, and *WCET*) into *MovementController*. The *crosscut association* specifies that *Deadline* must be initialized with 20 ms, *Priority* with 1, and *WCET* with 8 ms. Similarly, *DataFreshness* aspect adds a new attribute associated with *MovementInformation*'s *Speed* and *Angle* attributes. A different value to each attribute is specified in *crosscut associations*. However, it is important to highlight that *crosscut associations* are not "real" associations between aspects and classes in terms of UML association semantics. Instead, they are interpreted as informative relationships that do not produce any meta-model element in the associated elements.

Considering timing requirements handled by DERAF aspects, the form to specify such information is demonstrated in figure 5.11. For instance, deadlines are handled by *TimingAttributes*, and thus, they are specified as *crosscut associations* between this aspect and the affected active object classes. As DERAF defines high-level adaptation semantics (see Appendix A), the exact handling of deadlines is delegated to the target platform, which implements the pre-defined semantics of this aspect. In this sense, the UML model specifies that active objects behaviors are constrained by deadlines, which are dealt by *TimingAttributes*. However, there is no definition if this handling must be performed using timers, special APIs, or other programming abstractions. The target platform is responsible for this handling. Consequently, aspects adaptations must be map to constructions in such platform, defining the mentioned non-functional requirement handling. The same is valid to the other aspects that deal with timing issues, e.g. *PeriodicTiming*, *DataFreshness*, and other.

Additional examples of ACOD specification are provided in the case studies presented in Chapter 7, and also in Appendix B.

5.3.4. Join Points: Selecting Model Elements Affected by Aspects

Although the aspects specification is an important part of non-functional handling specification, equally important is the specification of which model elements are affected by aspects adaptations. Therefore, join points selection are specified using a subset of *Join Point Designation Diagrams* (JPDD) [121]. The main reason for using JPDD is the possibility to specify join points graphically, which facilitates the understanding about which element kind is selected. Additionally, JPDDs are considered more suitable to use in UML models than join points textual descriptions.

JPDD can capture model elements based on three different models: *(i)* control flow; *(ii)* data flow; and *(iii)* state. The first model allows elements selection based on the execution control flow depicted in sequence or activity diagrams, e.g. a JPDD selects actions performed in the behavior of a given method "a", which is called inside the behavior of a method "b". The second model allows the elements selection based on data passed from one method to other one, e.g. a JPDD selects a method behavior that has received a string starting with "s" character as parameter. The last model allows elements selection base on their explicit state described in a state diagram, e.g. a JPDD selects all objects that are in "state_A". As one can infer, elements selection can be performed statically or dynamically. The first model allows both dynamic and static selection; the other two only dynamic. Additionally, JPDDs can select elements (e.g. classes, attributes, and others) based their names rather than using the mentioned models.

AMoDE-RT modeling approach supports both control flow JPDDs, and elements selection based on naming patterns. However, there is a constraint: control flow JPDDs cannot specify multiple calling levels, i.e. only actions performed in the method behavior context can be selected. Furthermore, to specify which elements should be selected by JPDDs, this work follows the naming patterns presented in Stein et al. [121], as shown in table 5.2.

Elements selection is performed during the transformation of UML model into a DERCS model. JPDDs are evaluated using elements' static information, and hence, dynamic evaluation of JPDD is not supported. In this sense, the following model elements can be selected: *(i)* classes; *(ii)* attributes; *(iii)* methods; *(iv)* nodes; *(v)* sending message actions; *(vi)* object creation actions; *(vii)* object destruction actions; *(viii)* method return actions; and *(ix)* methods behavior. Structural elements (i–iv) are selected using the sort of JPDDs presented in figure 5.12; behavioral elements (v–ix) are selected by JPDDs depicted in figure 5.13.

Sequence diagrams and class diagrams are decorated with the «JPDD» stereotype to indicate that they are, in fact, the specification of join point selection rather than system specification. Additionally, elements selected by the join point are decorated with the «JoinPoint» stereotype, which defines some tags to identify precisely which elements are considered. The available tags are: *(i)* Classes; *(ii)* Object; *(iii)* Node; *(iv)* MessageDefinition; and *(v)* Behavior.

Table 5.2.: Naming pattern for elements selection in JPDD

Naming Pattern	Description
`*`	Indicate that any name matches with the pattern
`*Ending`	Indicate that any name that ends with the character sequence "Ending" matches with the pattern
`Start*`	Indicate that any name that starts with the character sequence "Start" matches with the pattern
`mthName '(' [parName [, parName]*] ')' ':' retTypeName`	This special naming pattern is used in sending message actions selection, where *mthName* is the message name pattern as described above; *parName* is the message parameter name pattern. It is an optional part. If method parameters should not be considered, the string ".." must be used. Otherwise, parameters naming pattern follows the above mentioned patterns; *retTypeName* indicates the method return type name, as described above.
`[local.   | remote.] objName ':'   className`	This special naming pattern is used in objects, classes or nodes selection. They are used to name lifelines in sequence diagram JPDD, where *local* or *remote* is a reserved word to indicate if the element communicates with, respectively, local or remote elements; *objName* is the object name, as described above; *className* is the class name, as described above.

To illustrate the specification of model elements selection, a brief discussion on which elements are selected by JPDDs depicted in figures 5.12 and 5.13 is provided. Structural elements are selected by JPDDs presented in figure 5.12:

- JPDDs in figures 5.12a and 5.12e select classes. The former selects all active object classes, while the later selects all classes whose objects send messages to remote objects;
- Figure 5.12b depicts the selection of all attributes, whose name starts with "sensor", from all passive objects that are accessed exclusively;
- The selection of all system nodes is shown in figure 5.12c;
- Figure 5.12d depicts a JPDD that selects all methods, whose name start with "set";
- All constructors of all active object classes are selected by the JPDD presented in figure 5.12f.

Regarding the selection of behavioral elements, the following elements are gathered by JPDDs depicted in figure 5.13:

- All actions related to messages whose name starts with "set", which are sent from any object to any passive object, are selected by the JPDD presented in figure 5.13a.
- JPDD presented in figure 5.13b selects all actions representing messages, whose name starts with "get", and are sent to any remote object.
- Figure 5.13c presents a JPDD that selects the periodic behavior («Join Point» stereotype's Behavior tag) executed by any active object. Thus, this JPDD selects all mes-
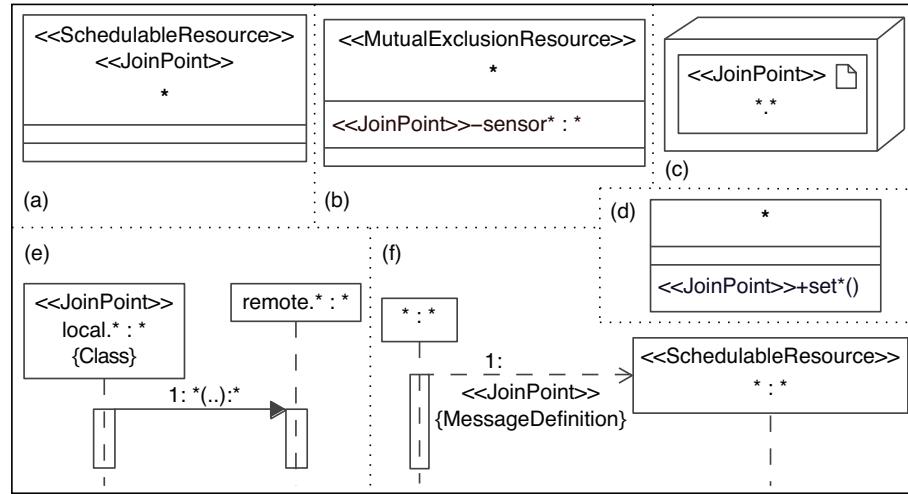
Figure 5.12.: JPDD for structural elements selection
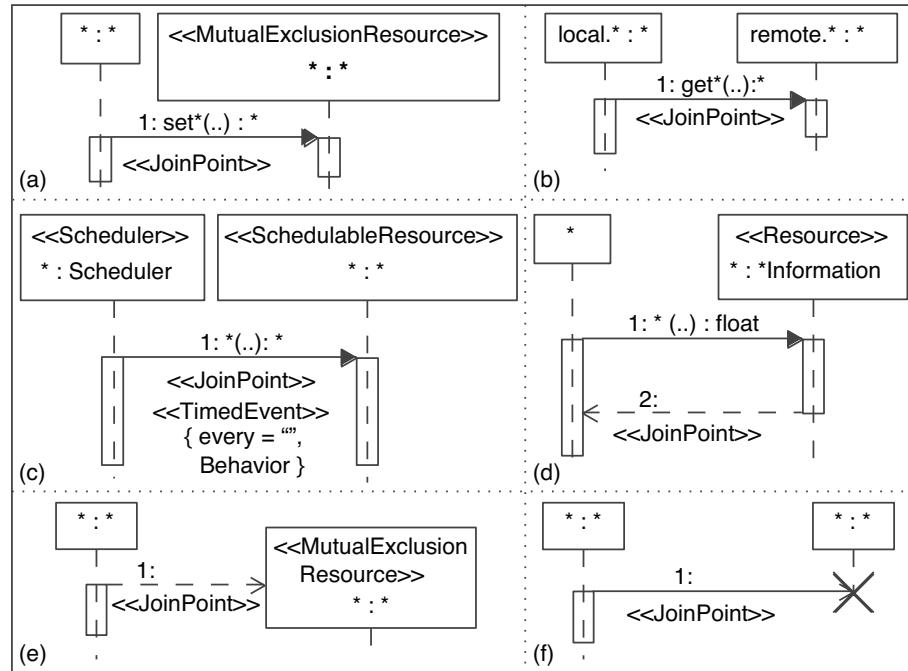
Figure 5.13.: JPDD for behavioral elements selection

sages sent from the scheduler to any active object, that are, in addition, are annotated with «TimedEvent» stereotype and `every` tagged value.

- All return actions from methods of all passive object classes whose name ends with "Information" are selected by the JPDD shown in figure 5.13d.
- Figure 5.13e depicts the selection of all actions that create passive objects.
- JPDD presented in figure 5.13f selects all actions that destroy any object.

5.4. Final Remarks

During the study to identify which diagrams are important to specify structure and behavior of distributed embedded real-time systems, the proposed modeling approach selects UML diagrams that have been considered more intuitive, in order to facilitate the interpretation of design intentions performed by different design teams. In this sense, modeling guidelines are defined (and must be followed) to enable system specification to be automatically extracted from UML models. Other goal is to use UML diagrams in its standard form, i.e. using the standardized graphical syntax without proposing any graphical extension. Hence, off-the-shelf UML modeling tools can be used to support AMoDE-RT modeling approach without any constraints.

In AMoDE-RT, the class diagram is the most important diagram to describe system's static structure. It provides all structural information for system objects. The activity diagram has been chosen to depict an overall view of system runtime phases, in which active objects' concurrent behavior can be seen with their timing constraints expressed using standard MARTE stereotypes. However, in AMoDE-RT modeling approach, the most important behavior diagram is the sequence diagram. Due to its intuitive graphical syntax, sequence diagram has been chosen to specify actions sequence execution instead of a textual actions language. In this sense, elements in activity and state diagrams are liked with sequence diagrams to indicate the behavior executed by classes associated these elements. Considering the subset of MARTE stereotypes used in AMoDE-RT modeling guidelines, table 5.3 shows all stereotypes that can be used to annotate UML diagrams' elements, along with a brief description of their usage.

Also with regard to the specification of non-functional requirements handling , this work does not propose any new UML graphical extensions to model AO concepts. As mentioned, the intention is to use UML standard diagrams, thus a lightweight extension in terms of a UML profile has been proposed. Commercial off-the-shelf modeling tools are able to specify both ACOD and JPDD diagrams. JPDD has been chosen due to its expressiveness to specify join points selection, and also to the lack of a consolidated standard for AO concepts modeling.

Finally, although DERAF aspects' pre-defined high-level semantics define non-functional requirements handling, aspects realization must be provided in further design phases using services provided by available platforms. Therefore, aspects adaptations must be implemented using constructions of a target platform, or reused from previous projects that had implemented these adaptations using the target platform. In this sense, platform support is crucial to allow the DERAF effective use. Although this is not the focus of this thesis, some DERAF aspects implementations are provided using platforms available in our research group. Empirically, we

Table 5.3.: Summary of MARTE stereotypes used in AMoDE-RT

MARTE stereotypes	UML elements	Usage
«SchedulableResource»	Class	Specifies active object classes
«Resource» or «MutualExclusionResource»	Class	Specifies passive object classes
«Scheduler»	Class	Specifies the scheduler of a computing node
«TimedEvent»	Operation, Message, Activity	Specifies behaviors that are triggered periodically
every		Indicates the time interval between two consecutive executions of the behavior
«RTFeature»	Activity	Specifies behaviors' timing characteristics
relDeadline		Indicates behaviors' relative deadline
wcet		Indicates behaviors' WCET

believe that all aspects are fully, or at least partially, implementable using platforms that are already available in industry or academy.

Chapter 6

Tool Support for the Proposed Approach

6.1. Introduction

Tool support is essential to improve a design method usage effectiveness. In MDE approaches, one important tool is the code generation one, which uses the produced models to create source code respecting system specification. Therefore, *Generation of Embedded Real-Time Code based on Aspects* (GenERTiCA) has been created to support the AMoDE-RT approach. As stated in chapter 4, GenERTiCA is a script-based code generation tool, which executes small scripts to produce code fragments that are merged to produce the expected source code files for a target platform. Figure 6.1 shows the three main features involved in the code generation approach implemented by GenERTiCA: *(i)* transformation of system specification from UML to DERCS model, which is more suitable than UML for code generation purposes; *(ii)* model-to-text mapping rules definition; and *(iii)* code generation and aspects weaving algorithm.

This chapter discusses GenERTiCA's features. Firstly, it will discuss the DERCS meta-model, and heuristics created to transform UML model elements into DERCS elements. Next, mapping rules specification is discussed, focusing on mapping rules file structure and scripts organization using the XML format. Additionally, mapping rules scripts are detailed. Finally, the algorithm used to produce code from model elements is discussed. Aspects weaving performed by GenERTiCA is also detailed.

6.2. A Platform Independent Model for Code Generation

UML is a complex modeling language, which allows system elements specification using different views. In its version 2.2, UML provides thirteen different diagrams: six for system structure specification, and seven for behavior specification. Although these different diagrams facilitate
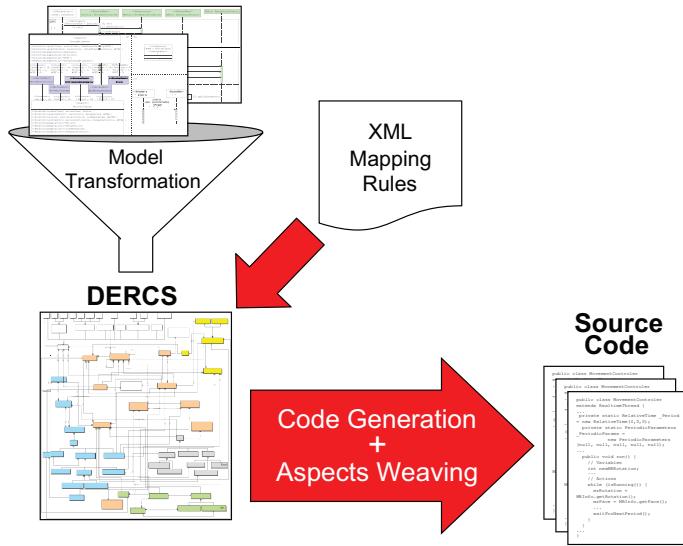
Figure 6.1.: GenERTiCA mains features overview

visualization of system features from different viewpoints, this diversity of diagrams may lead to an ambiguous specification, due to information overlapping and duplication. Furthermore, UML is considered a semi-formal language, due to the lack of formal semantics to define the interpretation of system specification information, which is usually spread in several diagrams. Consequently, computers cannot perform UML models automatic interpretation (or execution).

A candidate solution to these problems is to transform UML diagrams elements, which represent embedded system information, into elements of other model, providing the same abstraction level without binding system specification to any implementation platform. However, this transformation makes sense only if this other model can provide a more concise meta-model compared to the UML one. Hence, this works proposes the use of the so-called *Distributed Embedded Real-Time Compact Specification* (DERCS), aiming at providing a PIM suitable for code generation purposes. DERCS is based on a subset of both the UML meta-model and MARTE profile meta-model, and also the AO conceptual model [109], providing a model that includes OO and AO concepts. The main intention is to precisely and unambiguously represent the information on distributed embedded real-time system's structure, behavior and non-functional requirements handling.

DERCS meta-model defines a distributed embedded real-time system as set of communicating objects, which interact among each other to provide the expected system functionality. In other words, objects are the key elements in system specification, representing hardware and software components. System behavior is represented by both actions performed sequentially by objects, and objects interaction. There are two object types: active and passive. Active objects are autonomous entities that have their own flow of control (i.e. a particular thread), allowing concurrent actions to be executed in parallel with other active objects. Additionally, these objects can be compared to concurrent processes in multitask operating system, having characteristics, such as activation patterns (e.g. periodic, aperiodic, or sporadic), deadlines,

WCET, priorities, and others. On the other hand, passive objects are those that execute actions sequentially in response to messages received from other objects (active or passive). Passive objects can be seen as entities that provide useful information and services to active objects.

Likewise the UML meta-model, DERCS meta-model represents system structure elements using OO concepts. Figure 6.2 shows DERCS meta-model structural elements. An *object* is a *class* instance, which, in turn, represents elements structure in terms of attributes and methods. *Attributes* hold values to represent objects' state at a given instant, while *methods* represent messages signatures that can be received from other objects. Both can be inherited from the so-called superclasses. Concerning the *data types*, DERCS defines almost the same data types as UML. It is important to mention that classes can also define a set of explicit possible *states*, in which their instances can be during their lifetime. Class explicit states are represented by attributes whose data type is *StateDataType*. Each state is associated with *transitions*, representing state changes. Further, more than one incoming/outgoing transitions can be associated to the state. Concluding the discussion about system structure representation, as one can infer, there is no major difference from DERCS structural meta-model elements to UML ones.

System behavior is represented by elements presented in figure 6.3. A *behavior* consists of behavioral elements, which can be either *actions* or other behaviors, and *local variables*. Basically, behaviors can be triggered in response to messages received from other objects (i.e. a behavior is associated to a method body of a given class). In other words, behaviors can be seen
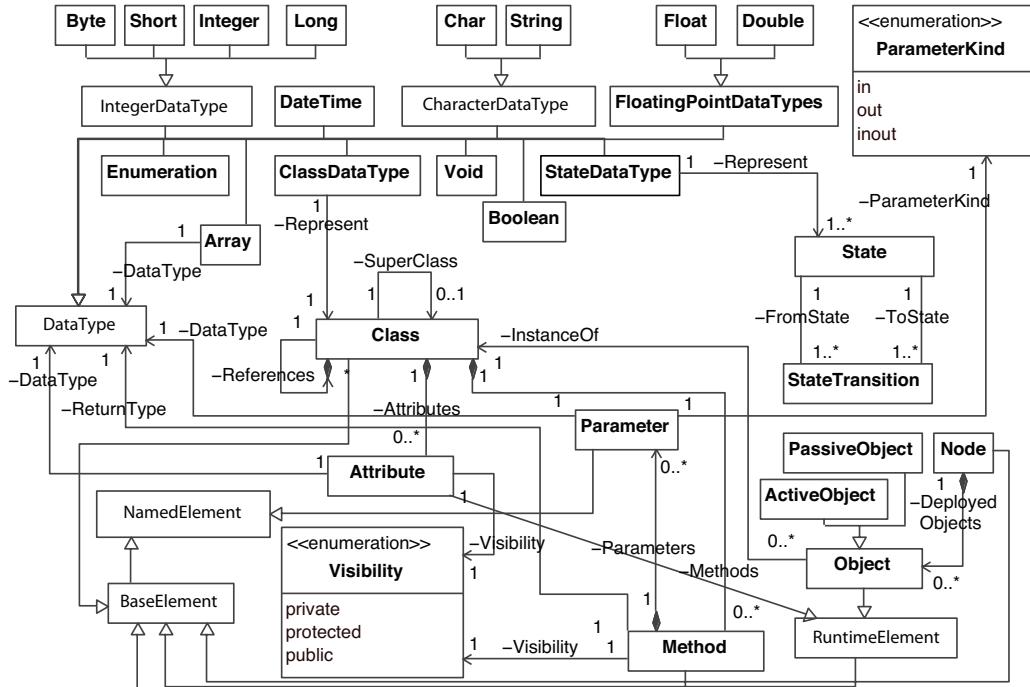
Figure 6.2.: DERCS meta-model: structural elements

as the execution of actions sequences that start in response to method calls. DERCS defines its actions model based on the UML meta-model, providing platform independent actions as follows:

- `AssignmentAction` represents a value assignment to an attribute or local variable;
- `ExpressionAction` represents mathematical or boolean expressions evaluation;
- `SendMessageAction` indicates the action of an object sending a message to another object;
- `ModifyStateAction` represents the action of changing object's explicit state;
- `CreateObjectAction` indicates an object creation, while `DestroyObjectAction` an object destruction;
- `ReturnAction` represents a method value return action;
- `InsertArrayAction` represent the action of inserting a new element in an array, while `RemoveArrayAction` represents the opposite, i.e. the action of removing an element from an array. In addition, `ArrayLengthAction` represents the array size information retrieval.

Moreover, DERCS defines that behaviors have pre- and post-conditions that must hold, respectively, before and after actions sequence execution. *Pre-conditions* indicate that behaviors start their actions execution only if the boolean expression holds. Likewise, *post-conditions* indicate that behaviors repeat actions sequence execution until the boolean expression become valid.

Behaviors can also be executed in response to events occurrences. In fact, an event is associated with an object that contains methods capable of handling this event. Thus, when an event occurs, it triggers a sending message action to one of the associated object's methods. DERCS defines two event types: internal and external. *Internal events* are detectable occurrences during system runtime, such as *messages sending/reception*, *entering* in an explicit state, and *exiting* from an explicit state. On the other hand, *external events* are occurrences which happened in the external environment in which the system is embedded. Furthermore, events specify *sequential* and *parallel* triggers. The former indicates that the triggered object's behavior must hold the execution until the executing behavior finishes. On the other hand, the later indicates that the associated object's behavior can start its execution concurrently with other executing behaviors.

Other important DERCS feature is objects distribution. Objects reside in computing devices (e.g. devices with general purpose processors and memory, or dedicates hardware devices), upon which their behavior is performed. `Node` element represents such devices, whose implementation represents either software or hardware platforms. Additionally, it can represent physically separated computing devices, making objects to be considered as local or remote objects (depending on the objects/device reference). However, it is important to highlight that, independently of objects implementation (i.e. software or hardware) or their deployment (i.e. local or remote), behavior semantics proposed in DERCS remain the same. Thus, actions are executed (behavior) in response to either messages received from other objects, or event occurrences. In other words, messages exchanged by objects implemented as software or hardware and/or deployed in different devices, at DERCS abstraction level, have the same semantics. The implementation of these different message exchange types is defined in the chosen target
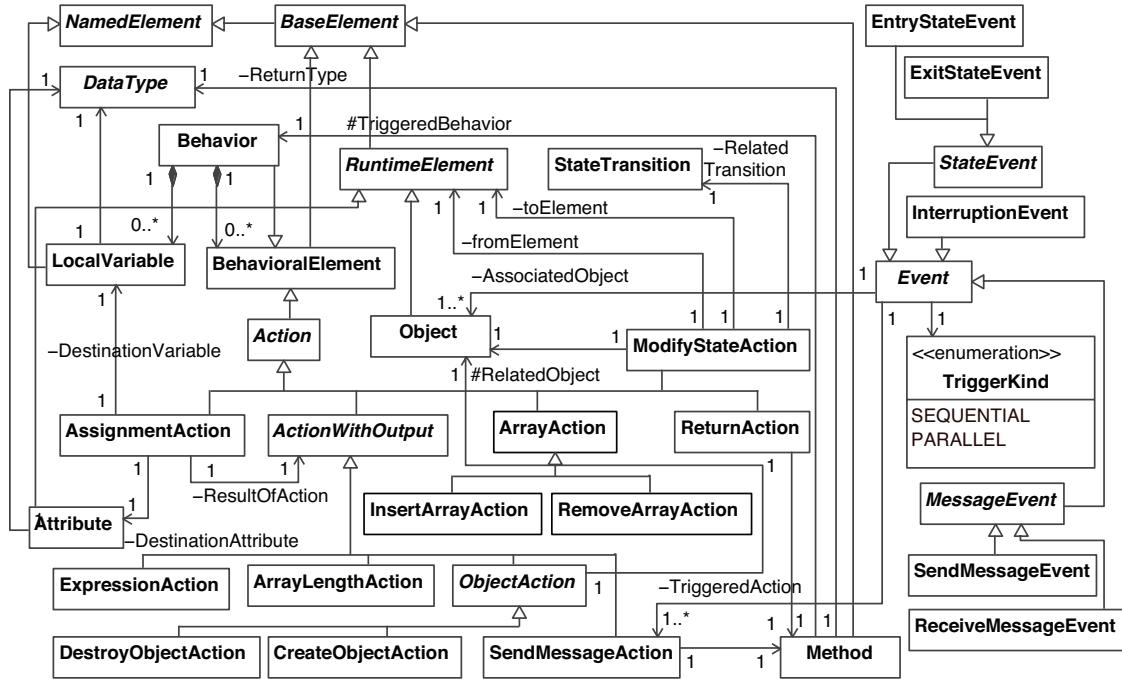
Figure 6.3.: DERCS meta-model: behavioral elements

platform that will realize the distributed embedded real-time system elements.

The most noticeable difference between DERCS and UML meta-models is the capability of representing AO concepts, as depicted in figure 6.4. *Aspects* consist of an *aspect adaptations* set, which contain both *structural* or *behavioral adaptations*, and a set of *pointcuts*. In essence, DERCS represents DERAF aspects specified in the ACOD (as discussed section 5.3.3); similarly, aspect adaptations represent specified adaptations provided by the selected DERAF aspects. In addition, the *crosscutting information* specified in ACOD's crosscut associations (between aspects and classes) is represented by Crosscutting and CrosscuttingInformation elements. *Pointcuts* represent the link between AspectAdaptations and Joinpoint elements, indicating the *relative position* in which adaptations must be performed. RelativePosition enumeration specifies one of the following options:

- BEFORE indicates that adaptations are performed *before* join point occurrences. It is used in pointcuts that link join points representing behavioral element (e.g. actions) selection with aspect adaptations;
- AFTER indicates that adaptations are performed *after* join point occurrences. It is also used in pointcuts related to behavioral join points;
- AROUND specifies that adaptations are performed in both *before* and *after* join point occurrences. It is used in pointcuts that link join points that select behavioral elements (e.g. actions or behaviors) with aspect's behavioral adaptations;
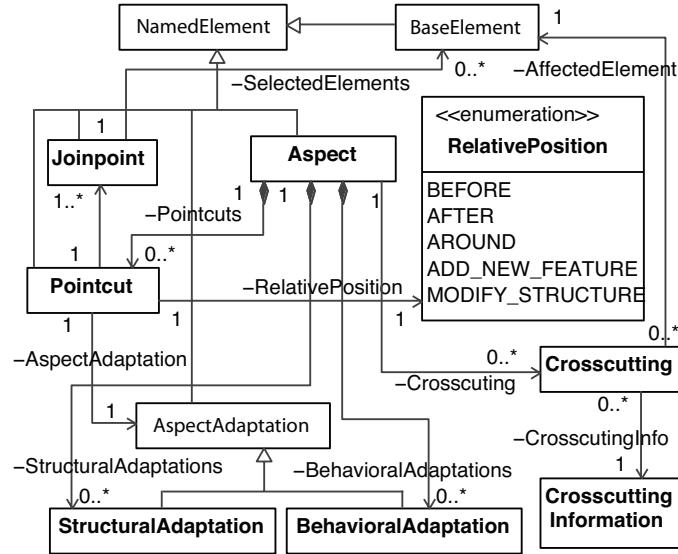
Figure 6.4.: DERCS meta-model: AO-related elements

- `ADD_NEW_FEATURE` specifies that new features (e.g. an attribute in a class, or a parameter in a method) are included by the aspect adaptation in the structural elements selected by the join point;
- `MODIFY_STRUCTURE` indicates that structural features of the selected elements are modified by the aspect adaptation. Likewise the previous relative position, this is used in pointcuts related to join points selecting structural elements.

Join points are represented by the `Joinpoint` element, which contains a list of selected base elements, i.e. those elements that extend the `BaseElement` class. In other words, the selected elements list consists of: instances of `Class`, `Attribute`, `Method`, and `Node`, in addition to all behavior-related elements, i.e. instances of all `Action` subclasses, and `Behavior` class. As it will be explained in the next section, the selection query specified in JPDD is evaluated, and all DERCS elements instances that match with the selection criteria are included in the `Joinpoint`'s elements selection list.

6.3. UML-to-DERCS Transformation

Based on the information provided in the previous section, it can be claimed that DERCS meta-model can represent structure and behavior in a more concise way than UML meta-model. DERCS uses fewer meta-model elements to represent the same information (i.e. system structure and behavior) compared to UML, which, in turn, has different element to represent similar features. In this sense, there is no direct one-to-one relationship among many DERCS elements and their similar counterpart in UML meta-model. Hence, to transform a UML model into a

DERCS model, some transformation heuristics had to be defined.

Considering the structural elements, the majority of them have a direct counterpart in the UML meta-model, as show in table 6.1. Thus, when GenERTiCA's transformation engine reads the UML model, and one of these elements is found, it does not need to interpret the UML meta-model element semantic regarding any transformation heuristic, i.e. it just creates the DERCS element that matches with the UML one. However, there are two exceptions: *(i)* method signatures; and *(ii)* associations between classes. An UML's `Kernel.Operation` element decorated with «`getter`» or «`setter`» stereotypes indicates an access method to a given attribute. Thus, the transformation heuristic understands such role, and creates not only one DERCS' `Method` element, but also its associated `Behavior` element, in which actions corresponding to the specified semantics (i.e. get/set attributes values) are inserted.

Associations among classes have also a special transformation heuristic. As stated in chapter 5, all associations must have at least one end specifying multiplicity equals to "1", and the navigable property set to true; the class representing this association end will receive elements related to the association. For "normal" associations, the transformation engine inserts a new `Attribute` element (related to the other association end), along with a `Parameter` element in class constructor, and an `AssignmentAction` to represent this new attribute initialization. Access methods, i.e. get/set methods, for the new attribute are also created as described in the previous paragraph. For aggregation relationships, the same transformation heuristic is applied. However, for composition relationships, the class receives two additional methods instead of the new parameter and its assignment action: *(i)* one method to create composite class parts; and *(ii)* another one to remove (or destroy) composite class parts. For both methods, the corresponding behavior is also created. To illustrate the mentioned heuristics, let's consider the *leftWheel*

Table 6.1.: UML-to-DERCS mapping table

UML meta-model	DERCS meta-model
`Kernel.Class`	`Class`
`Kernel.Property`	`Attribute`
`Kernel.Type` or `Kernel.PrimaryType`	`DataType` subclass
`Kernel.Operation`	`Method`
decorated with «`getter`»	`Method, Behavior, ReturnAction`
decorated with «`setter`»	`Method, Parameter, Behavior, AssignmentAction`
`Kernel.Parameter`	`Parameter`
`Kernel.ParameterDirectionKind`	`ParameterKind`
`Kernel.Association`	`Attribute, Method, Parameter, Behavior, AssignmentAction, ReturnAction`
if any association end defines `AggregationKind` as composite	`Attribute, Method, Parameter, Behavior, AssignmentAction, ReturnAction, CreateObjectAction, DestroyObjectAction`
`Kernel.InstanceSpecification` or `BasicInteractions.Lifeline`	
related to class decorated with «`SchedulableResource`»	`ActiveObject`
related to class decorated with «`MutualExclusionResource`» or «`Resource`»	`PassiveObject`

composite relationship between *MovementControlSystem* and *Actuator* depicted in figure 5.2. As one can see, *Actuator* association end is the navigable end (indicated by the arrow head). Consequently, *MovementControlSystem* receives a new attribute, whose name is *leftWheel* and the type is the *Actuator* class and, as this association is a composition, the mentioned methods are also added in *MovementControlSystem* instead of the new parameter and its assignment action in *MovementControlSystem*'s contructor.

As mentioned, UML has very different ways to specify system behavior. DERCS proposes a more simplified form for behavior representation (compared to UML meta-model elements). For that reason, there is no direct one-to-one mapping from UML behavior-related elements to DERCS ones. Thus some UML behavior diagrams interpretation heuristics have been created.

In AMoDE-RT approach, sequence diagram is the most important diagram to specify objects behavior, due to its capability of showing objects interactions, execution flow control (using *combined fragments* [92]), and also actions (using the reserved words presented in section 5.2, table 5.1). The whole behavior of a distributed embedded real-time system is specified using different sequence diagrams, i.e. behavior information must be extracted from more than one diagram. Additionally, there is no one-to-one relation between sequence diagrams meta-model elements and DERCS behavioral ones. Thus, to accomplish the UML-to-DERCS transformation, an interpretation heuristic has been defined. Sequence diagram messages are statically analyzed using a stack-based algorithm, which pushes messages (i.e. method calls) on the top of a "call stack" to discover which messages (i.e. actions) are nested inside the behavior of other methods. Algorithm 1 shows this static analysis.

For each message, a tuple *m = (Sender, Target, Behavior)* is created, where *Sender* is the message's sender lifeline[1]; *Target* is the message's target lifeline; and *Behavior* is the behavior associated to the method represented by the message. The algorithm analyzes all messages (respecting messages order depicted in the sequence diagram) to create the corresponding action, e.g. sending message, assignment, expression, etc. If the message represents a sending message action, this message's tuple is pushed on top of the call stack. If the following messages are sent from the same lifeline (i.e. same object) as tuple's target on stack's top, these messages represent actions performed within the context of the calling message's behavior.

It is important to highlight that combined fragments are also considered in sequence diagrams analysis. Combined fragments represent execution control flow in objects interactions, i.e. they can specify both conditional, or repeating interactions (as described in section 5.2.4). For each combined fragment, a behavior with pre-conditions (for combined fragments whose `interactionOperator` property is set to `alt` or `opt`), or post-conditions (for those specified with `loop` operator) is created. Therefore, messages enclosed by combined fragments represent actions performed within the context of branches (i.e. "ifs") or loops. When a combined fragment is detected, a new behavior is created, and inserted in the tuple's behavior on stack's top. Hence, actions created from messages enclosed by this combined fragment are inserted into the `Behavior` element related to the combined fragment.

---

[1]Lifelines are vertical lines depicted in sequence diagrams that represent objects and/or classes. The proposed transformation heuristics interprets lifelines as system's objects.

Algorithm 1 Extract behavioral information from sequence diagrams

---

  1: $stack \leftarrow \emptyset$
  2:
  3: **for all** m = message in Sequence Diagram **do**
  4:     **if** $stack = \emptyset$ **then**
  5:         PUSH($stack, m$)
  6:     **else**
  7:         **if** $stack$.Top.Target $= m$.Sender **then**
  8:             *// Action must be inserted into the method's behavior on the stack's top*
  9:             $action \leftarrow$ create an action from m
 10:         **else**
 11:             *// Action must be inserted into other method's behavior.*
 12:             POP($stack$)
 13:
 14:             *// Looking for the "right" method's behavior according the call stack...*
 15:             **while** $(stack \neq \emptyset) \land (stack$.Top.Target $\neq m$.Sender$)$ **do**
 16:                 POP($stack$)
 17:             **end while**
 18:
 19:             **if** $stack \neq \emptyset$ **then**
 20:                 *// The "right" method behavior could be found*
 21:                 $action \leftarrow$ create an action from m
 22:             **else**
 23:                 *// Message order violates call stack order, i.e. it is sent by a lifeline*
 24:                 *// (i.e. object) that have not sent any message before, breaking the*
 25:                 *// execution flow*
 26:                  throw an exception
 27:             **end if**
 28:         **end if**
 29:
 30:         insert $action$ in $stack$.Top.behavior
 31:
 32:         *// Potentially, all send message actions (including messages to the lifeline itself)*
 33:         *// trigger different behaviors, and hence, they must be pushed on the stack*
 34:         **if** $((action$ is a send message action $) \land (m$.Sender $\neq m$.Target$))$
             $\lor$ (m is a recursive message) **then**
 35:             PUSH($stack, m$)
 36:         **end if**
 37:     **end if**
 38: **end for**

The behavior transformation heuristic allows merging information from different sequence diagrams. For a given message *m*, if the following messages are nested messages (i.e. departing from *m*'s target lifeline), a DERCS `Behavior` element is created, and associated with the method represented by *m*. To illustrate this heuristic, let's consider the sequence diagram depicted in figure 5.5. *MovementController.run()* method has two nested messages *Actuator.setActValue* and *MovementController.processInfo*. Hence, a `Behavior` element containing two `SendMessageActions` is created and associated with *MovementController.run()* method. Similarly, *Actuator.setActValue()* method has five nested messages (e.g. messages 3, 4, 6, 7, 8, and 9), and also two combined fragments enclosing its nested messages. Thus, *Actuator.setActValue()*'s `Behavior` contains one `Behavior` representing the "loop" combined fragment, which, in turn, contains a `CreateObjectAction` and a `AssignmentAction` (related to message 3), another `Behavior`[2] (related to the "alt" combined fragment), and other `AssignmentAction` (related to message 9). *MovementController.processInfo()* method's behavior is extracted using the same heuristics.

As one case see, from a single sequence diagrams it is possible to extract different method behaviors, eliminating the need of creating one sequence diagram to each method behavior. However, if not carefully used, such approach can produce duplicated specification, e.g. the same method behavior specified twice, leading to ambiguities in behavior specification. To overcome this problem, a simple ambiguity detection heuristic has been created: if there is already a `Behavior` element associated with a *m* method (created from other sequence diagram), and there are messages nested to *m* in the current sequence diagram, this situation indicates that *m*'s behavior was specified twice. When this situation occurs, the transformation engine reports the detected ambiguity to system designers. To illustrate this situation, let's consider that sequence diagrams of figures 5.5 and 5.6 are specified in the same UML model. *MovementController.processInfo()* method has nested messages in both diagrams. The transformation engine will create a `Behavior` element to this method during the interpretation of figure 5.5's sequence diagram and, when the transformation engine tries interpret figure 5.6's sequence diagram, it will discover that there is already a `Behavior` element associated to *MovementController.processInfo()*. Consequently, the ambiguity is detected.

To summarize sequence diagram to DERCS elements transformation, table 6.2 presents the relationships among UML meta-model elements with DERCS ones.

State diagrams are used in AMoDE-RT modeling approach, and thus, need also transformation heuristics to derive DERCS behavioral elements from them. Two heuristics have been defined: *(i)* straightforward state machine mapping; and *(ii)* applying the *objects for states* design pattern [48]. The first heuristic produces *if-then-else* state machine implementations. More specifically, DERCS `StateDataType` elements are created to each state machine. Every UML `BehaviorStateMachines. State` element in the state machine is transformed to a DERCS `State` element, which is associated with the created `StateDataType`. Similarly, `BehaviorStateMachines. Transition` elements are transformed into `StateTransition` elements, whose guard condition, and from/to states are also obtained from `BehaviorStateMachines`. Moreover, the same transformation heuristics are applied to orthogo-

---

[2]This `Behavior` element contains actions related to messages 4–8

Table 6.2.: UML-to-DERCS behavior elements relationships

UML meta-model	DERCS meta-model

`BasicInteractions.Lifeline,`
`BasicInteractions.Message,`
`BasicInteractions.MessageOccurrenceSpecification,`
`Kernel.Operation`
	`Behavior,`
	`AssignmentAction,`
	`ExpressionAction,`
	`InsertArrayAction,`
	`RemoveArrayAction,`
	`ModifyStateAction`

`BasicInteractions.CallEvent,`
`BasicInteractions.CreationEvent,`
`BasicInteractions.DestructionEvent,`
	`SendMessageAction,`
	`CreateObjectAction`
	`DestroyObjectAction`

`BasicInteractions.Lifeline,`
`BasicInteractions.CombinedFragment,`
`BasicInteractions.InteractionFragment`
	`Behavior`

nal state: one `StateDataType` element is created to each orthogonal state region, and thus, DERCS `State` elements are created to each *AND-states* (i.e. concurrent sub-state).

According to AMoDE-RT guidelines, one state diagram is associated with only one class. Hence, an attribute (whose type is this state diagram's `StateDataType` element) and a method (which is responsible to execute different actions depending on the actual state) are created and added to the associated class. It is important to highlight that it is assumed that associations between state diagrams and classes represent the following execution semantics: instances of this class are active objects that execute the method related to the state machine. This method is triggered periodically, and its behavior executes concurrently with other active objects' behaviors. In this sense, the behavior related to this method performs a "common" *if-then-else* state machine implementation. Considering the state machines with orthogonal *AND-states*, additional attributes are created to each sub-state machine. However, instead of representing `StateDataType` elements, they represent sub-state machines' active objects. Therefore, when an object enters in a orthogonal state, sub-state machine active objects start to execute their behavior.

On the other hand, the second heuristic implements the *objects for states* design pattern [48], in which each state is represented as an object that implements behavior related to the state. *Objects for states* design pattern involves the following elements: *context*; *state*; and *concrete state subclasses*. To summarize, the context object has an attribute representing the state object, which is an instance of one state's subclass. The context delegates its methods execution to the state object. In the proposed transformation heuristic, the class associated with the state diagram is the *context*. This class receives a method representing the state diagram execution, and an attribute representing its state object, similarly to the first transformation heuristic. This method behavior has only one `SendMessageAction` action, representing the delegation of this method execution to the state object.

An abstract class is created to represent the state machine, and is used as the new attribute's type. This abstract class also has a method representing state machine execution, which is overridden by states' concrete subclasses. For each state, a `Class` element is created to represent the state's concrete subclass. This class extends the state diagram's abstract class, overriding

its abstract method using the behavior extracted from the sequence diagram associated with the state. At the end of this method behavior, additional `Behavior` elements with pre-conditions (representing the *if-then-else* statements) are inserted to represent state's outgoing transitions. Actions executed in these behaviors represent the destruction of context's current state object, and the creation of the next state object. For orthogonal *AND-state*, this heuristic follows the same approach as the first one: creates active object classes for each sub-state machine as explained earlier.

Both approaches have pros and cons. For example, (i) allows less memory usage but leads to extra runtime overhead because objects need to discover which actions must be performed in the actual state, by means of comparing all state machine's possible states (in the worst-case), in order to execute the correct actions for the current state. On the other hand, (ii) uses more memory because states are themselves objects (not only attributes representing states as enumerations or integer numbers), but allows less runtime overhead caused by the search for the correct actions to be executed when the object is in a specific state. The decision on which heuristic is applied depends on system constraints, and is made by designers before the UML-to-DERCS transformation process. Although important, none of these state diagrams transformation heuristics are implemented in the initial version of GenERTiCA's transformation engine prototype. In fact their implementation was not considered one of this thesis' main contributions, and thus, it was left to future work.

6.4. Mapping Rules

6.4.1. Overview

To generate code from the UML model, GenERTiCA adopts a script-based approach, in which small scripts define how to map model elements into target platform constructions, generating source code fragments that are merged to produce source code files. The proposed script-based code generation improves separation of concerns in mapping rules specification, because each script is concerned with the transformation of a single model element (or few of them) into source code fragment.

Mapping rules are described as XML [129] files, whose format is portable, and allows the specification of self-described content organized in a tree structure. These characteristics, and also because XML is a *de-facto* standard, have influenced its choice as the language used to describe GenERTiCA mapping rules. Furthermore, XML tree organization facilitates scripts storage in terms of platform mapping rules repositories, allowing scripts to be reused in further projects that use the same target platform. Hence, the design effort to derive system implementation from an UML model is decreased.

Leaf nodes of the mapping rules' tree contain scripts executed to generate code from a specific DERCS element (representing the correlated UML element). As mentioned, each script concentrates on generating a source code fragment related to a single DERCS element. The correct script is selected based on which element is being accessed by the code generation algo-

rithm (see next section), i.e. the leaf node must match with the DERCS element. These scripts
have complete access to DERCS model information, in order to be able to generate source code
as complete as possible. Consequently, the more complete code generation scripts are, more
source code is generated, and less effort is required to manually write additional code. One of
GenERTiCA's aims is to allow code generation as complete as possible, decreasing (or even
eliminating) the need of manual coding. However, this work does not define a new script lan-
guage or script execution engine. It rather uses a well-known open source scripting framework
called *Velocity* [3], which defines the *Velocity Template Language* (VTL) that provides all func-
tionalities required to assist the GenERTiCA code generation approach implementation. VTL
is a Java-like scripting language, which returns a string as result of script execution. Thus, the
generated source code fragment is obtained by means of accessing model information through
DERCS API.

Considering the mapping rules organization, one can see in figure 6.5 that the XML file root
is divided into a set of different target platforms, whose child trees represent mapping rules for
constructions in the target platform. There are two source code categories defined in a platform
mapping rules: *(i)* application code; and *(ii)* platform code. Both are divided in software and
hardware source code. In the *application* branch, software and hardware sub-trees have the
same structure, i.e. they have the same script types to generate code from DERCS elements; in
platform configuration branch the difference is that platform software elements provide services
instead of components as in the hardware ones.

6.4.2. Application Code

Application branch is subdivided in: *(i)* source code options; *(ii)* primary elements scripts;
(iii) scripts for class-related elements; *(iv)* scripts for behavior-related elements; and *(v)* script
specifying DERAF aspects implementation. Considering (i), figure 6.6 lines 01-29 show an
excerpt from a XML file with mapping rules to Java. The *<SourceOptions>* node manages
issues related to source code files creation, defining the source code file naming convention
(*<FileNameConvention>* node) and organization (*<SourceOrganization>* node). GenERTiCA
assumes that a target language may have both a declaration and an implementation file, such
as in C/C++ which defines header and implementation source code files. Thus, *<SourceOr-
ganization>* node defines how each of them is structured. If there are dependencies among
source code files, the *<SourceReference>* node indicates target language constructions to spec-
ify source code file references. It is important to note two attributes in *<SourceOrganization>*
node: *isAspectLanguage* and *hasClassesDeclaration*. The former indicates if the target lan-
guage is an AO-language or not. GenERTiCA will not perform aspects weaving if the attribute
value is "yes". It will interpret scripts in the *Aspect* branch as aspects constructions in the target
AO-language. *hasClassesDeclaration* attribute indicates if the target language requires a class
declaration before describing class implementation, such as in C/C++ languages.

Taking into account (ii), the *<PrimaryElements>* node (figure 6.6 lines 30-59) provides
scripts representing straightforward mappings from DERCS elements to primary elements con-
structions in the target language. On the other hand, considering (iii), the *class* branch provides
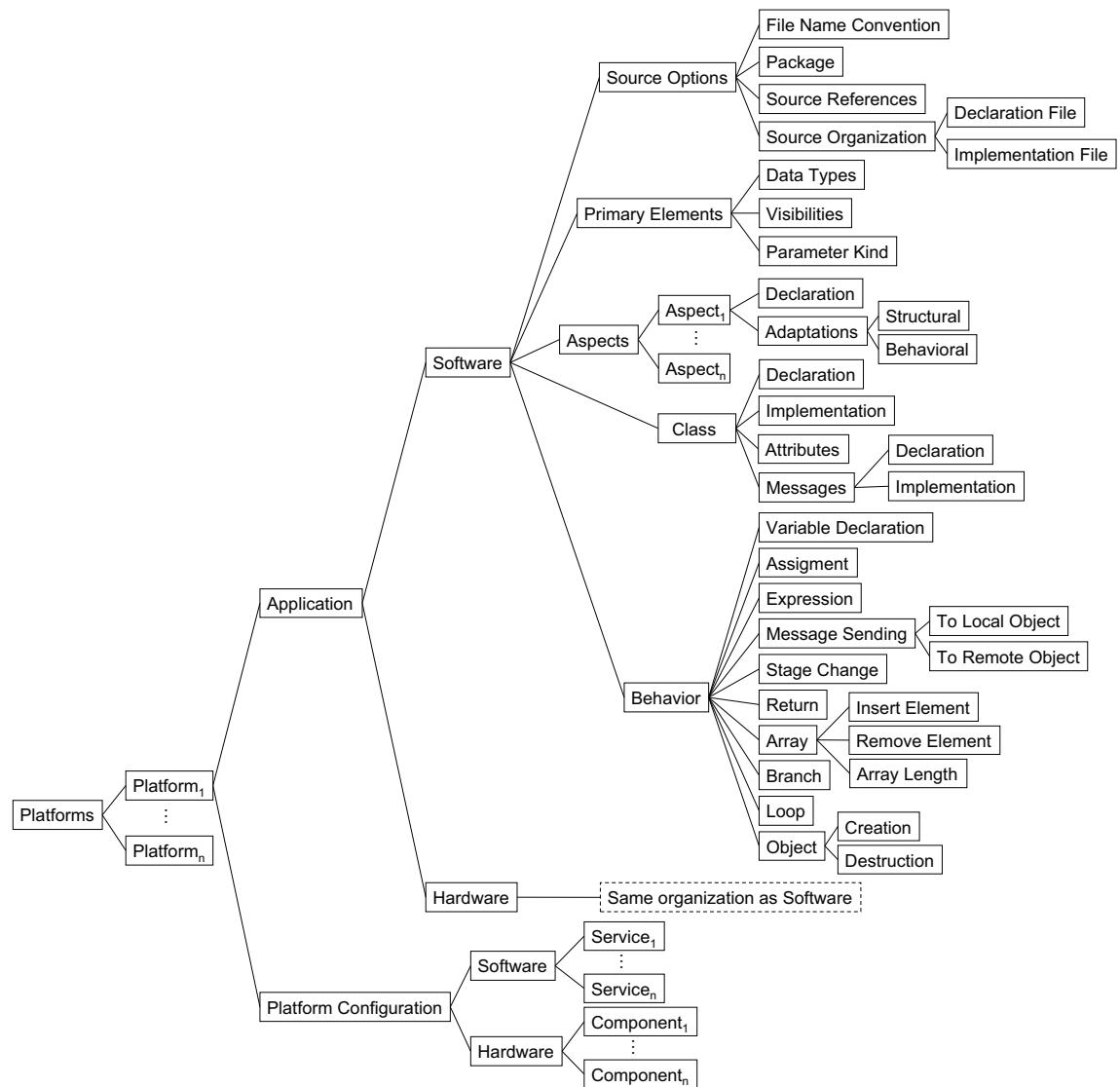
Figure 6.5.: Mapping rules XML organization

more complex scripts, which need to retrieve information on the DERCS element being evaluated, in order to generate the correct code fragment for that element. *<Class>* node defines, in the *<Implementation>* node, how to use target language constructions to describe the class implementation, in terms of attributes and methods. Additionally, if the target languages requires a class declaration construction, this node also provides means to specify this in the *<Declaration>* node.

However, *<Class>* node's most important children nodes are *<Attributes>* and *<Messages>* nodes. The later provides scripts to generate methods declaration and implementation based on information contained in a DERCS `Method` element. The former provides a script to transform a DERCS `Attribute` element into an attribute construction in the target platform language. A script to generate attribute declarations for a Java target platform is presented in figure 6.7. The code fragment produced by this script is shown in this figure's lower part. As one can see, this script is highly cohesive because it deals with only one element, i.e. the attribute, from which information is obtained by accessing context variables (those identifiers starting with a "$" character), or directly calling one of DERCS API methods of the *Attribute* element (e.g. line 03). It is important to note here that all methods (of all elements) available in the DERCS API can be used within the context of script.

Behavior branch (iv) provides key scripts to map DERCS behavioral elements into con-

```
01 <SourceOptions                          30 <PrimaryElements>
02     isAspectLanguage="no"               31   <DataTypes>
03     ClassesPerFile="1"                  32     <Array>$DataType.DataType[]</Array>
04     hasClassesDeclaration="no"          33     <Boolean>boolean</Boolean>
05     Indentation="5"                     34     <Byte>byte</Byte>
06     BlockStart="{" BlockEnd="}">        35     <Char>char</Char>
07   <FileNameConvention>                  36     <Class>
08     $Class.Name                         37       $DataType.Represent.Name
09   </FileNameConvention>                 38     </Class>
10   <Package>                             39     <DateTime>DateTime</DateTime>
11     package $Class.Package;             40     <Enumeration></Enumeration>
12   </Package>                            41     <Integer>int</Integer>
13   <SourceReference>                     42     <Long>long</Long>
14     import $ReferencedClass.Package     43     <Short>short</Short>
15     .$ReferencedClass.Name;            44     <String>String</String>
16   </SourceReference>                    45     <Void>void</Void>
17   <SourceOrganization>                  46     <Double>double</Double>
18     <DeclarationFile                    47     <Float>float</Float>
19         FileExtension="">               48   </DataTypes>
20       $SourceCode.ClassesDeclaration    49   <Visibilities>
21     </DeclarationFile>                  50     <Private>private</Private>
22     <ImplementationFile                 51     <Protected>protected</Protected>
23         FileExtension=".java">          52     <Public>public</Public>
24       $SourceCode.PackagesDeclaration   53   </Visibilities>
25       $SourceCode.ReferencesDeclaration 54   <ParameterKinds>
26       $SourceCode.ClassesImplementation 55     <In></In>
27     </ImplementationFile>               56     <Out></Out>
28   </SourceOrganization>                 57     <InOut></InOut>
29 </SourceOptions>                        58   </ParameterKinds>
                                           59 </PrimaryElements>
```

Figure 6.6.: Mapping rules: *<SourceOptions>* and *<PrimaryElements>* branches

```
01  <Attributes>
02    $VisibilityStr
03    #if ($Attribute.isStatic()) static #end
04    $DataTypeStr $Attribute.Name;
05  </Attributes>

private int windSpeed;
```

Figure 6.7.: Mapping rules: *<Attributes>* node

```
01  <SendMessage>                              26    <ToRemote>
02    <ToLocal>                                27      <Software>
03      <Software>                             28        #set($x=$Action.ParametersValuesCount
04        #if ($Action.ToObject !=             29              + 1 )
05              $Action.FromObject)            30      myMsg.setNrBytes($x);
06          $Action.ToObject.Name.             31      myMsg.addByte(
07        #end                                 32              $Action.RelatedMethod.ID);
08        $Action.RelatedMethod.Name(          33      #foreach($v in
09        #if($Action.ParametersValuesCount >  34              $action.getParametersValues())
10            0)                               35        myMsg.AddByte((byte)$v);
11          #foreach($param in                 36      #end
12                  $Action.ParametersValues)  37      localTp.sendMsg(conectionNumber,
13          #if ($velocityCount > 1), #end     38            myMsg,
14          #set( $x =  $velocityCount - 1)    39            timeOutParam.getmsgSendTime());
15          #if($Action.isParameterValue($x))  40      </Software>
16            $param                           41      <Hardware></Hardware>
17          #else                              42    </ToRemote>
18            $param.Name                      43  </SendMessage>
19          #end
20          #end
21        #end
22        );
23      </Software>
24      <Hardware></Hardware>
25    </ToLocal>
```
```
envInfo.getWindSpeed()                         myMsg.setNrBytes(1);
                                               myMsg.addByte(49);
                                               localTp.sendMsg(conectionNumber, myMsg,
                                               timeOutParam.getmsgSendTime());
```

Figure 6.8.: Mapping rules: *<SendMessage>* node

structions in the target language. There is one node to specify a script to each action available in the DERCS actions model, and also to behaviors with pre- (branch) and post-conditions (loop). Designers must only specify how to map individual DERCS actions into equivalent constructions in the target language. Code fragments related to actions are generated by these scripts, and merged to compose a behavior. This approach facilitates the specification of behavior mapping rules, because designers do not need to specify complex scripts that deal with all action types in the same script. Scripts have full access to actions information, as well as the behavior containing them. Thus, it is possible to create very specialized and elaborated scripts, as the mapping rules for `SendMessageAction` elements shown in figure 6.8. As mentioned, DERCS sending message semantics is the same for any kind of target object, i.e. local or remote objects, and/or objects implemented as software or hardware. The target platform is in charge to implement these sending message variations. In figure 6.8, lines 04-22 (left column) show the script to map actions that send messages to local objects in a Java platform. On the other hand, lines 28-39 (right side) depict the mapping of actions that send messages to remote objects using a communication API [34] in the same Java platform. GenERTiCA decides which script should be executed based on the information contained in the `SendMessageAction`, i.e. GenERTiCA compare the `Node` in which both sender and target objects have been deployed. If the `Node` is the same, GenERTiCA executes the script related to local messages, otherwise it executes the script related to remote messages. Examples of code fragments generated by both scripts are presented in the lower part of this figure 6.8.

The most important part of the application mapping rules specification is the set of scripts to describe DERAF aspects implementations. As stated in section 5.3.2, DERAF aspects high-level semantics do not define how to implement aspect adaptations; it must be done in implementation phase considering the target platform. Thus, in the proposed code generation and aspects weaving approach, DERAF aspects implementation is specified via scripts within the *aspects* branch (v). For each aspect, scripts representing its structural and behavioral adaptations are defined. GenERTiCA executes aspect adaptation scripts upon elements selected by the join points. More specifically, information contained in aspects' pointcuts specification is used to select which adaptations scripts must be executed to modify the elements gathered by join points. In other words, when the code generation algorithm analyzes a DERCS element (e.g. class, attribute, behavior, action, etc.) to generate its source code fragment, it also checks if this element is selected by any JPDD. If it is the case, scripts of aspects adaptations related to these JPDDs (as indicated in aspects' pointcuts specification) are executed, modifying either the generated code fragment, or the element itself. Thus, besides code generation, GenERTiCA also performs aspects weaving in both generated code fragments and DERCS elements.

There are two kinds of aspect adaptation implementations: one that modifies the generated code fragment; and one that modifies the selected element. The former is executed after the script defined in *<Class>* of *<Behavior>* branches for the selected element, modifying the generated code fragment to include changes promoted by the aspect adaptation. The later is executed before the mentioned branches' scripts, modifying the selected element at model level, i.e. the input DERCS model element is modified. Thereby, GenERTiCA is able to perform aspects model weaving. To illustrate these two types of aspect adaptation implementations, figure 6.9 presents the implementation of the *PeriodicTiming* aspect (see section 5.3.2) for the

RT-FemtoJava platform [65, 131].

In this example, six adaptations have been created: the first four modifying the generated source code fragment, and the two last modifying directly the affected element (indicated by the `ModelLevel` attribute). *PeriodicTiming* aspect affects active object classes that need to execute its behavior cyclically at a given frequency. Hence, adaptations affected these classes' attributes and behavior. *Period* structural adaptation adds two attributes in affected classes, as depicted in figure 6.10 lines 07-09; initialization code for these attributes is inserted in the class constructor by *SetPeriod* behavioral adaptation, as shown in lines 19-21. *FrequencyControl* appends code (after the last action) that controls the execution frequency of active objects' periodic behavior using RT-FemtoJava platform constructions, as presented in line 33. Similarly, *LoopMechanism* adaptation encloses the periodic behavior (and the code inserted by *FrequencyControl*) in a *while* construction, as depicted in lines 27 and 36. It is important to note that, to enable the expected behavior, *FrequencyControl* adaptation must be performed before *LoopMechanism*. GenERTiCA uses the `Order` attribute to control the execution order of adaptations scripts (lower numbers have higher execution priority). In this example, *FrequencyControl* order is 3 and *LoopMechanism* is 4, causing *FrequencyControl* script to be executed before *LoopMechanism* one, forcing the code inserted by the first script (line 33) to be enclosed by the *while* construction inserted by *LoopMechanism* script (lines 27 and 36).

```
01 <PeriodicTiming>
02  <Declaration></Declaration>
03  <Adaptations>
04   <Structural Name="Period" Order="1" ModelLevel="no">
05     private static RelativeTime _Period = new RelativeTime(0,0,0);
06     private static PeriodicParameters _PeriodicParams =
07           new PeriodicParameters(null, null, null, null, null);
08   </Structural>
09   <Behavioral Name="SetPeriod" Order="2" ModelLevel="no">
10     _Period.set(0,pPeriod,0);
11     _PeriodicParams.setPeriod(_Period);
12     setReleaseParameters(_PeriodicParams);
13   </Behavioral>
14   <Behavioral Name="FrequencyControl" Order="3" ModelLevel="no">
15     waitForNextPeriod();
16   </Behavioral>
17   <Behavioral Name="LoopMechanism" Order="4" ModelLevel="no">
18     while (isRunning()) $Options.BlockStart
19       $CodeGenerator.getGeneratedCodeFragment(1)
20     $Options.BlockEnd
21   </Behavioral>
22   <Structural Name="ModifyConstructor" Order="1" ModelLevel="yes">
23     $Message.addParameter("pPeriod", $DERCSFactory.newInteger(false),
24                  $DERCSFactory.getParameterIn());
25   </Structural>
26   <Behavioral Name="AdaptObjectConstruction" Order="1"
27           ModelLevel="yes">
28     $Action.addParameterValue($Crosscutting.getValueOf("Period"))
29   </Behavioral>
30  </Adaptations>
31 </PeriodicTiming>
```

Figure 6.9.: Mapping rules: *PeriodicTiming* aspect implementation

Moreover, as one can see, source code fragments inserted by the mentioned adaptations are exactly equal to their script in the mapping rules XML file, indicating that these adaptations are independent of affected elements information. However, although these scripts do not use any information of affected elements, designers need to be aware that both scripts could be applied only to behavior-related elements (e.g. `Behavior` or `Action` subclass). However, if these adaptations need to be applied to other DERCS elements (e.g. `Class`, `Method`, etc.), their script must be changed to provide additional modifications. On the other hand, there are adaptations scripts that are close related to affected elements, e.g. *ModifyConstructor* or *AdaptObjectConstruction*. These adaptations modify affected elements' DERCS specification rather than their generated source code fragment. Adaptations changing model elements are always executed before any other aspect adaptation, allowing model-level modifications to be visible for the non-aspect scripts (i.e. "normal" code generation scripts). Consequently, aspects model weaving occurs prior to code generation, and also aspects weaving in the generated fragments.

```
01 public class MovementControler extends RealtimeThread {
02   ...
03   protected EnviromentInformation envInfo;
04   protected int windSpeed;
05   ...
06   // PeriodicTiming.Period - Begin
07   private static RelativeTime _Period = new RelativeTime(0,0,0);
08   private static PeriodicParameters _PeriodicParams =
09           new PeriodicParameters(null, null, null, null, null);
10   // PeriodicTiming.Period - End
11   public void MovementControler(EnviromentInformation new_envInfo,
12     MovementInformation new_mMovInfo, MovementInformation new_bMovInfo,
13     MainRotorActuator new_mRAct, BackRotorActuator new_bRAct, int pDeadline,
14     // PeriodicTiming.ModifyConstructor - Begin
15     int pPeriod) {
16     // PeriodicTiming.ModifyConstructor - End
17     ...
18     // PeriodicTiming.SetPeriod - Begin
19     _Period.set(0,pPeriod,0);
20     _PeriodicParams.setPeriod(_Period);
21     setReleaseParameters(_PeriodicParams);
22     // PeriodicTiming.SetPeriod - End
23   }
24   ...
25   public void run() {
26     // PeriodicTiming.LoopMechanism (1) - Begin
27     while (isRunning()) {
28     // PeriodicTiming.LoopMechanism (1) - End
29       ...
30       windSpeed = envInfo.getWindSpeed();
31       ...
32       // PeriodicTiming.FrequencyControl - Begin
33       waitForNextPeriod();
34       // PeriodicTiming.FrequencyControl - End
35     // PeriodicTiming.LoopMechanism (2) - Begin
36     }
37     // PeriodicTiming.LoopMechanism (2) - End
38   }
39 }
```

Figure 6.10.: Source code fragment with modifications performed by aspect adaptations

Considering model level aspect adaptations, *ModifyConstructor* structural adaptation uses DERCS API to modify the constructor of affected classes to include a new parameter that represents initialization value of the *period* attribute inserted by *Period* adaptation. Moreover, *SetPeriod* adaptation adds the code in selected constructors' behavior to assign this new parameter's value to the *period* attribute. As aspects model weaving occur before code generation, the code generation script is able to include the new parameter in affected constructors' code fragment, as shown in line 15 of figure 6.10. Similarly, as the constructor of affected classes has been modified, actions that create objects from these classes must also be modified. Therefore, *AdaptObjectConstruction* adaptation script performs a model level adaptation in the mentioned actions. This adaptation uses the period information specified in ACOD's *crosscutting* relationships to include the correct information in the right object creation action.

As one can infer, this difference in aspect adaptation types allows flexibility in aspect implementation specification. Designers can choose the most suitable manner to implement DERAF aspects adaptations, taking into account the target platform. Additionally, the combination of aspects model and source code weaving opens room for new forms to describe aspects implementation, as well as allows new ways to explore how aspects modifications (performed on system functional (or base) elements) are implemented.

6.4.3. Platform Configuration

Platform configuration branch provides script to generate platform configuration files, or tailored source code files of frameworks, libraries, or APIs, which are generated based on services or components needed by application source code. GenERTiCA assumes that the target platform provides means (i.e. software services and hardware components) to support platform constructions described in application scripts. Considering the usually constrained execution environment of embedded systems, it makes sense to tailor the target platform, in order to provide only services and components required by the embedded application. In this sense, it is essential that target platforms provide means to allow their configuration in one of the following alternatives:

- **Configuration files**, which turn on/off services or compomponents. GenERTiCA can generate configuration files, allowing platform-specific tools to configure them, e.g. removing unused elements, or optimizing provided services;

- **Source code** availability. GenERTiCA can generate tailored source code based on the original code, optimizing the final target platform in terms of required footprint.

GenERTiCA platform configuration approach integrates DERAF aspects with platform configuration. More specifically, DERAF aspects are related to platform services and/or components, and thus, if an aspect is specified in the model, the service(s) and component(s) related to this aspect must be included in the final platform. The platform configuration specification is very pragmatic: *software* or *hardware* branches are divided in several files, which, in turn, are divided in parts (or fragments), as depicted in the example in figure 6.11. Platform configuration or source code files are, in fact, specified as a sequence of text fragments within *<Part>* nodes.

```
01 <PlatformConfiguration>
02  <Software>
03   <General OutputDirectory="./platform"></General>
04   <File Name="PlatFile_1.java" OutputDirectory="realtime"
05          Aspects="SchedulingSupport">
06    <Part>
07      ...
08      Configuration statements or source code fragment
09      ...
10    </Part>
11    <Part Aspects="TimingAttributes, PeriodicTiming">
12      ...
13      Configuration statements or source code fragment
14      ...
15    </Part>
16   </File>
17   <File> ... </File>
18   ...
19  </Software>
20 </PlatformConfiguration>
```

Figure 6.11.: Platform configuration XML structure

Hence, GenERTiCA creates platform configuration files from these fragments. Both *<File>* and *<Part>* nodes have an `Aspects` attribute indicating which DERAF aspects are related to, respectively, platform configuration (or source code) files and/or its text fragments. They are included in the generated platform configuration only if the model specifies any aspect in the list. On the other hand, if any *<File>* and/or *<Part>* node do not specify the `Aspects` attribute (or its value is an empty string), it means that the node's content must always be included in the generated configuration file.

6.5. Code Generation Process

As mentioned in the previous section, GenERTiCA adopts a script-based approach to produce source code and/or configuration files, for both application and target platform. Besides code generation, GenERTiCA also performs aspects weaving using aspects specified in the UML model, as well as their adaptations' implementation in form of scripts described in the mapping rules XML file.

Therefore, GenERTiCA's generation process involves two main phases: code generation/aspects weaving of application level elements, and configuration files generation or source code tailoring of the target platform. The former analyzes all elements in the model, trying to find the script in the mapping rules XML file that matches with each of them. On the other hand, the later reads all XML nodes related to platform configuration, checking if the associated aspects have been specified in the model, to generate the configuration file according to the specification.

To provide more details on the process followed by GenERTiCA to generated application code, figure 6.12 depicts the activity diagram representing this process. Source code is directly generated from the DERCS model, which is obtained from the original UML model via model

transformations, due to its capability of representing structure, behavior, and non-functional requirements handling in a more precise and unambiguous fashion than compared to UML. Thus, as can be seen, there are some DERCS key elements driving the code generation process: *(i)* `Class`; *(ii)* `Attribute`; *(iii)* `Method`; *(iv)* `Behavior`; *(v)* `Action`; *(vi)* `Aspects`; and *(vii)* `Joinpoint`.

Classes are the main elements in the code generation/aspects weaving process, due to their importance to the distributed embedded real-time system specification itself i.e. they represent structure and behavior of system objects, which, in turn, represent system elements. Therefore, GenERTiCA applies the code generation/aspects weaving algorithm to each class in the DERCS model classes list. For each class, its attributes, methods, as well as their associated behavior and actions list, are also used by the algorithm. As one can see, an activities execution pattern can be extracted (for each of these elements), representing this algorithm's kernel as follows:

1. Check if the element is affected by any model-level aspect adaptation, using pointcut and join points information. If it is the case, the adaptation is performed.

2. Try to find and execute the script that matches with the element being evaluated, e.g. if the element is a `AssignmentAction` the script in the *<Assignment>* is found and executed.

3. Check if the element is affected by any other aspect adaptation (i.e. those that modify the generated code fragment), and if so, execute all associated adaptations.

On the other hand, the platform configuration generation takes an inverse path, as shown in figure 6.13: the *<PlatformConfiguration>* branch drives the generation process, and the information on DERAF aspects is obtained from the model. Thus, for each *<File>* and *<Path>* node in this branch, at least one aspect in the associated aspects list must be found in the model to allow the generation of the file or the inclusion any of its parts.
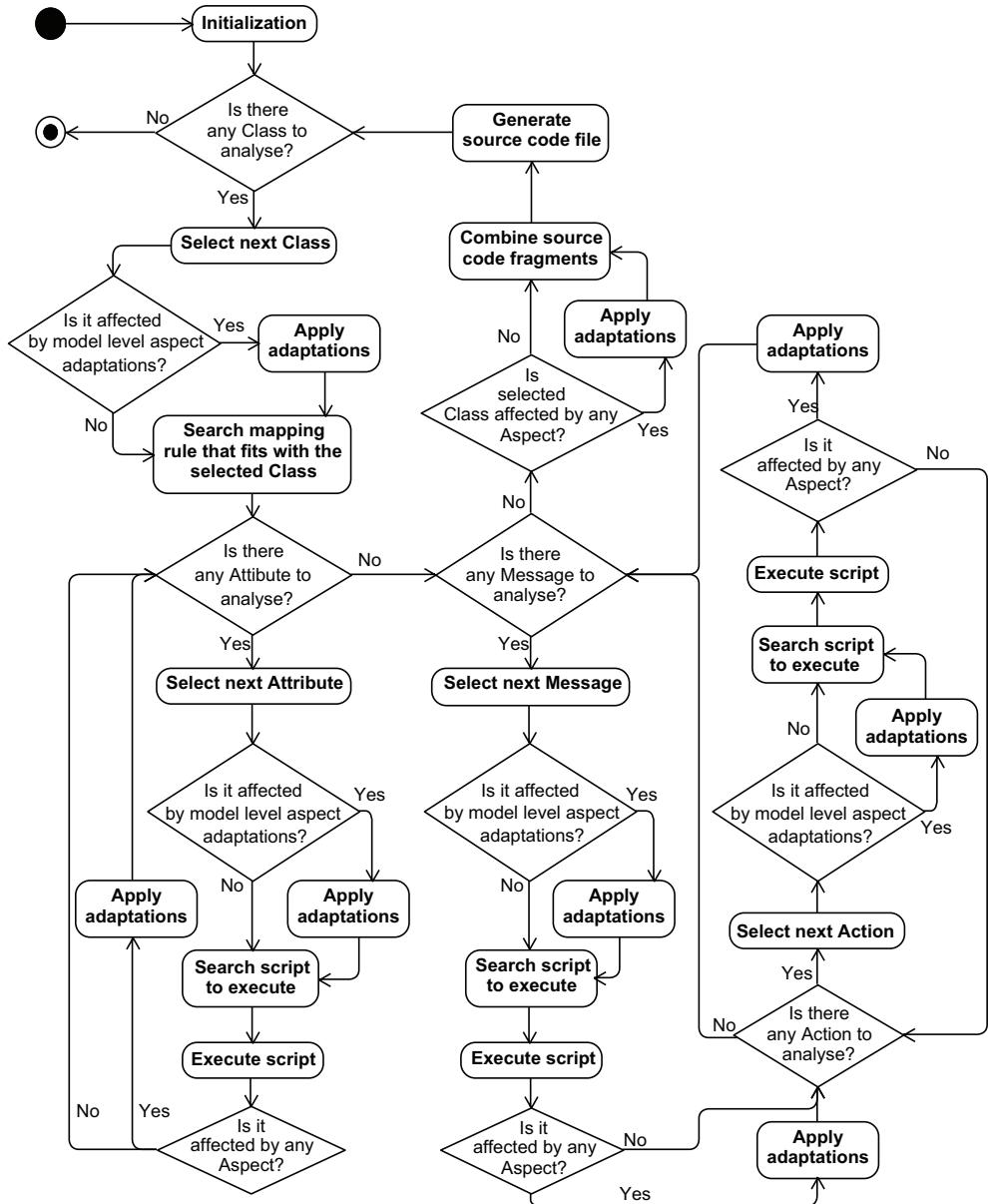
Figure 6.12.: GenERTiCA: application code generation flowchart
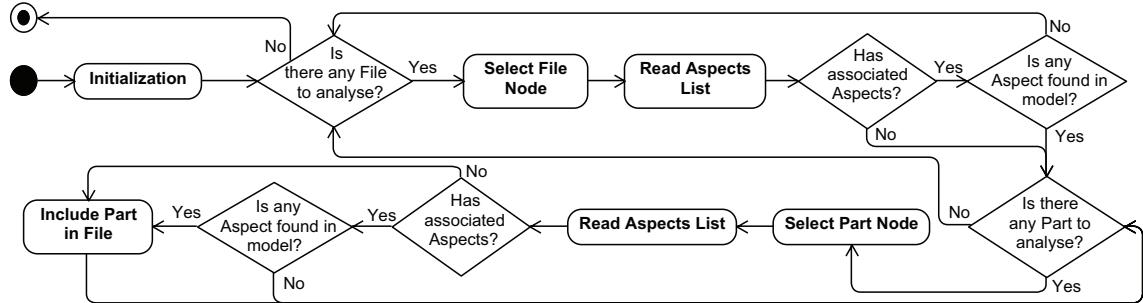
Figure 6.13.: GenERTiCA: platform configuration generation flowchart

6.6. Final Remarks

UML is not the most suitable modeling language to allow complete code generation, due to its various intentional semantic variation points. Therefore, GenERTiCA code generation approach transforms UML into the proposed DERCS model, trying to simplify the access to system specification information. Moreover, DERCS meta-model allows the same specification level as UML for describing structure and behavior, but using fewer meta-model elements, facilitating the mapping of model elements into constructions in the target platform language. However, more important is the separation of concerns in requirements handling specification provided by DERCS. DERCS AO concepts allow using AO-related elements at modeling level that can be further realized in both AO and non-AO target languages.

Regarding mapping rules description, the main reason for the choice of XML is because it represents a well-structured and self-described specification for organizing code generation scripts. Thus, other tools can use the mapping rules files for other purposes, such as evaluating different target platform constructions mapping rules to represent the same model element. Additionally, the XML format facilitates the creation of mapping rules repositories, from which tools can read their information to decide which file (or fragment) should be selected and reused.

GenERTiCA approach of using small scripts improves separation of concern in mapping rules specification because designers need only to take into account few elements for transforming a concepts in the model into constructions in the target language. Moreover, scripts allow aspects weaving at two levels: model and source code. In other words, aspects adaptations can modify both model elements and the source code fragment generated from them. Designers can choose the implementation form that better fits their needs. To the best of our knowledge, there is no other aspect weaving approach that allows both model and source code level aspects weaving.

Furthermore, aspect adaptation implementations are highly dependent on the target platform, i.e. the target platform must provide means to allow DERAF aspects semantics realization. However, it must be stated that this work does not intend to provide implementations for all aspects available in DERAF. Even so, the ones provided represent good examples on how

to implement other DERAF aspects. Besides, the proposed code generation/aspects weaving approach enables the use of both AO and non-AO languages as the target language in scripts specification, enabling more flexibility in the target platform selection.

Finally, it is important to highlight that, after code generation/aspects weaving process, separation in the handling of functional from non-functional requirements is missed, i.e. code representing non-functional requirements is intermixed with the code related to functional ones. However, it is not a problem since the RT-FRIDA approach (used in requirements specification) and also the mapping rules structure organization allow traceability in handling elements/construction from requirements to code, and vice-versa.

Chapter 7

Validation

7.1. Introduction

This chapter presents three case studies to illustrate and validate the AMoDE-RT approach, as well as the GenERTiCA code generation and aspects weaving tool. The first case study presents the movement control system of an unmanned aerial vehicle; the second one the control system of an industrial packing system; finally, the third one the movement control of an automated wheelchair. For each case study, two versions have been created: object-oriented and aspect-oriented. In addition, they have been compared using a subset of the software engineering metrics for AO systems presented in section 2.4.4.3. Mapping rules for two different platforms, namely the RT-FemtoJava and ORCOS platforms, have been specified to generate source code from the AO version of these systems examples.

7.2. Toolset Overview

In order to facilitate the understanding of the presented case studies, a brief description on the technologies used in implementation is presented. RT-FemtoJava and ORCOS platforms, which have been used to implement the AO version of these case studies, are presented. Thereafter, the assessment framework used to evaluate both versions of each case study is also presented.

7.2.1. RT-FemtoJava Platform

RT-FemtoJava platform is composed by a customizable Java processor [65], and a set of APIs to support real-time applications [131]. RT-FemtoJava processor implements a Java execution engine as hardware by means of a stack-based machine compatible with the Java Virtual Machine (JVM) specification [73]. Moreover, it adopts the Harvard organization, i.e. different memories for code (ROM) and data (RAM). There are different versions available for the RT-FemtoJava processor: 8, 16, 32-bits with different architectures (multicycle, pipeline, VLIW).

The choice of which version should be selected is made according to application requirements and constraints.

As RT-FemtoJava is a customizable processor, it is generated by the SASHIMI environment [65], which takes the compiled Java *bytecodes* as input to produce a VHDL description of the customized RT-FemtoJava, optimized for that Java binary code. In other words, SASHIMI analyses the compiled Java code, and automatically synthesizes an *Application Specific Instruction Processor* (ASIP), using only the instructions subset used by the target application. Hence, the synthesized processor control unit size is directly proportional to the number of different Java *opcodes* needed by the application software, optimizing the final footprint based on application requirements.

In addition to RT-FemtoJava processor, an API [131] based on the *Real-Time Specification for Java* (RTSJ) [13] was developed to facilitate application software development by raising the abstraction level of programming constructs. Thereby programmers do not have to worry about low-level details. This API covers the most important aspects of real-time programming like multithreading, real-time scheduling, and specification of timing issues. To clearly express timing and other constraints in the real-time application source code, this API introduces the concept of schedulable objects (i.e. active object), which are instances of classes that implement the RTSJ *Schedulable* interface (as *RealtimeThread* class). It also specifies a set of classes to store parameters that represent particular resource demands from one or more schedulable objects. For example, the *ReleaseParameters* class (super class from *AperiodicParameters* and *PeriodicParameters*) includes several useful parameters for the specification of real-time requirements, e.g. deadlines, activation period, and others. Moreover, it supports the expression of the following elements: *(i)* time values (absolute and relative time); *(ii)* clocks; *(iii)* periodic, sporadic and aperiodic tasks; *(iv)* scheduling policies; *(v)* timers; *(vi)* asynchronous events and their handlers; and *(vii)* pooling servers to minimize the disturbance caused by asynchronous events handlers execution. For details on the RTSJ-based API, interested readers are referred to [131]

As mentioned, RT-FemtoJava platform also has a communication API [34] that allows the establishment of a communication channel upon a network, in which objects residing in different processing nodes can exchange messages. Two communication models are supported by the communication API: client-server and publisher-subscriber. The former allows connection oriented and point-to-point communication, while the later connectionless and multicast communication. Additionally, distinct priorities and timing constraints can be associated with messages, improving the real-time constraints management. Moreover, the communication API is divided in transport, network, and data link layers, following the OSI/ISO reference model. In the current version, it implements a communication infrastructure following the CAN-bus [18] communication protocol.

7.2.2. ORCOS Platform

Organic Reconfigurable Operating System (ORCOS) platform provides a customizable RTOS, whose aim is to run it upon any kind of embedded hardware [125]. ORCOS implements a

fully object-oriented hybrid kernel (using C++ language), representing the evolution of the *DistRibuted Extensible Application Management System* (DREAMS) [38], a library-based construction set for operating system services. A remarkable feature of ORCOS/DREAMS is the ability to separate mandatory operating systems source code parts from optional ones by using a configuration mechanism. Thus, ORCOS can achieve small binary footprint using a configuration mechanism that uses an XML-based configuration language, named *Skeleton Customization Language* (SCL).

ORCOS kernel is divided in several independent modules, which are selected and integrated by means of the SCL. The following services are provided by ORCOS:

- **Memory management** is one of the most important modules, and is mandatory in any system configuration. There are separated memory spaces for each application task, and also for the ORCOS kernel. In addition, each task has its own memory manager, which is responsible for task's memory management strategy. Moreover, if the embedded hardware supports virtual memory, ORCOS is able to use it;

- **System calls** provide a manner for applications task to communicate with the ORCOS kernel. Thus, when a task needs to use any kernel functionality, it must use *syscall* API functions to trigger a hardware interrupt, which, in turn, is recognized by the kernel that executes the desired kernel functionality;

- **File system** uses the same approach as the Unix file system, i.e. all file system entries can be accessed through a unique path. Resources are registered inside the file system structure, and accessed through a POSIX-like set of kernel functions;

- **Processes**, in ORCOS, represent tasks and their set of executing threads. Real-time support is provided by ORCOS through real-time threads and real-time schedulers. Moreover, there are special kernel tasks/thread called *workerthreads* to support asynchronous interrupts for hardware devices I/O, timed functions calls, or periodic functions calls (likewise periodic threads but with the option of stopping function execution to allocate the *workerthread* to other purpose);

- **Scheduling** is other important module in ORCOS, and is divided in two steps: dispatching and scheduling. The later comprehends the set of rules to determine the order in which threads are executed, while the former executes instructions to allocate CPU to the thread selected to execute. The following scheduling policies are available: *Earliest Deadline First* (EDF), *Rate Monotonic* (RM), *Round Robin* (RR), and a priority-based scheduler;

- **Hardware Abstraction Layer (HAL)** provides an abstraction layer to access the real hardware, avoiding the ORCOS kernel to access it directly;

- **Power management** allows energy savings. The current ORCOS version allows only to halt the CPU execution every time the idle task is going to execute;

- **Communication** module allows inter-node and inter-process communication, defining a socket communication interface that uses different protocols to communicate. Each socket can be explicitly configured (at runtime) to define which protocol stack (OSI/ISO

reference model's transport and network layers) need to be used.

A detailed discussion on ORCOS is beyond the scope of this text. Thus, interested readers are addressed to [125] and [38].

7.2.3. Case Studies Assessment

Case studies presented in this chapter aim at assessing design improvements achieved by using the proposed AMoDE-RT design flow. Thus, for each case study, two versions have been designed: one using only OO concepts to specify both functional and non-functional requirements handling, and another one applying AO concepts to deal with non-functional requirements.

To compare the suitability of OO and AO concepts for distributed embedded real-time systems development, a set of quality metrics is calculated for each version. This work uses the assessment framework presented Sant'Anna et al. by [106] (see section 2.4.4.3) to infer the reusability quality of the produced UML models. Not all available metrics have been used to provide the qualitative assessment of both models. A subset of metrics had to be chosen and adapted based on their suitability for modeling instead of coding phase. Hence, implementation related metrics, such as *Lines of Code*, have not been used. Additionally, it is important to highlight that this assessment concentrates only on "reusability" instead of "reusability and maintainability" as proposed in the original assessment framework [106]. Figure 7.1 shows the selected metrics and their relations to provide the reusability quality assessment. For more details on each metric, readers are referred to section 2.4.4.3. In addition, to assist in metrics extraction, a plug-in for the MagicDraw modeling tool [85] has been developed and used to automatically calculate the metrics set.

Moreover, in order to be able to do a fair comparison between OO and AO models, the development of these models were done by two different persons, one person has modeled the OO version of all case studies, while the other one has modeled the AO version. The intention of this approach is to decrease the occurrence of any bias that may happen if the same person designs both versions of the same system.

Besides model assessment, other point evaluated in these case studies is the source code
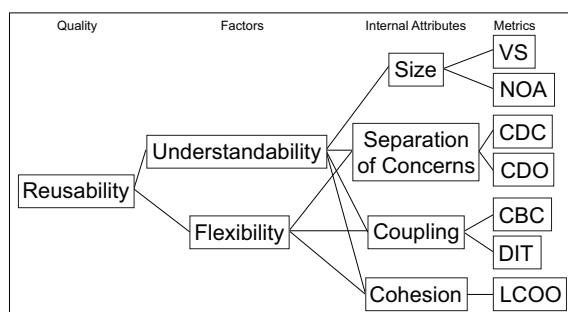
Figure 7.1.: Reusability quality model

generated from UML models. Statics about the amount of source code files, as well as generated lines of code, for each mentioned target platform are presented.

7.3. Case Studies

7.3.1. Unmanned Aerial Vehicle

Unmanned Aerial Vehicle (UAV) is an aircraft that flies without having an onboard pilot, and is used in activities where the human presence is avoided due to inherited risks, or simply to decrease operational costs. UAVs can fly a pre-programmed route or be remotely operated by a ground station. Reconnaissance support in natural disasters, monitoring and defect detection of transmission lines located in inhospitable places, and area surveillance and vigilance are some examples of UAV applications. An UAV is compounded of several subsystems, such as video recording and transmission, navigation, mission management, collision avoidance, self-diagnostic, and movement control.

This work focuses on the movement control subsystem of an unmanned helicopter, modifying the UAV movement control case study presented in [37] by means of providing a more detailed design. Summarizing, the helicopter control system is divided three different modules:

- **Sampling subsystem** is responsible to sample helicopter information (e.g. main and tail rotors pace), as well as environmental information (e.g. humidity, temperature, wind speed and direction). Sampled data have different utility lifetimes depending on the information kind, operation mode, and/or mission;

- **Control subsystem** uses the sampled data to control both helicopter main and tail rotors, allowing helicopter guidance and piloting. Basically, it implements a control system based on the method proposed by Seibel [112];

- **Actuation subsystem** takes the control values produced by the control subsystem, applying them in helicopter rotor engines.

Further, the helicopter control system has two interconnected real-time processing nodes: one is located close to the main rotor and the other one close to the back rotor. In other words, the designed control system is distributed over these two communicating nodes, employing both remote and local objects. For a complete description on this system's requirements, interested readers should refer to [37].

Figure 7.2 shows functionalities expected from the mentioned subsystem. According to AMoDE-RT modeling guidelines, functionalities affected by non-functional requirements (e.g. "Helicopter Movement Control") are decorated with non-functional stereotype annotations, e.g. such as «NFR_Timing». The following subsections provide more details on the modeled subsystem using AO and OO concepts.

123
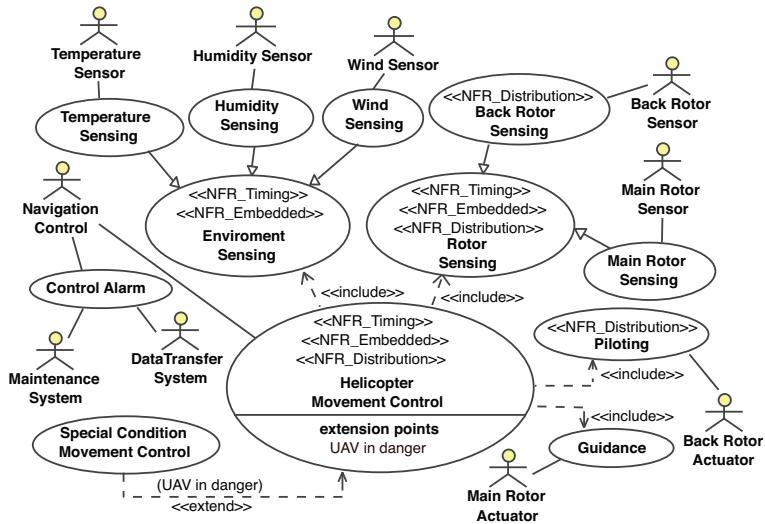
Figure 7.2.: UAV movement control use case diagram

7.3.1.1. Object-Oriented Version

The static structure of UAV movement control system is depicted with a class diagram. This diagram shows classes, their attributes and methods, and the relationships among classes. Figure 7.3-A depicts the UML class diagram created for the OO version. As suggested in AMoDE-RT modeling guidelines, classes representing active objects are decorated with the «SchedulableResource» stereotype from the MARTE profile, and passive objects with «MutualExclusionResource» stereotype.

Some classes depicted in figure 7.3-A (those with different filling color) are responsible to handle non-functional requirements as, for example, `Semaphore` class that is responsible to control the simultaneous access to shared passive objects, `Timer` that deals with timing requirements, or `EnergyController` that deals with energy consumption.

UAV control system's behavior was specified using only sequence diagrams, showing the behavior in terms of interactions among objects. Thirteen different sequence diagrams were created: *(i)* Helicopter movement control; *(ii)* Back rotor control; *(iii)* Main rotor movement encoder; *(iv)* Back rotor movement encoder; *(v)* Environment data acquisition; *(vi)* General Behavior *(vii)* General Behavior 2 *(viii)* General Behavior 3 *(ix)* Control Sub-System Initialization *(x)* Environment Sub-System Initialization *(xi)* Movement Sensing Sub-System Initialization *(xii)* Energy control; and *(xiii)* Task migration. Figure 7.4-A shows two fragments of the helicopter movement control sequence diagram: (A1) the start of `MovementController`'s periodic behavior responsible for controlling the helicopter movement, and (A2) the end of this active object method behavior.

In this diagram, the `Scheduler` object sends periodically an activation message (each 20 ms), which is annotated with the «TimedEvent» stereotype, to the MovementControl-
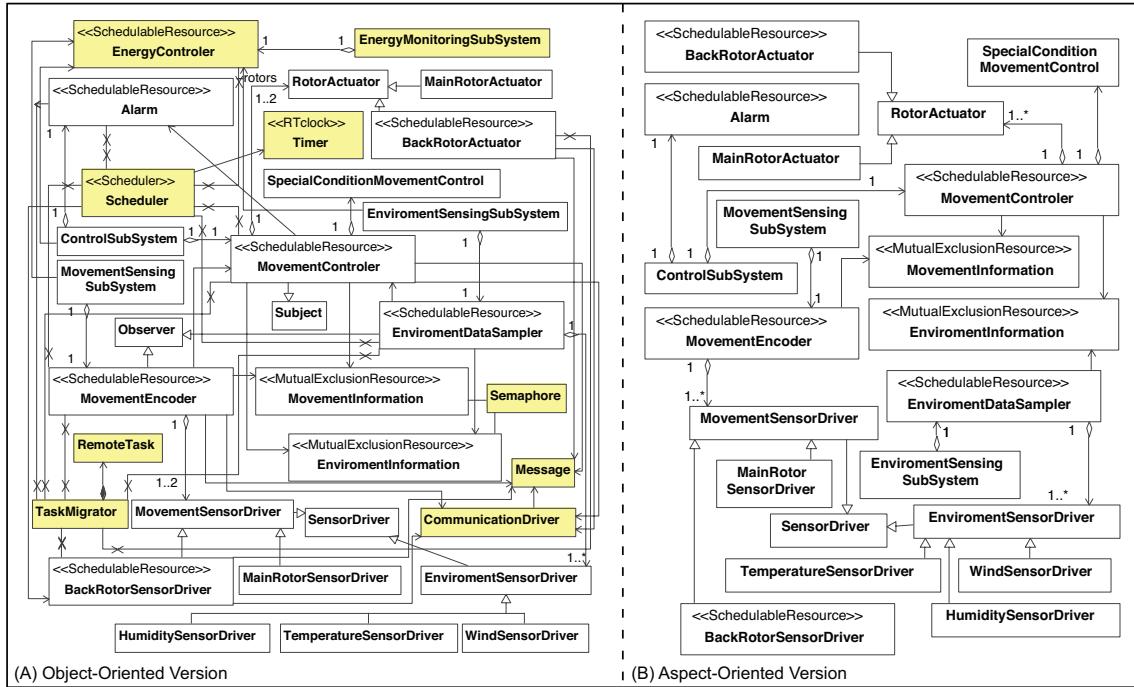
Figure 7.3.: UAV movement control class diagram
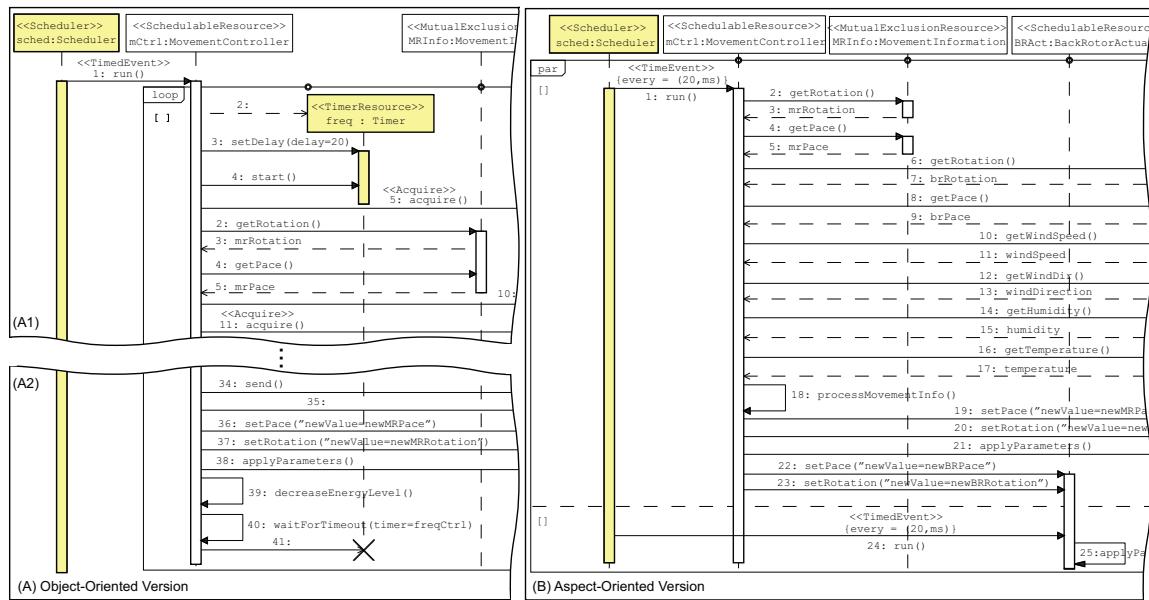
Figure 7.4.: Fragments of UAV movement control sequence diagram

`ler` object. A loop combined fragment, indicating the repetitive nature of the control task, encloses all performed actions. Timing and distribution requirements handling is performed by, respectively `Timer` and `Semaphore` classes (see figure 7.4-A1). Timer's timeout value is the value of the activation period assigned to `MovementController` object. At the end of the controller method (figure 7.4-A2), the execution is held until the timeout occurrence (message 40) to control the execution frequency. Figure 7.4-A1 also depicts the synchronized access (using a semaphore, as depicted in message 11) to `MovementInformation` object, whose attributes values are written by `MovementEncoder` active object, and read by `MovementController` active object. Therefore, before every access to the `MovementInformation` object, an permission must be requested, and after its use, the exclusive access must be released.

As stated before, the control system has one processing node at the main rotor and another one at the back rotor. The control task runs in the main rotor node while the back rotor actuation task runs in its own node. Thus, the movement control task must send the calculated actuation values to the back rotor node. Figure 7.4-A2 shows the handling of this communication non-functional requirement (messages 34 and 35), and also the application of calculated actuation values to the main rotor actuator. Furthermore, this diagram also shows other method related to control energy consumption (message 39).

7.3.1.2. Aspect-Oriented Version

The AO version uses DERAF aspects to specify the handling of non-functional requirements, i.e. the handling of each non-functional requirement is enclosed within the scope of a single element instead of being spread over several different elements.

Figure 7.3-B depicts the class diagram for the AO version. As can be observed, this diagram is simpler to visualize compared to the one in OO version, due to the elimination of classes that are not related with the application itself (i.e. classes that handle non-functional requirements). In other words, in AO version the handling of non-functional requirements is done using aspects from DERAF, which are specified in the ACOD. One may argue that the same visual simplification is achieved by means of separating functional from non-functional requirements handling classes into two different class diagrams. This claim is true, however, the use of aspects brings other advantages, such as a decrease in coupling among classes, reduction in the amount of model elements related to non-functional requirements handling, and others.

Considering the behavior specification, the number of required sequence diagrams was also reduced to nine. In AO version the following sequence diagrams of OO version have been eliminated: *(i)* Back rotor actuation; *(ii)* Back rotor movement encoder; *(iii)* Energy control; and *(iv)* Task migration. The last two diagrams (iii and iv) are not necessary anymore because the handling of energy control and task migration requirements have been delegated to, respectively, `EnergyControl` and `TaskMigration` aspects of DERAF (see section 5.3.2). Actions in the other two eliminated diagrams, i.e. (i) and (ii), were merged with, respectively, "Helicopter Movement Control" and "Main Rotor Movement Encoder" sequence diagrams. Figure 7.4-B shows the movement control diagram, which is equivalent to the same diagram in OO version.

As can be observed, all non-functional requirements handling elements have been removed, reducing considerably the size of diagrams in terms of number of messages (40% reduction) and lifelines.

Additionally, figure 7.4-B also shows the union of "Helicopter Movement Control" and "Back Rotor Actuation" sequence diagrams, which is represented by the parallel combined fragment ("par"), meaning that both interactions occur concurrently. Due to elimination of messages related to non-functional handling, only two messages remained from the original "Back Rotor Actuation". Thus, these messages were included into the "Helicopter Movement Control" sequence diagram (see messages 24 and 25) and, consequently, the "Back Rotor Actuation" could be eliminated in the AO version.

According to AMoDE-RT, DERAF aspects and join points are specified using a combination of ACOD and JPDD (see section 5.3). Figure 7.5-A shows a fragment of the UAV's ACOD, showing three aspects: `TimingAttributes`, `PeriodicTiming` and `Concurrency-Control`. The first two aspects insert new attributes to active objects classes (those annotated with «SchedulableResource» stereotype). Attributes related to deadline, priority and WCET are inserted by `TimingAttributes`, and the activation period by `PeriodicTiming` aspect. Values for this new attributes are specified in crosscut associations. It is important to emphasize that crosscutting associations do not insert by themselves new attributes into participating elements (class or aspect) as normal associations. Hence, they do not bind classes with aspects, and vice-versa.

As mentioned, the real link between aspects adaptations and affected elements (whose selection is specified with JPDDs) is specified by pointcuts description within aspects. Figure 7.5-B shows five examples of all JPDDs created in this case study:

1. `ActiveClass` join point (B1) represents the selection of all classes annotated with «SchedulableResource» stereotype;

2. `ActObjContructor` join point (B2) selects all actions that construct all active objects;

3. `PeriodicActivation` join point (B3) represents the selection of all messages, which are annotated with «TimedEvent» stereotype, sent by the scheduler to any active object;

4. `InfoObjCreation` join point (B4) selects all actions that construct passive objects (i.e. classes annotated with «MutualExclusionResource»), whose name ends with "Information";

5. `SetInfoValue` join point (B5) selects all messages with name starting with "set", which are sent to passive objects whose name ends with "Information".

As shown in figure 7.5-A, `PeriodicTiming` aspect uses JPDDs numbers 1–3, while `TimingAttributes` aspect uses only JPDDs numbers 1 and 2.

`ConcurrencyControl` aspect affects passive objects, which store information that can be simultaneously accessed by more than one active object. It assigns a concurrency control mechanism to each affected object during their instantiation, whose join point is captured by
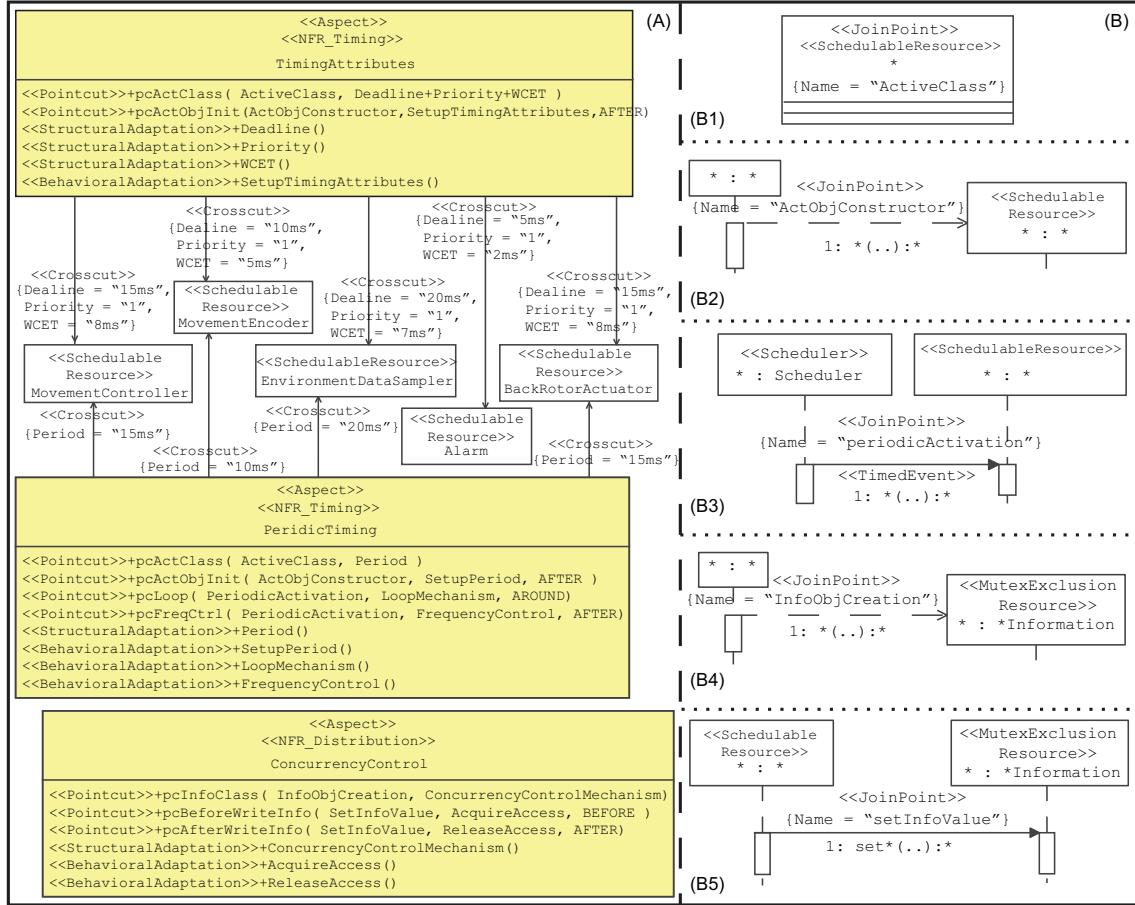
Figure 7.5.: UAV non-functional requirements handling: (A) ACOD, and (B) JPDD

the `InfoObjCreation` JPDD. Additionally, before every access (read or write) to a protected object, an access permission must be requested to this control mechanism. Similarly, the control mechanism should be informed that the object is no longer in use. Writing accesses are captured by `SetInfoValue` JPDD. It is important to highlight that according to DERAF premises of high-level aspects, at modeling level, it does not matter if this inserted control mechanism is a new attribute for each affected classes or simply an new global object, which is associated with the protected shared object. These are implementation specific issues, which should be decided at implementation phase.

As a first impression, one can think that the specification of ACOD and JPDD seems to require more effort but it is not true. The generic nature of JPDDs allows their re-use from previous modeled projects, as demonstrated in these case studies. Hence, many JPDDs have been simply reused without modification in the other case studies.

7.3.1.3. Results

Considering separation of concerns metrics, figure 7.6-A shows how effective was the application of DERAF aspects to handle time, distribution and embedded concerns. All non-functional requirements have better handling separation in the AO model compared to the OO one, i.e. the smaller amount of elements (classes and/or aspects) handling a concern, better separation of concerns is achieved, leading to a decrease in the scattering problem. The numbers presented confirm the simplification observed in the diagrams of AO version. The reduction ranges from 55% to 83% for the CDC and from 75% to 92% for the CDO metric. CDC/CDO became smaller in AO version because the way they are calculated (see section 2.4.4.3). For instance, in AO version, CDC for timing non-functional requirements considers only the following DERAF aspects: `PeriodicTiming`, `SchedulingSupport`, `TimingAtributes` and `TimePa- rametersAdapter`. On the other hand, in OO version, CDC takes into account classes specifically related to timing non-functional requirements handling (`Scheduler` and `Timer`) plus those related to functional requirements, which also deal with time issues (`MovementController`, `MovementEncoder`, `EnvironmentDataSampler`, `Back-RotorSensorDriver`, `BackRotorActuator`, `Alarm` and `EnergyController`). Consequently, in OO model, functional and non-functional requirements handling intermixing cause the inclusion of some functional elements/methods as non-functional elements/methods.

Considering the other metrics, figure 7.6-B depicts the results obtained. Analyzing coupling metrics, DIT results show that the use of aspect did not modified the inheritance tree. CBC results show, again, a decrease of more than 55% in the AO model. CBC takes into account each reference (e.g. attribute, method call, parameter) to other classes/aspects. Consequently, classes/aspects in AO version are more modular than in OO version, mainly, due also to the intermixed treatment of functional and non-functional requirements that happens in OO version. Observing the size metric, VS did not change, while NOA has a decrease of 52%. This happened because several non-functional-related attributes were moved from classes to aspects, which are woven into all affected classes in implementation phase.

(A) Separation of concerns metrics   (B) Coupling, Cohesion and Size metrics
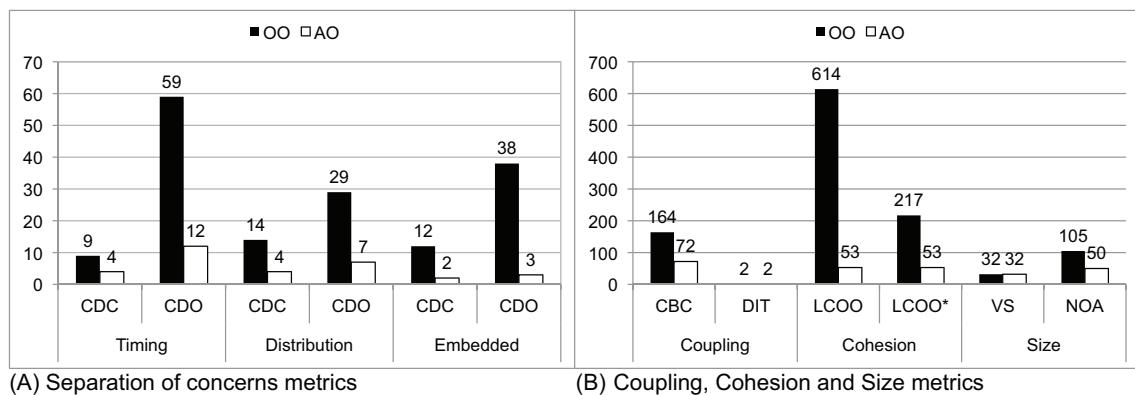
Figure 7.6.: Calculated metrics for the UAV control system

<<SchedulableResource>> **MovementController_OO**	<<SchedulableResource>> **MovementController_AO**
. . .	. . .
+getPriority() +setPriority() +getDeadline() +setDeadline() +getPeriod() +setPeriod() +getWCET() +setWCET() +getEnergyLevel() +decreaseEnergyLevel() +resetEnergyLevel() +waitForTimeout(timer) +changeControlPolicy() +run() : void +processInfo(r1:int, p1:int, r2:int, p2:int, ws:float, wd:float, h:float, t:float) : void +getLastValidComputation()	+run() : void +processInfo(r1:int, p1:int, r2:int, p2:int, ws:float, wd:float, h:float, t:float) : void +getLastValidComputation()

Figure 7.7.: Comparison of UAV's `MovementController` classes

Regarding cohesion, the difference of LCOO between AO and OO models is more than 91%. This decrease is primarily caused by elimination of *get/set* methods for attributes related to non-functional requirements handling. To illustrate this difference, figure 7.7 presents `MovementController` class for OO and AO versions. Moreover, LCOO metric does not distinguish the two kinds of *get/set* methods: (i) "raw" which have minimum impact on real cohesion; (ii) with computations, which have significant impact on real cohesion. As one can see, OO version's `MovementController` has nine "raw" *get/set* methods related to non-functional requirements. Therefore, to provide a fair assessment, LCOO for OO has been re-calculated excluding "raw" *get/set* methods (LCOO* in figure 7.6-B). Even in this situation, LCOO decrease is 75% in AO model. The obtained results show that using aspects improves model cohesion.

Besides the modeling approach, AMoDE-RT also supports code generation using the produced UML model as input. Mapping rules for two different platforms, i.e. RT-FemtoJava and ORCOS, have been created and used to produce the system source code in Java and C++. Tables 7.1 and 7.2 present some statistics about, respectively, the UAV movement control UML model and the generated source code.

Table 7.1.: UAV: Statistics of the UML model of AO version

Diagrams	**Amount**
Structural	2
Behavioral	10
ACOD	1
JPDD	16
DERAF aspects	10
Structural adaptations	15
Behavioral adaptations	18

The size of mapping rules XML files, in terms of code lines, is 803 lines for the RT-FemtoJava, and 749 lines for ORCOS platform. However, if the lines related to XML markup are not considered, the amount of lines for the mapping rules script is 388 and 332, respectively. These scripts represent the *Application* branch. ORCOS mapping rules file is smaller than the RT-FemtoJava one due to the later has more aspect adaptations scripts than the former. In fact, in ORCOS platform, several expected behaviors of aspect adaptations (e.g. `PeriodicTiming`'s `LoopMechanism` and `FrequencyControl`) are implicitly executed within platform context, and hence, they do not need extra lines of code in application code, only the correct configuration to enable such behavior. Furthermore, as some aspects used in the UML model are not supported by both target platforms, their adaptations insert only a comment (one line) indicating that aspects have performed adaptations in generated source code fragments. Despite not implementing real code, these scripts serve as demonstration of aspects weaving performed by GenERTiCA.

Considering generated platform configuration, the difference in amount of lines is even greater. RT-FemtoJava configuration has 2931 lines, and ORCOS 480 lines. This happened because ORCOS has already a configuration mechanism, and RT-FemtoJava none. Hence, on one hand, a configuration file is generated for ORCOS, and on the other hand, the entire RT-FemtoJava API code is tailored to include only lines that provide services required by aspects. In other words, by using this RT-FemtoJava configuration approach, GenERTiCA provides a preprocessor mechanism to Java, likewise the one natively supported in C/C++.

Regarding the generated source code for the UAV movement control system, all behaviors/actions specified with sequence diagrams have been mapped to source code constructions. Thus, for the RT-FemtoJava platform, the amount of application code lines is almost 3.3 times the amount of mapping rules scripts lines. For ORCOS, the rate is almost 4.2 times, demonstrating the effort reduction in application coding, and the importance of code generation tools in MDE approaches. In the other case studies, the mapping rules have been used without modification, and hence, the effort to obtain system implementation was even lower than in this case

Table 7.2.: UAV: Statistics of the generated source code

	RT-FemtoJava[+]	ORCOS
Mapping Rules (lines)	388/803	332/749
Application		
Source code files	21	42
Lines of Code	1264	1385
Binary Size (Kb)	5.41 (32)	147
Platform		
Source code files	21/38*	01/01
Lines of Code	2931*	480
Binary Size (Kb)	5.93 (50)	462

+ Numbers inside parentheses represent the bytecodes size generated by java compiler

* Considering RTSJ API [131] + API COM [34]

study.

Considering the size of compiled source code, a considerable difference between both platforms can be perceived. This is caused by the difference between instructions size of both target processors. ORCOS code has been compiled to a 32-bits PowerPC processor, whose instructions size is 4 bytes. On the other hand, RT-FemtoJava is a hardware implementation of the JVM specification and, according to Lindholm and Yellin [73], the size of JVM instructions is 1, 2 or 3 bytes. Consequently, PowerPC's binary size could be (in the worst-case) almost four times greater than RT-FemtoJava's binary size.

7.3.2. Industrial Packing System

This case study was inspired on the packing system presented in [60] and [19]. The system is composed of a robotic arm with a gripper, two conveyors, a storage unit and several sensors. The input conveyor brings individual parts, which are combined to form products. The conveyor stops when the sensor detects the presence of a part. Then the robotic arm will either put it in the storage unit or use it to assembly a product. The second conveyor brings empty boxes into which parts are inserted. This conveyor remains operating until its sensor detects an empty box at the expected position. When the product is completely assembled, the controller sends a command to the conveyor and it starts to move forward again. The controller is a periodic active object that verifies whether there are products to be assembly and/or parts to be place into the storage unit. When the new product requires a part, which is physically located at the parts conveyor, this part is taken from there, and used to mount the product; otherwise, the part is taken from the storage unit. This system was intended to be distributed, i.e. there are four different processing nodes: one responsible to control the products assembly process and the robotic arm; two nodes to control, respectively, the input parts conveyor and the assembled products output conveyor; and one to control the amount of parts in the storage unit.

The discussion starts exactly as in the UAV case study: firstly, the packing system func-
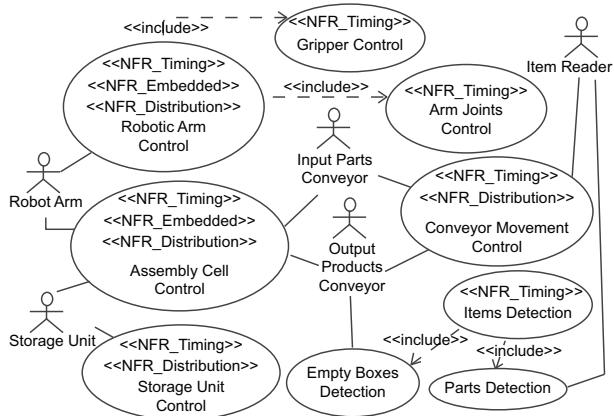
Figure 7.8.: Industrial packing system use case diagram

tionalities are specified with an use case diagram, as depicted in figure 7.8. Once again, one can see that several use cases are annotated with stereotypes related to non-functional requirements. Although this case study has more use cases decorated with "NFR_" stereotypes, it has fewer non-functional requirements. More specifically, there are fewer embedded non-functional requirements (if compared to the UAV case study), due to system size and absence of energy constraints in products packing system.

This case study have been performed as previous the previous one: i.e. object- and aspect-oriented modeling approaches are shown and compared with the assessment framework. Source code have also been generated for the two mentioned platforms, and their statistics are presented.

7.3.2.1. Object-Oriented Version

Following the approach adopted in the UAV case study, the static structure of the packing system is specified using the class diagram, as depicted in figure 7.9-A. The same MARTE profile stereotypes have been used, i.e. «SchedulableResource» and «MutualExclusion-Resource», to specify, respectively, active and passive objects. Non-functional requirements related classes are also emphasized with different colors.
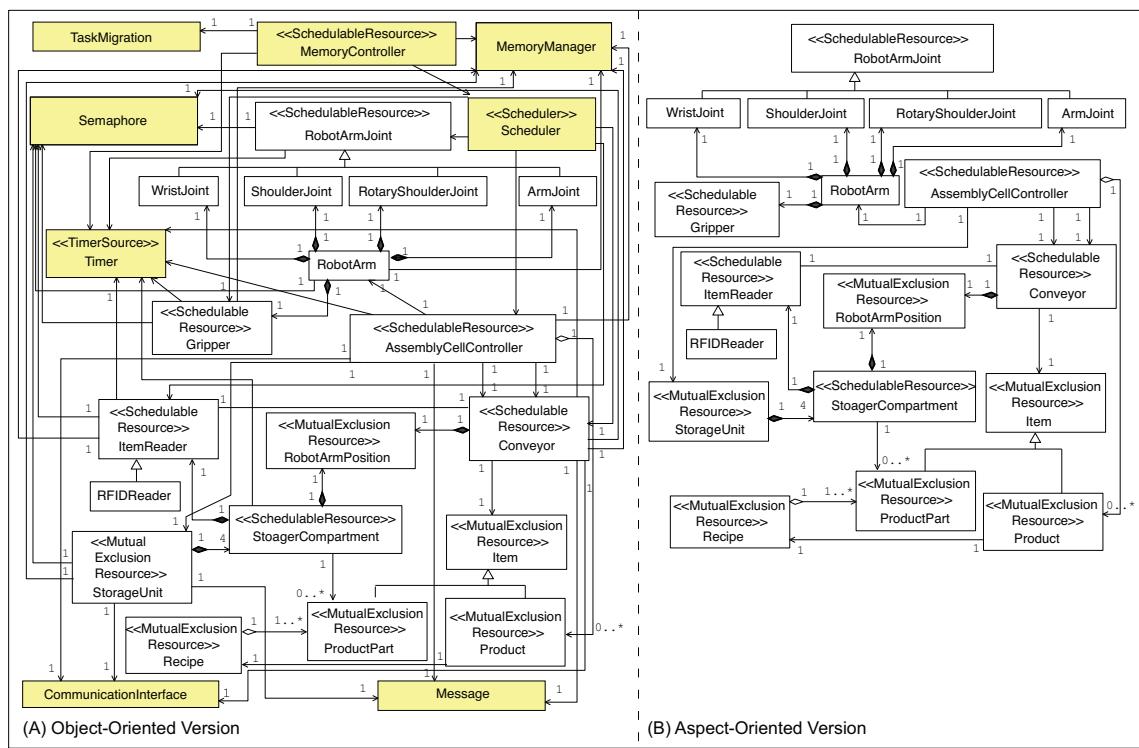
Figure 7.9.: Industrial packing system class diagram

For behavior specification, eleven sequence diagrams have been created: *(i)* Products assembly control; *(ii)* Conveyor control; *(iii)* Item detection; *(iv)* Robotic arm joints control; *(v)* Gripper control; *(vi)* Robotic arm movements control; *(vii)* Storage unit control; *(viii)* Controller sub-system initialization; *(ix)* Conveyor sub-system initialization; *(x)* Storage unit sub-system initialization; and *(xi)* Memory management and tasks migration. Figure 7.10-A shows two fragments of (i), as in the UAV case study: (a) the start of products assembly control execution, and (b) the end of this behavior. Repeating the UAV case study modeling pattern, the scheduler object sends a periodic message (annotated with «TimedEvent») to the AssemblyCellController active object, triggering its behavior execution at each 5 seconds. Classes specifying non-functional requirements handling can also be seen in these fragments, e.g. Timer and Message classes.

As stated before, active objects are spread into four nodes. The main control task (i.e. AssemblyCellController object) runs in the main node, and must access information from conveyors and storage objects located in other nodes. Thus, this object must send messages to these other objects, in order to collaborate with them to proceed with the products assembly and parts storage. For instance, figure 7.10-A shows the message sending that requests the position of the storage unit, into which a part should be placed (messages 45-48). Additionally, other messages related to other non-functional requirements handling, e.g. the memory control (message 51 and 55), are also depicted.
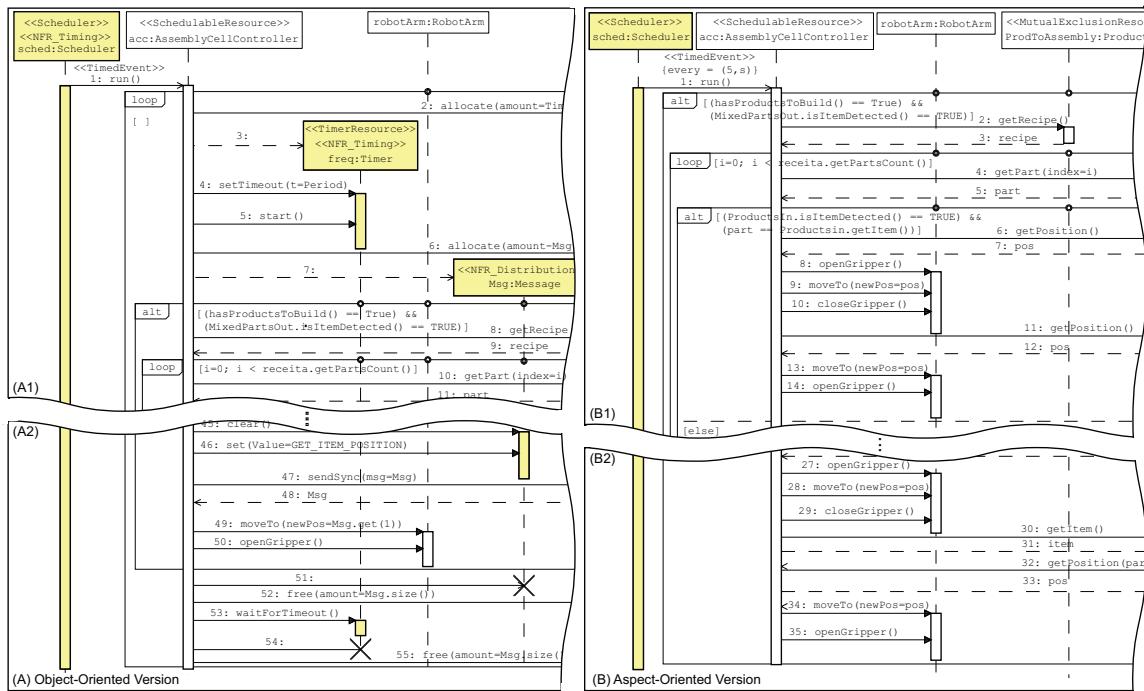
Figure 7.10.: Industrial packing system sequence diagram

7.3.2.2. Aspect-Oriented Version

DERAF was also used in the AO version of the industrial packing system to specify non-functional requirements handling. Figure 7.9-B shows the class diagram specifying the static structure. Again, the amount of classes, as well as the relations among them, have decreased. Thus the complexity of this diagram decreases in AO version, just like in the previous case study. Hence, the same statements made in the previous section hold in this case study.

In AO version, system behavior has been specified using less diagrams than the OO version, however, for the industrial packing system case study, the decrease happened only in one case: "Memory control and tasks migration" was removed due to the handling of memory control and task migration requirements to be delegated to, respectively, `MemoryUsageControl` and `TaskMigration` aspects. Figure 7.10-B shows two fragments of the products assembler control diagram, which are equivalent to those presented in figure 7.10-A. The amount of messages in this diagram is 36% smaller than the OO version one. As happened in the UAV case study, the complexity decrease for describing the same system features can be clearly perceived.

Considering the specification of non-functional requirements handling, figure 7.11 shows a fragment of ACOD, showing the reuse of `TimingParameters` and `PeriodicParameters` aspects. Not only aspects have been reused. As one can see, join points (i.e. JPDD)
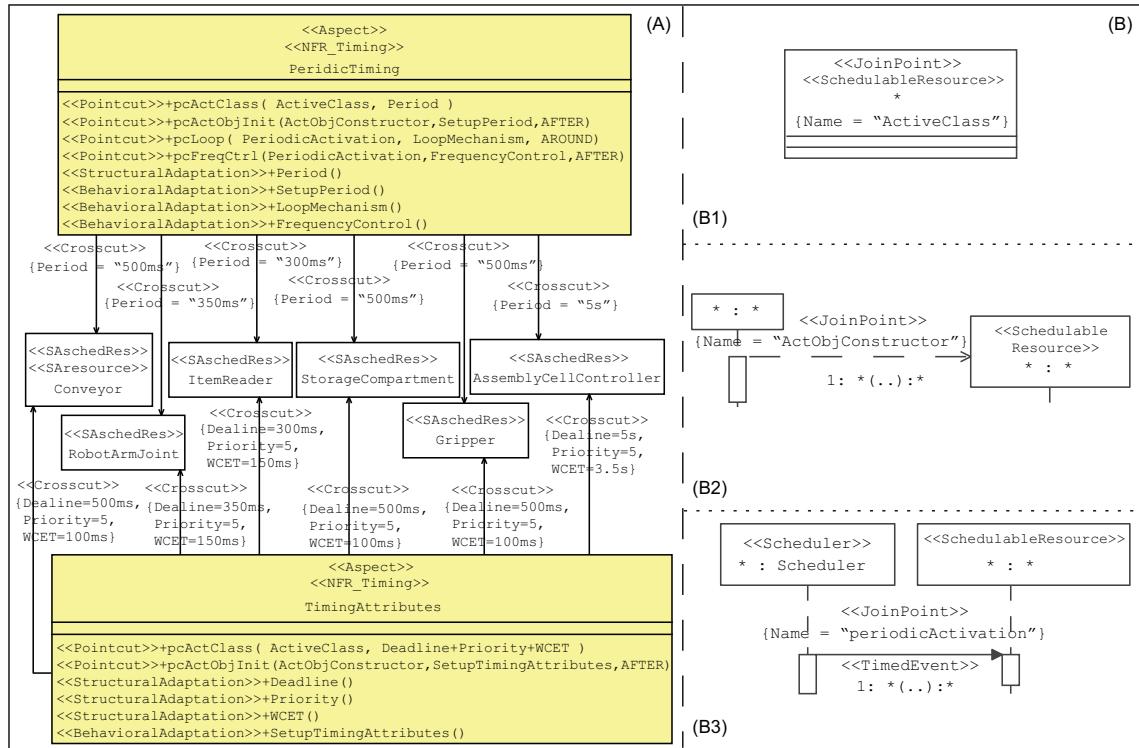
Figure 7.11.: Industrial packing system: reused elements in (A) ACOD, and (B) JPDD

used in pointcuts specification are also reused from the UAV case study. This shows the generality of DERAF and the AMoDE-RT approach, demonstrating that aspects can be reused at modeling level in different distributed embedded systems designs, due to their high-level semantics. It is important to highlight that, for aspects implementation, the mapping rules have also been reused without modifications, due to the fact that the target platform is the same, and the implementation follows the high-level pre-defined semantics of each aspect.

7.3.2.3. Results

Separation of concerns metrics depicted in figure 7.12-A shows that the same improvement achieved in UAV case study is obtained in the AO version of the industrial packing system. CDC metrics have been reduced at least 66% up to 81%, while CDO from 16% up to 75%. Although there is an improvement of concerns separation, it was not in the same degree as in the UAV case study, due to the amount of timing non-functional requirements present in both systems, as for example freshness requirements in UAV case study that do not exist in the industrial packing system.

The other metrics exhibit similar improvements.: *(i)* DIT did not change, i.e. aspects do not modify classes hierarchy; *(ii)* AO model is again more cohesive, as pointed by the decrease of 47% in CBC metric; *(iii)* VS in AO version indicates a small increase (i.e. one element), because in this version memory requirements are handled by two elements (`MemoryUsageControl` and `MemoryUsageMonitoring` aspects) instead of one element as specified in OO version; *(iv)* NOA, on the other hand, decrease almost 46% in AO version, showing that in spite of the increase in VS metric, the number of classes' internal elements has decreased. Regarding model cohesion, in AO version, LCOO decreases 58% when considering all kinds of methods, and 16% if "raw" *get/set* methods are excluded, showing that, in spite of the good cohesion in the OO version, the use of AO concepts improve system cohesion.

Considering the created UML model, and the implementation generated from it, tables 7.3

(A) Separation of concerns metrics   (B) Coupling, Cohesion and Size metrics
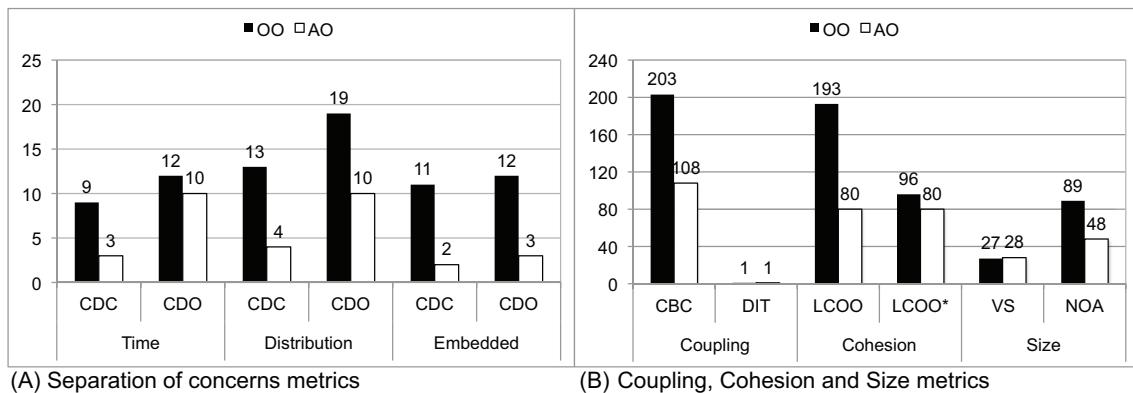
Figure 7.12.: Calculated metrics for the industrial packing system

Table 7.3.: Industrial packing system: Statistics of the UML model of AO version

Diagrams	Amount
Structural	2
Behavioral	12
ACOD	1
JPDD	13*
DERAF aspects	9
Structural adaptations	14
Behavioral adaptations	15

* 11 JPDDs have been reused from UAV

Table 7.4.: Industrial packing system: Statistics of the generated source code

	RT-FemtoJava[+]	ORCOS
Mapping Rules (lines)	388/803	332/749
Application		
Source code files	22	42
Lines of Code	1144	1343
Binary Size (Kb)	4.64 (29)	139
Platform		
Source code files	21/38*	01/01
Lines of Code	2931*	480
Binary Size (Kb)	6.12 (50)	462

+ Numbers inside parentheses represent the bytecodes size generated by java compiler

* Considering RTSJ API [131] + API COM [34]

and 7.4 show the statistics on the produced artifacts. In this case study, the reuse of previously created artifacts is highlighted in both AO-related elements specification and mapping rules. Considering the former, in addition to DERAF aspects reuse, JPDDs also have been reused. From the 13 JPDDs used in this case study, 11 have been reused from the UAV case study without any modification. The same happened with the mapping rules specification. Thus, none effort was necessary to generate 1144 and 1343 source code lines for, respectively, RT-FemtoJava and ORCOS platforms.

7.3.3. Wheelchair Automation

The third case study was an automation system for an electric-actuated wheelchair. Therefore, AMoDE-RT approach has been applied in the wheelchair's movement control system. Summarizing, the wheelchair movement control includes two engines (one for each wheel), a joystick to steer the wheelchair in terms of speed and direction, and two sensors to sample wheel rotation speed. Therefore, the system must perform the following concurrent activities:

- To sample the wheel sensors every 10 milliseconds to determine the movement speed and direction;

- To sample the current joystick position at the same period, i.e. 10 ms;

- To perform the wheelchair control algorithm every 50 ms, applying the calculated actuation value to left and right wheel engines;

- To monitor changes in the operation mode. The operation mode influences in the way of deadlines misses are treated: *(i)* signal the occurrence of missed deadlines; *(ii)* signal the occurrence of missed deadlines, and apply the last valid actuation value; or *(iii)* signal the occurrence of missed deadlines, and stop the wheelchair movement.

UML models of this case study have already been presented, as well as discussed, in [131] (OO version) and [37] (AO version). Thus, in this text, the goal of this case study is to discuss

(A) Separation of concerns metrics   (B) Coupling, Cohesion and Size metrics
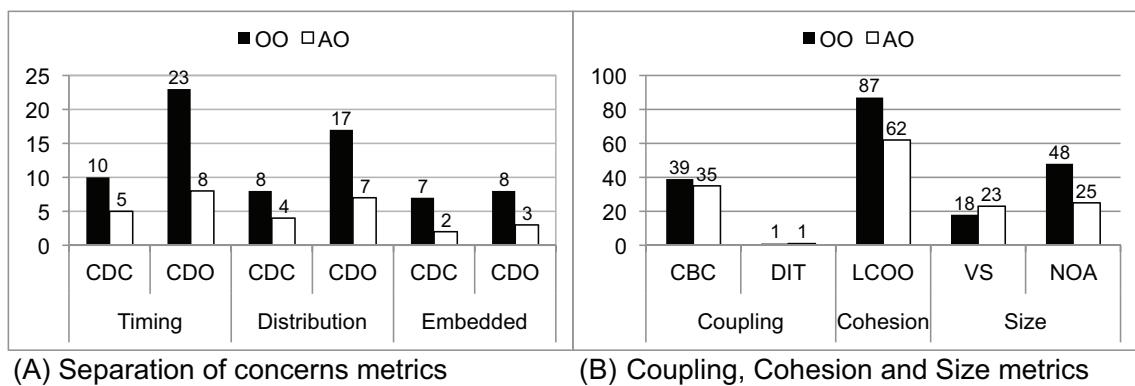
Figure 7.13.: Calculated metrics for the wheelchair movement control system

the calculated metrics for both version, and in addition, present results concerning the generated source code from the UML model of AO version. Moreover, this case study is slightly different from the one presented in [37] because AMoDE-RT modeling guidelines require some small modifications in diagrams, in order to allow code generation from the produced UML model.

Figure 7.13 shows the calculated metrics for the wheelchair case study. As occurred in the other cases studies, AO version has increased the separation of concern: CDC decreased 50% for timing and distribution and 71% for embedded non-functional requirements, and CDO from 58% to 65%. As it can be noted, comparing with the other case studies, the improvement obtained for CDC metric in this case study is not the same one obtained in UAV and industrial packing system. This situation happens due to the fewer amount of non-functional requirements that exist in wheelchair case study compared to the ones present in the other case studies. This can be also confirmed by CBC metric that decreased only 10%. In addition, the lower CBC absolute value in both versions indicates that the wheelchair movement control case study has a good degree of decoupling. Combining the later metric value with the lower LCOO absolute value, it can be stated that both OO and AO versions are well designed, due to, mainly, its small size in comparison with the other case studies. Moreover, one can see that, although NOA decreases almost 48%, the VS metric increases 27% in the AO version. As in the industrial packing system, the reason for this increase is that there are no one-to-one relation between non-functional requirements handling classes in OO version and aspects in AO version. Hence, as the system size is small, the relative impact of extra elements is greater than in the other case studies. However, as the other metrics have a significant improvement, this increase in vocabulary size is still acceptable.

Compared to the wheelchair case study presented in [37], the real contribution of this case study is the generation of source code. As the other two case studies, two tables presents some statistics: table 7.5 considers the produced UML model for the AO version; and table 7.6 considers the source code generated for the RT-FemtoJava and ORCOS platform.

Again, this case study emphasizes the reuse of previously created artifacts. In this case study, 100% of JPDDs could be reused from the UAV case study. Considering that both DERAF aspects and JPDDs have been reused, one can state that, by using AMoDE-RT, designers need

Table 7.5.: Wheelchair: Statistics of the UML model of AO version

Diagrams	**Amount**
Structural	1
Behavioral	6
ACOD	1
JPDD	14*
DERAF aspects	5
Structural adaptations	10
Behavioral adaptations	12

* all JPDDs have been reused from UAV

Table 7.6.: Wheelchair: Statistics of the generated source code

	RT-FemtoJava[+]	**ORCOS**
Mapping Rules (lines)	388/803	332/749
Application		
Source code files	12	24
Lines of Code	672	712
Binary Size (Kb)	2.81 (19)	133
Platform		
Source code files	13/30*	01/01
Lines of Code	1021*	480
Binary Size (Kb)	3.00 (28)	462

+ Numbers inside parentheses represent the bytecodes size generated by java compiler

* only RTSJ API [131]

only to concern with functional requirements specification, letting non-functional requirements handling specification to be composed by already created elements. Moreover, as experience is acquired in the development of others projects, the model elements repository grows in amount of elements, increasing the possibility of reusing more elements.

Considering the wheelchair control system implementation in this case study, GenERTiCA has generated 672 and 712 source code lines for, respectively, RT-FemtoJava and ORCOS platforms. Rates of generated source code lines per mapping rules scripts lines are 1.73 and 2.14, respectively. As one can conclude, implementation gains in this case study are not the same as the other cases studies mainly due to its size, i.e. wheelchair case study has 12 classes while the UAV and the industrial packing system have 22 classes each. In the same sense, this case study specified 6 behavioral diagrams, while the other ones 10 and 12 different diagrams each. However, in wheelchair case study, no additional mapping rules had to be defined, i.e. as the mapping rules files have already been specified in the first case study, it was necessary only to reuse them without modifications to obtain the mentioned amount of source code files.

7.4. Final Remarks

Taking into account the results obtained for all case studies, it can be stated that the use of AO concepts improves the reusability quality, even for small embedded real-time systems, as the case of the wheelchair movement control system. Almost all metrics have better values for AO model compared to OO one, ranging from 37% to 66% in average. Considering the understandability factor, key issues such as separation of concerns, cohesion and coupling improved around 45% in average. Although the number of components has increased a little bit (10% in average), the number of attributes decreased ca. 48%. For flexibility factor, AO model elements are more cohesive and decoupled compared to OO model. Separations of concerns results show that elements in AO model have more specific and well-defined roles than in OO model.

The difference in the absolute metrics values leads to the conclusion that improvements achieved with the use of AO concepts increase with the number of crosscutting non-functional requirements. Additionally, these case studies' metrics confirm that, using AO, the same benefits achieved in traditional information systems can be obtained in the design of distributed embedded real-time systems.

Further, as one can see in table 7.7, using DERAF aspects at modeling level allows their reuse in different designs. If the implementation follows the aspect adaptations high-level semantics, the aspects implementation can also be reused, as occurred in all presented the case studies. AMoDE-RT approach to specify join points selection also allows the reuse of JPDD (52% of all created JPDDs have been used in all case studies). However, it is worth to comment that JPDDs must specify generic selection of elements (e.g. `JPDD_ActiveObjectClass` or `JPDD_PeriodicBehavior`) to allow their reuse. Usually, JPDDs selecting specific elements (e.g. `JPDD_InfoAttributeRead`) are harder to reuse, due to their close relation with application specific elements (the mentioned JPDD, select attributes of classes whose name ends with "Information").

Considering the use of GenERTiCA, it must be stated that the amount of generated code is directly proportional to mapping rules scripts and diagrams specification completeness. In other words, if the UML model can provide complete information about system structure and behavior (following AMoDE-RT modeling guidelines), and mapping rules specification can map all elements available in the model into constructions available in a given target platform, it is likely that GenERTiCA can generate a large amount of source code. Considering the source code generated in presented case studies, one can see that it is possible to generate an amount of source code lines from 1.73 to 4.2 times the amount of mapping rules scripts lines [1].

Regarding the generated source code, source code files obtained after the code generation process are more complete than the ones obtained using available commercial or academic code generation tools, which usually only provide class skeletons and/or simple state machine related code. In addition, the aspects weaving performed by GenERTiCA allows the use of aspect adaptations in non-AO languages. Even considering these advantages, it must be highlighted that the generated code is not complete. There are several small issues that are highly dependent on the target platform, which cannot be solved using general approaches like GenERTiCA's one. For example, in all case studies, there is a need of filling the gap between the software objects representing hardware components (e.g. `WindSensorDriver` in UAV case study) and the real hardware. This kind of code is too specific to be specified in UML models or mapping rules, implying unnecessary details for a single element. Thus, programmers must code manually the corresponding methods in the generated source code files. Other example of platform-specific problems is the circular cross-reference problem in C++ source code files. This situation has occurred in the performed case studies, due to GenERTiCA's approach to specify references, which is strongly based on the Java language. A solution for this problem would be to pre-declare referenced classes inside class source code files. Thus, GenERTiCA code generation

---

[1]Mapping rules script's amount of lines for RT-FemtoJava and ORCOS platforms are, respectively, 388 and 332 lines. These numbers only represent script lines without considering XML marks, which, in fact, do not influence code generation

Table 7.7.: AO elements reused in the different case studies

	UAV	IPS*	Wheelchair
DERAF aspects			
TimingAttributes	X	X	X
PeriodicTiming	X	X	X
TimeBoundedActivity			
SchedulingSupport	X	X	X
Jitter			
DataFreshness	X		X
ToleratedDelay			
ClockDrift			
ConcurrentAccessControl	X	X	X
MessageSynchronization	X	X	
MessageAck	X	X	
MessageIntegrity			
MessageCompression			
TaskMigration	X	X	
NodeStatusRetrieval			
HwAreaMonitoring			
HwAreaControl			
EnergyMonitoring	X		
EnergyControl	X		
MemoryUsageMonitoring		X	
MemoryUsageControl		X	
JPDD			
JPDD_ActiveObjectClass	X	X	X
JPDD_ActiveObjectConstruction	X	X	X
JPDD_ActiveObjectConstruction_Action	X	X	X
JPDD_ActiveObjectConstructor	X	X	X
JPDD_ExclusiveGet	X	X	X
JPDD_ExclusiveObjectClass	X	X	X
JPDD_ExclusiveSet	X	X	X
JPDD_InfoAttributeRead	X		X
JPDD_InfoAttributeWrite	X		X
JPDD_InfoClassAttribute	X		X
JPDD_InfoObjectConstruction_2	X		
JPDD_InfoObjectConstruction_Action	X		X
JPDD_ObjectConstruction_Action		X	
JPDD_ObjectDestruction_Action		X	
JPDD_PeriodicBehavior	X	X	X
JPDD_SendMsgToRemoteObject	X	X	
JPDD_SubSystemClass	X	X	X
JPDD_SubSystemConstruction			
JPDD_SubSystemConstruction_2	X	X	X

* Industrial Packing System

algorithm must be extended to include this option.

There is another small technical problem in the code generation process implemented in the initial version of GenERTiCA: the expressions used inside the UML model must be specified using the target platform syntax. In other words, GenERTiCA reads expressions in the model, using them as they are (i.e. a text fragment) in the generated code. Consequently, if the target language changes, and the expressions syntax is not the same, expressions in the model must be fixed, otherwise the generated code incur to compilation errors. A solution to this problem would be to parse expressions specified in the UML model, converting them to the target platform syntax. A generic expressions language must be used for specifying expressions in UML diagrams. In this sense, OCL could be a reasonable option, but its characteristics and suitability for this purpose need to be evaluated before choosing it as this generic expressions language.

To conclude this chapter, it is worth to mention that the UAV case study is completely provided in the appendices. Interested readers can see the complete UML model, along with mapping rules files for the RT-FemtoJava platform.

Chapter 8

Conclusions and Future Work

This work has proposed an approach to design distributed embedded real-time systems using MDE techniques along with concepts of AO paradigm to cope with the increasing complexity associated with the design of modern systems. More specifically, the proposed approach has addressed the following topics: *(i)* manage the complexity of functional and non-functional requirements handling; *(ii)* support for separation of concerns; *(iii)* specification of system structure and behavior using a common language; *(iv)* improvement in design phases transition by providing adequate tool support. All ideas and elements involved in the proposed approach have been presented throughout this text.

AMoDE-RT design flow proposes solutions for all these issues, supporting a smooth transition from requirements specification to source code implementation, in order to fulfill gaps usually found in the design flow. Such quest is achieved using a combination of elements: *(i)* RT-FRIDA for requirements analysis; *(ii)* UML as specification language; *(iii)* DERAF aspects to handle non-functional requirements; *(iv)* modeling guidelines to homogenize the specification of system structure, behavior, and non-functional requirements handling; *(v)* DERCS as intermediate representation of such modeled information; *(vi)* transformation heuristics to convert UML model elements into DERCS elements; and *(vii)* GenERTiCA code generation tool to support the AMoDE-RT approach.

Besides requirements gathering, RT-FRIDA assists in linking requirements specification with design elements, improving requirements traceability. Further, traceability is still preserved in implementation, due to GenERTiCA approach that uses mapping rules to generated code fragments from model elemets. In other words, it is possible to compare generated source code lines with code generation/aspect adaptation scripts, relating them with model elements, to discover which requirements are handled by these code lines. In this sense, the effort to check if the system meets the requirements can be decreased.

This work has shown that UML and MARTE profile can be used to specify system expected functionalities in terms of structure, behavior, and also non-functional requirements handling. As MARTE provides stereotypes with standard semantics to express real-time and embedded systems features, its usage is preferable rather than "home-made" profiles, due to its already

accepted concepts and constructions that passed through a rigorous review process. Using a common and standard specification language facilitates the communication of design intention, reducing possible misunderstandings in specification interpretation. Further, UML raises the abstraction level used in design by shifting the focus from expected functions to system elements and their roles to accomplish the desired functionalities, representing abstractions closer to real world elements.

However, as UML has many variation semantic points, it is also important to define modeling guidelines and also interpretation semantics to minimize (or even remove) model specification ambiguities. AMoDE-RT modeling guidelines intend to provide flexibility in UML diagrams creation, but defining, at the same time, an interpretation semantic for modeled elements, allowing the integration of information specified in distinct diagrams (mainly in behavior diagrams).

UML sequence diagrams have been successfully used to describe actions performed within behaviors, eliminating the need of using textual action languages as current approaches suggest. AMoDE-RT transformation heuristics allow actions sequence extraction from several different sequence diagrams, enabling their association with other behavior diagrams, such as state diagrams, to provide graphical behavior specification.

Furthermore, this work results have shown that using AO concept in distributed embedded real-time systems design improves separation of concerns in the handling of functional and non-functional requirements. In this sense, DERAF is a remarkable contribution due to the lack of aspects with platform independent adaptation semantics created specifically to real-time and embedded systems domain. Due to its well-defined semantics, DERAF has been successfully used at both modeling and implementation levels. Moreover, the assessment presented in chapter 7 indicates improvements in design understandability and flexibility, and also in the reuse of previously developed artifacts (i.e. model and/or code). It was demonstrated that DERAF aspects and JPDDs can easily be reused in different designs.

Despite the lack of support for AO concepts in official UML specification, AMoDE-RT proposes to specify them using DERAF, ACOD and a set of JPDDs. Instead of proposing invasive extensions to UML meta-model elements, AMoDE-RT proposes a lightweight extension using UML's extensibility mechanism, i.e. a profile, allowing the use of off-the-self UML modeling tool to create AO elements with ACOD and JPDDs.

Similarly to other MDE approaches, the effectiveness of AMoDE-RT approach usage is highly dependent on tool support. Therefore, GenERTiCA has been created to assist in the automatic transformation of UML models into source code for different target platforms. Although UML and MARTE provide adequate constructions to specify features of distributed embedded real-time systems, they do not allow an unambiguous specification targeting source code generation. Consequently, the intermediate PIM called DERCS has been proposed to support code generation tools construction. The most remarkable difference between UML and DERCS is the representation of AO concepts, whose related elements stand for information specified in ACOD and JPDD. AMoDE-RT transformation heuristics extract information from ACOD and JPDD, allowing the creation of DERCS elements. In addition, it interprets JPDD

semantics gathering selected elements, associating these elements with the DERCS join points representation.

GenERTiCA code generation approach is different from the majority of code generation tools available; it allows the separation of concerns in mapping rules description by using small scripts responsible to generate source code fragments for structural and/or behavioral elements. It can be state empirically that this approach improves cohesion and reinforces designers focus on individual elements instead of the whole model. Besides not clearly demonstrated by case studies, we believe empirically that, using GenERTiCA approach, it is easy to reuse parts of mapping rules files in different designs, or using these parts as base to extend the mapping rule scripts with other constructions in the target platform.

A remarkable contribution of GenERTiCA is its ability to perform aspects weaving in generated code fragments, and also in the input DERCS model. This capability, along with the use of model-level aspects, allow to apply AO concept with non-AO target platforms, as demonstrated in case studies. Furthermore, model weaving provided by GenERTiCA could be also used in other tools, such as design exploration tools, to evaluate the impact of a given aspect implementation. In this sense, in spite of allowing different implementations, DERAF aspect semantics must be preserved to allow their high-level (re)use, i.e. the same platform can provide different forms to implement aspect adaptations, but this implementation must respect the pre-defined high-level semantics.

MDE, AOD, and code generation topics have still more issues to be investigated. This work development has leaded to other open problems regarding the mentioned topics. Thus, to conclude this text, a discussion on directions for future investigation are provided:

- Sequence diagram is not the most adequate diagram, in essence, to specify algorithmic behavior that do not represent object interactions. In behavior specifications, there are algorithms having more mathematical expressions calculation than object interactions. In these situations, activity diagrams are more suitable than sequence diagrams. Hence, a modification in AMoDE-RT modeling guidelines and transformation heuristics (to provide support for both sequence and activity diagrams to specify actions performed within a behavior) would allow designers to choose the one the better fits with the behavior characteristics;

- MARTE profile has a bunch of other stereotype to describe real-time features, e.g. `ResourceUsage`, `GRService`, `TimingResource`, and others. To investigate how to combine them with the AMoDE-RT approach is other research direction;

- To support other JPDD types would allow other advanced options for elements selection instead of only direct elements selection. This extension is very challenging due to expressiveness power of JPDD that would need elements evaluation considering, for example, execution flows, state machines, or indirect classes associations;

- To implement UML state diagrams transformation into DERCS elements according to AMoDE-RT transformation heuristics, as explained in chapter 6;

- MDE assumes that system implementation is obtained directly from models. To assure

that the automatically generated source code is functionally correct, the source model must also be correct. Thus, it is an interesting topic to investigate how to execute models. DERCS could be used as the base for a UML virtual machine that simulates the behavior specified in UML models, allowing early evaluation of system behavior;

- Following the model execution thread, it is also interesting to provide means for automatic UML model testing, likewise implementation-level approaches such as JUnit. Automatic model testing could allow automatic evaluation of model changes against expected behavior results;

- To extend GenERTiCA's code generation approach to overcome the problem of circular cross-reference, as mentioned in chapter 7;

- To investigate the use of OCL to support the specification of expressions in a programming language independent fashion and, in addition, to make GenERTiCA fully platform independent;

- To create mapping rules for other platforms, such VHDL, Verilog, and others;

- To apply AMoDE-RT approach in other application domains of embedded systems.

Appendix A

DERAF Detailed Description

This appendix provides a more exhaustive discussion on the high level semantics of DERAF aspects, representing the initial proposal for the handling of non-functional requirements presented in chapter 2.

A.1. Timing Package

As depicted in figure A.1, this package contains aspects to handle time-related requirements, such as deadlines for activities execution, WCET information, periodic tasks activation, and others.

TimingAttributes aspect is responsible to deal with active objects characteristics such deadline, priority, WCET, and absolute time instants on which their behavior must start and finish the execution. Attributes representing the mentioned characteristics are inserted in the affected active object classes, as well as methods and behavior to initialize and handle these attributes. It provides the following adaptations:

- *Deadline* inserts an attribute representing the active objects behavior deadline, i.e. an active object has only one main behavior, to which the deadline is related;
- *WCET* adds attributes to represent the WCET of active objects behaviors;
- *StartTime* inserts an attribute to specify the absolute time instant in which an active object can start their main behavior execution;
- *EndTime* inserts an attribute to represent the absolute time instant in which the execution of the active object main behavior is not allowed to execute. For instance, a periodic active object cannot be triggered to execute its behavior after its end time;
- *Priority* adds an attribute to represent the priority that an active object have to execute their behavior;
- *SetTimingAttributes* inserts the behavior responsible to initialize the inserted attributes values;

- *AddAccessMethods* adds access methods to the inserted attributes;

PeriodicTiming aspect provides means to trigger periodically an active object behavior execution. Thus, besides adding an attribute indicating the execution frequency in the affected active object class, this aspect must also enclose the affected behavior with a repetition mechanism, whose execution is controlled according the information stored in the mentioned new attribute. In other words, this aspect is used to deal with the handling of periodic active objects (or threads). It provides the following adaptations:

- *Period* inserts an attribute representing the activation period of periodic active objects behavior that is used to control the behavior execution frequency;
- *SetPeriod* inserts all code responsible to initialize the period attribute values, as well as the get/set methods responsible to access it;
- *LoopMechanism* encloses periodic active objects behavior with a mechanism, and hence, behavior's actions sequence is executed repeatedly;
- *FrequencyControl* adds a mechanism to control the execution frequency of periodic active object behavior. This mechanism is responsible to hold active object's behavior execution. One solution would be to inform the scheduler that the active object has executed its behavior, and can be suspended. Other implementation could be a busy wait.

SchedulingSupport aspect inserts a scheduler object in the affected computing nodes. This object is responsible to control active objects execution, indicating instants at which they must start performing their behavior. It provides the following adaptations:

- *Scheduler* adds a scheduling mechanism that follows a given scheduling policy;
- *SetupConcurrentActivities* inserts the behavior responsible to add active objects in the scheduling list, in order to perform the execution schedule.

TimeBoundedActivity aspect controls the execution time duration of an activity or action by counting the time elapsed since the start time instant. If maximum allowed duration is surpassed, this aspect provides means to abort the affected activity/action execution. Examples of this aspect use are: to restrict the maximum time a shared resource can be in exclusive access mode, or the maximal time amount an active object can wait for the reply of a remote objects. It provides the following adaptations:

- *TimeCountInfrastructure* adds a time counting mechanism (e.g. timer) associated with the affected element, which can be a new class attribute or a local variable in a method behavior;
- *StartCounting* inserts behavior to setup and start the time counting mechanism at the starting time the controlled action/activity;
- *StopCounting* adds behavior to stop the time counting mechanism right after the controlled action/activity is finished.
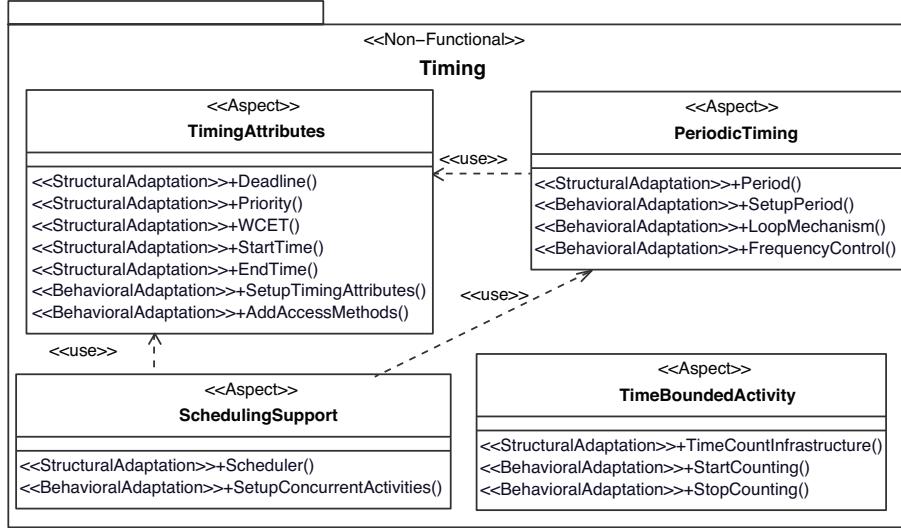
Figure A.1.: *Timing Package:* handling time non-functional requirements

A.2. Precision Package

Precision in meeting time requirements are handled by the aspects of this package, which concentrates efforts in features such as the maximum tolerated delay in starting activities, variance in activities timeliness, information's validity duration, or the deviation of local clock reference compared with the global one. Precision package aspects are depicted in figure A.2.

Jitter aspect measures the accuracy variance in activities performed by the system. This aspect provides means to measure the time before (or after) an observed activity happen, storing this information (the history must provide information of at least one time sample) to calculate the variance among the observed time instants. This aspect can be used, for example, to calculate the jitter in an periodic active object activation or execution, or to compute the time variance of a periodic message sending. It provides the following adaptations:

- *StartTime* adds, as the name indicates, behavior to measure the time point on which an activity starts;
- *VerifyToleratedJitter* inserts a behavior to calculate the variance in two consecutive time measurements of the same activity, comparing the result with the instant at which this activity is expected to be performed. If the variance violates the tolerated threshold, a corrective behavior can be executed.

ToleratedDelay aspect controls the maximum tolerated latency to the actual start of a given system activity. Thus, the time between the command and the execution of the observed activity must be measured and calculated. If the observed duration is greater than the maximum allowed latency, this aspect provides means to handle this exception. It provides the following adaptations:

- *StartTime* inserts behavior to measure the time instant at which an activity is commanded
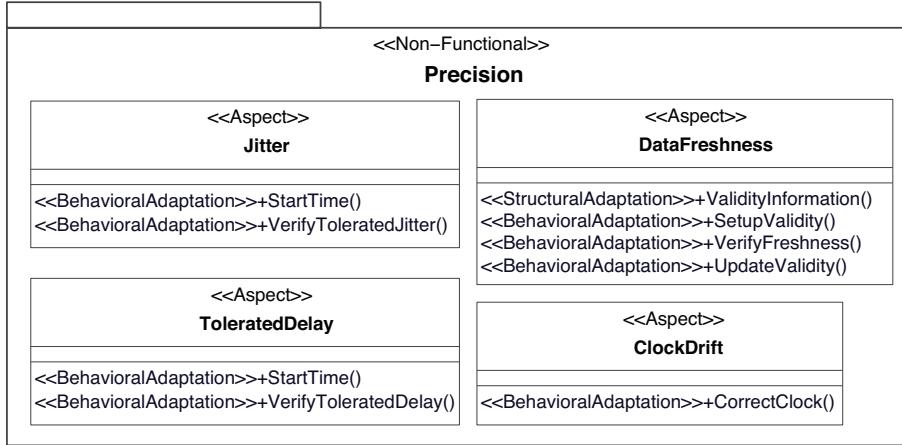
Figure A.2.: *Precison Package:* handling precision non-functional requirements

to start;

- *VerifyToleratedDelay* adds a behavior to measure the time point at which the observed activity actually starts. The time interval (i.e. delay) between command and execution starting instants is calculated and compared with the expected delay. If the threshold is violated, a corrective behavior can be executed.

ClockDrift aspect controls the clock deviation between the local time source and the global one. Assuming that the target platform provides means to allow clock synchronization, this aspect uses the global clock as reference to calculate the local clock deviation. Thus, designers must specify time instants (or system events, e.g. the starting of an behavior execution) at which the local clock must be compared with the global clock reference in order to check if there is a difference between the two measured values. It provides the following adaptation:

- *CheckClockDrift* reads the current time from both global and local clocks, and compares the time obtained from them. If the perceived different is outside the accepted threshold range, any corrective action (e.g. update the local clock reference) can be performed.

DataFreshness aspect is responsible to deal with the validity duration (or utility) of different system information [22]. For that, this aspect associates timestamps to affected data by adding new attributes to representing such information, as well as inserting behavior to control these data use. In other words, each time a controlled data needs to be read, its validity must be checked and, if it is out of validity, a corrective behavior must be performed, e.g. wait until the date to be updated, read data directly from its source, decrease the frequency at which periodic behaviors (which read the controlled data) are executed. Analogously, each time a controlled data is updated, its validity duration must also be updated. It provides the following adaptations:

- *ValidityInformation* adds an attribute indicating the validity period of the controlled attribute or object;
- *SetValidity* inserts the behavior that is responsible for initializing the validity period information;

- *VerifyFreshness* inserts a behavior to check data validity before all reading actions that access the controlled data;
- *UpdateValidity* adds the corresponding behavior that updates data validity after all actions that write/modify the controlled data.

A.3. Synchronization Package

This package provides aspects to deal with non-functional requirements related to the synchronization and the concurrent access control to shared resources. Figure A.3 depicts the available aspects.

ConcurrentAccessControl aspect provides means to control the concurrent access to objects, which share their attributes information with other objects. The access to object's different elements can be controlled: *(i)* the object itself; *(ii)* their attributes; and/or *(iii)* their methods. Therefore, depending on the controlled element, one or more arbiters (i.e. concurrency controller instances) are created. Every time an (active or passive) object needs to access controlled shared elements, it must request the access to them (i.e. request a lock) that are granted or not by the arbiter. Depending on the arbiter implementation (e.g. mutex, semaphore, monitors), and also to the number of objects that are accessing the shared resource at the moment, the access request can be authorized or not. Similarly, after the use of the shared resource, the object that had the access permission must notify the arbiter, indicating that it is leaving the shared resource and does not need to use it anymore. It provides the following adaptations:

- *ConcurrentControlMechanism* inserts an arbiter to control the access to shared resources;
- *AcquireAccess* adds the behavior that is responsible for requesting the access to shared resources before reading or writing information from/to the shared resource;
- *ReleaseAccess* inserts a behavior to notify the arbiter that the access to the share resource
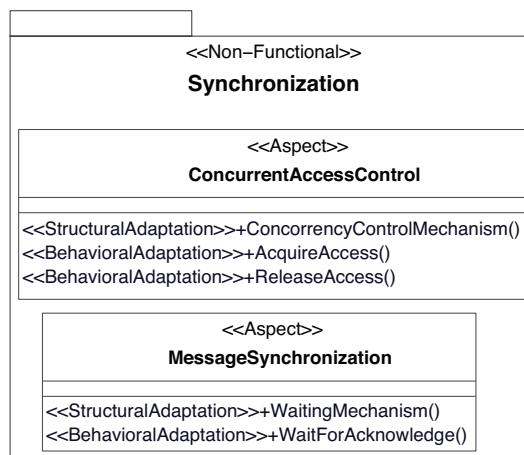
Figure A.3.: *Synchronization Package:* handling synchronization non-functional requirements

can be released;

MessageSynchronization aspect deals with holding behaviors execution until the arrival of an acknowledgement message (or a reply message) indicating that the (remote) object has received the message sent. It provides a waiting mechanism that could be implemented as either *(i)* a busy wait, i.e. a loop that waits until the acknowledgement message arrives; or *(ii)* using the system scheduler, which preempts the execution of the current active object, marking it as blocked, and thus, opening room for other active objects execution. Later, when the expected acknowledgement message arrives, the blocked active object is marked as ready to execute, and its execution is resumed following the scheduler's decision. It provides the following adaptations:

- *WaitingMechanism* inserts the acknowledgement waiting mechanism. In fact this adaptation makes more sense within the context of (ii), because the scheduler must be modified in order to realize this implementation;
- *WaitForAcknowledgement* adds the behavior that is responsible for waiting for the expected acknowledgement message;

A.4. Communication Package

This package provides aspects to deal with objects communication in terms of messages sending. The first intention was to cover the communication between objects that are located in computing devices that are physically separated. However, depending on application requirements, this package's aspects can also be used for specifying the communication of objects located in the same computing device. The available aspects are show in figure A.4.

MessageAck aspect provides an acknowledgment mechanism to notify reception of a message to its sender. In this sense, this aspect affects both sides of a message exchange: sender and destination objects. On one side, the sender object sends a messages and waits for an ac-
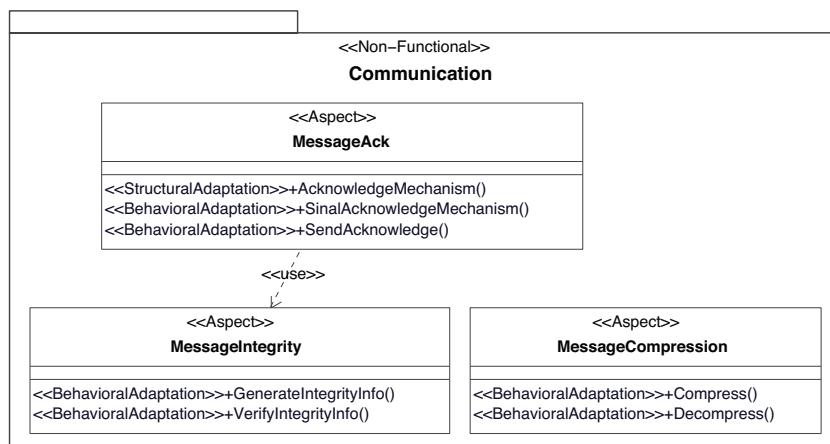
Figure A.4.: *Communication Package:* handling communication non-functional requirements

knowledgement of message reception. On the other side, the receiver objects needs to send an acknowledgement message after each received message. *MessageAck* is related with *MessageSynchronization* aspect. It provides the following adaptations:

- *AcknowledgeMechanism* adds the acknowledge mechanism. After a message reception, this mechanism must be notified about this arrival, and send an acknowledgement message to this message sender;
- *SignalAcknowledgeMechanism* adds the behavior, at the sender object side, that is responsible for notifying the acknowledge mechanism that a message has been sent and an acknowledge message must be received;
- *SendAcknowledge* inserts a behavior, at the receiver object side, that sends an acknowledge message, after an message reception, informing its sender that the message has been delivered to the destination object;

MessageIntegrity aspect is responsible for handling messages integrity by providing checking information within a message. Similarly to *MessageAck*, this aspect also affects both message's sender and receiver objects. Sender objects must generate integrity checking information, appending it in the message to be sent, while receiver objects must generate checking information from the received message, comparing it with the information that came with the received message. The acknowledgment mechanism must be notified whether the checking information match or not. It provides the following adaptations:

- *GenerateIntegrityInfo* inserts behavior, at the message sender side, before the message sending action, that executes a algorithm to generate checking information that is appended in the message being sent;
- *VerifyIntegrityInfo* adds behavior, at the message receiver side after the message reception, that executes an algorithm (the same performed at message sender side) to generate checking information of the received message, comparing the generated information with the one received in the message. If it matches, the acknowledge mechanism is notified, otherwise any other corrective behavior can be performed;

MessageCompression spect is in charge to compress/decompress messages in order to improve bandwidth utilization. Like the other aspects of this package, this aspect affects both message's sender and receiver objects. At sender side, the message is compressed using a compression algorithm, while at receiver side the message is decompressed using the same algorithm. It provides the following adaptations:

- *Compress* adds a behavior to compress the message being sent before sending it;
- *Decompress* adds a behavior to decompress the compressed message received before actually delivering it;

A.5. TaskAllocation Package

Aspects provided by this package handle non-functional requirements related to objects distribution on different computing devices at runtime. These aspects are typically related to distributed

system nodes that are physically separated. Figure A.6 depicts the available aspects.

NodeStatusRetrieval aspect includes a mechanism to gather information on the system dynamic characteristics, such as processing load, message sending and reception rates, and if the computing device is running. It provides the following adaptations:

- *ProcessingLoad* inserts a behavior to calculate the device's processing load, updating this information at every start/end of an active object behavior;
- *MessageThroughput* adds a behavior to calculate the sent messages rate, as well as the ratio for the received ones. This information is updated at every message sending/reception;
- *Alive* includes a new object in the computing device that is responsible to broadcast an "I'm alive" message to the other devices in the distributed system;

TaskMigration aspect adds a migration mechanism to move active objects from one computing device to another one. Therefore, active objects can migrate from one node to another, as well as from software to hardware, or vice-versa [1]. To accomplish this mission, the migration mechanism must provide means for saving/restoring the execution context of active objects, as well as for objects serialization and object's information sending. In fact, the decision on which objects must migrate is made by the aspects responsible to control embedded systems physical resources, such as *EnergyControl*, *MemoryUsageControl*, and *HwAreaControl*. Basically *TaskMigration aspect* provides only one adaptation, i.e. *Migrate* behavioral one, which adds the mentioned behavior related to the migration mechanism.

---

[1]Objects migration between software and hardware (at runtime) is usually known as "reconfiguration". However, in embedded systems domain, "reconfiguration" usually means to upload a bitstream into a FPGA device. Thus, in order to avoid misunderstandings, this text uses the term "reconfiguration" to refer to the later, while objects software-to-hardware and/or hardware-to-software reconfiguration are also called "migration"
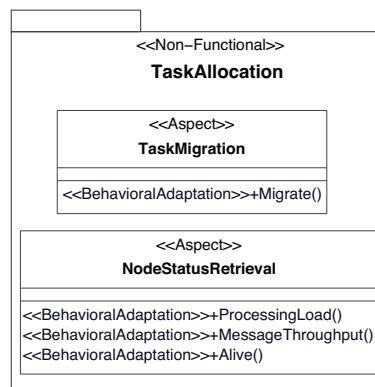
Figure A.5.: *TaskAllocation Package:* handling tasks allocation non-functional requirements

A.6. Embedded Package

Non-functional requirements related to physical resources availability, which are very common concerns in embedded systems design, are handle by this package's aspects. Energy consumption, memory usage, and hardware reconfigurable area can be cited as examples of such concerns. As depicted in A.6, the available aspects are concerned in monitoring and controlling the mentioned physical resources. Thus, depending on the physical resource being controlled, the control policy, and platform capabilities, different actions can be performed by these aspects as, for instance: *(i)* depending on the system requirements and runtime state, to remove objects related to non-critical activities; *(ii)* active objects migration; *(iii)* to loosen timing constraints; *(iv)* to decrease processor operation frequency; *(v)* to turn off unnecessary hardware components; It is important to highlight that this aspects are dependent on target platform capabilities, meaning that the platform must provide means to monitor and control system physical resources.

HwAreaMonitoring aspect is related to systems that use reconfigurable hardware devices, such as FPGAs. It provides a mechanism to monitor the reconfigurable area by which the remaining reconfigurable area (in terms of configurable logic blocks) is (re)calculated at each reconfiguration command. It provides the following adaptations:

- *IncreaseAreaUsage* inserts a behavior that increases the reconfigurable area usage amount, before all hardware reconfiguration actions, based on area required by the new hardware active objects;
- *DecreaseAreaUsage* adds a behavior that decreases the reconfigurable area usage amount, before all hardware reconfiguration actions, based on size information of the hardware active objects that are leaving the reconfigurable hardware device;

HwAreaControl aspect controls the hardware reconfigurable device usage by adding an arbiter to allow or deny every reconfiguration based on the information of this package monitoring aspects. In fact it provides only one adaptation: the inclusion of a new active object that accesses the information produced by the *HwAreaMonitoring* aspect to control the reconfigurable area use, taken actions as described earlier in this sub-section.

EnergyMonitoring aspect relies on the target platform to provide a mechanism to monitor energy consumed by system activities. This mechanism must measure the remaining energy level before the observed activities start, and after their completion. Further, it calculates the amount of energy that was consumed by these activities. It provides the following adaptations:

- *EnergyMonitoringMechanism* adds the energy monitoring mechanism;
- *InitialEnergyMeasurement* inserts a behavior responsible to measure the energy level before any activity execution;
- *CalculateEnergyConsumption* adds a behavior that also measures the energy level right after the execution of observed activity is finished, calculating the energy consumed by this activity, and also by the overall system;

EnergyControl aspect provides an object that uses information provided by the monitoring
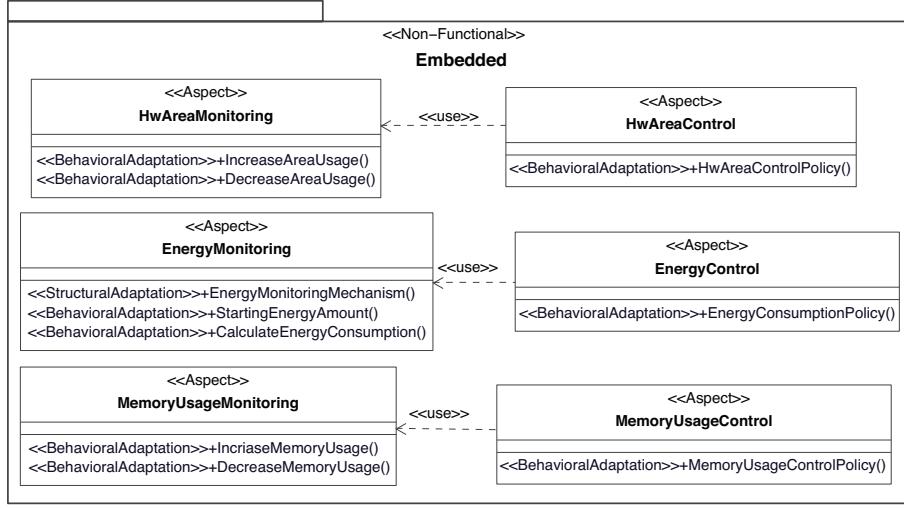
Figure A.6.: *Embedded package:* handling embedded non-functional requirements

aspects to control the energy consumption. To accomplish such goal, this object could perform the actions mentioned in the beginning of this subsection. This aspect provides only one adaptation, *EnergyConsumptionPolicy*, that includes an energy controller element in the system.

MemoryUsageMonitoring aspect is similar to the other two monitoring aspects but it is related to software rather than to hardware. It provides a mechanism that must calculate the overall memory usage of a computing device at every object allocation/deallocation. It provides the following adaptations:

- *IncreaseMemoryUsage* inserts a behavior to increase the monitoring element information on used memory amount before every action that allocates memory;
- *DecreaseMemoryUsage* inserts a behavior to decrease the used memory amount information before every action that allocates memory;

MemoryControl aspect uses the information provided by *MemoryUsageMonitoring* and *HwAreaMonitoring* aspects to control the memory allocation requests for objects allocation following an adopted memory control policy. Thus the *MemoryUsageControlPolicy* adaptation inserts this controller element in the system.

Appendix B

UML Models for the UAV Case Study
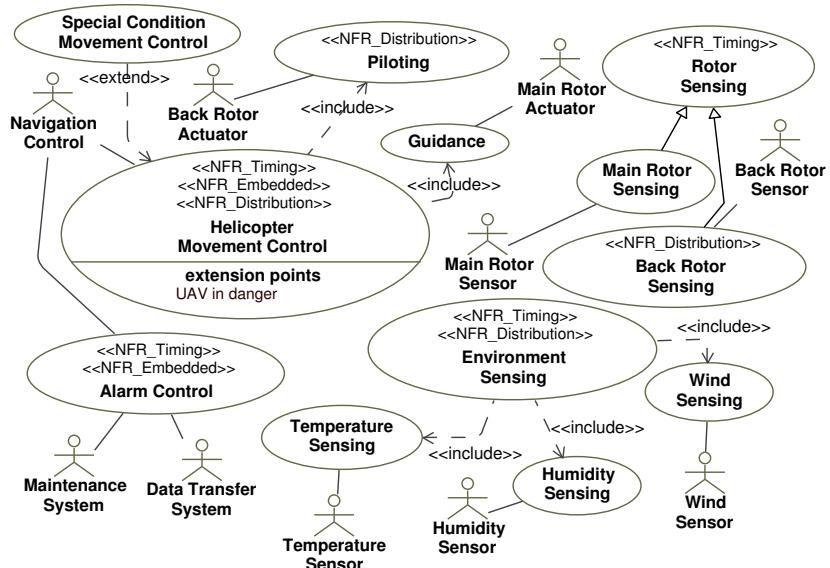
Figure B.1.: UAV movement control use case diagram
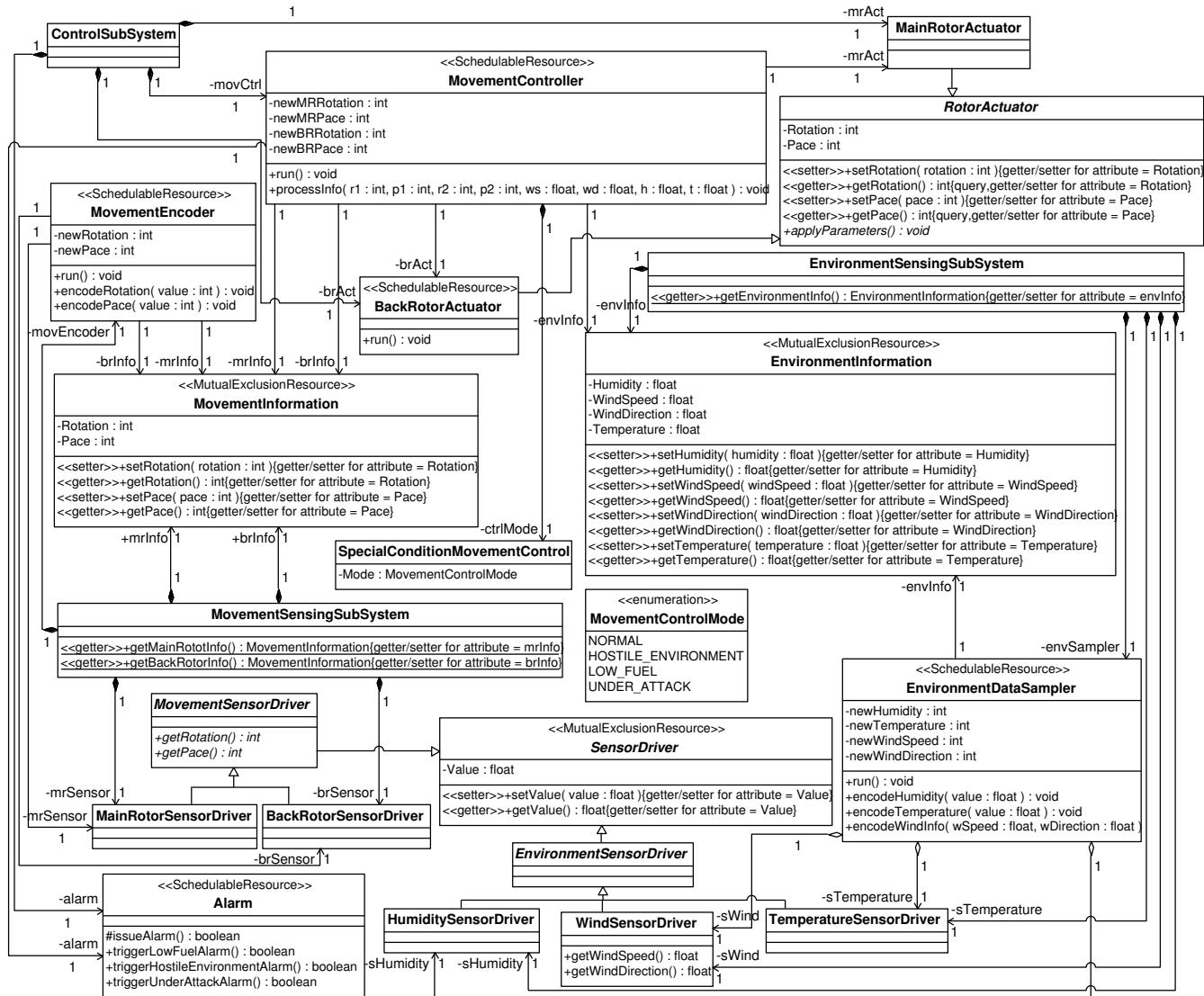
Figure B.2.: UAV movement control class diagram
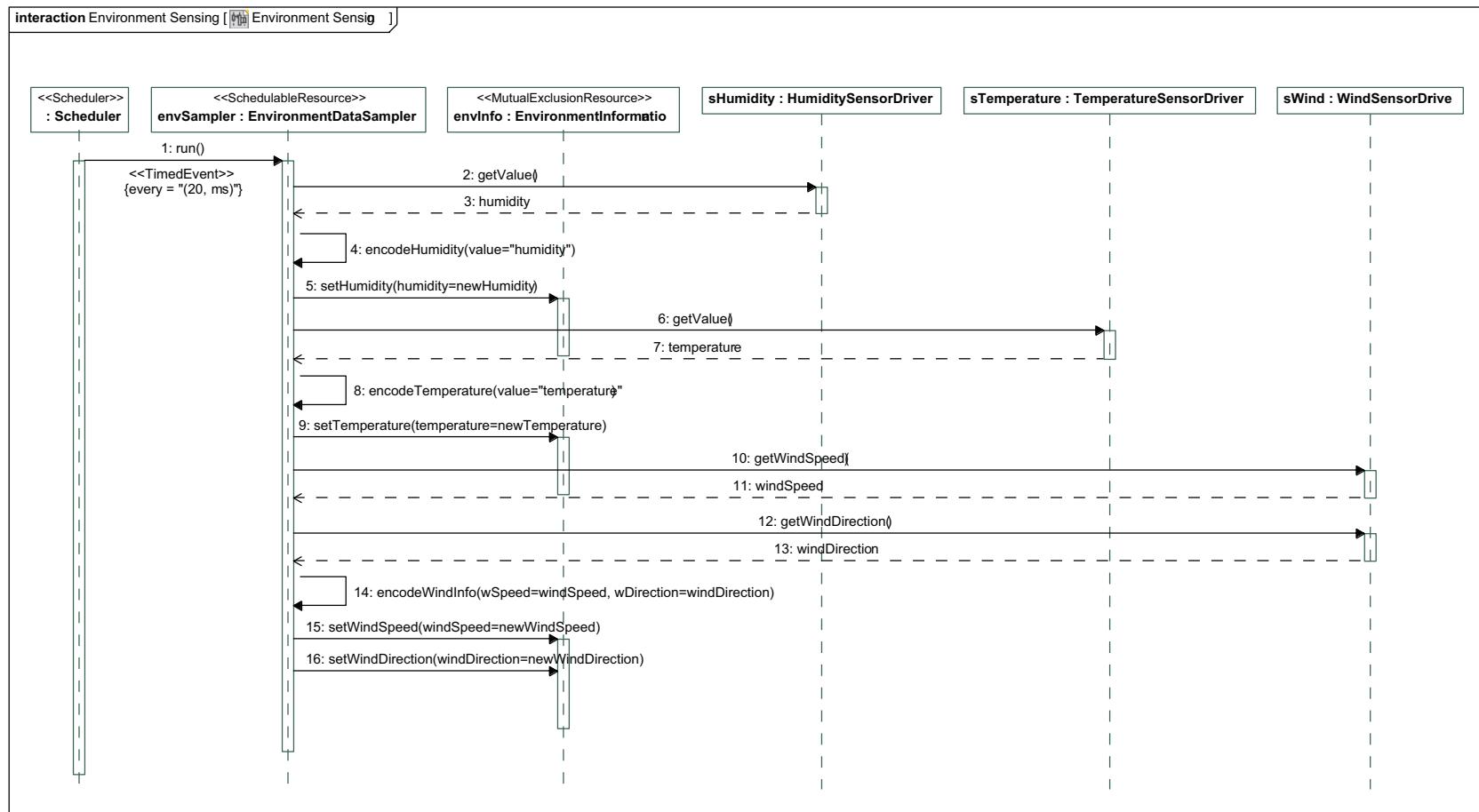
Figure B.3.: Environment sensing

interaction Movement Encode [Movement Encode]

<<Scheduler>>	<<SchedulableResource>>	mrSensor : MainRotorSensorDriver	<<MutualExclusionResource>>	brSensor : BackRotorSensorDriver	<<MutualExclusionResource>>
: Scheduler	movEncoder : MovementEncoder		mrInfo : MovementInformation		brInfo : MovementInformation

1: run():""

<<TimedEvent>>
{every = "(10, ms)"}

2: getRotation()

3: rotation

4: encodeRotation(value=rotation)

5: getPace()

6: pace

7: encodePace(value="pace")

8: setRotation(rotation=newRotation)

9: setPace(pace=newPace)

10: getRotation()

11: rotation

12: encodeRotation(value=rotation)

13: getPace()

14: pace

15: encodePace(value=pace)

16: setRotation(rotation=newRotation)

17: setPace(pace=newPace)

Figure B.4.: Main and back rotors sensing
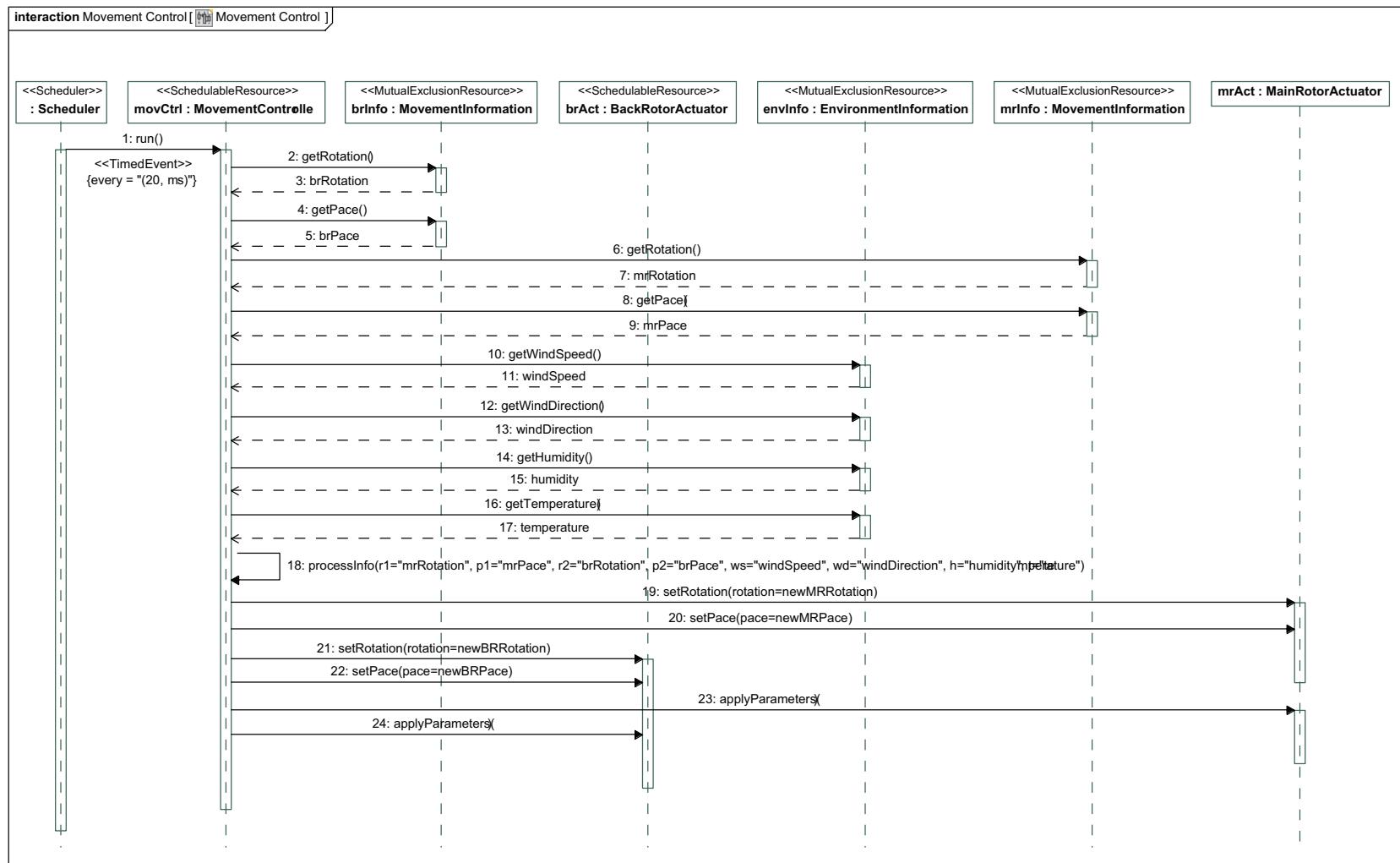
Figure B.5.: Helicopter movement control

Figure B.6.: Helicopter piloting
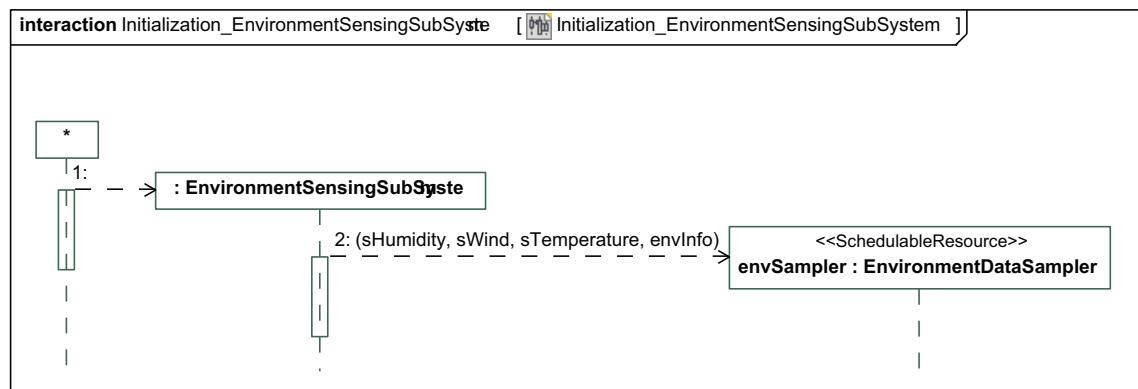
Figure B.7.: Environment sensing subsystem initialization
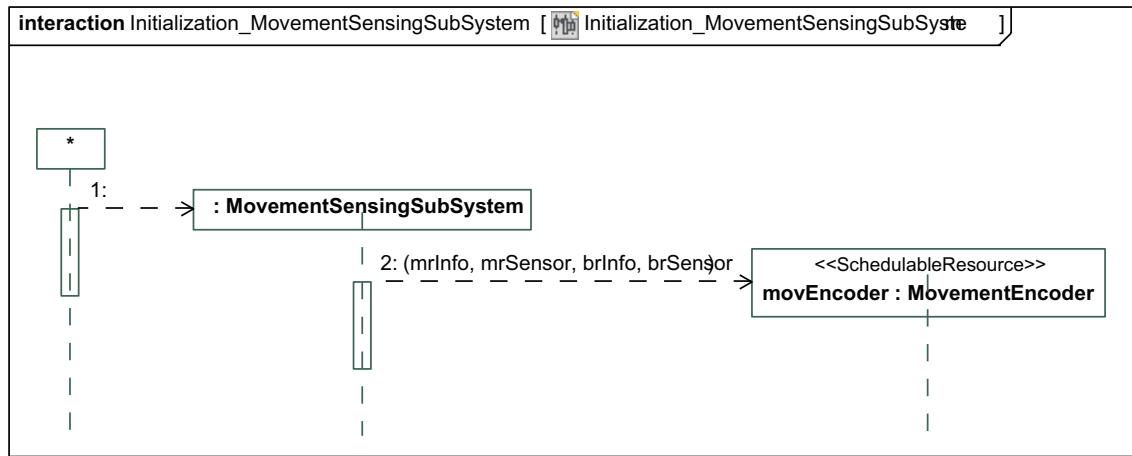
Figure B.8.: Movement sensing subsystem initialization
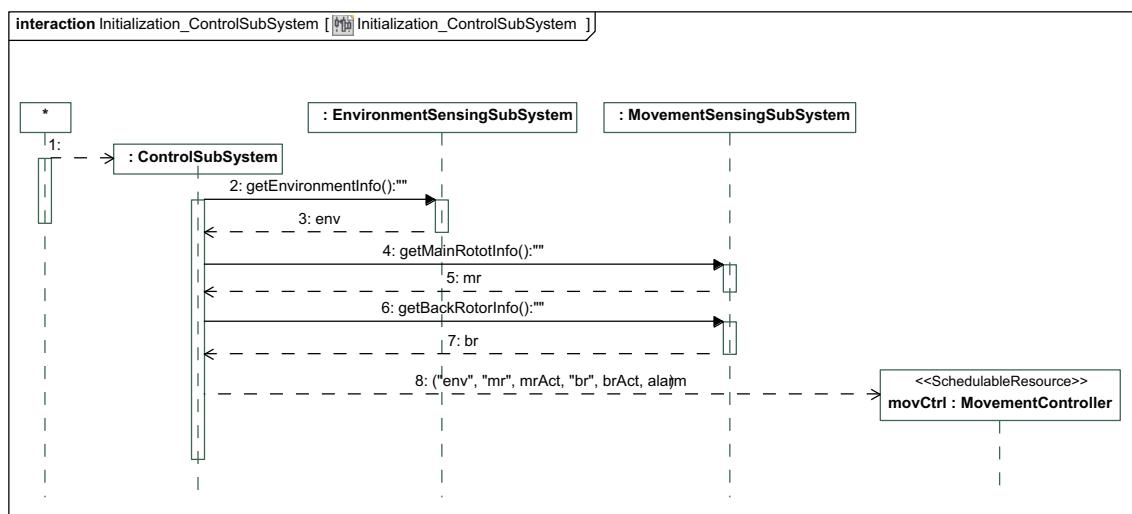
Figure B.9.: Control subsystem initialization

Figure B.10.: Other behavior: `WindSensorDriver.getWindSpeed()`
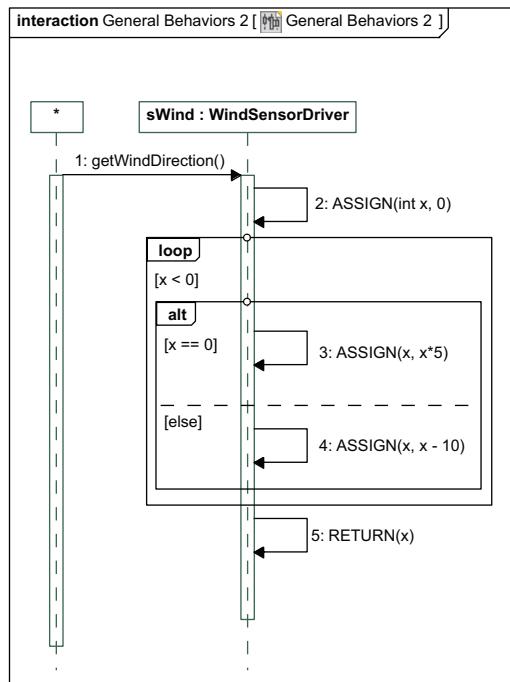
Figure B.11.: Other behavior: `WindSensorDriver.getWindDirection()`
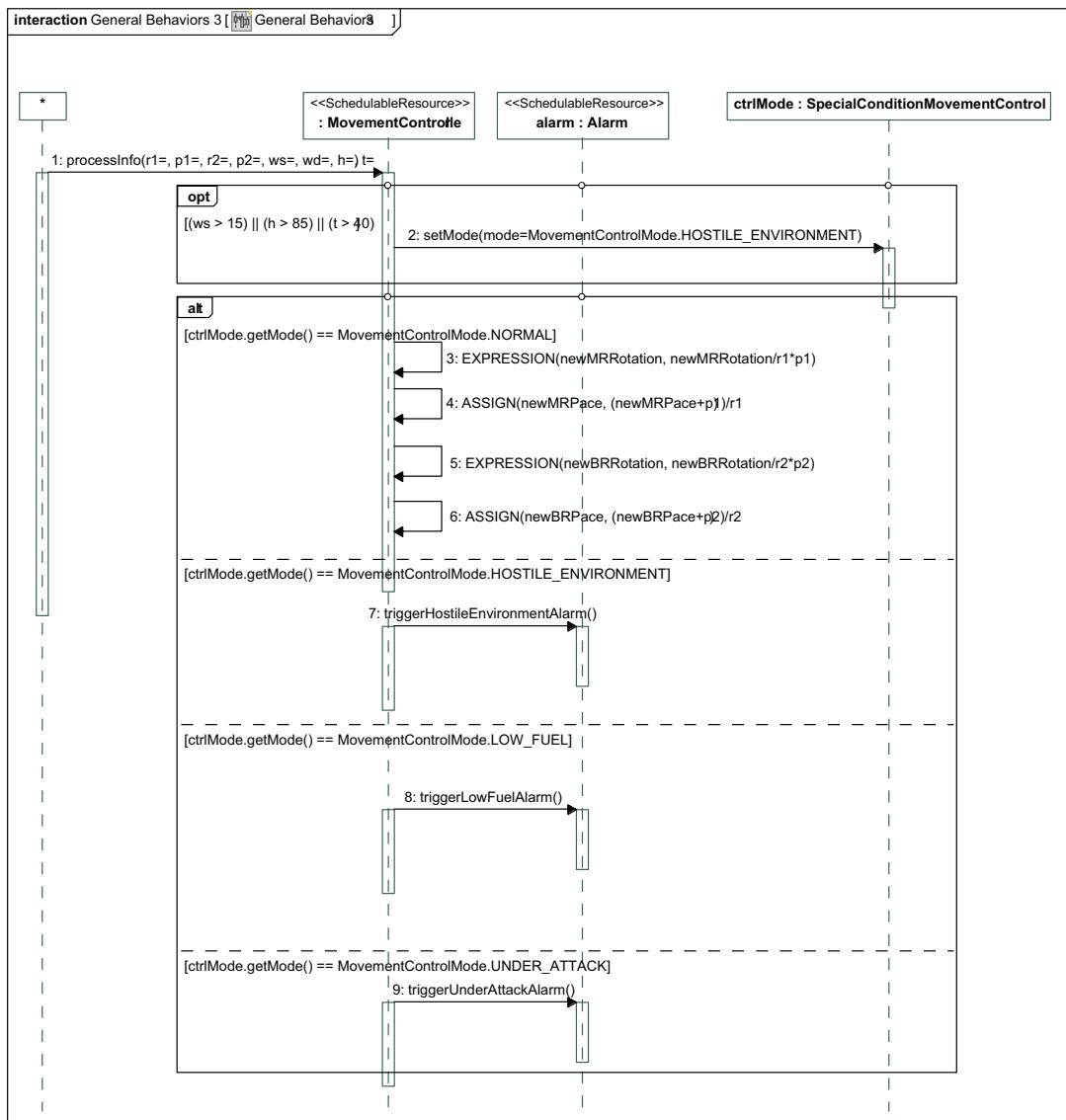
Figure B.12.: Other behavior: `MovementController.processInfo()`

Figure B.13.: UAV movement control deployment diagram

<<Aspect>>
TimingAttributes

<<Pointcut>>+pcActClass(JPDD_ActiveObjectClass, Deadline+Priority+WCET, ADD_NEW_FEATURE)
<<Pointcut>>+pcActClass2(JPDD_ActiveObjectClass, ModityClassStructure, MODIFY_STRUCTURE)
<<Pointcut>>+pcActObjInit(JPDD_ActiveObjectConstruction, SetTimingAttributes, AFTER)
<<Pointcut>>+pcActObjInit2(JPDD_ActiveObjectConstruction_Action, AdaptObjectConstruction, MODIFY_STRUCTURE)
<<Pointcut>>+pcActObjContructor(JPDD_ActiveObjectConstructor, ModifyConstructor, MODIFY_STRUCTURE)
<<StructuralAdaptation>>+Deadline()
<<StructuralAdaptation>>+Priority()
<<StructuralAdaptation>>+WCET()
<<StructuralAdaptation>>+ModifyConstructor()
<<StructuralAdaptation>>+ModityClassStructure()
<<BehavioralAdaptation>>+SetTimingAttributes()
<<BehavioralAdaptation>>+AdaptObjectConstruction()

<<Aspect>>
DataFreshness

<<Pointcut>>+pcInfoClassAttr(JPDD_InfoClassAttribute, ValidityInformation, ADD_NEW_FEATURE)
<<Pointcut>>+pcInfoClassObjInit(JPDD_InfoObjectConstruction_Action, SetValidity, AFTER)
<<Pointcut>>+pcReadAttrValue(JPDD_InfoAttributeRead, VerifyFreshness, BEFORE)
<<Pointcut>>+pcWriteAttrValue(JPDD_InfoAttributeWrite, UpdateFreshness, AFTER)
<<StructuralAdaptation>>+ValidityInformation()
<<BehavioralAdaptation>>+SetValidity()
<<BehavioralAdaptation>>+VerifyFreshness()
<<BehavioralAdaptation>>+UpdateFreshness()

<<Crosscut>>
{Validity = "100ms"}
Humidity, Temperature

<<Crosscut>>
{Validity = "25ms"}
WindSpeed, WindDirection

<<Crosscut>>
{Validity = "25ms"}

<<MutualExclusionResource>>
EnvironmentInformation

<<MutualExclusionResource>>
MovementInformation

<<Crosscut>>
{Deadline = "20ms",
Priority = "1",
WCET = "8ms"}

<<Crosscut>>
{Deadline = "10ms",
Priority = "2",
WCET = "5ms"}

<<Crosscut>>
{Deadline = "20ms",
Priority = "2",
WCET = "7ms"}

<<Crosscut>>
{Deadline = "10ms",
Priority = "1",
WCET = "3ms"}

<<Crosscut>>
{Deadline = "5ms",
Priority = "0",
WCET = "2ms"}

<<SchedulableResource>>
MovementController

<<SchedulableResource>>
EnvironmentDataSampler

<<SchedulableResource>>
Alarm

<<SchedulableResource>>
MovementEncoder

<<SchedulableResource>>
BackRotorActuator

<<Crosscut>>
{Period = "20ms"}

<<Crosscut>>
{Period = "10ms"}

<<Crosscut>>
{Period = "20ms"}

<<Crosscut>>
{Period = "10ms"}

<<Aspect>>
ConcurrentAccessControl

<<Pointcut>>+pcSharedObjClass(JPDD_ExclusiveObjectClass, ConcurrencyControlMechanism, ADD_NEW_FEATURE)
<<Pointcut>>+pcBeforeRead(JPDD_ExclusiveGet, AcquireAccess, BEFORE)
<<Pointcut>>+pcAfterRead(JPDD_ExclusiveGet, ReleaseAccess, AFTER)
<<Pointcut>>+pcBeforeWrite(JPDD_ExclusiveSet, AcquireAccess, BEFORE)
<<Pointcut>>+pcAfterWrite(JPDD_ExclusiveSet, ReleaseAccess, AFTER)
<<StructuralAdaptation>>+ConcurrencyControlMechanism()
<<BehavioralAdaptation>>+AcquireAccess()
<<BehavioralAdaptation>>+ReleaseAccess()

<<Aspect>>
PeriodicTiming

<<Pointcut>>+pcActClass(JPDD_ActiveObjectClass, Period, ADD_NEW_FEATURE)
<<Pointcut>>+pcActObjConstructor(JPDD_ActiveObjectConstructor, ModifyConstructor, MODIFY_STRUCTURE)
<<Pointcut>>+pcActObjInit(JPDD_ActiveObjectConstruction, SetPeriod, AFTER)
<<Pointcut>>+pcActObjInit2(JPDD_ActiveObjectConstruction_Action, AdaptObjectConstruction, MODIFY_STRUCTURE)
<<Pointcut>>+pcLoop(JPDD_PeriodicBehavior, LoopMechanism, AROUND)
<<Pointcut>>+pcFreqCtrl(JPDD_PeriodicBehavior, FrequencyControl, AFTER)
<<StructuralAdaptation>>+Period()
<<StructuralAdaptation>>+ModifyConstructor()
<<BehavioralAdaptation>>+SetPeriod()
<<BehavioralAdaptation>>+LoopMechanism()
<<BehavioralAdaptation>>+FrequencyControl()
<<BehavioralAdaptation>>+AdaptObjectConstruction()

<<Aspect>>
SchedulingSupport

<<Pointcut>>+pcSubSystemClass(JPDD_SubSystemClass, Scheduler, ADD_NEW_FEATURE)
<<Pointcut>>+pcSubSystemConstruction(JPDD_SubSystemConstruction_2, SetupConcurrentActivities, AFTER)
<<StructuralAdaptation>>+Scheduler()
<<BehavioralAdaptation>>+SetupConcurrentActivities()

<<Aspect>>
EnergyControl

<<Pointcut>>+pcSubSystemClass(JPDD_SubSystemClass, EnergyConsumptionPolicy, ADD_NEW_FEATURE)
<<StructuralAdaptation>>+EnergyConsumptionPolicy()

<<Aspect>>
MessageSynchronization

<<Pointcut>>+pcSubSystemClass(JPDD_SubSystemClass, WaitingMechanism, ADD_NEW_FEATURE)
<<Pointcut>>+pcRemoteMsgSending(JPDD_SendMsgToRemoteObject, WaitForAcknowledge, AFTER)
<<StructuralAdaptation>>+WaitingMechanism()
<<BehavioralAdaptation>>+WaitForAcknowledge()

<<Aspect>>
EnergyMonitoring

<<Pointcut>>+pcSubSystemClass(JPDD_SubSystemClass, EnergyMonitoringMechanism, ADD_NEW_FEATURE)
<<Pointcut>>+pcGetStartingEnergy(JPDD_PeriodicBehavior, StartingEnergyAmount, BEFORE)
<<Pointcut>>+pcCalcEnergy(JPDD_PeriodicBehavior, CalculateEnergyConsumption, AFTER)
<<StructuralAdaptation>>+EnergyMonitoringMechanism()
<<BehavioralAdaptation>>+StartingEnergyAmount()
<<BehavioralAdaptation>>+CalculateEnergyConsumption()

<<Aspect>>
MessageAck

<<Pointcut>>+pcSubSystemClass(JPDD_SubSystemClass, AcknowledgmentMechanism, ADD_NEW_FEATURE)
<<StructuralAdaptation>>+AcknowledgmentMechanism()
<<BehavioralAdaptation>>+SendAcknowledgment()()
<<BehavioralAdaptation>>+SinalAcknowledgmentMechanism()

<<Aspect>>
TaskMigration

<<Pointcut>>+pcSubSystemClass(JPDD_SubSystemClass, MigrationMechanism, ADD_NEW_FEATURE)
<<StructuralAdaptation>>+MigrationMechanism()
<<BehavioralAdaptation>>+Migrate()

Figure B.14.: Aspects Crosscutting Overview Diagram

package NFR[JPDD_ActiveObjectClass]

<<SchedulableResource>>
<<JoinPoint>>

*

Figure B.15.: JPDD: selection of active objects class

package NFR[JPDD_ExclusiveObjectClass]

<<JoinPoint>>
<<MutualExclusionResource>>

*

Figure B.16.: JPDD: selection of shared passive objects

package NFR[JPDD_InfoClassAttribute]

*Information

<<JoinPoint>>-* : *

Figure B.17.: JPDD: selection of passive class attributes

```
package NFR[ JPDD_InfoObjectConstruction_2 ]

            *Information

<<JoinPoint>>+*Information( .. ) : *{Behavior}
```

Figure B.18.: JPDD: selection of passive class constructor


```
package NFR[ JPDD_SubSystemClass ]

            <<JoinPoint>>
             *SubSystem
```

Figure B.19.: JPDD: selection of sub systems classes


```
package NFR [JPDD_SubSystemConstruction_2 ]

              *SubSystem

<<JoinPoint>>+*SubSystem(..):*(){Behavior}
```

Figure B.20.: JPDD: selection of sub systems constructor


```
interaction JPDD_ActiveObjectConstruction_Action [ JPDD_ActiveObjectConstruction_Action ]

       * : *
                        1:              <<SchedulableResource>>
                     <<JoinPoint>>            * : *
```

Figure B.21.: JPDD: selection of selection of active objects construction actions

Figure B.22.: JPDD: selection of active objects constructor behavior
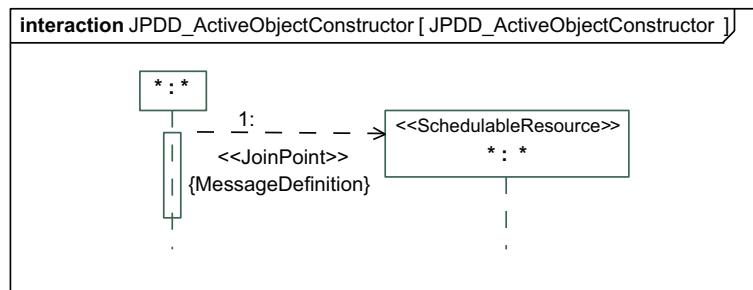
Figure B.23.: JPDD: selection of active objects constructor
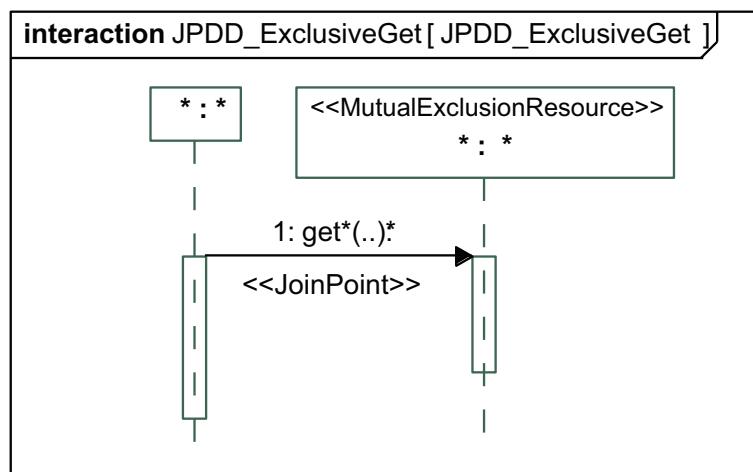
Figure B.24.: JPDD: selection of messages whose name starts with "get"
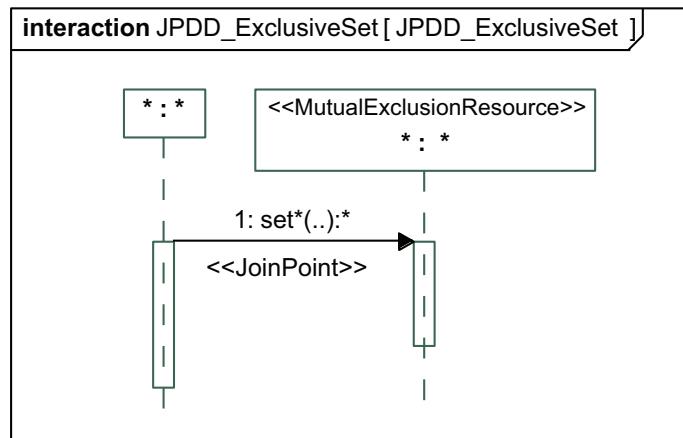
Figure B.25.: JPDD: selection of messages whose name starts with "set"
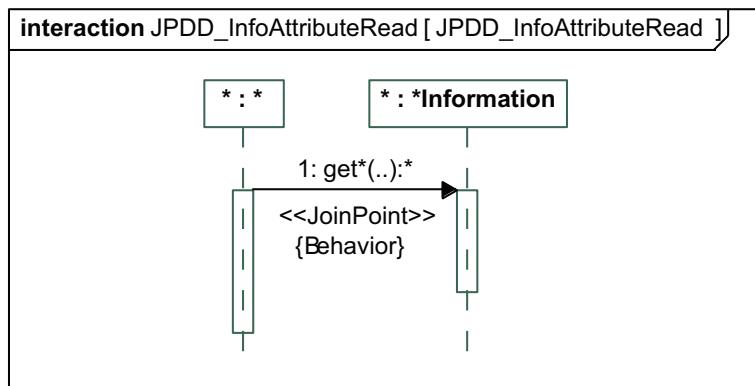
Figure B.26.: JPDD: selection of messages whose name starts with "get"
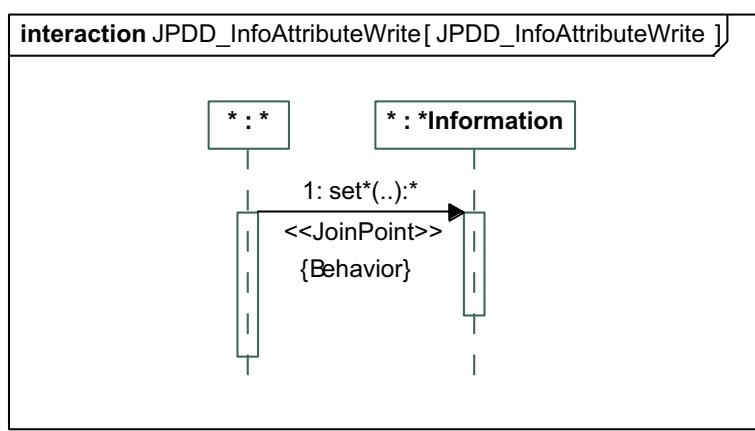
Figure B.27.: JPDD: selection of messages whose name starts with "set"
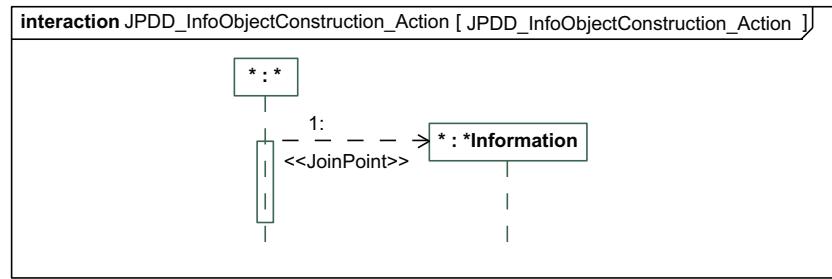
Figure B.28.: JPDD: selection of passive objects contruction action
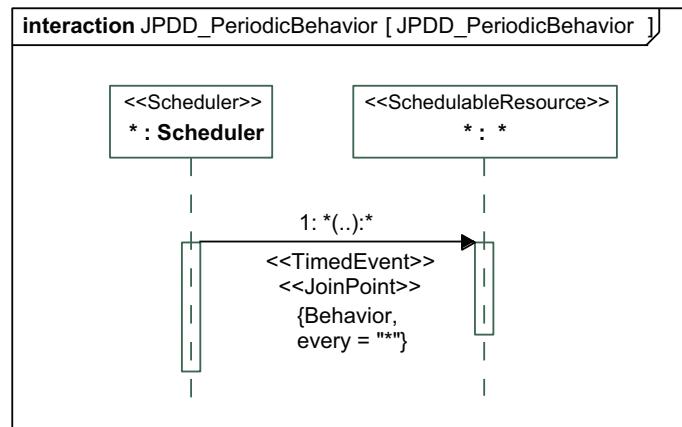
Figure B.29.: JPDD: selection of active objects periodic behavior
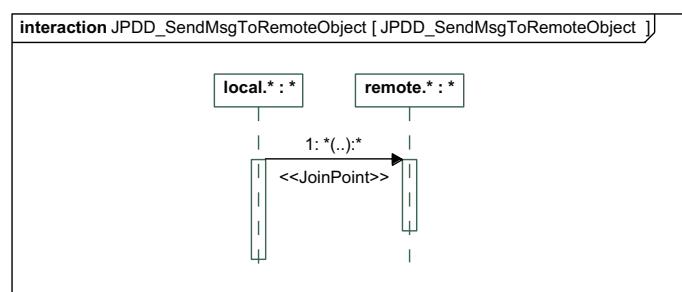
Figure B.30.: JPDD: selection of message sending action to remote objects
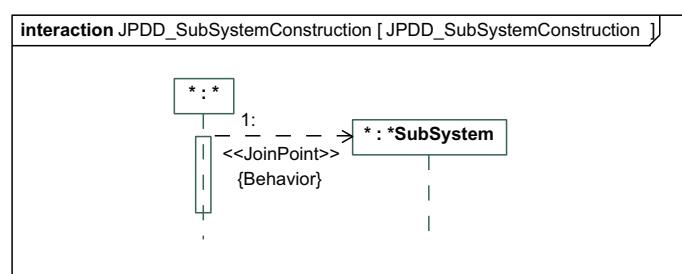
Figure B.31.: JPDD: selection of sub systems constructor behavior

Appendix C

Mapping Rules

C.1. Application

```xml
<?xml version="1.0" encoding="utf-8"?>
<Platforms>
  <!--**************************************************************************-->
  <!--*           Configuration for RT-FemtoJava Platform                    *-->
  <!--**************************************************************************-->
  <RT-FemtoJava>
    <!-- Mapping rules for APPLICATION CODE -->
    <Application>
      <Software>
        <!-- Source code generation options -->
        <SourceOptions isAspectLanguage="no" ClassesPerFile="1"
                       hasClassesDeclaration="no" Identation="5"
                       BlockStart="{" BlockEnd="}">
          <FileNameConvention>$Class.Name</FileNameConvention>
          <Package>package $Class.Package;</Package>
          <SourceReference>
            import $ReferencedClass.Package\.$ReferencedClass.Name;
          </SourceReference>
          <SourceOrganization>
            <DeclarationFile FileExtension="">
              $SourceCode.ClassesDeclaration
            </DeclarationFile>
            <ImplementationFile FileExtension=".java">
              $SourceCode.PackagesDeclaration
              \n$SourceCode.ReferencesDeclaration
              \n$SourceCode.ClassesImplementation
            </ImplementationFile>
          </SourceOrganization>
        </SourceOptions>


        <!-- Mapping rules for PRIMARY ELEMENTS -->
        <PrimaryElementsMapping>
          <DataTypes>
            <Array>
              #set( $n = $DataType.getSize() )
              #set( $s = $CodeGenerator.getDataTypeStr($DataType.DataType) )
              #if ($n > 0)
```

```
        $s[]
      #else
         ArrayList&lt;$s>
      #end
    </Array>
    <Boolean>boolean</Boolean>
    <Byte>byte</Byte>
    <Char>char</Char>
    <Class>$DataType.Represent.Name</Class>
    <DateTime>Date</DateTime>
    <EnumerationDefinition>
      public enum $DataType.Name
      $Options.BlockStart
      #foreach ($v in $DataType.Values)
        #if ($velocityCount > 1) , #end
        \n$v
      #end
      \n$Option.BlockEnd
    </EnumerationDefinition>
    <Enumeration>
      ${DataType.Name}
    </Enumeration>
    <Integer>int</Integer>
    <Long>long</Long>
    <Short>short</Short>
    <String>String</String>
    <Void>void</Void>
    <Double>double</Double>
    <Float>float</Float>
  </DataTypes>
  <DataTypeDefaultValues>
    <Array>null</Array>
    <Boolean>true</Boolean>
    <Byte>0</Byte>
    <Char>''</Char>
    <Class>null</Class>
    <DateTime>new Date(2000, 01, 01, 0, 0, 0)</DateTime>
    <Enumeration></Enumeration>
    <Integer>0</Integer>
    <Long>0</Long>
    <Short>0</Short>
    <String>""</String>
    <Void></Void>
    <Double>0.0</Double>
    <Float>0.0</Float>
  </DataTypeDefaultValues>
  <Visibilities>
    <Private>private</Private>
    <Protected>protected</Protected>
    <Public>public</Public>
  </Visibilities>
  <ParameterKinds>
    <In></In>
    <Out></Out>
    <InOut></InOut>
  </ParameterKinds>
</PrimaryElementsMapping>


<!-- Mapping rules for CLASSES -->
<Classes>
  <Declaration></Declaration>
```

```
<Implementation>
  import saito.sashimi.realtime.*;
  \n
  public
  #if ($Class.isAbstract())
    abstract
  #end
  class $Class.Name
  #if ($Class.SuperClass)
    extends $Class.SuperClass.Name
  #end
  \n$Options.BlockStart
  \n$CodeGenerator.getAttributesDeclaration(1)
  \n
  \n$CodeGenerator.getMessagesImplementation(1)
  \n$Options.BlockEnd
</Implementation>

<Attributes>
  $VisibilityStr
  #if ($Attribute.isStatic())
    static
  #end
  $DataTypeStr $Attribute.Name;
</Attributes>

<Messages>
  <Declaration></Declaration>
  <Implementation>
    #if ($DERCSHelper.isDestructor($Message) == false)
      $VisibilityStr
      #if ($Message.isStatic())
        static
      #end
      #if ($Message.isAbstract())
        abstract
      #end
      #if ($DERCSHelper.isNormalMethod($Message))
        $ReturnTypeStr
      #end
      ${Message.Name}(
      #if ($Message.ParametersCount > 0)
        #foreach( $param in $Message.Parameters )
          #if ($velocityCount > 1), #end
          $CodeGenerator.getDataTypeStr($param.DataType) $param.Name
        #end
      #end
      )
      #if (!$Message.isAbstract())
      $Options.BlockStart
        \n// Variables
        \n$CodeGenerator.getVariablesDeclaration(1)
        \n// Actions
        \n$CodeGenerator.getActionsCode(1)
      \n$Options.BlockEnd
      #else
        ;
      #end
    #else
      \n// ****************************
      \n//    destructor was ignored!
      \n// ****************************
    #end
```

```
      </Implementation>
    </Messages>
</Classes>


<!-- Mapping rules for BEHAVIOR, i.e. sequence of actions -->
<Behavior>
  <VariableDeclaration>
    $DataTypeStr $Variable.Name;
  </VariableDeclaration>

  <Branch>
    if ( $Branch.EnterCondition ) $Options.BlockStart
      #set( $ident = $IdentationLevel + 0)
      \n$CodeGenerator.getVariablesDeclaration($ident)
      \n$CodeGenerator.getActionsCode($ident)
    \n$Options.BlockEnd
    #if ( $Branch.hasAlternativeBehavior() )
      \n else $Options.BlockStart
        \n$CodeGenerator.getVariablesDeclaration($Branch.AlternativeBehavior,
                                                 $ident)
        \n$CodeGenerator.getActionsCode($Branch.AlternativeBehavior, $ident)
      \n$Options.BlockEnd
    #end
  </Branch>

  <Loop>
    #if ($Loop.NumberOfRepetitions > 0)
       for(int $IndexVariableName = 0; $IndexVariableName &lt;
                              $Loop.NumberOfRepetitions; $IndexVariableName++)
    #elseif ($Loop.ExitCondition)
      #if ($Loop.EnterCondition)
        ${Loop.EnterCondition};
      #end
      \n while ($Loop.ExitCondition)
    #end
    $Options.BlockStart
      \n$CodeGenerator.getVariablesDeclaration(1)
      \n$CodeGenerator.getActionsCode(1)
    \n$Options.BlockEnd
  </Loop>

  <Assignment>
    #if ($Action.isVariableAssignment())
      $Action.Variable.Name
    #else
      #if ($Action.Object)
        ${Action.Object.Name}.${Action.Attribute.Name}
      #else
        ${Action.Attribute.Name}
      #end
    #end
    =
    #if ($Action.isAssignmentOfValue())
      $Action.Value;
    #else
      $CodeGenerator.getActionCode($Action.Action)
    #end
  </Assignment>

  <Object>
    <Creation>
      #set( $x = 'nada' )
```

```
    new ${Action.Object.InstanceOf.Name}(
    #if ($Action.ParametersValuesCount > 0)
      #foreach( $x in $Action.ParametersValues )
        #if ($velocityCount > 1), #end
        $x
      #end
    #end
    );
  </Creation>

  <Destruction></Destruction>
</Object>

<Expression>
  #if ($DERCSHelper.isNormalMethod($Message))
    ${Action.Action.Expression}
  #else
    ${Action.Expression}
  #end
  ;
</Expression>

<Return>
  return
  #if ($Action.isAssignmentOfValue())
    ${Action.Value}
  #elseif ($Action.isAttributeAssignment())
    ${Action.Attribute.Name}
  #else
    $CodeGenerator.getActionCode($Action.Action)
  #end
  ;
</Return>

<StateChange></StateChange>

<SendMessage>
  <ToLocal>
    <Software>
      #if ($Action.getToObject() != $Action.getFromObject())
        #if ($Action.RelatedMethod.isStatic())
          ${Action.RelatedMethod.OwnerClass.Name}.
        #else
          ${Action.ToObject.Name}.
        #end
      #end
      ${Action.RelatedMethod.Name}(
      #if ($Action.ParametersValuesCount > 0)
        #foreach($param in $Action.getParametersValues())
          #if ($velocityCount > 1), #end
          #set( $x =  $velocityCount - 1)
          #if ($Action.isParameterValue($x))
            ${param}
          #else
            ${param.Name}
          #end
        #end
      #end
      );
    </Software>
    <Hardware></Hardware>
  </ToLocal>
```

```
    <ToRemote>
      <Software>
        #if ($Action.getToObject() != $Action.getFromObject())
          #if ($Action.RelatedMethod.isStatic())
            ${Action.RelatedMethod.OwnerClass.Name}.
          #else
            ${Action.ToObject.Name}.
          #end
        #end
        ${Action.RelatedMethod.Name}(
        #if ($Action.ParametersValuesCount > 0)
          #foreach($param in $Action.getParametersValues())
            #if ($velocityCount > 1), #end
            #set( $x =  $velocityCount - 1)
            #if ($Action.isParameterValue($x))
              ${param}
            #else
              ${param.Name}
            #end
          #end
        #end
      #end
      ); // ** REMOTE **
      </Software>
      <Hardware></Hardware>
    </ToRemote>
</SendMessage>

<InsertArrayElement>
  #if ($Action.isVariableAssignment())
    $Action.Variable.Name
  #else
    #if ($Action.Object)
      ${Action.Object.Name}.${Action.Attribute.Name}
    #else
      ${Action.Attribute.Name}
    #end
  #end
  .add(${Action.Element});
</InsertArrayElement>

<RemoveArrayElement>
  #if ($Action.isVariableAssignment())
    $Action.Variable.Name
  #else
    #if ($Action.Object)
      ${Action.Object.Name}.${Action.Attribute.Name}
    #else
      ${Action.Attribute.Name}
    #end
  #end
  .remove(${Action.Element});
</RemoveArrayElement>

<GetArrayElement>
  #if ($Action.isVariableAssignment())
    $Action.Variable.Name
  #else
    #if ($Action.Object)
      ${Action.Object.Name}.${Action.Attribute.Name}
    #else
      ${Action.Attribute.Name}
    #end
  #end
```

```
          .get(${Action.Element});
      </GetArrayElement>

      <SetArrayElement>
        #if ($Action.isVariableAssignment())
          $Action.Variable.Name
        #else
          #if ($Action.Object)
            ${Action.Object.Name}.${Action.Attribute.Name}
          #else
            ${Action.Attribute.Name}
          #end
        #end
        .set(${Action.Element});
      </SetArrayElement>

      <ArrayLength>
        #if ($Action.isVariableAssignment())
          $Action.Variable.Name
        #else
          #if ($Action.Object)
            ${Action.Object.Name}.${Action.Attribute.Name}
          #else
            ${Action.Attribute.Name}
          #end
        #end
        .size();
      </ArrayLength>
  </Behavior>


  <!-- Mapping rules for INTERRUPT HANDLING code -->
  <InterruptHandling>
  </InterruptHandling>


  <!-- Mapping rules for DERAF ASPECTS -->
  <Aspects>
    <!--****************************************************************-->
    <!--*                  Timing Package                            *-->
    <!--****************************************************************-->
    <TimingAttributes>
      <Declaration></Declaration>
      <Adaptations>
        <Structural Name="Deadline" Order="3" ModelLevel="no">
          private static RelativeTime _Deadline = new RelativeTime(0,0,0);
          \n
          \npublic void exceptionTask() {}
          \nprotected void initializeStack() {}
          \npublic void mainTask() {}
        </Structural>
        <Structural Name="Priority" Order="3">
        </Structural>
        <Structural Name="WCET" Order="3" ModelLevel="no">
          private static RelativeTime _Cost = new RelativeTime(0,0,0);
        </Structural>
        <Structural Name="ModityClassStructure" Order="0" ModelLevel="yes">
          $DERCSHelper.changeSuperClass($Class,
                  $DERCSFactory.newClass("RealtimeThread", null, true), true)
        </Structural>
        <Structural Name="ModifyConstructor" Order="0" ModelLevel="yes">
          $Message.addParameter("pDeadline", $DERCSFactory.newInteger(false),
                            $DERCSFactory.getParameterIn());
```

```
        $Message.addParameter("pCost", $DERCSFactory.newInteger(false),
                              $DERCSFactory.getParameterIn());
    </Structural>
    <Behavioral Name="SetTimingAttributes" Order="2" ModelLevel="no">
      \n_Deadline.set(0,pDeadline,0);
      \n_Cost.set(0,pCost,0);
      \ngetReleaseParameters().setDeadline(_Deadline);
      \ngetReleaseParameters().setCost(_Cost);
    </Behavioral>
    <Behavioral Name="AdaptObjectConstruction" Order="0" ModelLevel="yes">
      $Action.addParameterValue( $DERCSHelper.strTimeToInteger(
                              $Crosscutting.getValueOf("Deadline"), "ms" ) )
      $Action.addParameterValue( $DERCSHelper.strTimeToInteger(
                              $Crosscutting.getValueOf("WCET"), "ms" ) )
    </Behavioral>
    <Structural Name="AddAccessMethods" Order="3" ModelLevel="no">
      // TimingAttributes.AddAccessMethods
    </Structural>
    <Structural Name="StartTime" Order="3" ModelLevel="no">
      // TimingAttributes.StartTime
    </Structural>
    <Structural Name="EndTime" Order="3" ModelLevel="no">
      // TimingAttributes.EndTime
    </Structural>
  </Adaptations>
</TimingAttributes>

<PeriodicTiming>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="Period" Order="1" ModelLevel="no">
      \nprivate static RelativeTime _Period = new RelativeTime(0,0,0);
      \nprivate static PeriodicParameters _PeriodicParams =
                  new PeriodicParameters(null, null, null, null, null);
    </Structural>
    <Structural Name="ModifyConstructor" Order="1" ModelLevel="yes">
      $Message.addParameter("pPeriod", $DERCSFactory.newInteger(false),
                                      $DERCSFactory.getParameterIn());
    </Structural>
    <Behavioral Name="SetPeriod" Order="2" ModelLevel="no">
      \n_Period.set(0,pPeriod,0);
      \n_PeriodicParams.setPeriod(_Period);
      \nsetReleaseParameters(_PeriodicParams);
    </Behavioral>
    <Behavioral Name="FrequencyControl" Order="3" ModelLevel="no">
      waitForNextPeriod();
    </Behavioral>
    <Behavioral Name="LoopMechanism" Order="4" ModelLevel="no">
      while (isRunning()) $Options.BlockStart
        \n$CodeGenerator.getGeneratedCodeFragment(1)
      \n$Options.BlockEnd
    </Behavioral>
    <Behavioral Name="AdaptObjectConstruction" Order="1" ModelLevel="yes">
      $Action.addParameterValue( $DERCSHelper.strTimeToInteger(
                              $Crosscutting.getValueOf("Period"), "ms" ) )
    </Behavioral>
  </Adaptations>
</PeriodicTiming>

<SchedulingSupport>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="Scheduler" Order="0" ModelLevel="no">
```

```
        // SchedulingSupport.Begin
        \npublic static EDFScheduler scheduler = new EDFScheduler();
        \npublic void idleTask() {}
        \n// SchedulingSupport.End
      </Structural>
      <Behavioral Name="SetupConcurrentActivities" Order="0" ModelLevel="no">
        \n // SchedulingSupport
        \nScheduler.setDefaultScheduler(scheduler);
        \n
        #foreach( $Obj in $Message.TriggeredBehavior.BeharioralElements)
           #if ($DERCSHelper.isAssignmentOfActiveObject($Obj))
              #if ($Action.isVariableAssignment())
                 #set( $ObjName = $Obj.Variable.Name )
              #elseif ($Obj.Object)
                 #set( $ObjName = $Obj.Object.Name + '.' + $Obj.Attribute.Name)
              #else
                 #set( $ObjName = $Obj.Attribute.Name)
              #end
              \n
              \n${ObjName}.addToFeasibility();
              \n${ObjName}.start();
              \n
           #end
        #end
        \n
        \nscheduler.setupTimer();
        \nidleTask();
      </Behavioral>
    </Adaptations>
  </SchedulingSupport>

<TimeBoundedActivity>
  <Adaptations>
    <Structural Name="TimeCountInfrastructure" Order="0" ModelLevel="no">
      // TimeBoundedActivity.TimeCountInfrastructure
    </Structural>
    <Behavioral Name="StartCounting" Order="0" ModelLevel="no">
      // TimeBoundedActivity.StartCounting
    </Behavioral>
    <Behavioral Name="StopCounting" Order="0" ModelLevel="no">
      // TimeBoundedActivity.StopCounting
    </Behavioral>
  </Adaptations>
</TimeBoundedActivity>

<!--****************************************************************-->
<!--*              Precision Package                          *-->
<!--****************************************************************-->
<Jitter>
  <Adaptations>
    <Behavioral Name="StartTime" Order="0" ModelLevel="no">
      // Jitter.StartTime
    </Behavioral>
    <Behavioral Name="VerifyToleratedJitter" Order="0" ModelLevel="no">
      // Jitter.VerifyToleratedJitter
    </Behavioral>
  </Adaptations>
</Jitter>

<ToleratedDelay>
  <Adaptations>
    <Behavioral Name="StartTime" Order="0" ModelLevel="no">
      // ToleratedDelay.StartTime
```

183

```
      </Behavioral>
      <Behavioral Name="VerifyToleratedDelay" Order="0" ModelLevel="no">
        // ToleratedDelay.VerifyToleratedDelay
      </Behavioral>
    </Adaptations>
</ToleratedDelay>

<ClockDrift>
  <Adaptations>
    <Behavioral Name="CheckClockDrift" Order="0" ModelLevel="no">
      // ClockDrift.CheckClockDrift
    </Behavioral>
  </Adaptations>
</ClockDrift>

<DataFreshness>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="ValidityInformation" Order="0" ModelLevel="no">
      // freshness: ${Attribute.Name}
      \nprivate static AbsoluteTime ${Attribute.Name}_Validity =
                                        new AbsoluteTime(0,0,0);
      \nprivate static AbsoluteTime ${Attribute.Name}_NextValidity =
                                        new AbsoluteTime(0,0,0);
      \n
      \npublic void set${Attribute.Name}Validity(int newValidity)
                                                    $Options.BlockStart
      \n    ${Attribute.Name}_Validity.set(0,newValidity,0);
      \n$Options.BlockEnd
      \n//freshness: ${Attribute.Name}
      \n
    </Structural>
    <Structural Name="SetValidity" Order="0" ModelLevel="no">
      #set( $ObjName = '---')
      #if ($Action.isVariableAssignment())
          #set( $ObjName = $Action.Variable.Name )
      #elseif ($Action.Object)
          #set( $ObjName = $Action.Object.Name + '.' + $Action.Attribute.Name)
      #else
          #set( $ObjName = $Action.Attribute.Name)
      #end
      // begin of freshness setup
      \n
        #foreach( $NFR in ${Crosscutting.CrosscutingInformations} )
          #if ( $NFR.Name == "Validity" )
            #if ( $NFR.ElementName == $NFR.Name)
              #foreach ( $Attr in
                $Crosscutting.getAffectedElement().getAttributes() )
                \n${ObjName}.set${Attr.Name}Validity(
                  $DERCSHelper.strTimeToInteger( $NFR.Value, "ms" ));
                  // freshness
                \n
              #end
            #else
              \n${ObjName}.set${NFR.getElementName()}Validity(
                  $DERCSHelper.strTimeToInteger( $NFR.Value, "ms" ));
                  // freshness
              \n
            #end
          #end
        #end
      \n // end of freshness setup
    </Structural>
```

```
  <Behavioral Name="VerifyFreshness" Order="0" ModelLevel="no">
    #if ($Message.AssociatedAttribute)
      if (${Message.AssociatedAttribute.Name}_NextValidity.compareTo(
                        Clock.getTime()) >= 0) $Options.BlockStart
        \n$CodeGenerator.getGeneratedCodeFragment(1)
      \n$Options.BlockEnd
      \nelse $Options.BlockStart
      \n  ${Message.AssociatedAttribute.Name} =
                      ${Message.AssociatedAttribute.Name} * 90 / 100;
      \n$Options.BlockEnd
    #end
  </Behavioral>
  <Behavioral Name="UpdateFreshness" Order="0" ModelLevel="no">
    \n${Message.AssociatedAttribute.Name}_NextValidity.set(
                                        Clock.getTime());
    \n${Message.AssociatedAttribute.Name}_NextValidity.add(
                      ${Message.AssociatedAttribute.Name}_Validity);
  </Behavioral>
  </Adaptations>
</DataFreshness>

<!--*****************************************************************-->
<!--*                 Synchronization Package                      *-->
<!--*****************************************************************-->
<ConcurrentAccessControl>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="ConcurrencyControlMechanism" Order="0" ModelLevel="no">
      // ConcurrentAccessControl.ConcurrencyControlMechanism
    </Structural>
    <Behavioral Name="AcquireAccess" Order="0" ModelLevel="no">
      // ConcurrentAccessControl.AquireAccess
    </Behavioral>
    <Behavioral Name="ReleaseAccess" Order="0" ModelLevel="no">
      // ConcurrentAccessControl.ReleaseAccess
    </Behavioral>
  </Adaptations>
</ConcurrentAccessControl>

<MessageSynchronization>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="WaitingMechanism" Order="0" ModelLevel="no">
      // MessageSynchronization.WaitingMechanism
    </Structural>
    <Behavioral Name="WaitForAcknowledge" Order="0" ModelLevel="no">
      // MessageSynchronization.WaitForAcknowledge
    </Behavioral>
  </Adaptations>
</MessageSynchronization>

<!--*****************************************************************-->
<!--*                 Communication Package                        *-->
<!--*****************************************************************-->
<MessageAck>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="AcknowledgmentMechanism" Order="0" ModelLevel="no">
      // MessageAck.AcknowledgeMechanism
    </Structural>
    <Behavioral Name="SignalAcknowledgmentMechanism" Order="0"
                                                ModelLevel="no">
      // MessageAck.SignalAcknowledgeMechanism
```

```
      </Behavioral>
      <Behavioral Name="SendAcknowledgment" Order="0" ModelLevel="no">
        // MessageAck.SendAcknowledge
      </Behavioral>
    </Adaptations>
</MessageAck>

<MessageIntegrity>
  <Declaration></Declaration>
  <Adaptations>
    <Behavioral Name="GenerateIntegrityInfo" Order="0" ModelLevel="no">
      // MessageIntegrity.GenerateIntegrityInfo
    </Behavioral>
    <Behavioral Name="VerifyIntegrityInfo" Order="0" ModelLevel="no">
      // MessageIntegrity.VerifyIntegrityInfo
    </Behavioral>
  </Adaptations>
</MessageIntegrity>

<MessageCompression>
  <Declaration></Declaration>
  <Adaptations>
    <Behavioral Name="Compress" Order="0" ModelLevel="no">
      // MessageCompression.Compress
    </Behavioral>
    <Behavioral Name="Decompress" Order="0" ModelLevel="no">
      // MessageCompression.Decompress
    </Behavioral>
  </Adaptations>
</MessageCompression>

<!--****************************************************************-->
<!--*                 TaskAllocation Package                      *-->
<!--****************************************************************-->
<NodeStatusRetrieval>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="Alive" Order="0" ModelLevel="no">
      // NodeStatusRetrieval.Alive
    </Structural>
    <Behavioral Name="ProcessingLoad" Order="0" ModelLevel="no">
      // NodeStatusRetrieval.ProcessingLoad
    </Behavioral>
    <Behavioral Name="MessageThroughput" Order="0" ModelLevel="no">
      // NodeStatusRetrieval.MessageThroughput
    </Behavioral>
  </Adaptations>
</NodeStatusRetrieval>

<TaskMigration>
  <Declaration></Declaration>
  <Adaptations>
    <Behavioral Name="Migrate" Order="0" ModelLevel="no">
      // TaskMigration.Migrate
    </Behavioral>
    <Structural Name="MigrationMechanism" Order="0" ModelLevel="no">
      // TaskMigration.MigrationMechanism
    </Structural>
  </Adaptations>
</TaskMigration>

<!--****************************************************************-->
<!--*                   Embedded Package                          *-->
```

```
<!--**************************************************-->
<HwAreaMonitoring>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="HwAreMonitoringMechanism" Order="0" ModelLevel="no">
      // HwAreaMonitoring.HwAreMonitoringMechanism
    </Structural>
    <Behavioral Name="IncreaseAreaUsage" Order="0" ModelLevel="no">
      // HwAreaMonitoring.IncreaseAreaUsage
    </Behavioral>
    <Behavioral Name="DecreaseAreaUsage" Order="0" ModelLevel="no">
      // HwAreaMonitoring.DecreaseAreaUsage
    </Behavioral>
  </Adaptations>
</HwAreaMonitoring>

<HwAreaControl>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="HwAreaControlPolicy" Order="0" ModelLevel="no">
      // HwAreaControl.InsertControlMechanism
    </Structural>
  </Adaptations>
</HwAreaControl>

<EnergyMonitoring>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="EnergyMonitoringMechanism" Order="0" ModelLevel="no">
      // EnergyMonitoring.EnergyMonitoringMechanism
    </Structural>
    <Behavioral Name="StartingEnergyAmount" Order="0" ModelLevel="no">
      // EnergyMonitoring.StartingEnergyAmount
    </Behavioral>
    <Behavioral Name="CalculateEnergyConsumption" Order="0" ModelLevel="no">
      // EnergyMonitoring.CalculateEnergyConsumption
    </Behavioral>
  </Adaptations>
</EnergyMonitoring>

<EnergyControl>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="EnergyConsumptionPolicy" Order="0" ModelLevel="no">
      // EnergyControl.EnergyConsumptionPolicy
    </Structural>
  </Adaptations>
</EnergyControl>

<MemoryUsageMonitoring>
  <Declaration></Declaration>
  <Adaptations>
    <Structural Name="MemoryMonitoringMechanism" Order="0" ModelLevel="no">
      // MemoryUsageMonitoring.MemoryMonitoringMechanism
    </Structural>
    <Behavioral Name="IncreaseMemoryUsage" Order="0" ModelLevel="no">
      // MemoryUsageMonitoring.IncreaseMemoryUsage
    </Behavioral>
    <Behavioral Name="DecreaseMemoryUsage" Order="0" ModelLevel="no">
      // MemoryUsageMonitoring.DecreaseMemoryUsage
    </Behavioral>
  </Adaptations>
</MemoryUsageMonitoring>
```

```
          <MemoryControl>
            <Declaration></Declaration>
            <Adaptations>
              <Structural Name="MemoryUsageControlPolicy" Order="0" ModelLevel="no">
                // MemoryControl.MemoryUsageControlPolicy
              </Structural>
            </Adaptations>
          </MemoryControl>
        </Aspects>
      </Software>

      <Hardware></Hardware>
    </Application>


    <!-- Mapping rules for PLATFORM CODE -->
    <PlatformConfiguration>
      <Software>
        <SourceOptions OutputDirectory="platform"></SourceOptions>
        <Files xmlns:xi="http://www.w3.org/2001/XInclude">
          <xi:include href="./platform_RT-FemtoJava/AbsoluteTime.xml"/>
          <xi:include href="./platform_RT-FemtoJava/AbstractPoolingServer.xml"/>
          <xi:include href="./platform_RT-FemtoJava/AperiodicParameters.xml"/>
          <xi:include href="./platform_RT-FemtoJava/AsyncEvent.xml"/>
          <xi:include href="./platform_RT-FemtoJava/AsyncEventHandler.xml"/>
          <xi:include href="./platform_RT-FemtoJava/AsyncEventsMechanism.xml"/>
          <xi:include href="./platform_RT-FemtoJava/Clock.xml"/>
          <xi:include href="./platform_RT-FemtoJava/EDFScheduler.xml"/>
          <xi:include href="./platform_RT-FemtoJava/FixedPriorityHWScheduler.xml"/>
          <xi:include href="./platform_RT-FemtoJava/HighResolutionTime.xml"/>
          <xi:include href="./platform_RT-FemtoJava/HWRealtimeThread.xml"/>
          <xi:include href="./platform_RT-FemtoJava/InterruptPoolingMechanism.xml"/>
          <xi:include href="./platform_RT-FemtoJava/OneShotTimer.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PeriodicParameters.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PeriodicTimer.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PoolingServer2.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PoolingServer1.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PriorityParameters.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PriorityScheduler.xml"/>
          <xi:include href="./platform_RT-FemtoJava/PriorityScheduler2.xml"/>
          <xi:include href="./platform_RT-FemtoJava/RateMonotonicScheduler.xml"/>
          <xi:include href="./platform_RT-FemtoJava/RealtimeThread.xml"/>
          <xi:include href="./platform_RT-FemtoJava/RelativeTime.xml"/>
          <xi:include href="./platform_RT-FemtoJava/ReleaseParameters.xml"/>
          <xi:include href="./platform_RT-FemtoJava/Scheduler.xml"/>
          <xi:include href="./platform_RT-FemtoJava/SchedulingParameters.xml"/>
          <xi:include href="./platform_RT-FemtoJava/SporadicParameters.xml"/>
          <xi:include href="./platform_RT-FemtoJava/Timer.xml"/>
          <xi:include href="./platform_RT-FemtoJava/TimeTriggeredRealtimeThread.xml"/>
          <xi:include href="./platform_RT-FemtoJava/TimeTriggeredScheduler.xml"/>
        </Files>
      </Software>
      <Hardware></Hardware>
    </PlatformConfiguration>
  </RT-FemtoJava>
</Platforms>
```

C.2. Platform Configuration

```
<File Name="Scheduler.java" OutputDirectory="saito.sashimi.realtime"
```

```
    Aspects="SchedulingSupport">
<Fragment>
  package saito.sashimi.realtime;
  import saito.sashimi.*;

  public abstract class Scheduler implements TimerInterface  {
      protected static Scheduler c_defaultScheduler = null;
      protected static int m_MainBaseStackPointer = 0xFFFF;

      protected int m_currentTask = -1;
      protected boolean m_Processing = false;
</Fragment>
<Fragment Aspects="TimingAttributes">
      public final static int MAX_APERIODIC_TASKS = 16;
      protected static RealtimeThread m_AperiodicTaskList[] =
                              {null, null, null, null, null, null, null, null,
                               null, null, null, null, null, null, null, null};
      protected int m_AperiodicListCount = 0;
</Fragment>
<Fragment Aspects="PeriodicTiming">
      public final static int MAX_PERIODIC_TASKS = 16;
      protected static RealtimeThread m_PeriodicTaskList[] =
                              {null, null, null, null, null, null, null, null,
                               null, null, null, null, null, null, null, null};
      protected int m_PeriodicListCount = 0;
</Fragment>
<Fragment>
      public abstract boolean isFeasible();
      public abstract void runScheduler();
      protected abstract int getContextOffsetForStaticMethod();
      protected abstract int getContextOffsetForVirtualMethod();

      protected static int indexOf(RealtimeThread list[], int listCount,
                                   RealtimeThread schedulable) {
        int i = 0;
        for (; (i &lt; listCount) &amp;&amp; (list[i] != schedulable); i++);
        if (i &lt; listCount)
         return i;
        else
         return -1;
      }

      protected static void addToListOrderByPriority(RealtimeThread list[],
                    int listCount, RealtimeThread schedulable, int priority) {
        int i = 0;
        for(; (i &lt; listCount) &amp;&amp; (priority &lt;=
        ((PriorityParameters)list[i].getSchedulingParameters()).getPriority());
        i++);
        if (i &lt; listCount) {
          for(int j = listCount; j > i; j--)
             list[j] = list[j-1];
        }
        list[i] = schedulable;
      }

      protected boolean addToFeasibility( RealtimeThread schedulable ) {
</Fragment>
<Fragment Aspects="PeriodicTiming">
        if ((m_PeriodicListCount &lt; (MAX_PERIODIC_TASKS-1)) &amp;&amp;
            (indexOf(schedulable) == -1)) {
          m_PeriodicTaskList[m_PeriodicListCount] = schedulable;
          m_PeriodicListCount++;
          return true;
```

```
        }
        else
</Fragment>
<Fragment>
        return false;
      }

    protected boolean removeFromFeasibility( RealtimeThread schedulable ) {
</Fragment>
<Fragment Aspects="PeriodicTiming">
        if (m_PeriodicListCount > 0) {
          int i = 0;
          for(; (i &lt; m_PeriodicListCount) &amp;&amp;
                (m_PeriodicTaskList[i] != schedulable); i++);
          if (i &lt; m_PeriodicListCount) {
            m_PeriodicTaskList[i] = null;
            m_PeriodicListCount--;
            if (i &lt; m_PeriodicListCount) {
              int j = i;
              for(; (j &lt;= m_PeriodicListCount); j++)
                 m_PeriodicTaskList[j] = m_PeriodicTaskList[j+1];
              m_PeriodicTaskList[j] = null;
            }
            return true;
          }
          else
            return false;
        }
        else
</Fragment>
<Fragment>
        return false;
      }

    public static Scheduler getDefaultScheduler() {
        return c_defaultScheduler;
      }

    protected int indexOf( RealtimeThread schedulable ) {
</Fragment>
<Fragment Aspects="PeriodicTiming">
        int i = 0;
        for (; (i &lt; MAX_PERIODIC_TASKS) &amp;&amp;
               (m_PeriodicTaskList[i] != schedulable); i++);
        if (i &lt; MAX_PERIODIC_TASKS)
          return i;
        else
</Fragment>
<Fragment>
        return -1;
      }

    public boolean isAddedToFeasibility( RealtimeThread schedulable ) {
        return ( indexOf(schedulable) != -1 );
      }

    public static void setDefaultScheduler( Scheduler scheduler ) {
        c_defaultScheduler = scheduler;
      }

    public static void saveMainContext() {
        m_MainBaseStackPointer = FemtoJavaSO.saveCTX() +
                    getDefaultScheduler().getContextOffsetForStaticMethod();
```

```
        }

        public static void restoreMainContext() {
          c_defaultScheduler.m_Processing = false;
          FemtoJavaSO.restoreCTX(m_MainBaseStackPointer);
        }

        public boolean isAllTasksFinished() {
          boolean result = true;
  </Fragment>
  <Fragment Aspects="PeriodicTiming">
          for (int i=0; (i &lt; m_PeriodicListCount) &amp;&amp; result; i++)
            result &amp;= m_PeriodicTaskList[i].isFinished();
  </Fragment>
  <Fragment>
          return result;
        }

        public void setupTimer() {
          FemtoJavaInterruptSystem.setEnable(0x2F);
          FemtoJavaTimer.setTimer0(100);
          FemtoJavaTimer.startTimer0();
        }

        public void tf0Method() {
          FemtoJavaInterruptSystem.setEnable(0x6F);
          FemtoJavaTimer.stopTimer0();
          if (!c_defaultScheduler.m_Processing) {
            c_defaultScheduler.m_Processing = true;
            FemtoJavaInterruptSystem.setEnable(0x2F);
            c_defaultScheduler.runScheduler();
            c_defaultScheduler.m_Processing = false;
          }
        }

        public void tf1Method() {} // not used ! Used in Timer objects
    }
  </Fragment>
</File>
```

C.3. Source Code Generated by GenERTiCA

```
import saito.sashimi.realtime.*;
public class MovementController  extends RealtimeThread {
    private  SpecialConditionMovementControl ctrlMode;
    private  EnvironmentInformation envInfo;
    private  MovementInformation mrInfo;
    private  MainRotorActuator mrAct;
    private  MovementInformation brInfo;
    private  BackRotorActuator brAct;
    private  int newMRRotation;
    private  int newMRPace;
    private  int newBRRotation;
    private  int newBRPace;
    private  Alarm alarm;

    private static RelativeTime _Cost = new RelativeTime(0,0,0);
    private static RelativeTime _Deadline = new RelativeTime(0,0,0);

    public void exceptionTask() {}
    protected void initializeStack() {}
```

```
public void mainTask() {}

private static RelativeTime _Period = new RelativeTime(0,0,0);
private static PeriodicParameters _PeriodicParams =
                     new PeriodicParameters(null, null, null, null, null);

 public    MovementController(    EnvironmentInformation _envInfo  ,
                                  MovementInformation _mrInfo  ,
                                  MainRotorActuator _mrAct  ,
                                  MovementInformation _brInfo  ,
                                  BackRotorActuator _brAct  ,
                                  Alarm _alarm  ,  int pDeadline  ,
                                  int pCost  ,  int pPeriod  ) {
// Variables
// Actions
         ctrlMode = new SpecialConditionMovementControl();
         envInfo = _envInfo;
         mrInfo = _mrInfo;
         mrAct = _mrAct;
         brInfo = _brInfo;
         brAct = _brAct;
         alarm = _alarm;

     _Deadline.set(0,pDeadline,0);
     _Cost.set(0,pCost,0);
     getReleaseParameters().setDeadline(_Deadline);
     getReleaseParameters().setCost(_Cost);

     _Period.set(0,pPeriod,0);
     _PeriodicParams.setPeriod(_Period);
     setReleaseParameters(_PeriodicParams);
}
 public    EnvironmentInformation  getenvInfo(  ) {
// Variables
// Actions
     return  envInfo ;
}
 public    void  setenvInfo(    EnvironmentInformation _envInfo  ) {
// Variables
// Actions
     envInfo = _envInfo;
}
 public    MovementInformation  getmrInfo(  ) {
// Variables
// Actions
     return  mrInfo ;
}
 public    void  setmrInfo(    MovementInformation _mrInfo  ) {
// Variables
// Actions
     mrInfo = _mrInfo;
}
 public    MainRotorActuator  getmrAct(  ) {
// Variables
// Actions
     return  mrAct ;
}
 public    void  setmrAct(    MainRotorActuator _mrAct  ) {
// Variables
// Actions
     mrAct = _mrAct;
}
 public    MovementInformation  getbrInfo(  ) {
```

```
// Variables
// Actions
    return  brInfo ;
}
 public    void setbrInfo(    MovementInformation _brInfo  ) {
// Variables
// Actions
    brInfo = _brInfo;
}
 public   BackRotorActuator getbrAct( ) {
// Variables
// Actions
    return  brAct ;
}
 public    void setbrAct(    BackRotorActuator _brAct  ) {
// Variables
// Actions
    brAct = _brAct;
}
 public   Alarm getalarm( ) {
// Variables
// Actions
    return  alarm ;
}
 public    void setalarm(    Alarm _alarm  ) {
// Variables
// Actions
    alarm = _alarm;
}
 public    void run( ) {
// Variables
    int brRotation;
    int brPace;
    int mrRotation;
    int mrPace;
    float windSpeed;
    float windDirection;
    float humidity;
    float temperature;
// Actions
    while (isRunning()) {
        // EnergyMonitoring.StartingEnergyAmount
        brRotation =  brInfo.getRotation(); // ** REMOTE **
        // MessageSynchronization.WaitForAcknowledge
        brPace =  brInfo.getPace(); // ** REMOTE **
        // MessageSynchronization.WaitForAcknowledge
        // ConcurrentAccessControl.AquireAccess
        mrRotation =  mrInfo.getRotation();
        // ConcurrentAccessControl.ReleaseAccess
        // ConcurrentAccessControl.AquireAccess
        mrPace =  mrInfo.getPace();
        // ConcurrentAccessControl.ReleaseAccess
        // ConcurrentAccessControl.AquireAccess
        windSpeed =  envInfo.getWindSpeed();
        // ConcurrentAccessControl.ReleaseAccess
        // ConcurrentAccessControl.AquireAccess
        windDirection =  envInfo.getWindDirection();
        // ConcurrentAccessControl.ReleaseAccess
        // ConcurrentAccessControl.AquireAccess
        humidity =  envInfo.getHumidity();
        // ConcurrentAccessControl.ReleaseAccess
        // ConcurrentAccessControl.AquireAccess
        temperature =  envInfo.getTemperature();
```

```
            // ConcurrentAccessControl.ReleaseAccess
            processInfo(mrRotation, mrPace, brRotation, brPace, windSpeed,
                        windDirection, humidity, temperature);
            mrAct.setRotation(newMRRotation);
            mrAct.setPace(newMRPace);
            brAct.setRotation( newBRRotation); // ** REMOTE **
            // MessageSynchronization.WaitForAcknowledge
            brAct.setPace( newBRPace); // ** REMOTE **
            // MessageSynchronization.WaitForAcknowledge
            mrAct.applyParameters();
            brAct.applyParameters(); // ** REMOTE **
            // MessageSynchronization.WaitForAcknowledge
            // EnergyMonitoring.CalculateEnergyConsumption
            waitForNextPeriod();
        }
    }
     public    void processInfo(    int r1  , int p1  , int r2  , int p2  ,
                        float ws  , float wd  , float h  , float t   ) {
    // Variables
    // Actions
        if ( (((int)ws) > 15) || (((int)h) > 85) || (((int)t) > 40) ) {
            ctrlMode.setMode(SpecialConditionMovementControl.HOSTILE_ENVIRONMENT);
        }
        if ( ctrlMode.getMode() == SpecialConditionMovementControl.NORMAL ) {
            newMRRotation =  newMRRotation/r1*p1 ;
            newMRPace = (newMRPace+p1)/r1;
            newBRRotation =  newBRRotation/r2*p2 ;
            newBRPace = (newBRPace+p2)/r2;
        }  else {

            if ( ctrlMode.getMode() ==
                    SpecialConditionMovementControl.HOSTILE_ENVIRONMENT ) {
                alarm.triggerHostileEnvironmentAlarm();
            }  else {

                if ( ctrlMode.getMode() ==
                            SpecialConditionMovementControl.LOW_FUEL ) {
                    alarm.triggerLowFuelAlarm();
                }  else {

                    if ( ctrlMode.getMode() ==
                        SpecialConditionMovementControl.UNDER_ATTACK ) {
                        alarm.triggerUnderAttackAlarm();
                    }
                }
            }
        }
    }
}
```

Bibliography

[1] Ákos Lédeczi et al. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, 2001.

[2] Per Andersson and Martin Höst. Uml and systemc: A comparison and mapping rules for automatic code generation. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, pages 199–209. Springer Netherlands, 2008.

[3] Apache. Apache velocity project, 2008. http://velocity.apache.org/.

[4] Deborah J. Armstrong. The quarks of object-oriented development. *Communication of the ACM*, 49(2):123–128, 2006.

[5] Tero Arpinen et al. Configurable multiprocessor platform with rtos for distributed execution of uml 2.0 designed applications. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1324–1329, Leuven, 2006. European Design and Automation Association.

[6] Artisan. Artisan real-time studio, 2008. http://www.artisansoftwaretools.com/products/artisan-studio/.

[7] Felice Balarin et al. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[8] Krishnakumar Balasubramanian et al. A platform-independent component modeling language for distributed real-time and embedded systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–199, Los Alamitos, 2005. IEEE Computer Society.

[9] Krishnakumar Balasubramanian et al. Weaving deployment aspects into domain-specific models. *International Journal of Software Engineering and Knowledge Engineering*, 16(3):403–424, 2006.

[10] Silvia Castro Bertagnolli. *FRIDA: um método para elicitação e modelagem de RNFs*. PhD thesis, Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2004.

[11] Danilo Beuche et al. The pure family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 45–53, Washington, 1999. IEEE Computer Society.

[12] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, May 2005.

[13] Gregory Bollella et al. *The Real-Time Specification for Java, version 1.0.2*. Addison Wesley Longman, second edition, 2001.

[14] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Massachusetts, 1994.

[15] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition)*. Addison-Wesley, 2005.

[16] Matteo Bordin and Tullio Vardanega. Real-time java from an automated code generation perspective. In *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 63–72, New York, 2007. ACM.

[17] Borland. Borland together, 2008. http://www.borland.com/us/products/together/.

[18] BOSCH. Can 2.0 protocol specification, 1991. http://www.can-cia.org/index.php?id=164.

[19] James Brusey et al. Auto-id based control demonstration - phase 2: Pick and place packing with holonic control. Technical report, Cambridge University, 2003. http://www.ifm.eng.cam.ac.uk/automation/publications/documents/CAM-AUTOID-WH011.pdf.

[20] Sven Burmester et al. The fujaba real-time tool suite: Model-driven development of safety-critical, real-time systems. In *Proceedings of the 27th International Conference on Software Engineering*, pages 670–671, New York, 2005. ACM.

[21] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications*, pages 25–40, Berlin, 2005. Springer.

[22] A. Burns et al. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46(4):305–325, 2000.

[23] Alan Burns and Andrew J. Wellings. Hrt-hood: a structured design method for hard real-time systems. *Real-Time Systems*, 6(1):73–114, 1994.

[24] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, Harlow, second edition, 1997.

[25] Luigi Carro and Flavio Rech Wagner. Sistemas computacionais embarcados. In *Jornadas de Atualização em Informática*, number 22, pages 45–94. SBC, Campinas, 2003.

[26] Vaclav Cechticky et al. A uml2 profile for reusable and verifiable software components for real-time applications. In *Proceedings of the 9th International Conference on Software Reuse*, pages 312–325, Berlin, 2006. Springer.

[27] Rong Chen et al. Uml and platform-based design. In Luciano Lavagno, Grant Martin, and Bran Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 107–126. Kluwer Academic Publishers, Norwell, 2003.

[28] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[29] Siobhán Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming: Special issue on Unified Modeling Language*, 44(1):71–100, 2002.

[30] Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, Upper Sadde River, 2005.

[31] Siobhán Clarke and Robert J. Walker. Towards a standard design language for aosd. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 113–119, New York, 2002. ACM.

[32] Edmund H. Conrow and Patricia S. Shishido. Implementing risk management on software intensive projects. *IEEE Software*, 14(3):83–89, 1997.

[33] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[34] Elias Teodoro da Silva Jr. *Middleware Adaptativo para Sistemas Embarcados e de Tempo-Real*. PhD thesis, Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2008.

[35] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Commununication of the ACM*, 9(9):671–678, 1966.

[36] Abhijit Davare et al. A next-generation design framework for platform-based design. In *Proceedings of the Conference on Using Hardware Design and Verification Languages*, 2007.

[37] Edison Pignaton de Freitas. Metodologia orientada a aspectos para a especificação de sistemas tempo-real embarcados e distribuídos. Master's thesis, Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2007.

[38] Carsten Ditze. *Towards Operating System Synthesis*. PhD thesis, Department of Mathematics and Computer Science, University of Paderborn, Paderborn, 2000.

[39] Francisco A. M. do Nascimento, Marcio F. da S. Oliveira, Marco A. Wehrmeister, Carlos E. Pereira, and Flávio R. Wagner. Mda-based approach for embedded software generation from a uml/mof repository. In *Proceedings of the 19th annual Symposium on Integrated Circuits and Systems Design*, pages 143–148, New York, 2006. ACM.

[40] DomainSolutions. Codegenie mdd, 2008. http://www.domainsolutions.co.uk/.

[41] Martyn Edwards and Peter Green. Uml for hardware and software object modeling. In Luciano Lavagno, Grant Martin, and Bran Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 127–147. Kluwer Academic Publishers, Norwell, 2003.

[42] Robert E. Filman et al., editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[43] Robert France et al. Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, 151(4):173–185, Aug. 2004.

[44] Lidia Fuentes, Jorge Manrique, and Pablo Sánchez. Pópulo: a tool for debugging uml models. In *Proceedings of the 30th International Conference on Software Engineering*, pages 955–956, New York, 2008. ACM.

[45] Lidia Fuentes, Mónica Pinto, and José María Troya. Supporting the development of camdaop applications: an integrated development process. *Software-Practice & Experience*, 37(1):21–64, 2007.

[46] Lidia Fuentes and Pablo Sánchez. Elaborating uml 2.0 profiles for ao design. In *Proceedings of the 8th Workshop on Aspect-Oriented Modelling (AOM)*, 2006.

[47] Lidia Fuentes and Pablo Sánchez. Designing and weaving aspect-oriented executable uml models. *Journal of Object Technology*, 6(7):109–136, 2007.

[48] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing, Boston, 1995.

[49] Murray Gell-Mann. *The quark and the jaguar: adventures in the simple and the complex*. W. H. Freeman & Co., New York, 1995.

[50] Gentleware. Poseidon for uml, 2008. http://www.gentleware.com/uml-software-pe.html.

[51] Sébastien Gérard and Bran Selic. The uml – marte standardized profile. In *Proceedings of the 17th World Congress of the International Federation of Automatic Control*, pages 6909–6913, 2008.

[52] GSRC. Metropolis: Design environment for heterogeneous systems, 2002. http://www.gigascale.org/metropolis/index.html.

[53] A. N. Habermann, Lawrence Flon, and Lee Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, 1976.

[54] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[55] William H. Harrison, Charles Barton, and Mukund Raghavachari. Mapping uml designs to java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 178–187, New York, 2000. ACM.

[56] William H. Harrison, Harold L. Ossher, and Peri L. Tarr. Asymmetrically vs symmetrically organized paradigms for software composition. Technical report, IBM Watson Research Center, 2002. http://domino.watson.ibm.com/library/cyberdig.nsf/papers/2A4097E93456D0CF85256CA9006DAC29/$File/RC22685.pdf.

[57] Jan Hendrik Hausmann and Stuart Kent. Visualizing model mappings in uml. In *Proceedings of the ACM Symposium on Software Visualization*, pages 169–178, New York, 2003. ACM.

[58] Marcelo Victora Hecht, Eduardo Piveta, Marcelo Pimenta, and Roberto T. Price. Aspect-oriented code generation. In *Anais do XX Simpósio Brasileiro de Engenharia de Software*, pages 209–223, Porto Alegre, 2006. Sociedade Brasileira da Computação.

[59] Jack Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, 2003.

[60] Steve Hodges et al. Auto-id based control demonstration - phase 1: Pick and place packing with conventional control. Technical report, Cambridge University, 2003. http://www.ifm.eng.cam.ac.uk/automation/publications/documents/CAM-AUTOID-WH-006.pdf.

[61] IBM. Ibm rational rose technical developer, 2008. http://www.ibm.com/software/awd tools/developer/rose/.

[62] IBM. Ibm telelogic rhapsody, 2008. http://modeling.telelogic.com/products/rhapsody/software/developer/index.cfm.

[63] IBM. Ibm telelogic tau, 2008. http://www.telelogic.com/products/tau/tau/index.cfm.

[64] ISO/IEC. Systems and software engineering - recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, Jul. 2007.

[65] Sergio Akira Ito et al. Making java work for microcontroller applications. *IEEE Design and Test of Computers*, 18(5):100–110, 2001.

[66] ITRS. International technology roadmap for semiconductors 2007 edition: Design. Technical report, International Technology Roadmap for Semiconductors, 2007. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Design.pdf.

[67] Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming*, pages 220–242, Berlin, 1997. Springer-Verlag.

[68] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. Addison-Wesley, Boston, 2000.

[69] Petri Kukkala et al. Uml 2.0 profile for embedded system design. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 710–715, Washington, DC, USA, 2005. IEEE Computer Society.

[70] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[71] Leslie Lamport. The +cal algorithm language. Technical report, Microsoft Research, 2007. http://research.microsoft.com/users/lamport/pubs/pluscal.pdf.

[72] Phillip A. Laplante. *Real-Time Systems Design and Analysis : an Engineer's Handbook*. IEEE Press, New York, second edition, 1997.

[73] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley, Boston, 1999.

[74] Daniel Lohmann et al. Pure embedded operating systems - ciao. In *Proceedings of the International Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.

[75] Daniel Lohmann et al. Interrupt synchronization in the ciao operating system: Experiences from implementing low-level system policies by aop. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, New York, 2007. ACM.

[76] Quan Long et al. Consistent code generation from uml models. In *Proceedings of the Australian Software Engineering Conference*, pages 23–30, Los Alamitos, 2005. IEEE Computer Society.

[77] LSE. Sistemas eletrônicos embarcados baseados em plataformas, 2003. http://www.inf.ufrgs. br/lse/pag_projeto.php?cod_projeto=1.

[78] Grant Martin and Wolfgang MÜller, editors. *UML for SOC Design*. Springer-Verlag, Netherlands, 2005.

[79] Stephen J. Mellor et al. An action language for uml: Proposal for a precise execution semantics. In *Proceedings of the First International Workshop on The Unified Modeling Language. UML'98: Beyond the Notation*, pages 307–318, London, 1999. Springer-Verlag.

[80] MicroTool. objectif – the tool for model-driven development with uml, 2008. http://www.microtool.de/objectif/en/index.asp.

[81] Richard Nass. An insider's view of the 2008 embedded market study. *Embedded Systems Design*, 21(9), September 2008. http://www.embedded.com/design/testissue/210200580.

[82] Kathy Dang Nguyen, Zhenxin Sun, P. S. Thiagarajan, and Weng-Fai Wong. Model-driven soc design via executable uml to systemc. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 459–468, Washington, 2004. IEEE Computer Society.

[83] Elisabetta Di Nitto et al. Deriving executable process descriptions from uml. In *Proceedings of the 24th International Conference on Software Engineering*, pages 155–165, New York, 2002. ACM.

[84] Natsuko Noda and Tomoji Kishi. Aspect-oriented modeling for embedded software design. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 342–349, Washington, 2007. IEEE Computer Society.

[85] NoMagic. Introducing magicdraw, 2008. http://www.magicdraw.com/.

[86] OMG. Common warehouse metamodel (cwm), 2003. http://www.omg.org/spec/CWM/1.1/.

[87] OMG. Model-driven architecture, 2004. http://www.omg.org/mda.

[88] OMG. Uml profile for schedulability, performance, and time, version 1.1, 2005b. http://www.omg.org/technology/documents/formal/schedulability.htm.

[89] OMG. Meta object facility (mof) 2.0, 2006. http://www.omg.org/spec/MOF/2.0.

[90] OMG. Object constraint language (ocl) 2.0, 2006. http://www.omg.org/spec/OCL/2.0/.

[91] OMG. Xml metadata interchange (xmi) 2.1.1, 2007. http://www.omg.org/spec/XMI/2.1.1/.

[92] OMG. Unified modeling language (uml), version 2.2, 2008. http://www.omg.org/spec/UML/2.2/Beta1/Superstructure/PDF.

[93] OMG. Mof query/views/transformations, 2008a. http://www.omg.org/spec/QVT/1.0.

[94] OMG. Uml profile for modeling and analysis of real-time and embedded systems (marte), 2008b. http://www.omg.org/cgi-bin/doc?ptc/2008-06-08.

[95] OMG. Uml profile for for modeling quality of service and fault tolerance characteristics and mechanisms, v1.1, 2008c. http://www.omg.org/spec/QFTP/1.1/.

[96] Isabelle Perseil and Laurent Pautet. Foundations of a new software engineering method for real-time systems. *Innovations in Systems and Software Engineering*, 4(3):195–202, Oct. 2008.

[97] Mónica Pinto, Lidia Fuentes, and Jose María Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, pages 118–137, New York, 2003. Springer-Verlag.

[98] Mónica Pinto, Lidia Fuentes, and José María Troya. A dynamic component and aspect-oriented platform. *The Computer Journal*, 48(4):401–420, 2005.

[99] Raj Rajkumar. Model-based development of embedded systems: The sysweaver approach. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 35–46. Springer Netherlands, 2007.

[100] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A soc design methodology involving a uml 2.0 profile for systemc. In *Proceedings of theDesign, Automation and Test in Europe Conference and Exhibition*, pages 704–709, Washington, 2005. IEEE Computer Society.

[101] Linda H. Rosenberg. Applying and interpreting object oriented metrics. Technical report, NASA Software Assurance Technology Center, 2003. http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply.pdf.

[102] J. Rothenberg. The nature of modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors, *Artificial Intelligence, Simulation & Modeling*, pages 75–92. John Wiley & Sons, New York, 1989.

[103] SAE. Architecture analysis & design language, 2006. http://www.sae.org/technical/standards/AS5506.

[104] Pablo Sánchez et al. Aspect-oriented model weaving beyond model composition and model transformation. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, pages 766–781, Berlin, 2008. Springer-Verlag.

[105] Alberto Sangiovanni-Vincentelli. The tides of eda. *IEEE Design & Test of Computers*, 20(6):59–75, Nov. 2003.

[106] Claudio Sant'anna et al. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, number 17, pages 19–24, 2003.

[107] Tim Schattkowsky and Wolfgang Mueller. Model-based specification and execution of embedded real-time systems. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1392–1393, Los Alamitos, 2004. IEEE Computer Society.

[108] Tim Schattkowsky, Wolfgang Mueller, and Achim Rettberg. A model-based approach for executable specifications on reconfigurable hardware. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 692–697, Washington, 2005. IEEE Computer Society.

[109] Andrea Schauerhuber et al. Towards a common reference architecture for aspect-oriented modeling. In *Proceedings of the 8th International Workshop on Aspect-Oriented Modeling*, number 3rd, 2006. http://dawis2.icb.uni-due.de/events/AOM_AOSD2006/Schauerhuber.pdf.

[110] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, Feb. 2006.

[111] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21:294–324, 1998.

[112] Conrado Werner Seibel. *Uma metodologia Formal para o Planejamento e Controle de Missões de Aeronaves Não-Tripuladas*. PhD thesis, Departamento de Engenharia Elétrica, Universidade Federal de Santa Catarina, Florianopólis, 2001.

[113] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003a.

[114] Bran Selic and Leo Motus. Using models in real-time software design. *IEEE Control Systems Magazine*, 23(3):31–42, June 2003b.

[115] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, sixth edition, 2001.

[116] Olaf Spinczyk and Daniel Lohmann. The design and implementation of aspectc++. *Knowledge-Based Systems: Special Issue on Creative Software Design*, 20(7):636–651, 2007.

[117] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Willey, 2006.

[118] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[119] John A. Stankovic et al. Vest: An aspect-based composition tool for real-time systems. *Real-Time and Embedded Technology and Applications Symposium, IEEE*, 0:58, 2003.

[120] Dominik Stein et al. A uml-based aspect-oriented design notation for aspectj. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 106–112, New York, 2002. ACM Press.

[121] Dominik Stein et al. Expressing different conceptual models of join point selections in aspect-oriented design. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 15–26, New York, 2006. ACM.

[122] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Upper Saddle River, second edition, 2007.

[123] Aleksandra Tesanovic et al. Aspects and components in real-time system development: Towards reconfigurable and reusable software. *Journal of Embedded Computing*, 1(1/2005):17–37, Jan. 2005.

[124] Shiu Lun Tsang, Siobhán Clarke, and Elisa L. A. Baniassad. An evaluation of aspect-oriented programming for java-based real-time systems development. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, number 7, pages 291–300, Los Alamitos, 2004. IEEE Computer Society.

[125] UPB. Organic reconfigurable operating system, 2008. https://orcos.cs.uni-paderborn.de/orcos/.

[126] Klaas van den Berg, Jose Maria Conejero, and Ruzanna Chitchyan. Aosd ontology 1.0: Public ontology of aspect-orientation. Technical report, AOSD-Europe, 2005. AOSD-Europe-UT-01.

[127] Yves Vanderperren, Wolfgang Mueller, and Wim Dehaene. Uml for electronic systems design: a comprehensive overview. *Design Automation for Embedded Systems*, 12(4):261–292, 2008.

[128] Stamatis Vassiliadis et al. The hipeac roadmap on embedded systems. Technical report, European Network of Excellence on High-Performance Embedded Architecture and Compilation, 2005. http://www.hipeac.net/roadmap.

[129] W3C. extensible markup language (xml) 1.0 (fourth edition), 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[130] W3C. Xsl transformation (xslt) 2.0 - candidate recomendation, 2006. http://www.w3.org/TR/xslt20.

[131] Marco Aurélio Wehrmeister. Framework orientado a objetos para projeto de hardware e software embarcados para sistemas tempo-real. Master's thesis, Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

[132] Wayne Hendrix Wolf. *Computers as Components : Principles of Embedded Computing System Design*. Morgan Kaufmann, San Francisco, 2001.

[133] Lichen Zhang and Ruicheng Liu. Aspect-oriented real-time system modeling method based on uml. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 373–376, Washington, 2005. IEEE Computer Society.

List of Abbreviations

AAM   Aspect-oriented Architecture Model

ABS   Anti-lock Bracking System

AMoDE-RT Aspect-oriented Model-Driven Engineering for Real-Time systems

API   Application Programming Interface

AADL   Architecture Analysis & Design Language

AO   Aspect-Orientation

AOD   Aspect-Oriented Design

AODM   Aspect-Oriented Design Modeling

ASIP   Application Specific Instruction Processor

CWM   Common Warehouse Meta-model

DERAF   Distributed Embedded Real-time Aspects Framework

DERCS   Distributed Embedded Real-time Compact Specification

DERTS   Distributed Embedded and Real-Time System

DREAMS DistRibuted Extensible Application Management System

DSML   Domain-Specific Modeling Languages

GenERTiCA Generation of Embedded Real-Time Code based on Aspects

HDL   Hardware Description Language

HW   Hardware

IEEE   Institute of Electrical and Electronics Engineers

IP   Intellectual Property

JPDD   Join Point Designation Diagrams

MAC   Media Access Control

MARTE	Modeling and Analysis of Real-time and Embedded systems
MDA	Model-Driven Architecture
MDD	Model-Driven Design
MDE	Model-Driven Engineering
MOF	Meta-Objects Facilities
OMG	Object Management Group
OO	Object-Orientation
ORCOS	Organic Reconfigurable Operating System
PIM	Platform Independent Model
PSM	Platform Specific Model]
QoS	Quality of Service
QVT	MOF Query/View/Transformation
RTSJ	Real-Time Specification for Java
RTOS	Real-Time Operating System
SAE	Society of Automotive Engineers
SCL	Skeleton Customization Language
SEEP	*Sistema Eletrônicos Embarcados baseados em Plataformas*
SoC	System-on-Chip
SPT	UML profile for Schedulability, Performance and Time
SW	Software
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WCET	Worst Case Execution Time
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

List of Publications

2009

[1] Edison P. Freitas, Rodrigo S. Allgayer, Marco A. Wehrmeister, Carlos E. Pereira, and Tony Larsson. Supporting platform for heterogeneous sensor network operation based on unmanned vehicles systems and wireless sensor nodes. In *Proceedings of IEEE Intelligent Vehicles Symposium*, pages 786–791, Los Alamitos, 2009. IEEE Computer Society.

[2] Edison P. Freitas, Tales Heimfarth, Marco A. Wehrmeister, Flavio R. Wagner, Armando M. Ferreira, Carlos E. Pereira, and Tony Larsson. Using a link metric to improve communication mechanisms and real-time properties in an adaptive middleware for heterogeneous sensor networks. In *Advances in Information Security and Assurance*, pages 422–431. Springer, Berlin, 2009.

[3] Edison P. Freitas, Marco A. Wehrmeister, Armando M. Ferreira, Carlos E. Pereira, and Tony Larsson. Multi-agents supporting reflection in a middleware for mission-driven heterogeneous sensor networks. In *Proceedings of 3rd International Workshop on Agent Technology for Sensor Networks*, pages 25–32, 2009.

[4] Marcio F. S. Oliveira, Marco A. Wehrmeister, Francisco A. Nascimento, Carlos E. Pereira, and Flavio R. Wagner. *High-level Design Space Exploration of Embedded Systems Using the Model-Driven Engineering and Aspect-Oriented Design Approaches*, chapter 5, pages 114–146. Information Science Reference, Hershey, 2009.

[5] Marco A. Wehrmeister, Edison P. Freitas, and Carlos E. Pereira. An infrastructure for uml-based code generation tools. In *Proceedings of 3rd IFIP International Embedded Systems Symposium*. Springer, 2009. (to appear).

[6] Marco A. Wehrmeister, Edison P. Freitas, and Carlos E. Pereira. Using genertica to generation code from rt-uml : a case study. In *Proceedings of 13th IFAC Symposium on Information Control Problems in Manufacturing*, pages 678–683. Elsevier Science, 2009.

2008

[1] Edison P. Freitas, Marco A. Wehrmeister, Carlos E. Pereira, and Tony Larsson. Real-time support in adaptable middleware for heterogeneous sensor networks. In *Proceedings of International Workshop on Real Time Software (co-located with International Multiconference on Computer Science and Information Technology)*, pages 593–600, Los Vaqueros Circle, 2008. IEEE Computer Society Press.

[2] Edison P. Freitas, Marco A. Wehrmeister, Carlos E. Pereira, and Tony Larsson. Reflective middleware to support mission-driven heterogeneous sensor networks. In *Proceedings of Workshop on Sensor Networks and Applications (co-located with 21st Symposium on Integrated Circuits and Systems Design)*, pages 1–6, Porto Alegre, 2008. Universidade Federal do Rio Grande do Sul.

[3] Edison P. Freitas, Marco A. Wehrmeister, Carlos E. Pereira, and Tony Larsson. Using aspects and component concepts to improve reuse of software for embedded systems product lines. In *Proceedings of 12th International Software Product Line Conference*, pages 105–112, Limerick, 2008. University of Limerick.

[4] Carlos E. Pereira, Marcelo Götz, Marco A. Wehrmeister, Edison P. Freitas, and Elias T. Silva Junior. Real-time distributed embedded systems: Infra-structure for bio-inspired automation systems. In *Self-optimizing Mechatronic Systems: Design the Future*, pages 449–468. Heinz Nixdorf Institute, Paderborn, 2008.

[5] Marco A. Wehrmeister, Edison P. Freitas, Dalimir Orfanus, Franz Rammig, and Carlos E. Pereira. A case study to evaluate pros/cons of aspect- and object-oriented paradigms to model distributed embedded real-time systems. In *Proceedings of 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, pages 44–54, Los Alamitos, 2008. IEEE Computer Society Press.

[6] Marco A. Wehrmeister, Edison P. Freitas, Dalimir Orfanus, Franz Rammig, and Carlos E. Pereira. A comparison of the use of aspects and objects for modeling distributed embedded real-time systems with rt-uml. In *Anais do X Workshop de Sistemas de Tempo-Real e Embarcados (em conjunto com XXVI Simpósio Brasileiro de Redes de Computadores)*, pages 1–8, Porto Alegre, 2008. Sociedade Brasileira da Computação.

[7] Marco A. Wehrmeister, Edison P. Freitas, Dalimir Orfanus, Franz Rammig, and Carlos E. Pereira. Evaluating aspect and object-oriented concepts to model distributed embedded real-time systems using rt-uml. In *Proceedings of Trienal World Congress of the International Federation of Automatic Control*, pages 6885–6890. Elsevier Science, 2008.

[8] Marco A. Wehrmeister, Edison P. Freitas, Dalimir Orfanus, Franz Rammig, and Carlos E. Pereira. Genertica: A tool for code generation and aspects weaving. In *Proceedings of 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 44–54, Los Alamitos, 2008. IEEE Computer Society Press.

2007

[1] Edison P. Freitas, Marco A. Wehrmeister, Elias T. Silva Junior, Fabiano C. Carvalho, Flavio R. Wagner, and Carlos E. Pereira. Deraf: A high-level aspects framework for distributed embedded real-time systems design. In *Early Aspects: Current Challenges and Future Directions (Lecture Notes in Computer Science)*, pages 55–74. Springer, Berlin / Heidelberg, 2007.

[2] Edison P. Freitas, Marco A. Wehrmeister, Elias T. Silva Junior, Fabiano C. Carvalho, Flavio R. Wagner, and Carlos E. Pereira. Using aspect-oriented concepts in the requirements analysis of distributed real-time embedded systems. In *Embedded System Design: Topics, Techniques and Trends*, pages 221–230. Springer, Boston, 2007.

[3] Elias T. Silva Junior, Marco A. Wehrmeister, Flavio R. Wagner, and Carlos E. Pereira. An approach to improve predictability in communication services in distributed real-time embedded systems. In *Proceedings of 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 121–126, New York, 2007. ACM Press.

[4] Marco A. Wehrmeister, Edison P. Freitas, Franz Rammig, and Carlos E. Pereira. Combining aspects-oriented concepts with model-driven techniques in the design of distributed embedded real-time systems. In *Proceedings of 19th Euromicro Conference on Real-Time Systems, Work-In-Progress Session*, pages 49–59, Singapore, 2007. National University of Singapore.

[5] Marco A. Wehrmeister, Edison P. Freitas, Flavio R. Wagner, and Carlos E. Pereira. An aspect-oriented approach for dealing with non-functional requirements in a model-driven development of distributed embedded real-time systems. In *Proceedings of 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 49–52, Washington, 2007. IEEE Computer Society.

2006

[1] Edison P. Freitas, Marco A. Wehrmeister, Elias T. Silva Junior, Fabiano C. Carvalho, Flavio R. Wagner, and Carlos E. Pereira. Using aspects to model distributed real-time embedded systems. In *Anais do III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (em conjunto com XX Simposio Brasileiro de Engenharia de Software)*, pages 1–11. SBC, 2006.

[2] Francisco A. Nascimento, Marcio F. S. Oliveira, Marco A. Wehrmeister, Flavio R. Wagner, and Carlos E. Pereira. Mda-based approach for embedded software generation from a uml/mof repository. In *Proceedings of 19th Symposium On Integrated Circuits And Systems Design*, pages 143–148, New York, 2006. ACM Press.

[3] Marco A. Wehrmeister, Fernando H. Ataide, Fabiano C. Carvalho, and Carlos E. Pereira. A comparative study of embedded protocols for safety-critical control applications. In *Proceedings of 12th IFAC Symposium on Information Control Problems in Manufacturing*, pages 87–94. Elsevier Science, 2006.

[4] Marco A. Wehrmeister, Leandro B. Becker, and Carlos E. Pereira. Optimizing the generation of object-oriented real-time embedded applications based on the real-time specification for java. In *Proceedings of IEEE/ACM Design, Automation and Test in Europe*, pages 1–6, Los Alamitos, 2006. IEEE Computer.

2005

[1] Elias T. Silva Junior, Marco A. Wehrmeister, Leandro B. Becker, Carlos E. Pereira, and Flavio R. Wagner. Design exploration in hw/sw co-design of real-time object-oriented embedded systems: the scheduler object. In *Proceedings of 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 378–388, Washington, 2005. IEEE Computer Society.

[2] Elias T. Silva Junior, Marco A. Wehrmeister, Fabiano C. Carvalho, Leandro B. Becker, Carlos E. Pereira, and Flavio R. Wagner. Exploração do espaço de projeto em hw/sw co-design de sistemas tempo-real embarcados orientados a objetos: o objeto escalonador. In *Anais do VII Workshop de Tempo Real (em conjunto com XXIII Simpósio Brasileiro de Redes de Computadores)*, pages 09–16, 2005.

[3] Marco A. Wehrmeister, Fernando H. Ataide, Fabiano C. Carvalho, and Carlos E. Pereira. Assessing the use of rt-java in automotive time-triggered applications. In *From Specification to Embedded Systems Application*, pages 223–234. Springer-Verlag, New York, 2005.

[4] Marco A. Wehrmeister, Leandro B. Becker, and Carlos E. Pereira. Applying the seep method in the design of a real-time embedded control system for a motorized wheelchair. In *Proceedings of 10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 147–184, Los Alamitos, 2005. IEEE Computer Society.

[5] Marco A. Wehrmeister, Leandro B. Becker, and Carlos E. Pereira. An approach for designing real-time embedded systems from rt-uml specifications. In *Proceedings of 16th International Federation of Automatic Control World Congress*. Elsevier Science, 2005.

[6] Marco A. Wehrmeister, Leandro B. Becker, and Carlos E. Pereira. Metodologia de projeto orientada a objetos baseada em plataformas para sistemas tempo-real embarcados. In *Anais do VII Workshop de Tempo Real (em conjunto com XXIII Simpósio Brasileiro de Redes de Computadores)*, pages 01–08, 2005.

[7] Marco A. Wehrmeister, Leandro B. Becker, and Carlos E. Pereira. Object-oriented methodology to the development of embedded real-time systems. In *Proceedings of 3rd IEEE International Conference on Industrial Informatics*, pages 68–73, Los Alamitos, 2005. IEEE Computer Society.

[8] Marco A. Wehrmeister, Leandro B. Becker, Carlos E. Pereira, and Flavio R. Wagner. An object-oriented platform-based design process for embedded real-time systems. In *Proceedings of 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 125–128, Los Alamitos, 2005. IEEE Computer Society.