Alma Mater Studiorum – Università di Bologna
Università degli Studi di Padova

DOTTORATO DI RICERCA IN

Informatica

Ciclo XXIII

Settore scientifico-disciplinare di afferenza: INF/01

Definition, realization and evaluation of a software reference architecture for use in space applications

Presentata da: Marco Panunzio

Coordinatore Dottorato

Prof. Simone Martini

Relatore

Prof. Tullio Vardanega

Esame finale anno 2011

Abstract

A recent initiative of the European Space Agency (ESA) aims at the definition and adoption of a software reference architecture for use in on-board software of future space missions. The initiative responded to high-level strategic goals that ESA set in the last few years in dialogue with its industrial environment. This PhD project was launched in the intent of supporting the ESA initiative with a scientifically-based contribution.

In the initial phase of the PhD project we gathered all the industrial needs relevant to ESA and all the main European space stakeholders: national space agencies, system and software prime contractors and the main software suppliers. The results of this initial phase were submitted to and discussed in a top-level working group comprised of representatives of all those stakeholders. Exploiting knowledge from previous investigations funded by ESA and entertaining focused discussions with the system and software prime suppliers of ESA, we were able to consolidate a set of technical high-level requirements for the fulfillment of those industrial needs. The conclusion of that phase of the investigation is that the best solution for the fulfillment of the high-level requirements is the definition, realization and adoption of a software reference architecture.

The software reference architecture that we are constructing is built on four fundamental constituents: (i) a component model, to design the software as a composition of individually verifiable and reusable software units; (ii) a computational model, to relate the design entities of the component model, their execution needs and their extra-functional properties for concurrency, time and space, to a framework of analysis techniques which ensure that the architectural description is statically analyzable in the dimensions of interest; (iii) a programming model, which consists in a tailored subset of a programming language and a set of code archetypes, and is used to ensure that the implementation of the design entities conforms with the semantics, the assumptions and the constraints of the computational model; and (iv) a conforming execution platform, which is in charge of preserving at run time the system and software properties asserted by static analysis and it is able to notify and react to possible violations of them.

The nature, feasibility and fitness of constituents (ii), (iii) and (iv), in the context of a Model-Driven Engineering approach, were already investigated by the author during the ASSERT project (FP6 IST-004033 2004-8), prior to the launch the PhD project. The results of ASSERT,

including the feedback received by the industrial partners of the space domain participating to that project, provided considerable confidence on the overall goodness of the approach we are proposing.

In the core of the PhD project we focused on the design of a component model for use in on-board software, which completes the 4-constituent foundation to the proposed software reference architecture and also is the central contribution of this PhD thesis.

The design of the proposed component model was centered on: (i) rigorous separation of concerns, achieved during software design with the support for design views (following and specializing the standard ISO 42010/IEEE 1471) and by careful allocation of concerns to the three software entities of the approach: the component, the container and the connector; (ii) the support for specification and model-based analysis of extra-functional properties; (iii) the inclusion in the component model of space-specific concerns while consciously trying to keep them separate from domain-neutral concerns.

In this thesis we also discuss a concrete incarnation of the proposed component model, experimentally built around a domain-specific language, a graphical editor and associated design environment for schedulability analysis and code generation. Thanks to our involvement in the CHESS project (ARTEMIS JU grant nr. 216682, 2009-2012) we were also able to compare the results of this effort with an alternative implementation of the same component model based on the extension of a subset of the UML MARTE profile.

Disclaimer This work was supported by the Networking/Partnering Initiative of the Directorate of "Technical and Quality Management" at ESA/ESTEC and by the CHESS project ("Composition with Guarantees for High-integrity Embedded Software Components Assembly", ARTEMIS Joint Undertaking grant nr. 216682).

The views presented in this PhD thesis are however those of the author's only, and do not engage those of the European Space Agency or of the members of the CHESS consortium.

Acknowledgments

During the late summer 2007, almost at the tail end of the ASSERT project in which I was working, my supervisor proposed me to apply to the PhD program in Computer Science.

At the beginning I was not really convinced; and after some discussions, I had only one request: I wanted an industrially-relevant research topic, and opportunities for frequent contacts with industrial stakeholders to receive feedback from them.

A few months later, after passing the admission exam, I started my PhD.

In these years there have been a few periods of despair, others of doubt and reflection, others of enthusiastic breakthrough. In any case, all accompanied by tenacious commitment.

And as regards the industrial relevance of the various strands of my research, I was never disappointed.

Therefore, I want to gratefully thank my thesis supervisor, Prof. Tullio Vardanega; for his support and guidance throughout the PhD and my previous research experiences; for the decisive discussions when the road ahead was unclear; and for setting up all the conditions that made it possible to work on a very interesting topic, strongly tied to industrial needs.

I want to thank Jean-Loup Terraillon and Andreas Jung at ESA-ESTEC. The periods spent at the research center of the European Space Agency were opportunities for interesting discussions. My collaboration with the SAVOIR-FAIRE working group was crucial to submit the ideas of this thesis to the industrial stakeholders and receive feedback and recommendations.

I also want to thank Alain Rossignol and Luc Planche, for the frank discussions we had during my short visit at Astrium Toulouse.

I am sincerely grateful to Gérald Garcia and Xavier Olive, for the invaluable discussions we had during my 10-day visit at Thales Alenia Space - Cannes. The confirmations and the valuable suggestions I received during that visit were the turning point of this investigation.

I want to thank Prof. Fabio Panzieri and Prof. Michael González Harbour, for their guidance during the various stages of this PhD.

I am also very grateful to the external referees of this PhD thesis:
Prof. Michel R.V. Chaudron and Prof. Ivica Crnkovic. Their feedback and precise comments were helpful to improve the quality of this thesis and the presentation of the results.

E per finire, voglio ringraziare di cuore i miei genitori. Vi ringrazio per tutto il supporto che mi avete dato in questi anni, e per essere sempre disponibili ad aiutarmi in ogni modo.

Table of contents

List of Figures

List of Tables

List of acronyms

Table 1: List of acronyms.

Acronym	**Meaning**
AADL	Architecture Analysis & Design Language
AIT	Avionics Integration Test
APEX	Application/Executive
AOCS	Attitude and Orbit Control System
ASSERT	Automated proof-based System and Software Engineering for Real-Time systems
ATAM	Architecture Tradeoff Analysis Method
ATV	Automated Transfer Vehicle
AUTOSAR	Automotive Open System Architecture
BC	Bus Controller
BSP	Board Support Package
BSW	Basic Software
C-by-C	Correctness by Construction
CBSE	Component-Based Software Engineering
CCM	CORBA Component Model
CHESS	Composition with Guarantees for High-integrity Embedded Software Components Assembly
CNES	Centre National d'Études Spatiales
CORBA	Common Object Request Broker Architecture
DFACS	Drag Free Attitude Control System
DHS	Data Handling System
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DSL	Domain-specific language
DSM	Domain-specific metamodel

Table 1 – continued from previous page

Acronym	Meaning
ECU	Electronic Control Unit
EGSE	Electrical Ground Support Equipment
ESA	European Space Agency
FDIR	Fault Detection, Isolation and Recovery
GMV	Grupo Mecánica del Vuelo
GNC	Guidance, Navigation and Control
GOCE	Gravity Field and Steady-State Ocean Circulation Explorer
GPS	Global Positioning System
HW	Hardware
ICD	Interface Control Document
IDL	Interface Definition Language
IMA	Integrated Modular Avionics
INTEGRAL	International Gamma-Ray Astrophysics Laboratory
ISS	International Space Station
LISA	Laser Interferometer Space Antenna
LwCCM	Lightweight CCM
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDE	Model-Driven Engineering
MIAT	Minimum inter-arrival time
MOF	Meta Object Facility
NASA	National Aeronautics and Space Administration
NPI	Networking/Partnering Initiative
OBC	On-board computer
OBCS	Object Control Structure
OBOSS	On-board Operations Support Software
OBSW	On-board software
OCL	Object Constraint Language
OMG	Object Management Group
OPCS	Operational Control Structure
PI	Provided Interface
PIM	Platform Independent Model
PSM	Platform Specific Model
PUS	Packet Utilization Standard
RI	Required Interface
RT	Remote Terminal

Table 1 – continued from previous page

Acronym	**Meaning**
RTOS	Real-Time Operating System
RTK	Real-Time Kernel
R/F	Radio Frequency
RUP	Rational Unified Process
SAM	Schedulability Analysis Model
SRR	System Requirement Review
SW-PDR	Software Preliminary Design Review
SW-SRR	System Software Requirement Review
SWRR	Software Requirement Review
STR	Star Tracker
SW	Software
TC	Telecommand
TM	Telemetry
TSP	Time and Space Partitioning
TTRM	Telemetry, Telecommand and Reconfiguration Module
UART	Universal Asynchronous Receiver-Transmitter
UML	Unified Modeling Language
V&V	Verification and Validation
XML	eXtensible Markup Language
WCET	Worst-case Execution Time

Chapter 1

Problem statement

European space industry has entered an economic era in which the funding availed to future space missions are not expected to grow significantly. At the same time, future missions are requested to achieve more and more challenging scientific goals. The steady growth in complexity that accompanies the definition of those missions is at odds with a season of capped budgets for Earth observation, science studies and space exploration.

One of the major consequences of this situation is that the activities that contribute the most to the value added of the mission, i.e. mission analysis and system engineering, will play an even bigger role in the overall economy of the project, with a proportional increase of the time and cost invested on them. The implication of this trend is that realization activities, and software development among them, are pushed forward in the project schedule and compressed (cf. figure 1.1 and 1.2).

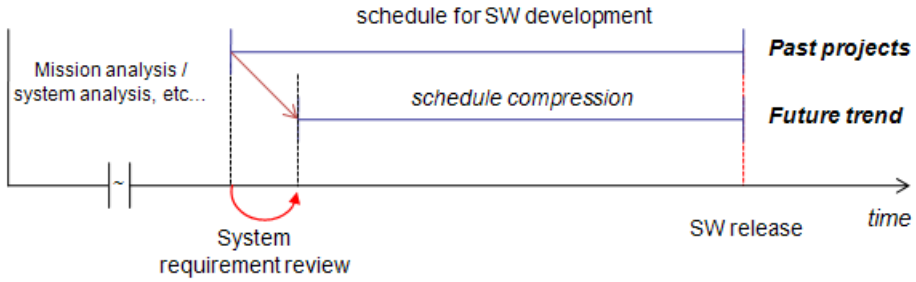

Figure 1.1: The software schedule for past and future projects. The increased effort for earlier design stages pushes forward the start of software development, hence compressing its schedule.

Interestingly, the complexity of the software product is foreseen to increase significantly to keep pace with the rising mission needs, while the cost of software development is expected to fit in the same or perhaps even decreased budget envelope.

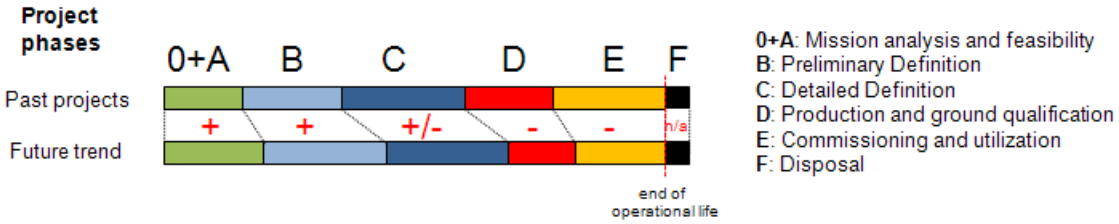

Figure 1.2: The effort proportionally allocated to the various project phases in past and future projects. The figure shows the increased effort for early phases and a reduction of the allocation to ”realization activities”.

This situation is therefore calling for a rise in the *cost-effectiveness* of software development, thus ultimately increasing the ”value” of the software product delivered with a given budget. The best way to succeed in this challenge is to strengthen the efficiency of software development and singling out and abating the recurring costs.

It is therefore necessary to first understand where this is possible, i.e. identify on which factors we can intervene and with how much degree of freedom; and subsequently, to investigate the definition of a solution.

In a typical satellite we can identify two main constituent parts (see figure 1.3): the payload and the service module (also known as platform). The payload part comprises the instruments necessary for the scientific mission (telescopes, spectrometers, detectors, etc..). The service module instead is used to govern the satellite position, orientation and manoeuvres (e.g. perform attitude or orbit changes), communicate with ground, and ensure that the thermal and power needs are satisfied; for this reason the service module is equipped with various sensors (gyroscopes, Earth sensors, magnetometers, thermisters, etc...), actuators (reaction wheels, thrusters, heaters, etc..) or other equipment (solar arrays).

The payload of the satellite obviously is mission-specific: hence, it will differ in form and instruments from one mission to another. Conversely, the service module may present a recurrent structure (i.e. shape and decomposition in subsystems) in several missions.

For example, the INTEGRAL spacecraft used the same service module as the XMM-Newton space telescope, and the Planck and Herschel spacecrafts shared a service module that was very similar both at structural and avionics level.

The on-board software of a satellite conceptually mirrors the same (physical) separation of the satellite structures. Accordingly, we can typically identify a separation of the software in two parts: the payload software and the platform software. Also in this case, the payload software is vastly if not completely different from one mission to the other; there are situations instead in which the platform software may assume a recurrent organization. In software engineering parlance, the ”organization” of the software is the *software architecture*.

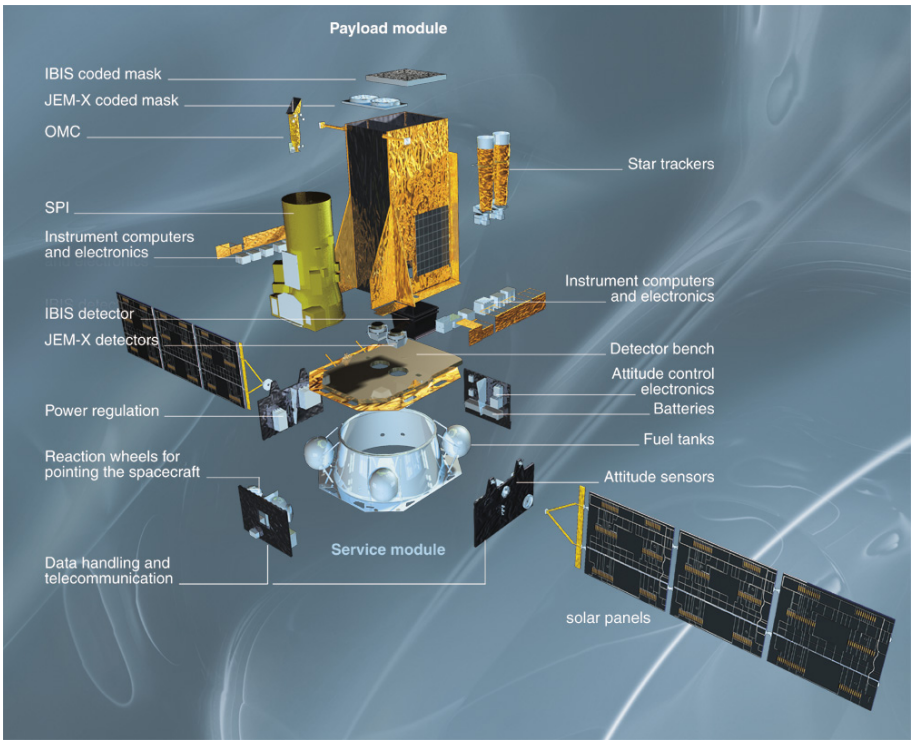

Figure 1.3: A diagram of the INTEGRAL spacecraft, which outlines the subsystems and instruments of the payload and the service module. *Credits: ESA*

The platform software comprises a set of traditional *functional contents*: the Attitude and Orbit Control System (AOCS) or one of its possible adaptations for particular missions, like the Drag-Free Attitude Control System (DFACS) of the GOCE satellite [32], or a Guidance, Navigation and Control (GNC) system [83] of a deep-space mission; the Data Handling System (DHS); the thermal management subsystem; the power management subsystem; etc.. Additionally it may require the inclusion of more advanced subsystems that realize the functional requirements of future missions, like advanced autonomy and planning or formation flying.

On-board software for satellites can be classified as high-integrity real-time software; the realization of the *functional contents* is therefore subject to stringent requirements at both process and product level in dimensions such as: time and space predictability, safety, dependability, security. As a consequence, the output of the various stages of the development process are strictly regulated by domain-specific standards (like ECSS-E-ST-40C [46] on software development and ECSS-Q-ST-80C [48] on software product assurance). Moreover, the software product is subject to extensive verification and validation campaigns to ascertain its quality, that is to say to verify it performs as expected while fulfilling all *extra-functional* requirements.

It is important to appreciate that the value added of the software products resides in its functional contents (what the software does). All the remaining software parts and all the development activities instead contributes to the *quality* of the final product.

Given this clarification, a reduction in the overall cost for software under a comparable complexity for the functional contents can be achieved only by producing a final product with the same level of quality with a reduced effort. Alternatively, the reduced effort necessary to achieve the same level of quality allows to keep constant the overall budget of the satellite while developing more complex functional contents.

In this context the software architecture then plays a truly crucial role, as it expresses the architectural framework that hosts the functional contents, and its underlying architectural assumptions and methodological principles are among the major contributors to the attainment of the quality of the software product. In essence, we regard the software architecture as the instrument that contributes to the realization of our goal.

The space domain, in the specific European context, is heavily influenced by the current structure of the space market that is dominated by two prime contractors: Astrium Satellites (now a business unit of EADS Astrium) and Thales Alenia Space (owned by Thales Group and Finmeccanica). The two competitors (for economic and historical reasons) adopted different concepts, methodologies and technologies with respect to on-board software development and this led to distinct ways of producing the final software product. Therefore, the reconciliation of those two worlds in terms of overall approach to the problem, methodology, architectural solutions and interoperability of products is quite hard. To confirm the extent of this diversification, at the present state of the art, a software supplier can competitively provide its services as subcontractor only in a single supply chain.

As a response to these overall situation, the European Space Agency (ESA) launched an initiative that aims at the definition and realization of a *software reference architecture*, to be used in as many ESA missions as possible.

The rationale behind this decision is to foster the convergence to a single, common and agreed architecture that would be able to ensure the needed quality to the software product and it is adopted in several ESA missions. The net outcome of such adoption would be the possibility to focus on the definition of the functional contents rather than recurrently investing unnecessary engineering effort on the quality side of the problem; furthermore it would enable prime contractors to compete on functionality and performance without recurrently investing budget on quality (which is attained by adhering to the common solution), and software suppliers to compete on both supply chains.

The novel approach shall be able to address two development models that are inherent to the domain. The first model is *incremental development*, as the on-board software is not simply a product to be released at the end of the development, but incremental releases of it are necessary during the development schedule to perform, in coordination with other satellite development

teams, additive hardware/software integration and validation activities. The second model is *iterative development*, which is determined by the fluctuation and late finalization of system and software requirements experienced by almost every project; iterations therefore risk to be destructive backtracks of design and implementation decisions.

In figure 1.4 we depict an example of how the two development models coexist by extending the classical V-model development.

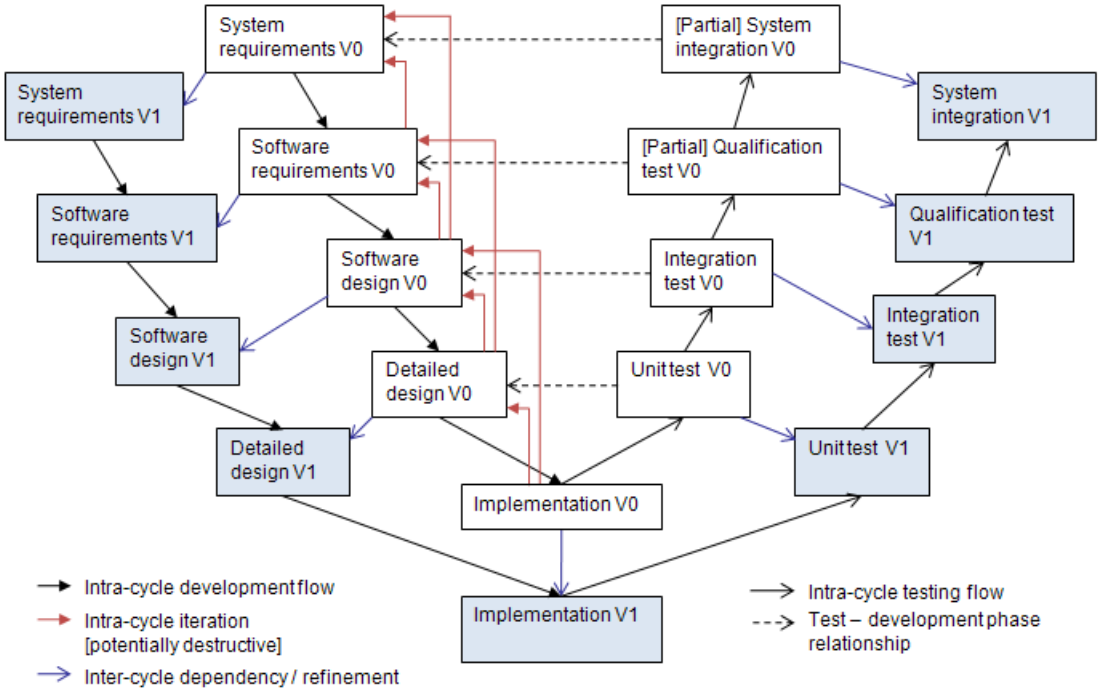

Figure 1.4: A classic V-model augmented to show iterative cycles and incremental development. For the sake of clarity we omitted the iterations inside the V1 iteration and the relationships between the right and left branch of the V1 iteration.

A first development iteration (termed ”V0”) comprises a set of sequential activities depicted on the left branch of the figure: from the creation of (a subset of the) system requirements to software requirements and so on, up to the implementation. Each activity is mirrored by the corresponding V&V activity in the right branch. During this sequence of activities, the development actors may be forced to backtrack to the previous activity to reconsider some design choices. Severe design problems may force to amend the output of even earlier phases (for example software requirements may be found infeasible during detailed design or implementation), with the obvious consequences in time and cost. The subsequent development iteration (”V1”) incrementally adds contents to the system and hence all its various activities are depen-

dent from the previous iteration. A novel approach shall demonstrate its capability of taming the destructive potential of iterative development and use instead iterations as refining or confirmative steps of the development in the context of an incremental development.

The investigation of this novel approach to the development of on-board software focuses primarily on the application to the platform software for Earth observation, science and deep-space missions, as those are the mission types for which domain experts deem it is possible to produce more easily a common recurrent solution and thus factorize the investment.

Therefore, the reference architecture initiative is currently not intended to address the software for launcher vehicles, re-entry vehicles, and rovers. Additionally, it will not address specifically software for payloads, even though the underlying principles and methodology might be applicable even in that context.

Let us now elaborate on the concept of *software architecture*, in particular on the various aspects it addresses. Subsequently, we focus on the advantages and implications of elevating a software architecture to the role of *reference architecture*.

1.1 The role of the software architecture

One disconcerting situation in the software engineering practice is the informal and liberal interpretation of the concept of *software architecture*. This factual truth is patently at odds with the centrality of that concept for the discipline.

Software architecture is probably the most important concept of software engineering, yet if we were to ask a definition and explanation of it to three different software developers or software engineers, almost for sure the answers would all be different and incomplete.

The main misunderstanding is that software architecture is often liberally used as a synonym of *software design*. Software design instead is (partly) just one of the concerns addressed by the software architecture, whose important role is to govern the software and the process by which it will be built and operated. More importantly, the software architecture deals with the *principles* guiding the design and evolution of a software system. This aspect is fundamental and for sure *precedes in importance* software design, and for this reason shall not be neglected.

To confirm that the notion of software architecture is far from having a universally accepted meaning, the reader can appreciate the multitude of definitions that were given in the course of time and were gathered by a brilliant exercise of the Software Engineering Institute [128]; the ”bibliographic definitions” therein can give an idea of the situation in the academic world; the ”community definitions” instead are a clear indication of the plethora of interpretations spread among software practitioners. From a quick scrutiny, the reader can understand that even if we discount the poor elegance or the lack of conciseness of some of the definitions, they are not all equivalent, and some of them clearly address only a subset of the concerns of a software

architecture.

For our investigation, we elected as a reference the definition of architecture that was given in the IEEE 1471 standard [69], later adopted and approved by ISO/IEC as ISO 42010 [76], as we consider it is the best and most authoritative:

"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution"

Starting from this definition we then reason on what are the concerns addressed by the software architecture:

- *Software decomposition*: the organization of the software in terms of parts, so that each single part has an own architectural cohesiveness (it addresses a single well-defined part of the problem), and interactions between parts are minimized as to reduce unnecessary dependencies (i.e. coupling), hence incidental complexity of understanding, V&V, operation and maintenance.

- *Externally visible attributes of software "components"*: the attributes of those software parts (which we call, for the moment only informally, components) that are externally visible. Those attributes represent the features or needs that are specific to the component yet can influence other parts of the software or its overall properties. The other attributes shall remain as (invisible) internal details and will not be used for the overall reasoning at the level of the software architecture;

- *Relationship between software "components"*: how components relate one another to provide services and fulfill needs;

- *Extra-functional concerns*: the software architecture is the level of reasoning where most of the extra-functional concerns are addressed;

- *External interfaces*: how the software interacts with the external environment (for example, by commanding sensors or actuators, and acknowledging external interrupts);

- *Principles for the development and evolution of the software*: a process that addresses the design of software, establishes the relevant rules for its development and provides means to perform software maintenance and evolution and dictates the supported form of reuse.

- *The rules in place to warrant the consistent relationship between all the aspects above*: an overall methodology that is able to consistently harmonize all these aspects; additionally, the methdology shall provide criteria to sanction if a software part can be included in the system by abiding by the principles underlying the architecture or shall be rejected as its inclusion would break the overall consistency of the software.

The latter aspect is very important. Every software has (at least implicitly) some software architecture [69, 117]. However, if we want to establish a software architecture that is supporting our goals, we have to define the methodology that underpins the development approach and warrants the consistency for all the aspects we mentioned above.

Reporting and refining an analogy used by Bran Selic during the discussion held at the FESA workshop[1] in April 2010, the software architecture is to the software product as a *constitution* is to a sovereign state: all the laws of the state must be compatible with the constitution, that is, they shall not violate or contradict implicitly or explicitly the fundamental principles established by the constitution and elected as the pillars of the state. Laws created by the legislative power that are incompatible with those principles are discarded after an assessment (typically performed by the supreme body of the judiciary power). This analogy is quite fascinating and, limiting exclusively to the mentioned aspect, a good representation of our context (even though extending the analogy to other aspects unfortunately makes it misleading).

The central role of the software architecture makes it fundamental to contribute to the value added of the final software product.

In fact, a poor software architecture hinders the fulfillment of functional, behavioural, extra-functional and life-cycle requirements. Conversely, a good software architecture warrants the quality of the final product and contributes to its value added.

In order to complete the picture, reasoning by exclusion we can enumerate aspects that are not of pertinence of a software architecture, and that we can score out from the investigation.

- *Detailed design*, which is the refinement of the software decomposition performed at architectural level by providing the internal software organization of ”components”;
- *Algorithm design*, which is the design of the algorithmic behaviour of the software to fulfill the functional requirements;
- *Software implementation*, which is the activity to realize the implementation of the whole software.
- *Hardware architecture*, which is obviously not part of the software architecture. However we have to notice that we need to complement the software architecture with a description of the hardware, limited to the aspects that impact software and its extra-functional properties in particular: therefore we need to specify the hardware aspects relevant for communication, analysis and code generation purposes. There is no interest in complete modeling of the hardware.

Figure 1.5 recapitulates these concepts.

[1] ”Formalisms for Description And Visualization of Embedded Systems Architectures - Current State Of Practice, Needs And Research Topics” http://www.kth.se/polopoly_fs/1.58948!EventSummary_extended.pdf

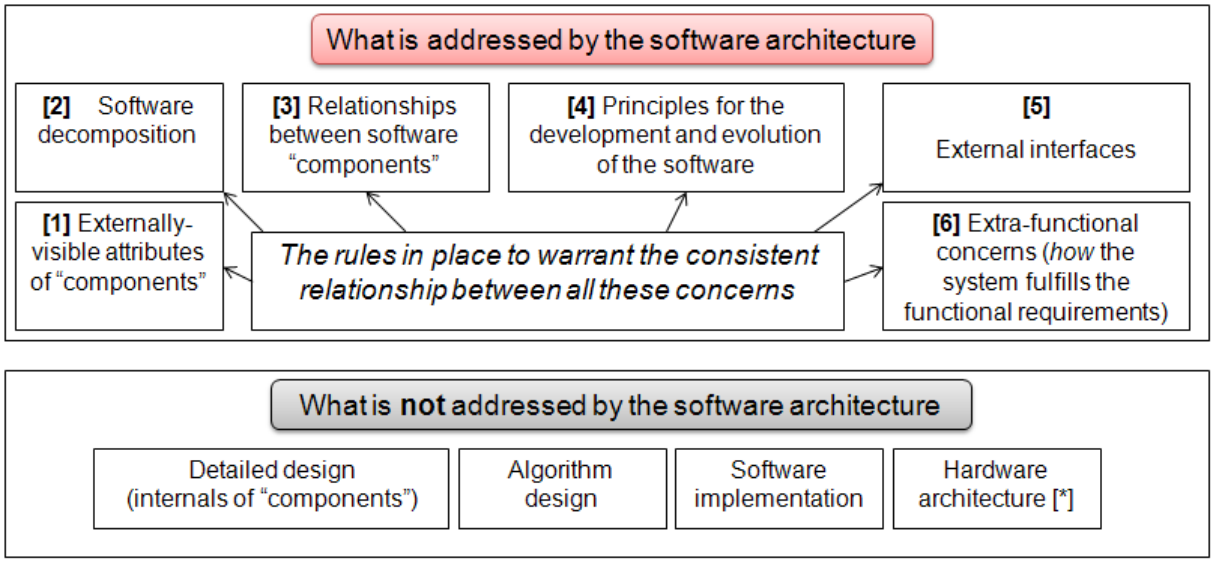

Figure 1.5: Recapitulation of the concerns addressed by the software architecture and those outside its perimeter.

1.2 On the concept of reference architecture

In the previous section we highlighted the confusion around the term ”software architecture”. It is not surprising that also the concept of *reference architecture* does not have a commonly accepted definition.

The most referred definition of reference architecture is that of the ”Rational Unified Process” (RUP) [80]: *”a predefined architectural pattern, or set of patterns, possibly partially or completely instantiated, designed, and proven for use in particular business and technical contexts, together with supporting artifacts to enable their use. Often, these artifacts are harvested from previous projects.”*

The definition is somewhat generic, yet it includes interesting aspects. In the following we try to elaborate and refine it.

It is important to recognize the difference between a software architecture and a software reference architecture. The software reference architecture prescribes the form of the concrete software architectures for a set of systems for which it was developed. We could then say that the reference architecture is a form of ”generic” software architecture, which prescribes the founding principles, the underlying methodology and the architectural practices that were recognized by the domain stakeholders as the best solution to the construction of a certain class of software systems. The architecture of one target software system can be considered as a sort

of ”instantiation” to the specific system needs of the software reference architecture.

The other interesting part of the RUP definition is that a reference architecture is ”proven for use in particular business and technical contexts”. This underlines:

1. the importance of the elicitation of the industrial needs and technical requirements that define the reference architecture;

2. the need of a *validation in use* and of an *evaluation* of the reference architecture. The validation demonstrates empirically that the software architecture(s) resulting from the application of the software reference architecture to concrete systems covers and fulfills satisfactorily all the industrial needs that originated it. The evaluation ascertains the goodness of the reference architecture by analyzing to what extent it satisfies the industrial needs. The evaluation can be facilitated by using appropriate methodological support. The most known evaluation method for software architectures is the Architecture Trade-off Analysis Method (ATAM) [54, 11]; it would require some adaptation to apply it to the level of reference architecture.

There are very different kinds of reference architectures, and an interesting work by Angelov et al. [6] can help us understand our stance. In that article, the authors define a set of dimensions that help the definition of a classification for reference architectures. They single out a set of dimensions and sub-dimension that characterize the definition and application of a reference architecture.

Their dimensions are: *context*, *goal* and *design*. The context dimension discriminates: *where* the reference architecture will be used (i.e. within a single organization or multiple organizations), *who* defines it (i.e. user organizations, software organizations, research centers, standardization bodies), *when* is it defined (*preliminary* reference architectures are defined before an implementation of it exists, or *classical* when experience on the application on the target class of systems has already been acquired).

The goal dimension describes the main usage of the reference architecture: it may be a *standardization* of existing software architectures, or a *facilitation* for the design of concrete architectures (by providing guidelines, methodologies and patterns of the software architecture).

The design dimension describes the main design choices of the reference architecture: *what* is described (components, interfaces, protocols, guidelines, etc...), the *abstraction level* of the description (concrete, semi-concrete, or abstract; corresponding according to the authors to decreasing levels of dependence with technological and implementation decisions) and the level of formality of the description (*informal, semi-formal, formal*).

We now anticipate how the software reference architecture of our investigation would be classified according to this criteria. For the context dimension: (i) it is defined for use *by* and *across* multiple organizations (a single architecture instantiation for a satellite project may be

shared by a software prime and all its software suppliers); (ii) it is defined by a working group comprised of staff members of the major stakeholders, supported by members of the academic world; (iii) it is a *preliminary* architecture, even though part of the stakeholders (the prime contractors) already have experience and their solutions for the *present* target systems.

For the goal dimension: (i) at the present time the reference architecture is at the *facilitation* stage. If the investigation demonstrates the feasibility and goodness of fit of the reference architecture, the stakeholders may be favourable to its standardization.

For the design dimension, the reference architecture: (i) describes the rules for the specification of components, interfaces, interactions, extra-functional attributes and how they relate to the analysis theories of interest, a design flow for the creation of the design entities, and the complete methodological background for the consistent relationship of all those aspects; (ii) can be considered as *semi-concrete* as it will abstract as much as possible from technological aspects and implementation solutions (that can be chosen or adapted by the users, provided that there is complete adherence with the methodological background), yet the space of the solutions from the technological point of view is constrained in some aspects (for example, the choice of the execution platform); (iii) it will adopt a semi-formal description language so that the resulting architecture description is predictive (i.e. it is possible to reason on and analyze its extra-functional concerns).

1.3 Contribution and structure of this thesis

The PhD project of the author was launched in January 2008 with the intent of supporting the ESA initiative with a scientifically-based contribution toward the definition, realization and evaluation of the software reference architecture. Since October 2008, the author started to collaborate with ESA in the scope of the Networking/Partnering Initiative (NPI) of the Directorate of ”Technical and Quality Management” at ESTEC, the research center of ESA at Noordwijk, The Netherlands. The author was able to work directly at ESTEC, as a visiting student in the ”Software Systems Engineering” section.

The collaboration with ESA allowed the collaboration of the author with the SAVOIR-FAIRE working group, comprised of staff members of ESA, the French and German space agencies (CNES and DLR), software prime contractors (EADS Astrium, Thales Alenia Space, Swedish Space Corporation), the main software suppliers (Scisys, GMV, Terma, Space Systems Finland), and tool vendors (Intecs).

The relationship with SAVOIR-FAIRE was fundamental to better understand the industrial needs and requirements of the European stakeholders, submit proposals on the various topics of the investigation to their scrutiny, and receive valuable feedback to drive the investigation.

The structure of this thesis is as follows.

In chapter 2, we elaborate on the various industrial needs that motivate the adoption of a single common software architecture. We show that some of those needs are common to all software domains; a few others are in common or similar to the needs of other high-integrity real-time domains; finally, a sizeable subset of them are specific to the European space domain. We motivate why the software reference architecture is an adequate solution capable of fulfilling all such needs.

The elicitation and consolidation of these needs was initially performed thanks to focused discussions with ESA staff members. The task was then facilitated and its results greatly improved thanks to short visits performed at the premises of EADS Astrium (Toulouse) and Thales Alenia Space (Cannes), which allowed the author to have interesting and valuable discussions with managers and engineering teams of the prime contractors.

Those discussions led to the derivation and formalization of a set of high-level requirements that are to be considered as the technical objectives to be achieved by the software reference architecture. These high-level requirements were subject to thorough discussion in the SAVOIR-FAIRE working group, which provided feedback and recommendations for their finalization.

The approach which we propose to base the software reference architecture on aims at two overarching goals: *composability* and *compositionality*. According to Sifakis [127]:

- *composability rules* "allow inferring that a component's properties are not affected when its structure is modified. That is, if components $gl\{U1, .., Un\}$ and $gl'\{U1, .., Un\}$ respectively satisfy the properties P and P', then the component $gl \oplus gl'\{U1, .., Un\}$ satisfies $P \wedge P'$. Composability means stability of component properties across integration.";

- *compositionality rules* "allow inferring global system properties from component properties. That is, if for $i = 1, .., n$ a set of components U_i satisfy properties P_i , then it is possible to guarantee that the component $gl(U_1, .., U_n)$ satisfies some property $gl'(U_1, .., U_n)$ where gl' is an operator on properties depending on gl".

Those definitions are in wide use in the real-time and formal methods communities. In contrast, the software engineering and component-based software engineering communities use the same two terms with other meanings; curiously, what the former terms "compositionality", it is referred as "composability" in the latter (see for example the work by Crnkovic et al. [39]).

For our investigation, we adopt the definitions by Sifakis, narrowed down as follows:

- *composability*, that is achieved when the properties (needs and obligations) of individual components are preserved on component composition and deployment on the target system;

- *compositionality*, that is achieved when the properties of the system as a whole can be derived (economically and conclusively) as a function of the the properties of its constituting components.

As regards composability, we stress the nature of properties (needs and obligations) and explicitly require that they are preserved both at component composition (at design time) and on deployment on the target. For compositionality, we note that we are interested in composability functions that provide conclusive results in any circumstance on the properties of interest.

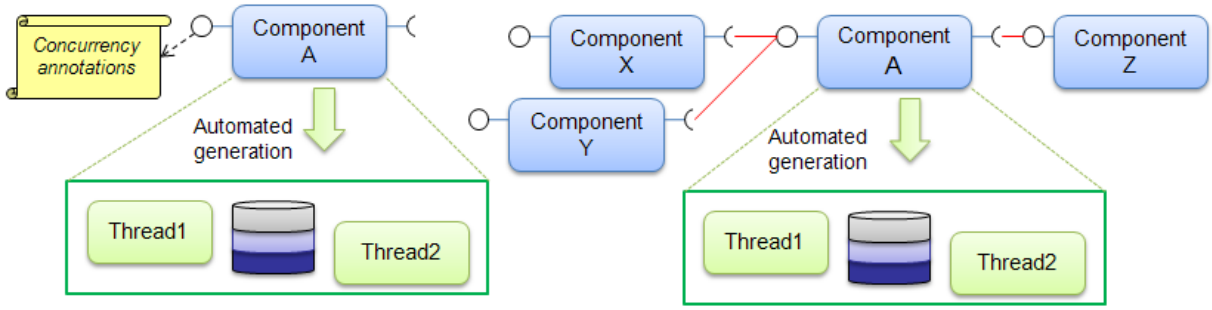

Figure 1.6: An example of composable property: the number of threads and protected objects generated in response to the extra-functional annotations to the component interface.

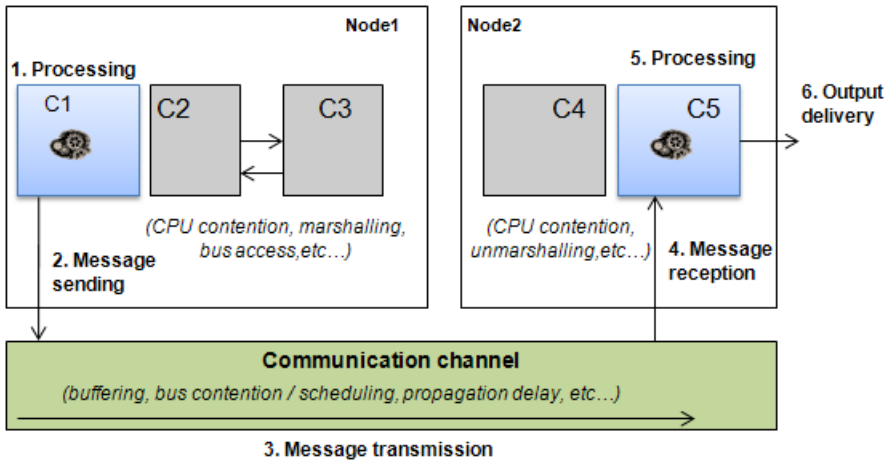

Figure 1.7: An example of compositional property: end-to-end latency.

An example of composable property in our approach is the number of threads and protected objects generated per component, which depends exclusively on the extra-functional annotations to the component interface (see fig. 1.6). The number is invariant across component composition (which and how many other components are later connected to the component of interest).

An example of compositional property is the end-to-end latency in the delivery of an output at the end of a chain of activities (see fig. 1.7). The latency (i.e., the worst-case response time

of the end-to-end chain of activities) is compositional if it is possible to calculate it exclusively from the properties explicitly exposed by components (and those describing the communication channel).

Composability and compositionality shall be consistently achieved by a development methodology that exhibits the crucial property of *composition with guarantees* [138], which may be regarded as a form of composability and compositionality that can be assured by static analysis, guaranteed throughout implementation, and actively preserved at run time.

The goal of composition with guarantees can be achieved with an approach centered on four constituents [108]:

1. a *component model*, to design the software as a composition of individually verifiable and reusable software units;

2. a *computational model*, to relate the design entities of the component model, their execution needs and their extra-functional properties for concurrency, time and space, to a framework of analysis techniques which ensures that the architectural description is statically analyzable in the dimensions of interest;

3. a *programming model*, which consists in a tailored subset of a programming language and a set of code archetypes, and is used to ensure that the implementation of the design entities conforms with the semantics, the assumptions and the constraints of the computational model;

4. a *conforming execution platform*, which is in charge of preserving at run time the system and software properties asserted by static analysis and it is able to notify and react to possible violations of them.

Compositionality is earned at the level of the *computational model* and *component model*, as it depends on the adopted body of analysis theories as well as how the architectural entities in use relate to them. Composability is earned at the level of the *component model*. Composability and compositionality, augmented with *property preservation* [137] (i.e. preservation of the analyzed properties at run time), lead to the achievement of composition with guarantees.

In order to increase the industrial applicability of the approach, and as a response to industrial needs of the space domain, we further include in the formulation of the software reference architecture: (a) a development process centered on Model-Driven Engineering [120]; (b) the provisions for domain-specific aspects, which complement the approach, yet are consistent to all its underlying principles.

Figure 1.8 recapitulates the goals of our approach and the constituents that we need for their achievements. We also depict in square brackets the concerns of the software architecture addressed by each constituent (cf. fig. 1.5).

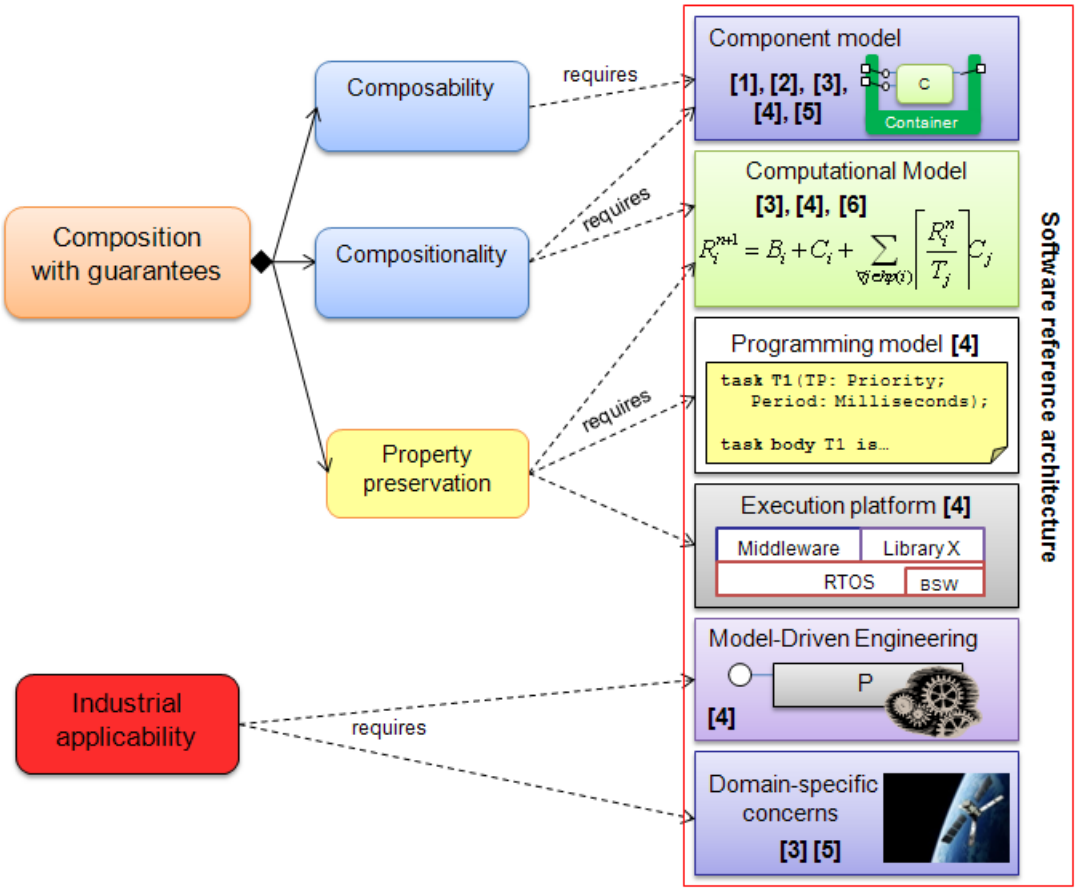

Figure 1.8: The goals we set for our investigation and the constituents of the software reference architecture that permit their achievement. We depict in square brackets the concerns of the software architecture dealt by each constituent (cf. fig. 1.5

We conclude chapter 2 by examining the state of the art of similar approaches, to elaborate on the concepts that inspired our work and on the improvements that we introduce with respect to existing approaches.

The central contribution of the thesis is the definition of a software reference architecture which provides a comprehensive approach to fulfill the industrial needs we gathered in the first part of our investigation.

Not all the constituents of the software reference architecture were investigated for the first time in this thesis. The nature, feasibility and fitness of constituents 2, 3 and 4, in the context of a

Model-Driven Engineering approach [120], were already investigated also by the author during the ASSERT project (FP6 IST-004033 2004-8), prior to the launch of the PhD project. The results of ASSERT [7, 24, 106, 21], including the feedback received by the industrial partners of the space domain participating to that project, provided considerable confidence on the overall goodness of the approach we are proposing, and encouraged us to base the foundation of the software reference architecture on the same principles.

In order to complete the 4-constituent foundation to the proposed software reference architecture, in the core of the PhD project we focused on the design of a component model for use in on-board software [109]. The inclusion of the component model in the approach permits to achieve the goal of composability and compositionality in the form that we termed *composition with guarantees*.

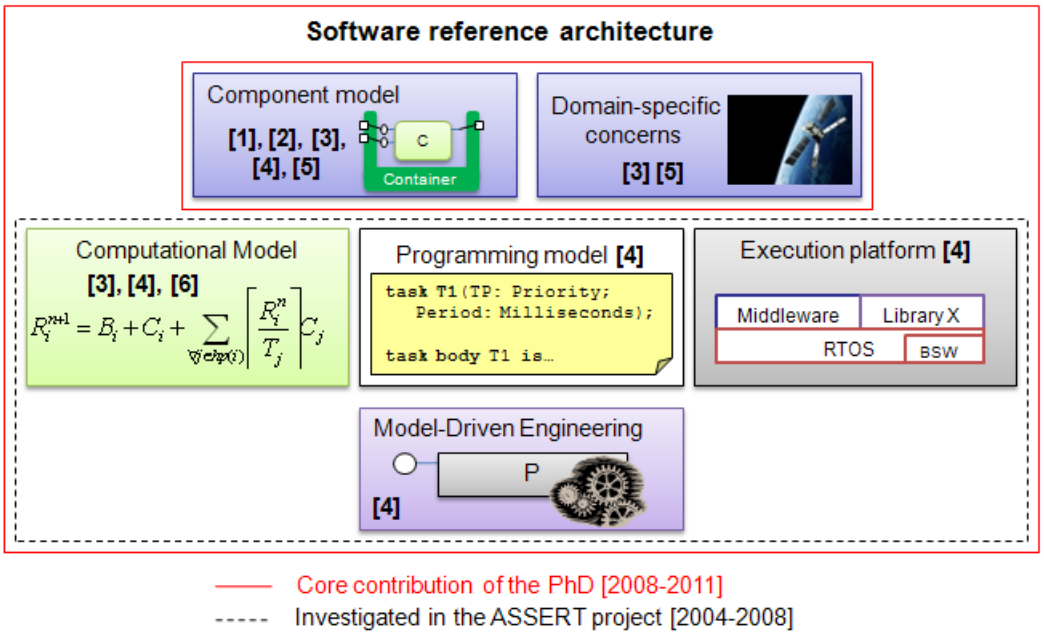

Figure 1.9: Our proposed approach. In the picture we depict the constituents already investigated in the scope of the ASSERT project. We depict in square brackets the concerns of the software architecture dealt by each constituent (cf. fig. 1.5). The contribution of the PhD to the approach is also highlighted.

In figure 1.9 we depict the constituents of the software reference architecture; the concerns of a software architecture that are addressed by each of the four central constituents of the approach and by MDE and domain-specific aspects; and we clarify the contribution of this PhD thesis.

In chapter 3 we elaborate on the most important elements on which the component model is centered: (i) rigorous separation of concerns, achieved during software design with the support for design views (following and specializing ISO 42010/IEEE 1471 [76, 69]) and by careful allocation of concerns to the the various software entities of the approach; (ii) the support for specification and model-based analysis of extra-functional properties; (iii) the inclusion in the component model of space-specific concerns while consciously trying to keep them separate from domain-neutral concerns.

Additionally we clarify and refine the notion of property preservation that was initially investigated during ASSERT.

In the same chapter we also discuss a concrete incarnation of the proposed component model, experimentally built around a domain-specific language, a graphical editor and associated design environment for schedulability analysis and code generation.

Thanks to our involvement in the CHESS project (ARTEMIS JU grant nr. 216682, 2009-2012) [138, 105] we are also able to compare the results of this effort with an alternative implementation of the same component model based on the extension of a subset of the UML MARTE profile [99].

We highlight the current achievements of the two implementation strands, the different difficulties encountered in the implementation and the remaining challenges ahead of us.

In chapter 4, we provide a qualitative evaluation of the software reference architecture, by elaborating on the technological feasibility of the approach, discussing on the coverage of the requirements by our solution and reflecting on the goodness of the approach with the help of a case study and the preliminary feedback provided by industrial partners.

Finally, in chapter 5, we draw some conclusions on the current results of the initiative and discuss the current and future activities related to the software reference architecture.

Chapter 2

A software reference architecture for on-board space applications

In this chapter we elaborate on the proposed approach to the definition and realization of a software reference architecture for on-board space applications.

Initially, in section 2.1 we discuss the industrial needs elicited with interactions and focused discussions with the main stakeholders of the European space domain.

Later on in section 2.2, we describe the technical requirements that represents the baseline that has to be fulfilled by the a novel development approach that aims at fulfilling all the industrial needs elicited by the stakeholders. We report on the activities that permitted the derivation of those requirements, the criteria that were used to assess the difficulty of realization and establish their importance.

In section 2.3 we describe the overall approach we propose for the software reference architecture. As we argument therein, the software reference architecture shall be centered on four fundamental constituents: (i) a component model; (ii) a computational model; (iii) a programming model; and (iv) a conforming execution platform. We describe in details the role of each of the constituents, and for each of them we report and discuss the most similar approaches in the state of the art. Finally, we elaborate on the interactions and possible overlapping of the four constituents and we motivate why all of them are necessary to fulfill the technical requirements for the software reference architecture.

Finally in section 2.4 we examine the state of the art of similar approaches, to elaborate on the concepts that inspired our work and on the improvements that we introduce with respect to existing approaches.

2.1 Industrial needs of the European space domain

The ESA initiative for the definition of a software reference architecture is a harmonization of methodologies and technologies around the Agency charter (and its *geographic return* policy, presented later on in this section), seeking to earn the different benefits relevant for the involved stakeholders: ESA itself as the procurement agency, software prime contractors and software suppliers.

One of the first goals of the doctoral thesis consisted in gathering all the industrial needs set as strategical goals by ESA, or related to the other stakeholders, in order to have a clear picture of all the strategic and economic objectives to fulfill for future ESA missions. The finalization of these goals is important as the understanding of their impact is fundamental to drive the investigation towards a good solution.

The industrial needs were gathered out of discussions with staff members of ESA and discussed in various meetings of the SAVOIR-FAIRE working group, comprised of staff members of national space agencies (CNES [France], and DLR [Germany]), software prime contractors (EADS Astrium [France], Thales Alenia Space [France], Swedish Space Corporation [Sweden]) and the main software suppliers (Scisys [United Kingdom], GMV [Spain], Terma [Denmark], Space Systems Finland [Finland]) and tool vendors (Intecs [Italy]).

In table 2.1 we present an outline of the industrial needs. We also complemented each industrial need with a personal evaluation of its nature: some industrial needs in fact can be considered common to all software domains including mainstream or enterprise software (albeit the context, hence the possible solutions, are completely different); a subset of the industrial needs are common or very similar to those of comparable high-integrity real-time domains; finally, a sizeable subset of them instead is specific to the space domain in general and the European context in particular.

In the reminder of the section we provide a detailed description of the industrial needs and we comment on their relationships and their collective impact.

ID	Industrial need	Type
IN-01	*Reduced development schedule*	C
IN-02	*Higher cost-effectiveness of software development*	C
IN-03	*Support for incremental and parallel software development*	C
IN-04	*Multi-team development and product policy*	HI/DS
IN-05	*Quality of the software product*	HI
IN-06	*Lower effort intensiveness of Verification and Validation*	HI
IN-07	*Role of software suppliers*	DS
IN-08	*Mitigation of the impact of late requirement definition or change*	HI
IN-09	*Simplification and harmonization of Fault Detection Isolation and Recovery*	DS
IN-10	*Lower effort for flight maintenance*	DS
IN-11	*Future needs*	HI/DS

Table 2.1: The main industrial needs of the European space domain. Where the entry reads C, the need is common to all software domains; where it reads HI, it is similar or common to other high-integrity real-time domains; where it reads DS, the need is domain specific.

Industrial Need 01 - Reduced software development schedule. Future projects will require software to be developed in a shorter schedule for several reasons. First of all, the definition and finalization of software requirements occur later in the project schedule than they used to. In fact, the mission definition and system engineering phases take longer as they are the project activities where most of the eventual value added resides. Since the release of the product is usually tied to astronomical events (which define or constrain the launch window) or other external factors (for example, the procurement of the launcher vehicle), the implementation activities, including software development, are consequently compressed within residual time, typically towards the tail-end of development.

Moreover, as we discuss in the description of IN-02, staggered incremental releases of the on-board software are required so that electrical integration and HW/SW integration tests can be performed incrementally and the relevant workmanship is more efficiently deployed.

For all these reasons, although the software itself only accounts for a minor fraction of the total cost of the satellite (around 10%), delayed software releases may have large impact on the overall project schedule as they hold the work of the HW/SW integration [and system] team, and consequently increase cost.

Even though the reasons that generate this industrial need are specific to the space domain, the call for reduced development schedule and shorter time-to-market is common to all software domains.

Industrial Need 02 - Higher cost-effectiveness of software development. The budget availed for software development is not going to rise in the foreseeable future. Instead, it may be cut down in favor of other cost elements. Conversely however, the performance and complexity of core functions of the satellite platform (Attitude and Orbit Control System, Data Handling System, etc.) may well grow in response to the demands of end users. Additionally, new complex functions may also be required (i.e., formation flying, advanced autonomy, etc.).

An explicative example of the trend for future needs is given by LISA (Laser Interferometer Space Antenna), a joint ESA/NASA mission for the detection of gravitational waves. It is undergoing a feasibility study and it is proposed for launch in 2020. The mission comprises three satellites flying in formation, which form an almost equilateral triangle with an arm length of 5 millions of kilometers. Each satellite uses two telescopes to detect the laser light emitted by the other two satellites. Each satellite has also two free-floating test masses, each of them the endpoint of one of the incoming laser beams. A misalignment of milli-arcseconds is sufficient to disrupt a laser link, hence the instruments and the software shall be able to detect variations in the micro-arcsecond range. Gravitational weaves will alter the distance between satellites and the interference will be measured on the test masses with an interferometer that has a required precision in the range of the *picometer*. Furthermore, each satellite shall compensate the interference of the solar pressure with micro-propulsion thrusters (with a force in the range of the *micronewton*). These extraordinarily challenging mission requirements will be satisfied only by achieving an unprecedented precision in the AOCS control laws, which can be considered as a big leap in complexity with respect to current state of the practice.

This scenario calls for better cost-effectiveness in the software development, thus ultimately for increasing the ”value” of the software product delivered in a given budget envelope.

Increased cost-effectiveness is achieved in one of three ways: (i) developing the same set of functions for less budget; (ii) developing the same set of functions but under more stringent requirements (for example more robust and accurate control laws) for the same budget; (iii) increasing the number of functions implemented for the same budget.

Cost-effectiveness can be increased by singling out and abating recurring costs. This permits specialized resources to focus their skills on their side of the problem and yield as much value added as possible off it (the functional contents) and to reduce development costs while delivering the same set of functions. Abatable recurring costs in this context also arise from those parts of the software that do not directly deliver value added and are not mission-specific, e.g. device drivers, the real-time operating system or kernel, communication services, archetypal parts of the application software (which are mostly related to the tasking and real-time aspects of the software). The more they can be factored in the recurrent automated elements of the development the better for the economy of the development.

This industrial need may be considered common to all software domains; the means to fulfill it shall be specific to the space domain and in accord with all the other industrial needs.

Industrial need 03 - Support for incremental and parallel software development. The novel development approach shall support different system and software development practices.

First of all, it is a common need to require an early release of an initial version of the software (”V0”) to start and support electrical and avionic integration tests.

Then, at least two main development models are relevant for the domain. The first is *incremental development*, in which the on-board software is built incrementally, gradually adding more functionalities. A possible implementation of this model would consist in creating a preliminary release of the software for electrical and avionics integration, then adding commandability and observability of the software via telecommands (TC) and telemetry (TM) (see section 3.7.2), then integrating the AOCS, which represent the central and most complex application of the platform software, then proceed with the integration of other functional applications (thermal management, power management, mission management, etc..) and at the end integrating the autonomy capabilities of the spacecraft.

The second development model of interest is *parallel development*, in which the on-board software is divided in parts which are developed by separate development teams, possibly under the responsibility of different companies.

The most common example of that development model is the parallel development of the AOCS and DHS applications, which in the past were systematically allocated to distinct processor units under the responsibility of distinct companies. More recently, this strategy is adopted also to facilitate the application of the geographical-return policy (see IN-04). The Herschel and Planck spacecrafts for example were designed using this strategy.

The novel development approach shall not get in the way of the desired development model and arguably helping to manage and mitigate the effort due to communication between different teams, especially when borders between companies have to be crossed.

This industrial need may also apply in other high-integrity real-time domains.

Industrial Need 04 - Multi-team development and product policy. Decomposition of the software product and the possibility to easily and cleanly subcontract part of it to different suppliers are crucial factors to the geopolitical economy of the European space domain. This need originates from the *geographical return* policy sanctioned in the ESA charter, which requires to ”ensure that all Member States participate in an equitable manner, having regard to their financial contribution, in implementing the European space programme”.

There are different needs in this regard:

- the enforcement of subcontracting schemes to software primes or software suppliers only;
- the subcontracting of distinct processing units to different companies (as mentioned in IN-03), including the responsibility for the deployed software;

- the subcontracting from software primes to software suppliers by enabling the maximum flexibility in the choice of the subcontractor, in order to maximize the adherence of the bid to the contingent needs originated by the geographical return policy.

This industrial need is space specific and it is related to the specific European context.

Industrial Need 05 - Quality of the software product. The quality exhibited by the on-board software (in both functional and extra-functional terms) shall be no less than attained with current practices. Adhering to a well-defined development process and adopting qualified methodologies and technologies shall stay as central prerequisites to the novel development approach.

This industrial need is common to all other high-integrity real-time domains.

Industrial Need 06 - Lower effort intensiveness of Verification and Validation. In the space domain, Verification and Validation (V&V) activities are by far the largest and most labor-intensive contributor to the software development cost: they typically range 50% to 60% of the total cost. The new development approach shall therefore strive to contain the labor intensiveness of V&V.

This industrial need is common to all high-integrity real-time domains.

Industrial Need 07 - Role of software suppliers. The current structure of the European market is characterized by the tangible dominance of two prime contractors: Thales Alenia Space and Astrium Satellites (EADS Astrium). This sort of market duopoly promoted a diversification of concepts, methodologies and technologies used for the development of on-board software to the extent that a software supplier can competitively provide its services only in a single supply chain.

A strategic goal of ESA consists in allowing suppliers to provide software to every prime contractor, without the need to adapt the software to the specific development policies of each prime. This would promote the competition of suppliers in terms of software functionalities, performance and cost.

However, since the amount of ESA satellites is limited, the market is not big enough to sustain the presence of several software suppliers. Therefore, it is not foreseen a significant change of the players in the market, but rather a change of the focus of their business activities.

This is a space-specific industrial need and it is related to the peculiar European context.

Industrial need 08 - Mitigation of the impact of late requirement definition or change. The definition of new software requirements or their change unfortunately may occur during the whole software lifecycle.

The causes of this situation can be ascribed to the activities that precede software development and directly influence it. The most typical causes are in fact: a late refinement of system design; the evolution of the operational level; the late finalization of the policy for the system-level *Fault Detection, Isolation and Recovery* or of the mission management. Finally, software modification may be required to compensate for hardware problems found during system integration. The compression of the schedule for software development (as described in IN-01) is expected to increase this trend.

Late definition of software requirements and requirements changes can (and arguably do) disrupt the schedule for software development as they may impose longer time-to-release and may occur when fundamental decisions on software architecture and software design have already been fixed.

The novel development approach shall be able to mitigate the effects we described.

This industrial need may also apply in other high-integrity real-time domains.

Industrial need 09 - Simplification and harmonization of Fault Detection Isolation and Recovery. Fault Detection, Isolation and Recovery (FDIR) is one of the most complex parts of the on-board software.

For future missions, a simplification and hopefully harmonization of the FDIR approach is advocated. This need has to be attacked both at system and software level. For the former, system engineers have to rationalize the definition of the FDIR strategy. For the software side of the problem, a simplification and rationalization of the software provisions for FDIR is needed.

FDIR is typically the most problematic part of the on-board software. This is typically the result of at least two factors: (i) often the software part of the FDIR is not considered as an integral part of the application under development; its development proceed separately with loose coordination with the designers of the application. As a results, FDIR is something that adds to the nominal execution modes of the application; (ii) it is the software part whose design and finalization occurs later in the development, thus forcing to costly backtracks of parts of the software already close to completion.

For those reasons, FDIR risks to be disorderly and intimately coupled with other functional contents, ultimately breaking a sound separation of concerns in software development.

The novel approach should aim at providing a clean separation between the FDIR strategy (that is the policy to be realized to fulfill the applicable system and software requirements) and the mechanisms to realize it (which are the subject of the advocated rationalization). Such a separation is also expected to mitigate the effects of late definition of FDIR requirements that we mentioned in the previous industrial need.

FDIR is present in all missions and is one of the most important contributors to the complexity of the overall on-board software. According to a technical report by NASA [40], the *incidental complexity* of FDIR may become the most hindering factor to the achievements of

the requirements of future space missions, as opposed to the *essential complexity* that is introduced by the challenges of the requirements.

This industrial need is space specific.

Industrial need 10 - Lower effort for flight maintenance. Software flight maintenance is part of the specificity of the space domain. In contrast to similar domains (like civil avionics, or automotive) where it is always possible to physically access the system to perform maintenance operations, after the launch of the satellite all software maintenance operations have to be performed remotely.

For what concerns software, remote flight maintenance may be required to adjust parameters of the software, to upload a software patch that is able to mitigate the consequences of faulty hardware, or to correct software bugs. The need to replace in-flight parts of the on-board software is thus inherent to the domain.

Flight maintenance contributes to the operational costs of the satellite (hence the budget to operate the satellite rather than building it). Reduction of the effort of the maintenance operations, as well as a harmonization of the maintenance strategy will decrease the time and thus operational cost of maintenance.

In-flight maintenance operations often require a reboot of the on-board software after the upload of a new software image. Reboot constitutes a risk for the spacecraft, as it cannot process the commands from ground until the data handling system is restored to an operational status. Problems that occur in the early stages of the bootstrap may thus compromise the spacecraft mission or leave the operational team with crippled means to communicate and operate the spacecraft for further troubleshooting.

It would be desirable then to minimize the risk of those in-flight maintenance operations by updating parts of the software without having to reboot the processor unit. We can also take advantage of the component-oriented approach to investigate the possibility to reload and restart at run-time components, which would earn us also a better control on the abstraction and granularity of the software to uplink (by reasoning in term of components to uplink and not of memory regions).

Another interesting maintenance scenario occurs when managing a constellation of satellites. Every satellite of a constellation typically is launched with the same on-board software. However, during the lifetime of the mission, different patches may be applied to distinct satellites. Therefore the on-board software of the different satellites evolves separately and starts to diverge. Easy recording of the evolution, annotation of the justification for the modification and tracement of the modifications to each version of the on-board software would decrease the maintenance effort through the whole software life-cycle.

Industrial Need 11 - Future needs. As the novel development approach is targeting future ESA missions, it shall also cope with future needs related to software.

The novel approach is not expected to address directly all these needs during its initial investigation and preliminary definition. However all these needs should be formulated and a preliminary assessment shall be performed to understand if their inclusion in the novel approach is compatible or may break fundamental assumptions.

The most important needs that were identified are:

- Multi-core processors will eventually enter the picture even in the on-board software domain. The novel approach shall not be undermined by that shift in hardware technology.

- Execution of software of different criticality levels in the same processor board. The applicable standard for safety assurance (ECSS Q-ST-80C [48]) classifies the on-board software with a set of safety levels according to the possible consequences of its failure. The classification is comparable with the more known DO-178B standard [115] for civil avionics (level A is the highest level of criticality, level D is the lowest). The standard prescribes the activities and the output documentation that has to be produced in order to provide evidence that the software exhibits the requested level of quality.

  In almost all cases, the platform on-board software (everything except the payload software) ranges in an interval of criticality level between B and C. There are just a few examples of on-board software classified to the highest safety level (level A), so that if the software is not executed, or if it is not correctly executed can cause or contribute to a system failure resulting in catastrophic consequences, like loss of life, loss of a system or of a launcher site facility or lead to long-term detrimental effects on the environment (cf. [48, 47]).

  One example is the safety module of the ATV (Automated Transfer Vehicle) which controls the automated docking with the International Space Station (ISS). In that case, the level A software was segregated in a separate processor module with quadruple modular redundancy, in order to meet the stringent requirements imposed by the standard.

  In future missions instead, it is foreseen that level A software may be integrated with the rest of the platform on-board software (level B) and executed on the same processor.

  Another possible future scenario is the integration of payload software (which typically has criticality level D) with the platform on-board software (criticality B). At the moment, the payload software is typically executed on a distinct on-board computer.

- Another future trend is the greater importance of security concerns. Future missions may require the integration of applications with different security levels or dual-use missions

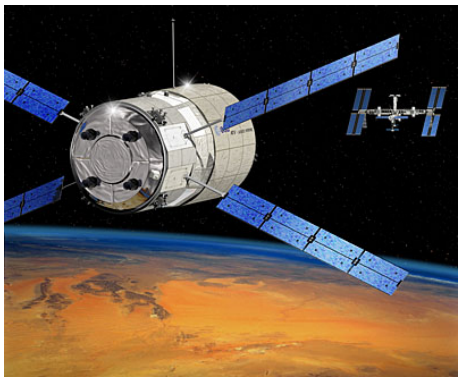

Figure 2.1: The Automated Transfer Vehicle (ATV) approaching the International Space Station (ISS). *Credits: ESA*

in which security-critical software produced by distinct companies is deployed on the same processing unit.

- As a response to the safety and/or security needs just described, in the future it is foreseen that there will be the adoption of some form of Time and Space Partitioning (TSP). The central concept of TSP is the partition, which is a logical container for a software application, so that *""the behaviour and performance of software in one partition must be unaffected by the software in other partitions"* [118]. In particular, for what concerns the time dimension: *"Temporal partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter, and duration of scheduled access to it"*. For what instead concerns the spatial dimension: *"Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partitions"*.

  A recent initiative of ESA is trying to define an incarnation suitable for space of Integrated Modular Avionics (IMA), an approach to Time and Space Partitioning of the civil avionics domain [87, 144]. The dominant incarnation of IMA for avionics is the ARINC 653 standard [1], which defines a IMA architecture centered on two-level scheduling, with a static global scheduler [10, 52, 14] to warrant temporal isolation and a set of partitioning requirements for the realization of a strategy to the attainment of spatial isolation. The standard specifies the Application/Executive interface (APEX), a software interface between a partitioning operating system on top of the avionics computer and the application software deployed in the partitions; the ARINC architecture prescribes

the semantics of intra- and inter-partition communications, and describes how to perform partition configuration. All APEX services are complemented by an API specification in a high-order language, allowing the implementation in various languages and on top of various ARINC-compliant hardware.

2.1.1 Discussion

The industrial needs that we described in this section were collected with several interviews and discussions with space stakeholders and then evaluated and refined during the meetings of the SAVOIR-FAIRE working group.

Unfortunately SAVOIR-FAIRE did not explicitly assign relative importance or priority to the needs. This lack of classification is not ideal to completely understand the picture even if it is clear that some of these needs are more important; for example IN-01 (Reduced software development schedule) and IN-02 (Increase the cost-effectiveness of software development) are of foremost importance, and this is not really a big surprise, as they are exactly the needs from which we started our narration in chapter 1.

Fortunately, as we will see in section 2.2, this lack of explicit assignment of importance is partially covered by the technical requirements that were later derived, which are instead prioritized.

We later classified the needs according to their domain neutrality or domain specificity and reasoned on the aspects they impact on.

In figure 2.2, we highlight the relationship between a subset of the industrial needs, to visually depict how they represent opposing forces in the development process. The figure represents on the x axis the time of the project development. We highlight on the left side two project activities that are mandated by the ECSS-E-ST-40C standard [46]: the system requirement review (SRR) and the software requirement review, which is notionally carried out either in a dedicated activity (SWRR) or, for projects that are behind of schedule, just before the software preliminary design review (SW-PDR). Then, on the right side, we highlight the moment in which there is a first release of the software (nicknamed ”V0”), that is used for electrical and avionics integration tests (AIT), and the moment of the final release of the software.

The effects of the compressed development schedule are represented by IN-01: the finalization of the system requirements (hence also system design) occurs later than before while the software release is tied to external factors. The later completion of system design instead implies that the finalization of software requirements will occur later in the development process; this is represented by the arrow for IN-08. Finally, with the IN-03 arrow we depict the need to provide an early release of the initial version of the software for electrical and AIT. The diagram on the top part of the figure complements the diagram with the amount of persons involved in software-related activities during the timeframe of development. The peak on the

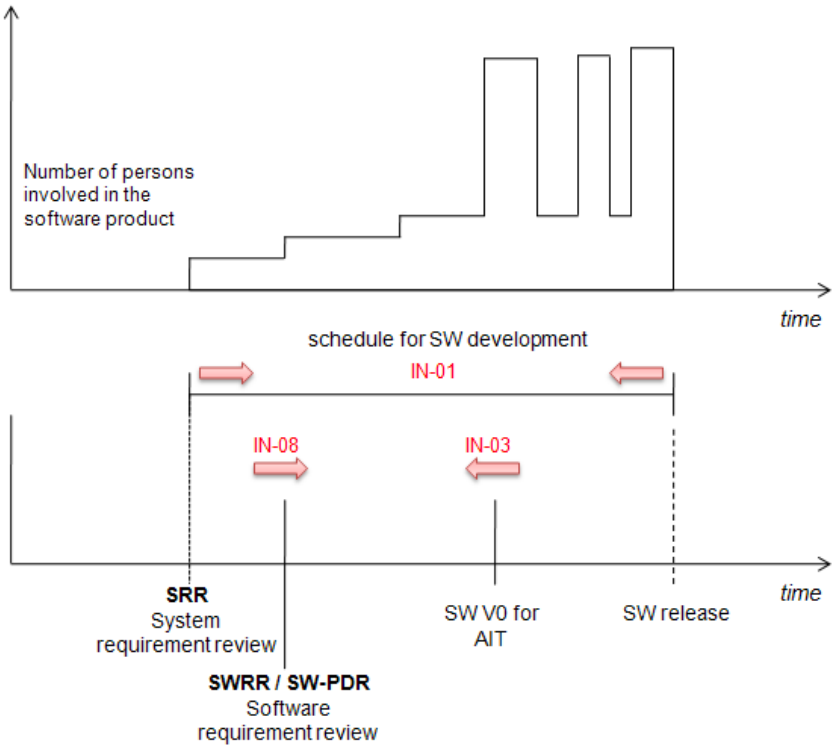

Figure 2.2: Relationship between a selection of the industrial needs and the directions of the forces they act as opposing forces in the development process. On top, a diagram which depicts the amount of persons involved in software development. The peak at the release of V0 and subsequent incremental releases is due to the amount of persons (external to the software development team) involved in the HW/SW integration.

vicinity of the release of ”V0” shows that in that time window, in addition to the software team, the HW/SW integration team is mobilized and is waiting for the V0 release of the software. Delays in the release hold unproductively the integration team, thus increasing the time and cost of development. This reasoning of course applies also to the integration of subsequent incremental releases of the software, as shown by the subsequent peaks in the diagram.

2.1.2 Derivation of high-level requirements

The analysis of the industrial needs discussed in section 2.1 compounded by technical and programmatic discussions held within the SAVOIR-FAIRE working group, led to the decision to derive a set of high-level technical requirements to use as the baseline for the investigation of a novel development approach to on-board software.

The derivation activity was productive, especially thanks to the visits the author was able to perform at the premises of the two prime contractors, EADS Astrium (Toulouse) and Thales Alenia Space (Cannes). This close relationship with the major players of the domain facilitated faster discussion and exchange of ideas and rapid corrections in response to feedback.

The results of the activity were later submitted and discussed within the SAVOIR-FAIRE working group, where, after some reviews, the collective set of high-level requirements ultimately received the approval of all the convened stakeholders.

Table 2.2 highlights the list of high-level requirements that are strategical in the build up of the novel development approach and the industrial needs from which they were derived.

Table 2.2: The high-level requirements derived from the industrial needs.

ID	**High-level requirement**	**Derived from**
HR-01	*Separation of concerns*	IN-02: Higher cost-effectiveness of software development, IN-05: Quality of the software product, IN-06: Lower effort intensiveness of Verification and Validation, IN-07: Role of software suppliers, IN-09: Simplification and harmonization of Fault Detection Isolation and Recovery
HR-02	*Software reuse*	IN-01: Reduced development schedule, IN-02: Higher cost-effectiveness of software development, IN-07: Role of software suppliers
HR-03	*Reuse of V&V products*	IN-01: Reduced development schedule, IN-02: Higher cost-effectiveness of software development, IN-06: Lower effort intensiveness of Verification and Validation, IN-07: Role of software suppliers
HR-04	*Component-based approach*	IN-01: Reduced development schedule, IN-02: Higher cost-effectiveness of software development, IN-03: Support for incremental and parallel software development, IN-04: Multi-team development and product policy, IN-05: Quality of the software product, IN-06: Lower effort intensiveness of Verification and Validation, IN-07: Role of software suppliers, IN-10: Lower effort for flight maintenance

Table 2.2 – continued from previous page

ID	High-level requirement	Derived from
HR-05	*Static analyzability*	IN-05: Quality of the software product, IN-06: Lower effort intensiveness of Verification and Validation
HR-06	*Property preservation*	IN-05: Quality of the software product, IN-06: Lower effort intensiveness of Verification and Validation
HR-07	*Late accommodation of modifications in the software*	IN-01: Reduced development schedule, IN-08: Mitigation of the impact of late requirement definition or change
HR-08	*Hardware/Software independence*	IN-05: Quality of the software product, IN-08: Mitigation of the impact of late requirement definition or change
HR-09	*Support for heterogeneous software*	IN-04: Multi-team development and product policy, IN-07: Role of software suppliers
HR-10	*Provision of mechanisms for FDIR*	IN-05: Quality of the software product, IN-06: Lower effort intensiveness of Verification and Validation, IN-09: Simplification and harmonization of Fault Detection Isolation and Recovery
HR-11	*Software observability*	IN-05: Quality of the software product, IN-06: Lower effort intensiveness of Verification and Validation
HR-12	*Software update at run time*	IN-10: Lower effort for flight maintenance

Figure 2.3 depicts the relationship between the industrial needs and the high-level requirements we drew from them. We omit from the figure ”Industrial Need 11 - Future needs”, as we intentionally did not addressed in the first iteration of the investigation so that no high-level requirements were derived from it.

The reader may notice the complex relationship between industrial needs and high-level requirements: almost all the high-level requirements contribute to the fulfillment of at least two industrial needs; some of them are essential for the fulfillment of four or more high-level requirements. We defer to section 2.2.1 for a more detailed quantitative evaluation of the requirements.

We anticipate that after a scrutiny and evaluation of the set of requirements, the SAVOIR-FAIRE working group concluded that the best solution to fulfilling *all* of them is the adoption

of a software reference architecture provably capable of addressing them both collectively and individually.

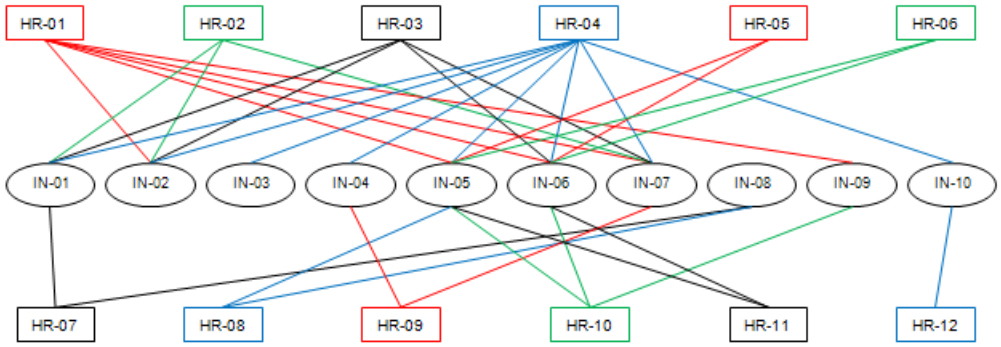

Figure 2.3: Relationship between the industrial needs and the high-level requirements derived from them.

Let us now proceed to describe the requirements in detail and report further results from the evaluation.

2.2 Technical requirements

High-level Requirement 01 - Separation of concerns. In the context of high-integrity real-time systems, separation of concerns [44] is used to cleanly separate different aspects of software design, in particular functional and extra-functional concerns. As a consequence of that, it enables separate reasoning about those concerns. Experience from previous projects makes us believe that when it comes to reuse, separation of concerns may permit reuse of functional concerns independently from extra-functional concerns, thus increasing the probability of software reuse.

Separation of concerns fosters the realization of software that is cleanly defined. It contributes to attaining quality in the software. The separation of functional and extra-functional concerns permits a rationalization of the V&V activity, than can be divided on a per-concern basis. Clean separation of functional concerns can facilitate software suppliers, which can develop software addressing only functional concerns and possibly validating it in isolation (w.r.t. functional requirements).

High-level Requirement 02 - Software reuse. Software reuse is instrumental to reducing the cost of development. The novel approach shall maximize the reuse potential of the functional aspects independently from extra-functional aspects.

Reuse of the functional part of the software under different extra-functional requirements requires: (i) a clean separation of the functional and extra-functional aspects (see also HR-01: Separation of concerns); (ii) a clear and possibly semi-formal description of the extra-functional attributes that are emerging properties of the functional code, like the worst-case execution time (WCET) for a specific target platform and the extra-functional constraints that ought to be applied to the functional code and may restrict its reuse (for example, an AOCS control law could fulfill its robustness requirements only if executed at the frequency of, say, 8Hz, and this constraint shall be known and annotated appropriately in the software design description in order to consider it in the reuse scenario).

For on-board software, our primary aim is the reuse of software across subsequent projects of the same organization; secondarily, software reuse across different prime contractors and different execution platforms.

High-level Requirement 03 - Reuse of V&V products. There is little interest in reuse of software without also reusing the output products (i.e. documentation and tests) of the Verification and Validation activities already performed on it.

The qualification of the software artifact is in fact sanctioned by those activities and each time a software artifact is reused, it is potentially losing the pedigree of qualified software. As the volume of V&V effort required by on-board software is massive, if software reuse does not permit to reuse its V&V value too, then its attractiveness decreases and may become nil.

In a development approach with strict separation of concerns, the approach shall aim to the reuse of the majority of tests for the functional part of the software. Exceptions to this reuse goal are for example numerical accuracy tests, which may depend on the target platform and thus may not hold their validity across different execution platforms.

The achievement of separation of concerns (HR-01) is thus an essential enabler to the reuse of V&V products, even though it *does not imply* the successful achievement of the goal.

This requirement relates back to the notion of *composability* we introduced in section 1.3. In fact the reuse of V&V products depends on whether properties asserted on a component hold true in different contexts of use or when the component is integrated with other components. The requirement instead is not concerned with emergent (system) properties, which we address through compositionality.

In spite of its importance, the fulfillment of this high-priority requirement spans in a longer time frame as it requires an investigation on the methodological, technical, management and normative support to enable and accept at the customer side (i.e. ESA) the reuse of qualification proofs, and a case study to confirm the feasibility and effectiveness of the approach.

High-level Requirement 04 - Component-based approach. The novel approach shall define a component-based approach for the design of on-board software. It shall allow the preservation

of properties in terms of *composability* (preservation of properties verified on a component in isolation once the component is assembled with other components) and *compositionality* (the possibility of determining a property of a component assembly from some transformation of properties of the constituting components). The composability or compositionality of properties are determined by the methodology that underpins the approach.

In Component-Based Software Engineering, the whole software is designed as a composition of components. The domain may require the adoption of hierarchical composition or decomposition of components. In an approach that also requires separation of concerns (HR-01), components are pure functional units (they comprise only sequential code; timing and concurrency aspects in particular are dealt with externally to the component). Components are the unit of reuse (the smallest reusable software artifacts) and therefore the development process shall support the reuse of components in different projects or context, as per HR-02 and HR-03.

High-level Requirement 05 - Static analyzability. Early analysis can confirm (or outline problems on) the software design in the related dimension of concern. As opposed to analysis performed on the already implemented software, the earlier the analysis is performed, the less is the effort required to remedy to detected problems.

The design process and methodology used for the novel approach shall support the verification at design time of functional and extra-functional attributes, through, e.g. schedulability analysis [123], queuing analysis and form of dependability analysis.

There are other types of analysis of interest which instead require the implemented source code, like, e.g. stack analysis and timing analysis [143].

High-level Requirement 06 - Property preservation. When certain extra-functional attributes are used as input for some form of analysis, and the analysis confirms the feasibility of the system in the dimension of interest, those inputs become system *properties*. From that moment they shall be considered as constraints on the system as they specify the ”frame” (of output values/effects/behaviour) in which the system behaviour is consistent with what was predicted during the analysis.

In order to ensure consistency between the analyzed model of the software and the system at run time, those properties have to be enforced or preserved during software execution [137]. This is achievable by enforcing in the implementation (constant) controllable properties, like the period or the minimum inter arrival time (MIAT) of a task and monitoring properties that depend on system execution (as the execution time of a task). Adequate mechanisms shall be provided to realize appropriate handling and reactions to violation of those properties.

Property preservation is related to *composability*, as it ensures that composable properties – asserted at design time – still hold during system execution.

High-level Requirement 07 - Late accommodation of modifications in the software. The novel approach shall facilitate the accommodation of modifications of the software late in the software life cycle. Very often in fact, system integration discloses system-level problems. At that time of the life cycle, software is the only element that still has some degree of freedom in its on-going implementation. For that reason, the fulfillment of new requirements specified to tackle the emerged integration problems is forcedly delegated to the software.

Although part of this problem can be avoided through an improvement of the system requirement definition and system software requirement definition (the subset of the system requirements that impacts software development), as finalized by their respective project reviews (SRR and SW-SRR respectively), this need is often unavoidable.

The software development model that is adopted shall then be *iterative* in nature by permitting iterations to be steps that are confirmative or refining rather than destructive of the work (thus avoiding the detrimental effects of backtracks typical of the waterfall model [116]). The novel approach must be able to accommodate late changes, by modifying or substituting software parts (i.e. components) and minimizing the re-engineering and test effort.

We contend that a development process defined as an incarnation of Model-Driven engineering is the best instrument to tame the iterations of the development as it naturally shifts the iterations in the design and specification phase, and avoids costly backtracks of already implemented parts of the software.

High-level Requirement 08 - Hardware/Software independence. Hardware/software independence permits to develop software that is independent from hardware features. It is typically achieved by carefully isolating hardware-dependent parts of the software in separate modules which are accessible through a defined interface. An example is the Board Support Package (BSP) used by a real-time operating system or kernel (RTOS/RTK). As long as the interface does not change, the software is isolated from changes in the hardware-dependent layer.

Hardware/software independence promotes a layering of the software architecture that is beneficial from the point of view of the quality of the product. It also helps to mitigate the impact of possible hardware changes during the software life cycle.

High-level Requirement 09 - Support for heterogeneous software. It is not realistic to hypothesize the unification of the scattered range of technologies currently adopted by the various key players of the domain. Hence our approach should support interoperability at least between: (i) components written in Ada and C (manually coded or generated with Matlab or Simulink); (ii) the execution platforms in use by prime contractors.

High-level Requirement 10 - Provision of mechanisms for Fault Detection Isolation and Recovery. The novel approach shall provide mechanisms to handle the spacecraft depend-

ability, i.e. the implementation of Fault Detection, Isolation and Recovery (FDIR).

The FDIR is a very complex software subsystem that fulfills the dependability concerns and its implementation spreads over multiple spacecraft subsystems. The FDIR system and software requirements are often consolidated late in the project schedule. To complicate the picture, even though the FDIR responds to dependability requirements (hence extra-functional in nature), very often the implementation of FDIR is breaking the separation of concerns of software.

A possible solution is to provide a common set of software-based mechanisms for FDIR. A difficulty in this field is the reconciliation of the practices of the two prime contractors, which often diverge in their design choice. A valuable strategy would consist in providing a set of functionalities and design patterns that cover the essential mechanisms for the software realization of the FDIR strategy.

High-level Requirement 11 - Software observability. The novel approach shall facilitate *software observability*, which is the possibility to extract information on the current and past status of the software using exclusively the output data retrieved with the services specified by the operational scenario of the satellite mission.

Observability is required: (i) before the launch of the satellite, for software integration and validation and interoperability with the *ground segment* of the system using the *Electrical Ground Support Equipment* (EGSE); and (ii) after launch, during satellite operation, as only remote communication with the satellite is possible, and is carried out in the form of telecommands and telemetry (TM/TC) via the uplink and downlink channels of the satellite (see section 3.7). It is essential that mechanisms that facilitate observability are built in the software architecture, in order to avoid the need for post-launch updates or ad-hoc patches of the software in case the ground team that operates the satellite requires more information to analyze the behaviour of the software or to identify the root causes of a failure occurred on board.

High-level Requirement 12 - Software update at run time. It shall be possible to update a single software component (including its bindings to other components) at run time, without having to reboot the whole on-board computer.

The reboot of an on-board computer is a safety risk for the mission, and decreases the time dedicated to scientific experiment due to the reduction of the mission availability.

2.2.1 Requirement importance, prioritization and evaluation

In this section we present a recapitulation (see table 2.3) of the discussions held about the importance of each high-level requirement, the estimated feasibility of realization and the timeframe to address the requirement.

Requirements where classified according to their importance as either *mandatory* or *optional*. The fulfillment of all mandatory requirements is necessary for an initial implementation of the software reference architecture as a successful solution. The reader can notice that only two requirements are considered optional for this initial version of the software reference architecture: software observability and software update at run time. It is foreseen that their realization does not undermine the overall approach to the software reference architecture and thus the SAVOIR-FAIRE working group decided to postpone their fulfillment for the sake of priorities.

SAVOIR-FAIRE expressed also an evaluation on the difficulty of the realization of the requirements. They where classified according to the following scale: easy, medium, difficult, very difficult). We can see that the vast majority of requirements are considered of medium complexity. The working group specified also the timeframe on which each requirement is expected to be investigated to provide a satisfactory solution. The timeframe is a combined index of the importance of the requirement in the overall picture and the perceived complexity to fulfill it. The requirement are expected to be fulfilled in the short, medium or long term. In the following, we comment on the classification.

First of all, requirement HR-01 on separation of concerns, which is marked difficult, due to the many aspects of the methodology it impacts. We discuss extensively about separation of concerns later in section 3.1. Despite the complexity of the requirement, separation of concerns shall be considered from the beginning of the investigation, as it is the cornerstone of the whole methodology together with the component-oriented approach.

Requirement HR-03 on the reuse of V&V products was instead split in three parts. The reuse of unit and integration tests, is considered easy. The reuse of validation test cases is instead considered difficult. In fact, validation test cases would be reusable if the software part candidate for reuse were: (i) developed to fulfill the functional (and extra-functional) requirements applicable to the first context of use; (ii) a-priori developed to fulfill a broader set of requirements that constitutes its *validation envelope*. Finally the reuse of the *results* of the validation tests is considered the most challenging requirement of the whole set, to the extent that most probably it is more economic to simply reuse the validation test cases and re-run them.

Requirement HR-10 has been split in two parts: the part of the mechanisms provided for FDIR which resides int the application layer are considered of easy realization; the part of the mechanisms which are either factorized in a *reusable* modular library or require support from the underlying real-time operating system (RTOS), are deemed of medium complexity, as their implementation may require the modification and *re-qualification* of existing RTOS or libraries. For this reason the latter part of the requirement is projected in a longer timeframe.

A similar reasoning applies to requirement HR-08 due to the effort necessary to define and consolidate or modify the hardware-dependent part of the interface layer.

Unfortunately, the classification of the difficulty of realization of the requirements as dis-

High-level requirement	**Importance** [**M**andatory / **O**ptional]	**Feasibility** [**E**asy / **M**edium / **D**ifficult / **V**ery **D**ifficult)]	**Timeframe** [**S**hort/**M**edium/ **L**ong term]
HR-01: Separation of concerns	M	D	S
HR-02: Software reuse	M	M	S
HR-03: Reuse of V&V products	M	E (UIT) D (Val Tests) VD (Val Results)	S M L
HR-04: Component-based approach	M	M	S
HR-05: Static analyzability	M	M	S (capturing the properties of interest) M (tool support)
HR-06: Property preservation	M	M	M
HR-07: Late accommodation of modifications in the software	M	M	M
HR-08: Hardware/Software independence	M	E (application level) M (execution platform)	S
HR-09: Support for heterogeneous software	M	M	M
HR-10: Provision of mechanisms for FDIR	M	E (application level) M (execution platform)	S L
HR-11: Software observability	O	M	L
HR-12: Software update at run time	O	M	L

Table 2.3: Requirement importance, feasibility and the proposed timeframe for their fulfillment.

cussed by the SAVOIR-FAIRE working group is flat. In our opinion, a better classification would have discriminated the difficulty of realization of one and the same requirement on multiple axes: the *theoretical* axis, that means the requirement requires the theoretical or methodological consolidation of some foundational aspects for its realization; the *process* axis, that

refers to the difficulty to create a satisfactory solution that matches the current industrial process or the applicative normative support of the domain; and the *technological* axis, which refers to the technological complexity to realize a satisfactory solution fulfilling the requirement.

High-level requirement	**Feasibility** [**E**asy / **M**edium / **D**ifficult / **V**ery **D**ifficult)]	**Scientific foundation**	**Process**	**Technological**
HR-01: Separation of concerns	D	D	D	M
HR-02: Software reuse	M	M	M	M
HR-03: Reuse of V&V products	E (UIT) D (Val Tests) VD (Val Results)	E D VD	M D VD	E M M
HR-04: Component-based approach	M	M	M	M
HR-05: Static analyzability	M	D	M	M
HR-06: Property preservation	M	M	M	M
HR-07: Late accommodation of modifications in the software	M	M	M	M
HR-08: Hardware/Software independence	E (application level) M (execution platform)	E M	E M	E M
HR-09: Support for heterogeneous software	M	E	M	M
HR-10: Provision of mechanisms for FDIR	E (application level) M (execution platform)	E M	M M	E M
HR-11: Software observability	M	M	M	M
HR-12: Software update at run time	M	M	E	M

Table 2.4: Requirement difficulty, with breakdown along the scientific foundation, process and technological axes.

We therefore accompany those requirements with our supplemental classification of their difficulty according to the breakdown we described. The classification is established according to our own understanding of the problem and is reported in table 2.4.

There is not enough *direct* information that permits us to evaluate qualitatively the industrial needs. We can weight them only quantitatively, so as to have some clue on which are the industrial needs that require more effort to be satisfied.

It is apparent (cf. fig. 2.3) that the industrial need that mobilizes more technical requirements is ”IN-05: Quality of the software product”, that is the creation of an approach that is attaining the same level of quality of the software while achieving all the other industrial needs. The lower effort intensiveness of Verification and Validation is the industrial need with the second biggest number of child requirements. It is also valuable to notice the sizeable intersection between the set of child requirements of IN-05 and IN-06, that visibly shows that the reduction of the effort for V&V is ultimately related to the quality of the software.

The third industrial need in this ranking is ”IN-07: Role of software suppliers”, the fulfillment of which entails the possible reorganization of the space market.

Finally, the reduced development schedule and higher cost-effectiveness of development (IN-01 and IN-02), which constitutes the premise of our work and are the foremost goals of the investigation, are related to four requirements each.

The other industrial needs are addressing more specific problems and this is the reason why they are quantitatively related to fewer requirements.

By cross-checking in table 2.4 the estimated difficulty of fulfillment of the requirements we can indirectly argue that IN-01, IN-05, IN-06 and IN-07 are the industrial needs which require more effort to be satisfied.

2.2.2 Discussion

The analysis of the technical requirements by the SAVOIR-FAIRE working group led to the conclusion that the best solution to fulfill all of them is the definition and adoption of a *software reference architecture*.

The software reference architecture consists in a *single*, *agreed* and *common* solution (defined and adopted by all the stakholders) for the definition of the software architecture of systems that are the targets of this investigation.

As we describe in the next section, this also earns us the advantage of permitting to base the solution on the lessons learned and the solid results from past experience (notably, the ASSERT project [7], partially funded by the European Commission, led by ESA, and with a total budget of 15 Million Euro).

2.3 Approach

The comprehensive goal of this PhD thesis is the definition, realization and evaluation of the overall approach to enable the creation of a software reference architecture for the development of on-board software.

Our role in the investigation was in fact to investigate the methodological, process and technological part of the problem, in order to avail to the domain experts a comprehensive approach

that has the potential to fulfill all the industrial needs defined in section 2.1 and comprises all the elements needed to allow them to complete the definition of the reference architecture.

The adoption of design methodologies that generate *non-composable* software artifacts is one of the main factors that contribute to the increase of time and cost of system production. Software analysis and verification that only become possible when the software is completely integrated are scarcely useful and fragile to even the slightest change in the system implementation. Extra-functional properties are especially exposed to this risk. Considering that in the development of a high-integrity real-time system, the cost of the software verification and validation activities may reach up to 60%-70% of the total development cost, it is clear that any improvement in this area may reduce the overall time and cost of production considerably.

The approach we propose to adopt for the software reference architecture at two overarching goals:

- *composability*, that is achieved when the properties (needs and obligations) of individual components are preserved on component composition and deployment on the target system;
- *compositionality*, that is achieved when the properties of the system as a whole can be derived (economically and conclusively) as a function of the the properties of its constituting components.

Composability and compositionality shall then be consistently achieved by a development methodology that exhibits the crucial property of *composition with guarantees* [138], which may be regarded as a form of composability and compositionality that can be assured by static analysis, guaranteed throughout implementation, and actively preserved at run time.

We argue that this property can be achieved with the combined use of:

1. a *component model*, to design the system as a composition of reusable software units endowed with known functional and extra-functional attributes;
2. a *computational model*, to relate the components, execution needs and extra-functional attributes to a set of analysis techniques, thus making the architecture description analyzable;
3. a *programming model*, that is a subset of a target programming language complemented by a set of code archetypes which convey in the implementation the semantics, the assumptions and the constraints of the computational model;
4. an *execution platform* capable of hosting and executing components and policing their compliance with the computational model, while also actively preserving the system and component properties asserted during static analysis.

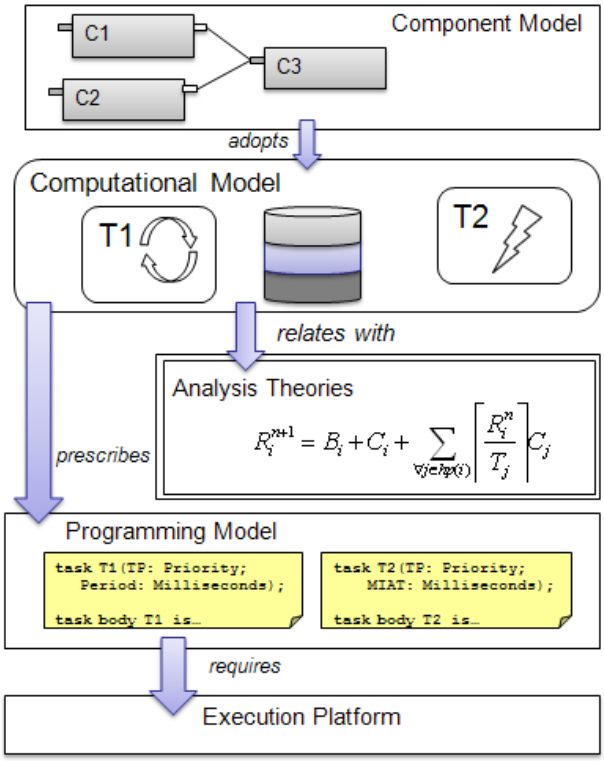

Figure 2.4: The four constituents of our overall approach and their relationships.

We anticipate that the attainment of composition with guarantees in a component-oriented approach requires to solve the following problems:

1. how to specify components and their contractual interfaces;
2. what is the semantics of the contractual interfaces;
3. how to assemble components and what is their communication semantics;
4. how to analyze the system design and extra-functional needs and meaningfully back propagate analysis results to the relevant design entities;
5. how properties entailed by the validated design specification are preserved down to implementation;
6. how properties are enforced at run time.

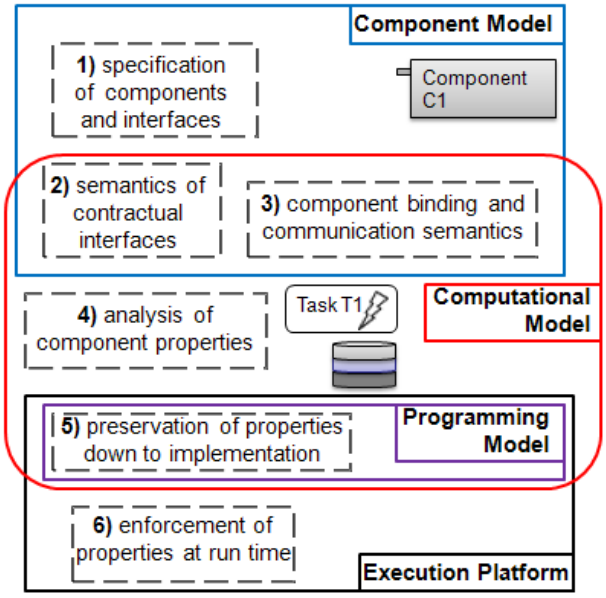

Figure 2.5: Concerns addressed by the various constituents and the overlapping responsibility among them.

In figure 2.5 we depict how we map those problems to the four founding constituents we propose in our approach.

We argue that all these items can be addressed by the combination of a component model (for items 1, 2, 3), a computational model (for items 2, 3, 4) and the programming model that implements it (for item 5), and a suitable execution platform (for items 5 and 6). The interaction between those constituents is delicate and careful attention shall be paid not to jeopardize their controlled interplay.

Multiple incarnations of the approach we propose are possible. However, we argue that its full potential can be unleashed with its adoption in an educated form of Model-Driven Engineering (MDE), by which we may also succeed in taming the destructive effects of iterative development.

As we outlined in chapter 1, constituents 2,3 and 4 (i.e. the computational model, the programming model and the execution platform) were subject of investigation during the ASSERT project [7], for which the author worked prior to the launch of the PhD activity. ASSERT successfully demonstrated the feasibility of a MDE development process for on-board software [24, 106]. The project focused on achieving the goal of *compositionality*, for which the computational model is instrumental.

We based our investigation on the results of ASSERT, and focused on the achievement of

composability, and in extension of *composition with guarantees*. For the former, we investigated the definition and realization of a component model that responds to the needs and requirements elicited in the investigation; for the latter, we revisited the results of ASSERT w.r.t. the computational model, programming model and execution platform, to strengthen them with the notion of *property preservation*.

The resulting MDE approach centered on those four constituents is valid and applicable to high-integrity real-time systems in general. However, we contend that while the achievement of a domain-neutral approach is an interesting result, it is not enough to qualify as a complete approach to enable the definition of the software reference architecture.

In section 3.7 we elaborate on how to complement the approach with domain-specific concerns. That part of the investigation is quite interesting as the domain-specific concerns shall be included in a manner that is not breaking the consistency and assumptions of the overall approach. We maintain that the inclusion of domain-specific concerns to complement the domain-neutral part of a component model is essential for the industrial adoption of the approach, as we deliver industry from performing *a-posteriori* costly (and potentially unsafe) extensions to the approach in order to fulfill the uncovered domain-specific requirements or adapt the approach to the development process in use.

We argue that the overall approach we describe is capable of responding to all the industrial needs of the domain by fulfilling their derived technical requirements, and it is thus sufficient to enable domain experts to select and gather the architectural best practices of the domain and factorize the definition of a recurrent software architecture for the future missions of interest.

2.3.1 Component model

In this section we provide a generic introduction to component-based software engineering (CBSE) to understand the role of the component model in our overall picture. We refer to chapter 3 for a detailed description of the component model we defined and we are implementing.

In component-oriented engineering [79, 130], the whole software is built as a composition of *components*. A component is a reusable software unit which exposes an *interface* to the system. Interaction with the component can be performed only through its interface, which encompasses: (i) the set of *provided services* that the component offers to the system; (ii) the set of *required services* that the component requires from the system in order to discharge its services. The component model further prescribes additional architectural features that are used to create and manipulate components: the possible dichotomy between component types and instances, component hierarchy, component inheritance.

We say that components have *contractual interfaces* because they stipulate an obligation between themselves and the rest of the system: as long as the needs of individual components are satisfied, the component provides its services and works as expected. This property is

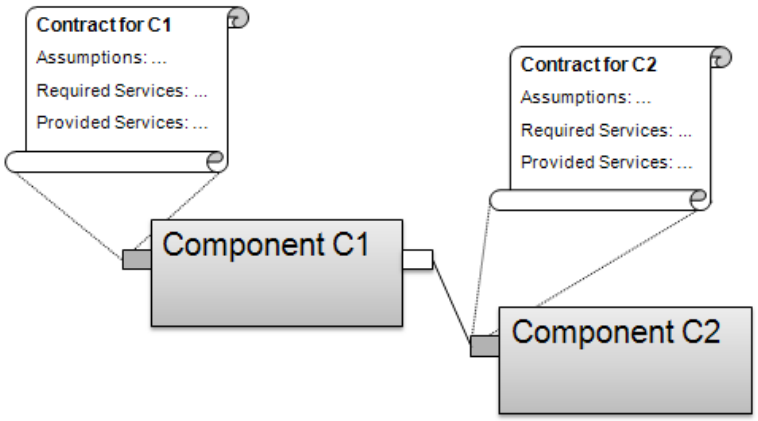

Figure 2.6: A generic depiction of components and their contractual interfaces.

engaging and essential for a methodology that aims at systematic reuse of software: components can be reused in different contexts and environments, as well as bound to other components, as long as their contractual needs are satisfied.

As our work is specifically concerned with high-integrity real-time systems, we must pay attention to the fact that a range of extra-functional properties become of interest and influence *how* services are performed. There can be different approaches to complement a component description with extra-functional aspects. For example: (i) by decorating the functional description of an interface with extra-functional attributes; or (ii) by providing a separate model for the description of the extra-functional aspects of interest such as a resource-consumption model, or a reliability model (as in the ROBOCOP component model [95]).

When the former approach is chosen, component binding becomes not only the syntactic matching between the provided and required services of components, but also the assurance of the semantic information conveyed by the extra-functional attributes attached to the interface.

The component model can be either *static* or *dynamic*. In the former case, it supports only contracts established and fixed once and for all at design or deployment time. In the latter case, the framework may support various forms of dynamic reconfiguration: in a minor form when it is possible to replace a component under the condition that the new component still honors all component matching in which the substituted component was involved and all extra-functional properties and assumption are kept valid; to a greater extent either when some form of run-time renegotiation of the contracts is allowed or the system is open and new components can be created and added to the system after start time. An example of such dynamic nature is given by the FRESCOR framework [30, 51] which support both reconfiguration of contracts and inclusion at run-time of new components.

2.3.2 Computational model

The seminal work by Liu and Layland [84] was the inception of a long string of fundamental results in the investigation of the timing behavior of real-time systems. Real-time systems theory bases its results on the concept of workload model. This is an abstract model of the system that is used to reason about the timing properties of the entities that compose the system at run time. The workload model typically comprises:

1. the taxonomy of the run-time entities of the systems, seen almost exclusively in terms of tasks that contend for access to logical and physical processing resource, and are characterized by their timing and execution attributes (worst-case execution time, release period, urgency, etc..);

2. the scheduling algorithms chosen to arbitrate the use of the processing resources;

3. the enumeration of logical or physical resources other than the CPU which the system tasks may require during computation, and the protocol rules in place to grant access to them (like for example the Priority Ceiling Protocol [124, 58]).

It therefore appears that the classic notion of workload model does not account for any aspects that concern or proceed from the software architecture; since timing and concurrency are the only dimensions of interest to the classic workload models, it is very difficult to relate them to notions relevant to software architectures. This shortcoming can be observed also in more advanced workload models: for instance energy-aware models [16], although they predominantly target mobile computing and soft real-time systems; elastic scheduling models [29, 35]; or frameworks for compositional analysis (for example [112] or [126]), which focus on the composition of timing properties, without a viewpoint related to software architecture.

In order to consistently relate a strand of interest of real-time theory to software architecture, and in particular to a component model, we need an abstract model that stands between them and that conveniently captures the concerns of both dimensions: a computational model. The notion of computational model:

1. describes how computations are carried out in a system by prescribing the form of legal basic entities and their associated properties in the dimension of concurrency, time and space, and how entities communicate to one another;

2. provides an abstraction of a system that can be analyzed in accord with an associated analysis framework and its underlying theory;

3. permits to consistently relate the run-time entities that correspond to architectural components and the extra-functional attributes and properties that may be specified in component interfaces, as well as to relate component interactions to the workload model that underpins the analysis framework;

4. requires the explicit adoption of a programming model for the target implementation, which rejects all language constructs that do not conform with the synchronization and execution semantics permitted by the analysis framework and that can thus undermine the assumptions made by the analysis.

The kind of analysis that can be used in practice strictly depends on the chosen computational model and in particular on the specific semantic restrictions on execution and communication that are imposed on the system.

The constraints placed on a system that adopts a given computational model have two contrasting effects:

- to make the system analyzable: they prohibit semantics that cannot be analyzed according to the body of reference analysis theories;

- to reduce the freedom of expression availed to software specification: the constraints restrict what the software can do, and therefore retain as legal and meaningful only what the support theory can analyze.

Intuitively, the more restrictive the computational model, the simpler (i.e. composed of basic entities that entail simple semantics) the systems that comply with it, and the simpler the analysis techniques that can be used. The tradeoff between the reduced expressive power and the increased analyzability should be carefully evaluated for the target application domain, as it may greatly impact the development at the level of the process (which may become too rigid and prescriptive) as well as of the product (which may become too inflexible).

The key argument here is that: (i) an approach for real-time systems makes sense and serves its purpose solely if it always leads to an analyzable implementation; (ii) the analysis that is performed on the abstract description of the system is meaningful only as long as the system implementation conforms with the semantics prescribed by the computational model and assumed by the analysis.

For what concerns (i), it follows that it is the body of the chosen analysis theories that dictates the spectrum of the admissible implementations.

When some real-time theories assume questionable simplifications to ease the mathematical solution of the problem, they provide valid or useful results (not devalued by their excessive pessimism) only if those specific assumptions hold on the system operation; the net consequence is also that they narrow the admissible form of the implementation to the extent that

it may be considered poor and uninteresting. An example is the infamous assumption of task independence, which is patently at odds with architectural best practices and is able to defeat even a simple producer-consumer collaboration pattern.

For what concerns (ii), the programming model is in charge of conveying in the implementation the execution and synchronization semantics assumed by the analysis.

2.3.3 Programming model and execution platform

The programming model that realizes a given computational model, together with a conforming execution platform are the two constituents that we use to achieve the following goals: (i) ensure that the implementation fully conforms with the semantics prescribed by the computational model and assumed by the analysis; (ii) complement composability and compositionality ensured at the level of the component and computational model with *property preservation*, that is, ensuring that the contracts stipulated between components (extra-functional needs and provisions) and sanctioned by analysis (as analysis results) are respected at run-time.

The achievement of those two goals warrants that the system as it was represented for analysis purposes is a faithful representation of its implementation and the results of the analysis performed on the system model are valid prediction of the system at run time.

The *programming model* denotes a subset of the constructs of a programming language, coupled with a set of suitable code archetypes. The subset of the programming language is instrumental to achieve (i), the code archetypes, together with the execution platform are necessary to realize (ii).

A concurrent subset for real-time systems tailored for this goal typically defines [136]:

1. a given set of concurrency constructs;
2. a range of communication and synchronization means;
3. a given notion of time;
4. a given interface with the external environment.

Each computational model may have multiple programming models, one for each target implementation language (see figure 2.7).

The Ravenscar Profile [26] of the Ada programming language [73] is an example of a concurrent language subset tailored for the implementation of high-integrity real-time systems. A suitable restriction of the Real-Time Specification for Java (RTSJ) [132] can be an equivalent subset [81] for the Java programming language. If coupled with suitable code archetypes, they can be considered as programming models for the Ravenscar Computational Model [27], a priority-driven computational model for high-integrity real-time systems.

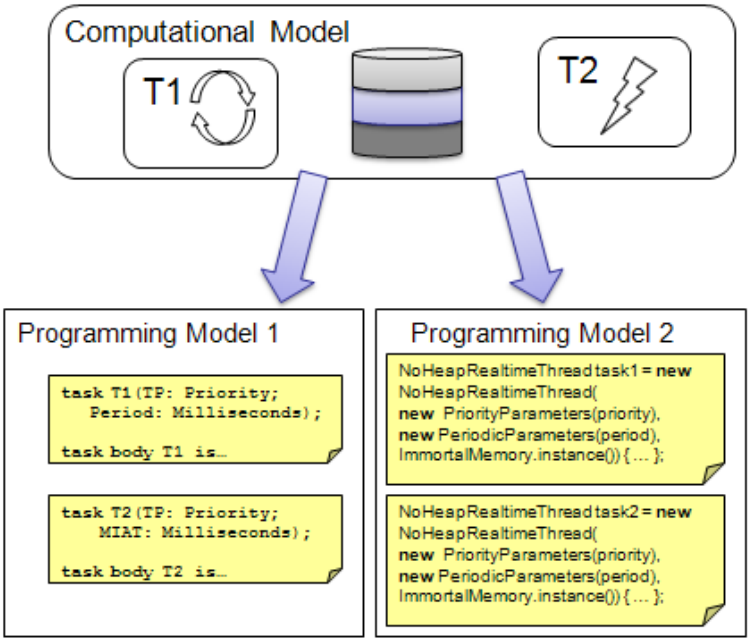

Figure 2.7: Relationship between a computational model and its realizing programming models.

For what concerns instead property preservation, we are interested in three distinct and complementary provisions:

1. enforcement of the timing properties that are to stay constant during execution;
2. monitoring of the timing properties that are inherently variable in so far as they are influenced or determined by system execution;
3. trapping and handling of the violations of asserted properties.

In class 1 we include the provisions needed to enforce the period and static offset for cyclic tasks, and the minimum inter-arrival time (MIAT) for sporadic tasks. Those values are stipulated as constants that feed schedulability and performance analysis and they must therefore be obliged as exactly as possible during execution. Needless to say, the granularity and accuracy of the software clock and the absolute vs. relative nature of delays have a tremendous influence on the degree of preservation that can be attained [145].

Provisions in class 2 concern the monitoring of the execution time of tasks and their deadlines, and monitoring of blocking times due to shared resource contention. Those attributes are intrinsically variable. Deadlines in particular, even if they are specified as relative in the design space, are transformed in the implementation in absolute deadlines; therefore they take effect

basing on the absolute notion of time, and this prevents any drift caused by the interference of tasks or run-time mechanisms (see also section 3.8.2.4).

The ability to monitor those variable attributes is the prelude to being able to detect the violations of their bounds as well as identify the failing party. The bound assumed for task execution time is the worst-case value of it, known as worst-case execution time (WCET), which should not be overrun. Deadlines should not be missed, which we can detect by observing whether the jobs of tasks always complete before their respective deadline.

Finally, class 3 requires adequate mechanisms to detect such violations, together with the ability to act on the violation event following some user-defined fault handling policy.

The responsibility of the provisions above is shared between programming model (specifically, the adopted property-preserving code archetypes) and the execution platform.

The last responsibility that falls on the execution platform consists in rejecting all programs that express semantics prohibited by the adopted computational model and thus not expressed by the compliant programming model.

With this requirement we mean that the execution platform shall be capable of executing *exclusively* the tailored language subset delimited by the programming model. This is not a limitation: on the contrary, the reduced run-time support can arguably be realized in a real-time operating system or kernel of lower complexity, leading to application with less overhead in space and time.

Finally, it is advisable, or better still necessary, to pay careful attention to the requirements that have to be fulfilled by the execution platform. This may increasingly include numerous middleware-level services, such as, e.g., deployment, intra- and inter-processor message transmission, distribution transparency, context management, time distribution, etc. The implementation of such middleware services should most definitely conform with the restrictions of the adopted computational model. Moreover, the middleware itself may have a non-negligible impact on the timing properties of the system, which cannot and must not be ignored by static analysis. The compliance of the middleware to the computational model and the knowledge of its design and behavior permit to consistently integrate it in the system representation that is subject to analysis, providing more accurate predictions on the timing behaviour of the system.

2.3.4 Interactions between constituents

Component model and computational model. The adoption of a computational model for the development of high-integrity real-time systems is a delicate issue as it partially overlaps the aspects regulated by the computational model.

We see three areas in which there is direct interaction between the two concepts:

1. semantics of non-functional properties of components;

2. semantics of communications between components;

3. implementation of the component model.

Components expose contractual interfaces in force of which they stipulate some obligations toward the system: as long as the needs of the component are satisfied, the component is able to discharge its services as specified. Earlier in this thesis we insisted that extra-functional attributes must be present in the contractual agreements that interconnect components. Component binding must consequently also cater for the assurance of the extra-functional semantics entailed in the interconnected interfaces. The most essential aspects of such extra-functional semantics concern execution, communication and isolation.

This observation calls for the selection of the extra-functional attributes which are to be attached to component interfaces. As a general rule, the extra-functional attributes that decorate a component interface should be: (i) *necessary*, so as to fully describe the component in the dimensions of interests — concurrency, time, space, safety —; and (ii) *valid*, in that they should enable meaningful forms of static analysis.

It is interesting to notice that this selection is done *exclusively* at the level of the component model, but it has a considerable impact on the overall framework. A component model that is not able to capture all the attributes of interest makes the overall framework unable to analyze and later preserve them.

The selection of the extra-functional properties that are to decorate component interfaces favors the adoption of domain-specific languages and methods: in that manner in fact the needed attribute slots can be more conveniently tailored in the best accord with the specific needs of the domain of interest.

While a component model may not prescribe the use of a specific computational model, the semantics implicitly or explicitly entailed by the component model should be compatible with the semantics of the computational model that the developer wishes to adopt. Due to the potentially overlapping concerns of the two notions, the component model should ideally be neutral with respect to the computational model and should facilitate a later adoption of a computational model without semantic distortion.

We can therefore see a clear link between the computational model and the component model. The computational model is the foundation on which the system can be analyzed for execution behavior, and, for what concerns its dimensions of pertinence, it dictates what the implementation can and cannot do: this prescription is necessary to have an implementation that exhibits the semantics specified in the computational model and assumed in the analysis.

The component model does thus adopt a computational model; only if the semantics expressed in the former is compatible with the semantics prescribed by the latter, then the architectural description of the system may be analyzable according to the analysis theories that underpin the computational model, and it is possible to produce system implementations that

exhibit the asserted properties. The best possible condition is when the extra-functional attributes of the component model exactly mirror those of the adopted computational model. In fact in the case the component model features a superset of the properties of the computational model, those properties cannot be used to analyze the system and thus there is no obvious way to provably ensure that they can be preserved across the development process. Conversely, if the component model is able to express a subset of the attributes of the computational model, then the design framework cannot fully benefit from the analysis capabilities and it cannot be engineered to provide for the preservation of those properties.

2.3.5 Model-Driven Engineering

In the last decade, Model-Driven Engineering (MDE) [120] emerged as a new trend for software development. The adoption of MDE promises to considerably reduce the cost of development and time-to-market owing to the increased abstraction of the design and to its automation capabilities, which permit the automated generation of lower-level artifacts, like documentation, analysis models, source code.

MDE best demonstrated its potential in domains like enterprise computing [50], where it has already been successfully applied.

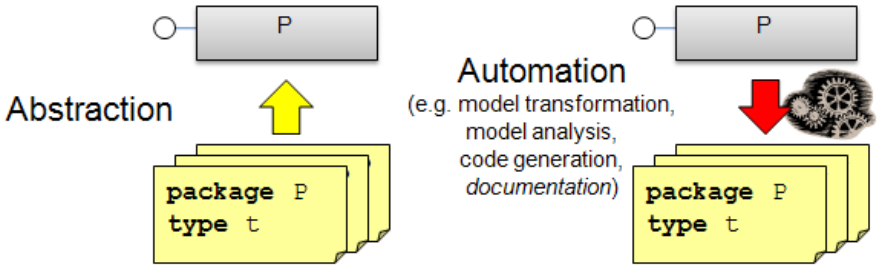

Figure 2.8: The greater abstraction level and the automation capabilities provided by a Model-Driven Engineering approach.

The use of MDE has been advocated also for high-integrity real-time systems [121, 65], however its adoption is less obvious than in other industrial domains. In order for MDE to be adopted, it must be proved to deliver value in the effectiveness of verification and validation and in the coverage of extra-functional needs, which differentiate high-integrity real-time industry from other domains.

Positive experiences on the application of MDE to the design of this class of systems exist, although with only limited coverage of the overall development process.

In MDE, the principal design artifact is a *model*, which is an abstract representation of the system under development which encompasses system and software architecture. Models con-

form with a *metamodel*, which describes the syntax of entities that may populate models, as well as their relationships and the constraints in place between them. The metamodel thus is the means to actively constrain the design space of the MDE infrastructure. MDE uses at least two kinds of model: *Platform Independent Models* (PIM), which are free from platform- or language-dependent aspects and *Platform Specific Models* (PSM), which embed those aspects and are necessary for implementation. A *model-to-model transformation* can combine the description of the target platform and deployment information to generate a PSM from a PIM (the user model). A platform-specific model is a representation of the system implementation.

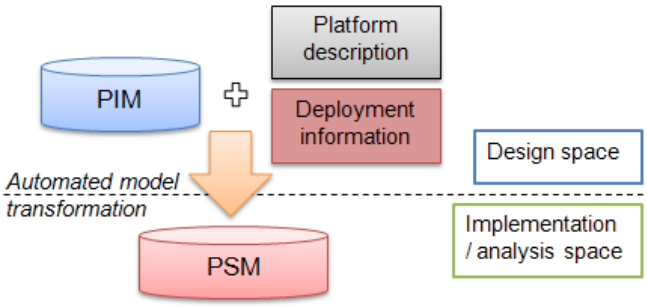

Figure 2.9: Transformation from Platform Independent Model to Platform Specific Model

In fact, a PSM produced with a rigorous design methodology is a most faithful description of the system and it is subject to various forms of analysis such as schedulability analysis, information flow analysis, safety analysis, dependability analysis etc.. These forms of model-based analysis can apply to very early stage of development, well ahead to system implementation. MDE therefore leverages early analysis results to tradeoff design decisions on successive increments of the system. On the contrary, traditional programmatic approaches can only use verification and validation results to confirm or negate the fulfillment of system requirements; and those results are produced too late, when the system has little or no freedom left in time, cost and implementation.

In an MDE approach, the source code for the implementation is nothing more than another design product, and can be automatically generated with a model-to-code transformation from a validated PSM.

2.4 State of the art

In this section we introduce a selection of state-of-the art approaches that are similar to the approach we propose; we outline their overall approach and main characteristics.

BIP. In the last decade there has been fervent activity at Verimag for the design of a component-based framework for the development of heterogeneous systems. Early work [59] recognized the difficulty int reconciling components assuming different kinds of *interactions* and *execution paradigms*.

Interactions can be either: (i) *atomic*, if the resulting behaviour of components that initiate an interaction cannot be altered by other interactions; or (ii) *non atomic*, in the form for example of a method call or the sending of a message, in which components that do not participate to the interaction of interest can interfere during its execution. Process algebrae [8, 4] and synchronous languages [15] assume atomic interactions, while languages like UML [101], SDL [71], Java [131] or Ada [73] assume non-atomic interactions.

Interactions can involve: (i) *strict* synchronization; or (ii) *non-strict* synchronization. Strict interactions can be triggered only if all the participating actions can occur; an example is the atomic rendezvous in CSP [67]. Synchronous languages instead have non-strict interactions since output actions can occur even if they do not match with an input actions. Inputs instead require synchronization with an output in order to be triggered. Interactions are *binary* if they involve only two components or *n-ary* if there are three or more components involved.

There exist two well-known execution paradigms: *synchronous execution* and *asynchronous execution*. Synchronous execution assumes that the system runs in a sequence of *global* steps and that the environment does not change during the execution of a step. Asynchronous execution does not assume any notion of global execution step; this is typically assumed by the UML world as well as programming languages like Java, Ada or multi-threaded execution platforms in general.

The framework of [59] aims at a correct-by-construction design by achieving component *composability*, which guarantees that properties verified on a component in isolation are preserved after integration with other components; and component *compositionality*, which permits to derive the global properties of the system from the local properties of its components.

The work later evolved in the conception of the BIP framework (Behaviour, Interaction, Priority) [13]. In BIP the authors further the separation of concerns present in previous works by creating components as superposition of three layers: (i) a lower behaviour layer; (ii) an intermediate layer that describes with a set of connectors the interactions between transitions of the behaviour; and (iii) an upper layer with a set of priority rules to determine the scheduling policies for interactions. The product of two components is the result of the separate composition of their layers.

The framework provides atomic components, which are the basic building blocks of the system, and permit the creation of compound components, which are obtained by successive composition of their constituents. Notably, the authors describe the operational semantics of BIP, and realized an infrastructure to generate C++ code from BIP systems and an execution platform to run it either using a multi-threaded execution (each atomic component has its own

thread of control) or using a single threaded execution (the execution engine is the only thread).

The two major contributions of BIP are the modeling of heterogeneous systems and the overall approach that reduces the gap between the analyzed system and the implementation. It is worth noticing however, that the framework does not provide any mechanism to enforce properties or monitor and react to run-time violations of properties.

Predictable Assembly from Certifiable Components The *Predictable Assembly from Certifiable Components* (PACC) project [142, 66] by CMU/SEI is based on core principles that are very close to the overall approach we propose. The technology used in the project is named PECT (Prediction-enabled Component Technology). The authors recognize that predictability and analyzability stem from the analysis theory; and therefore the design space shall be constrained so as to create solely components –which for their form and semantics— are amenable to static analysis. The design space shall then be permeated by "analytic constraints" with the goal of making the behaviour of assembly of components predictable; and the *properties* of components shall be unambiguously defined.

The approach supports the definition of hierarchical components and explains how analysis for various extra-functional dimensions can be introduced in the approach (e.g. latency analysis, safety analysis, availability analysis). The approach proposes a distinction between the "abstract component technology" (specified by the approach) and one or more incarnation of the abstract methodology in a concrete component model, a conforming "run-time environment" and a set of analysis theories (called "reasoning framework"). For this reason, PACC focuses on the overall methodology, the abstract definition of the component model and the relationship between the different analysis theories (to reason on how they can be soundly integrated without reciprocal conflicts).

In essence, PACC does neither adopt nor promote a specific component model, it only declares the minimum features and principles that have to be fulfilled by a candidate component model in order for it to deliver tangible benefits.

As an example, in [66] the authors adopt the Construction and Composition Language (CCL) [141] for the development of a substation automation system, targeting Microsoft .NET on a Windows 2000 operating system extended with some limited support for fixed-priority scheduling.

CORBA Component Model and Lightweight CCM One of the most interesting examples of component model is the CORBA Component Model [96] (CCM) promoted by the Object Management Group (OMG). The component model targets enterprise software, yet specialized version of CORBA or restrictions of CCM are amenable also to embedded (real-time) systems.

The specification describes: (i) the syntax and semantics of the component model, which is based on the CORBA Interface Definition Language (IDL); (ii) defines a MOF-compliant

metamodel for IDL and the extension to IDL which comes from the component model; (iii) a *Component Implementation Framework* (CIF) that provides a programming model for the implementation of components and encapsulate the complexity of CORBA services which them transparent to the programmer; (iv) a container programming model.

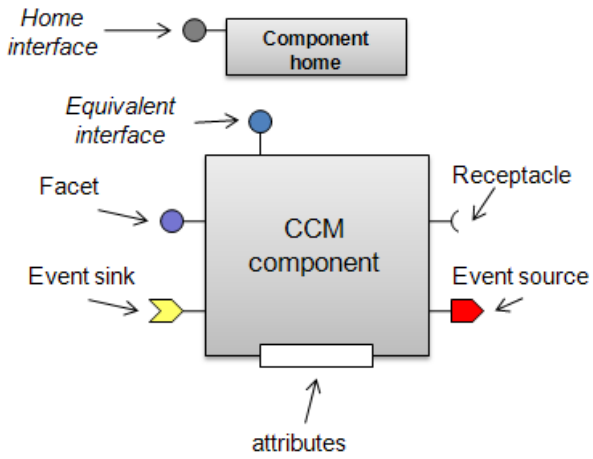

Figure 2.10: A component in CCM

CCM components are first created and then localized through a *home interface*. A CCM component exposes a single *equivalent interface* to clients. The equivalent interface allows a client to navigate among the component's services and to connect to them. Components have: provided ports to expose their services; required ports to bind to other components to fulfill their needs; and a set of attributes, which are typed parameters exposed through *getter* and *setter* operations. These attributes are primarily intended to be used for component configuration.

Provided ports can be: *facets*, for synchronous invocation of component operations; *event sinks*, to receive external events. Required ports can be: *receptacles*, that accept object references on which it is possible to invoke operations of other components; *event sources*, that send events according to a publish-subscribe paradigm. They are further categorized in *emitters*, if the event source can be connected to a single event sink, or *publishers*, if multiple event sinks are notified when events are generated. One clear advantage of the approach is that components can benefit from existing CORBA services [98] and the interoperability of implementations in the CORBA ecosystem.

Lightweight CCM (LwCCM) is a conformance point of CCM. LwCCM is a subset of the full-fledged CCM specification that targets distributed real-time systems. LwCCM removes some of the features of full-fledged CCM, mainly in the areas of persistence management, security and transactions. Nevertheless, LwCCM components are interoperable with CCM components. In fact, they expose regular interfaces still compatible with components compliant

with CCM, but some of their framework services are disabled.

LwCCM uses CORBA as the underlying architecture for distributed object computing. LwCCM is quite interesting since it standardizes the development, packaging, configuration, and deployment of component-based DRE systems.

LwCCM has however some shortcomings when it comes to its application to real-time systems: extra-functional attributes are not considered and their inclusion is thus forcedly a vendor-specific extension; its type system is weak, as it is comparable with that of the C language; and LwCCM components are flat and no provisions for hierarchical composition or decomposition are provided.

The adoption of a component model (and associated framework) that is regulated by a standard is particularly attractive, but unfortunately, the advantage may be dissipated if any domain-neutral or domain-specific customization to the component model is necessary; this in fact may place the component model outside the normative frame of its specification.

Nevertheless the base structure of LwCCM and its design entities are interesting for our application domain, and, as we will see in the next chapter, it was the major inspiration for our component model.

Ada-CCM Ada-CCM [85, 86] is a component-based framework which follows the LwCCM components specification style and programming model. The framework however does not use CORBA as the communication middleware. Communication between remote components is performed via specialized distributed components called *connectors*, which use either the Real-Time Ethernet Protocol (RT-EP) [89] or GLADE [111], an implementation of the Ada Distributed Annex.

The framework generates the code for containers and the support infrastructure in Ada 2005 [73] according to the component interfaces written in an IDL. The business code of components is manually written in Ada by the developer.

The internal structure of components, their external interface and the deployment plans for components comply with the ”Deployment and Configuration of Component-based Distributed Applications Specification” (DEPL) [97] by OMG.

DEPL’s Component Implementation Description was extended with metadata that describe the timing behaviour of components. The component developer must supply those metadata, which are later woven to a description of component interactions, and platform resources so as to automatically generate a real-time model (real-time situation model) of the system that is given as input to the MAST analysis suite [57]. Separate analysis scenarios shall be specified for every distinct operational mode; however the framework seems not to natively support (in containers) mechanisms to perform mode changes.

The target platform is MaRTE-OS [3], an Ada run-time kernel that implements the Minimal Real-Time POSIX subset [70].

SaveComp Component Model SaveComp Component Model (SaveCCM) [63, 2] was created during the SAVE project and targets heavy vehicular systems. The industrial requirements of interest for the domain were gathered and analyzed in [94]. In SaveCCM, a system is composed by a set of interconnected components which interacts through input and output ports. Trigger input ports are control flow entry points; exchange of data is performed through data ports. The model supports both time-triggered and event-triggered activation events and components are hierarchical (using an abstract component called *assembly*). Components are exclusively passive units (hence they do not comprise threads). The interface of the component can be decorated with quality attributes to feed analysis in various extra-functional dimensions and code generation. Unfortunately, only examples of those attributes are provided, without a precise enumeration.

Each component is in an idle state, and becomes active when all its input ports are triggered. The component processes the inputs and writes to the output ports; then it returns to the idle state. SaveCCM also provides an explicit "switch" entity (akin to a component in the graphical representation), which is used to modify the component interconnections to support operational modes. The switch simply facilitates the specification of a pattern of output ports that are triggered according to the validity of a condition on the triggered input ports and their data. No additional computation is allowed by the switch apart from that guarded evaluation.

The design syntax of SaveCCM is based on textual XML. A graphical notation based on a modified subset of UML2 component diagrams is available.

The semantics of the language is formally defined and is defined in [34] a two-step transformation, first to an intermediate language called "SaveCCM core", and then to timed automata with tasks.

A set of automatic model transformations can turn the design model in a representation amenable to various forms of static analysis: timing automata with tasks, to perform Response Time Analysis [77] with the TIMES analysis tool [5]; or finite state process models to perform model checking of properties such as absence of deadlock or perform reachability analysis with the LTSA tool [88]. Unfortunately we were not able to understand if the analysis results are reported back on the original model (as opposed to being separately available only on the tools integrated in the infrastructure) and in which form they are presented to the end user.

The design environment can generate C glue code to interface with the RTXC multitasking RTOS[1] for the CrossFire MX[2] Electronic Control Unit (ECU). No provisions for run-time monitoring of the asserted properties is provided.

PROGRESS Component Model The PROGRESS component model (ProCom) is inspired by the outcome of the SAVE project and by the Rubus component model [62] and targets dis-

[1] http://www.quadros.com
[2] http://www.cc-systems.com

tributed embedded real-time systems for the heavy vehicular, automotive and telecommunication domain.

The authors of SaveCCM recognized that their component model is suitable for small-scale systems defined with a fine-grained granularity; however the high-level concerns typical of the early design stages of a large-scale distributed embedded system were difficult to express. Another prominent need is the provision for high-level early analysis, which, although providing only coarse-grained results, is useful to drive design decisions especially for component deployment to processing units.

Those considerations led to the creation of ProCom. The component model distinguishes two granularity levels for the software specification: (i) a higher level, where the system is modeled as a set of active, concurrent subsystems which communicate through message passing; (ii) and a lower level, for the specification of the internals of the subsystems. The two levels are addressed by two separate languages ProSys for the former, and ProSAVE for the latter (i.e. an evolution of SaveCCM).

ProSys high-level components are deployed to *virtual nodes*, which are then allocated to the physical architecture; logical and physical deployment is performed using a *separate* deployment model [33]. Virtual nodes are logical units for the specification of budgets with respect to the CPU time and memory consumptions. The approach supports early analysis on those dimensions, to assess the goodness of the overall design before proceeding with the implementation. A hierarchical scheduling model is adopted to provide real-time guarantees on the CPU budget; each virtual node is mapped to a *server*. In [33], no hierarchical scheduling model is specifically indicated; well-known approaches such as [41] or [125] are singled out as possible candidates.

Most probably the authors will adopt a subset of AADL or SysML as design formalism for the deployment model.

ProSAVE components provides trigger and data ports similarly to SaveCCM. However, they also encompass a notion of ”service” that is used to correlate cohesive trigger or data ports at the input or output side in a port group. When a trigger port of a component is activated, then all the data ports of the same group (hence service) are read. Data ports supports exclusively a *push-data* semantics: the transfer of data to an input data port overwrites the previous value (hence, no buffering of data is possible). Components are characterized also by attributes which are specified on a port, a port group, at service level in the case of – for example – the WCET or a restriction of the values produced by a data port; alternatively, attributes are related to the whole component (such as the memory footprint). A complete framework for the management of extra-functional attributes was developed in the context of PROGRESS [122]. It permits to distinguish between attributes obtained with different methods (for example an estimation of the WCET, a WCET obtained with measurement-based analysis or static analysis) or derived from other attributes (for example, the memory footprint of a component, derived from the memory

footprint of its subcomponents). The framework also supports configuration management (i.e. versioning) of attributes.

ProCom Components are directly connected with a *connection* between their output and input ports. Alternatively, the component model offers explicit *connectors* related to data flow (such as ”data muxer”, a ”data or”, a ”data demuxer”) or control flow (such as fork and join). Clocks have to be explicitly specified in the design to provide the sources for periodic triggers.

ROBOCOP The ROBOCOP component model (Robust Open Component Based Software Architecture) [95], targets the consumer electronic domain and is inspired by CCM [96] and Koala [135].

The authors started their endeavour by considering the aspects that require particular attention for the target domain: (i) *upgradability*, to extend the life time of devices by uploading improved version of the software; (ii) *extensibility*, to add functionalities to the device; (iii) *low resource consumption*, particularly in footprint size, due to the limited hardware capabilities of devices. (iv) *support for third-party components*, which influences the strategy for packaging of components.

In ROBOCOP, a component is a collection of related models which are used to trade components between parties. Those models are human-readable models (like documentation) or machine-oriented models (a simulation model, a resource model, an interface model, a security model, etc..). In each of these models it is possible to describe the attributes of the component relatively to the dimension of interest.

The ROBOCOP component is much more similar to a package that is used to share components with various stakeholders and is different from the unit of deployment of the approach (which would be an executable component). What is termed component in the classical CBSE acceptation, it is called *service*.

A component is developed and published in a repository. At this stage, the component is still generic as it is not bound to any target platform. Components are then tailored for execution on a specific target platform, loaded on the device, registered and instantiated for execution.

Services are specified with a ROBOCOP IDL (RIDL), inspired by the CORBA IDL [98], that is used to declare the provided and required ports (typed interfaces) of each service and its public attributes. The interfaces are COM [25] interfaces.

An underlying run-time environment caters for component registration and instantiation. It can be extended to address other optional concerns, such as resource management; those other concerns are left optional to avoid overhead for users that are not requesting them.

ROBOCOP is one of the few component models that provide some kind of domain-specific support for its application domain instead of being classifiable as a component model for embedded real-time systems in general (we clarify our stance in this regard in section 3.7). For example, the *download framework* [95], which provides a complete infrastructure for the down-

load, substitution and loading of new versions of components, and responds to important concerns for the domain of consumer electronics.

ROBOCOP was later extended during the ITEA projects Space4U and Trust4All, with a framework (called DeepCompass [18, 17]) for scenario-based analysis. The framework enables early performance analysis and architecture optimization, by comparing how various hardware and software architectures better fit some evaluation criteria (e.g. responsiveness or reliability).

A scenario model is built at design time by combining the base software design (i.e. the set of existing components) together with a description of the input events of the system. The latter are used to understand the components that are activated and therefore deduce the potential workload scenario for the system. The framework automatically generates an "executable system model" which includes the tasks that are executed during the scenario, their complete call graph, the execution sequence between tasks and the resources that are accessed by them.

ASSERT The ASSERT project [7] aimed at the definition of a MDE design process for the development of on-board software for satellites, centered on the principles of Correctness-by-Construction [36] and separation of concerns [44].

The cornerstone of the project is the adoption of the Ravenscar Computational Model (RCM) [27]. The design space is fixed as to reflect (albeit at a higher abstraction level) the assumptions and constraints of the RCM; since the design space is thus constrained to solely express the semantics allowed by the computational model and actively forbids the creation of entities outside that perimeter, the designed systems are amenable to static analysis by construction.

ASSERT demonstrated the feasibility and effectiveness of the combined use of a computational model, a programming model and a conforming execution platform to achieve that goal. The adopted programming model was the Ada Ravenscar Profile [26] combined with a set of appropriate code archetypes [113]. The target platform was GNATforLEON [134], an Ada 2005 cross-compilation system for the LEON 2 [53], a 32-bit radiation-hardened processor based on the SPARC V8 architecture.

ASSERT lacked an explicit notion of component model. The project investigated two different solutions for the specification of design entities: one based on AADL [119], the other on a domain-specific language called HRT-UML/RCM [24], loosely inspired by UML.

The author worked in the HRT-UML/RCM track prior to the launch of his PhD activity. In that line of work he defined a set of analysis equations [107] specialized for the target platform in use (the priority band architecture [114]), and their implementation in an extended version of the MAST analysis suite, nicknamed MAST+. Finally, he designed and developed the support for model-based schedulability analysis [106, 21], which performs schedulability analysis directly on the architectural design of the system. The necessary information is extracted from the user model and used to feed MAST+. The results of the analysis are then seamlessly propagated back to the user model and attributed to the correct design entities in the form of read-only

attributes. The feasibility and goodness of the approach were sanctioned by an industrial evaluation performed by a software prime contractor, which reimplemented a sizeable subset of one of its satellite subsystems.

The concepts investigated in ASSERT and the feedback received from industry constituted the basis of our present investigation. The core of our PhD focused on the definition and realization of the component model to complete the constituents necessary for the software reference architecture.

Chapter 3

A component model for on-board space applications

In this chapter we elaborate on the component model we propose to complete the 4-constituent approach described in section 2.3.

In section 3.1 we highlight the two principles on which we base our work: *separation of concerns* and *correctness by construction*. In section 3.2 we provide a description of the software entities that compose our approach: the *component*, the *container*, the *connector*. We clarify therein our specific interpretation of these classic concepts of component-based software engineering and we elaborate on their role in the approach, the concerns they address and how they relate to the underlying execution platform. In section 3.3 we elaborate on the design views as the means to enforce separation of concerns in the design space. In section 3.4 we discuss on the general form assumed by our CBSE approach and the action to be performed for the creation or the reuse of components. In section 3.5 we elaborate on the design entities and the steps for the creation of a software model in our approach. In section 3.6 we comment on the design flow that is promoted by our approach, how we allocate the design steps to the various design views and the precedence constraints introduced between those steps. In section 3.7 we clarify what we consider the differences between a domain-neutral and a domain-specific approach, and we discuss how we can include domain-specific concerns (i.e. specific to the space domain and not to high-integrity real-time systems in general) in our approach, without breaking its fundamental assumptions and nullifying its advantages. We discuss what are the repercussion at design, methodological and technological level of this aspect of the problem. Section 3.8 discusses our initial incarnation of the approach, centered on the creation of a domain-specific language (DSL) specified through a domain-specific metamodel and on a design editor on top of it.

Section 3.9 reports on our work for the CHESS project, where we are realizing the same

component model using an extension of the UML MARTE profile as the design language. We compare the two approaches and highlight the advantages and disadvantages of the two solutions.

Our preliminary evaluation of the work discussed in this chapter is presented in chapter 4.

3.1 Founding principles

The component-oriented approach we propose is centered on two important principles: Correctness by Construction and separation of concerns.

The first reflections on the concept of correctness by construction can be traced back to Dijkstra. During his ACM Turing lecture in 1972 [43], he was reasoning on the techniques then commonly used to ascertain the correctness of a program, that is first to make the program and then to test it (which sadly, still is the most common approach also 39 years later). He argued that "*program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence*". He was advocating a constructive approach to program correctness: after arriving at the structure of a convincing proof of correctness for a program, a programmer could develop a program fulfilling the proof's requirement. The important difference of the approach w.r.t. formal verification is that the proof is not constructed after the creation of the program, but the development of the proof and program proceed "hand-in-hand". The reasoning can be extended to think that a program can be the result of successive mathematical transformations, starting from an initial specification and producing an executable program.

In the 90's Praxis High-Integrity Systems (now Altran Praxis) devised a convincing process and technological incarnation of those ideas. Praxis' Correctness by Construction (C-by-C) [61, 36] is in fact a software production method that fosters the early detection and removal of development errors to build safer, cheaper and more reliable software. It is centered on the use of the SPARK language [12], based on a subset of the Ada 95 programming language.

Among the practices engaged in the realization of these goals in Praxis' manifesto, we single out:

1. the use of formal and precise tools and notations for any product of the development cycle, whether document or code;

2. the use of tools to verify the product of each development step: as formal tools are used to develop every single artifact, it is possible to constructively reason on their correctness;

3. the conscious effort to say things only once so as to avoid contradictions and repetitions;

4. the conscious effort to design software that is easy to verify, by e.g., using safer language subsets or explicit coding and design patterns.

From this description it is clear that C-by-C is not applied to individual development stages (requirement analysis, design, implementation, deployment, configuration and their corresponding V&V activities, etc...) but permeates the whole development process.

When errors or wrong decisions propagate from one development stage to the other, the later they are recognized, the higher the cost to remedy them, as typically the designer is forced to perform costly backtracks. C-by-C recognizes this situation, and strives to reduce as much as possible the errors locally.

At their outset these activities reflected a source-centric development approach. In our work, we attempt to cast them to a component-oriented approach based on Model-Driven Engineering. We need to adapt those practices to apply them to: (i) the design of components, hence the organization and provisions of the MDE design space and the design language availed to the designer; (ii) the provision for verification and analysis capabilities of the design environment to sanction the well-formedness and goodness of fit of the design products; (iii) the production of lower-level artifacts from the design model; in our vision the production shall be as automated as possible (where the ideal situation is the complete automation of every implementation and documentation activity proceeding from the design model).

Two alternate solutions are possible when setting the MDE design space:

- to allow the user the largest freedom of expression in the software specification and modeling and then verifying *a-posteriori* the correctness of the model against some correctness criteria;

- to restrict the expressive power of the user up front by propagating to the design space the applicable constraints of the approach and enforcing them actively so that the resulting model is correct by construction against the domain-neutral and domain-specific criteria considered in the overall approach.

In the former approach, the design space is intentionally void of any semantics or constraint specific to the adopted development approach and the user's meaning can be expressed with full freedom; the aspects and constraints that matter for the adopted approach are introduced at a later point via criteria that inform model verification. A model specified in this manner may of course fail some *a-posteriori* checks and thus incur the need for possibly escalating modifications, whose repercussions are difficult to gauge in effort, time and cost.

In the latter approach instead, the design environment supports the desired semantics and constraints directly in the user's design space. In that manner we have a-priori guarantees that the user model is correct against those constraints; this also means that it can safely be used as

input for the subsequent MDE activities (for example, to feed model transformations to create one or many platform-specific models, which represent aspects of the implementation).

Separation of concerns, a long-known but often neglected practice first advocated by Dijkstra in [44], is used to cleanly separate different aspects of software design. As a major advantage, it enables separate reasoning and focused specification.

In our approach, we strive to attain as much as possible separation between functional and extra-functional concerns. In our vision, the separation of the functional part from extra-functional concerns permits to reuse the sequential code of the functional part independently of the realization of the extra-functional concerns that applied when the component was created; this may considerably increase the reuse potential of the software (i.e. the opportunity of its possible reuse under different functional and extra-functional requirements).

Separation of concerns applies to various aspects of software development. In the following sections we discuss how this practice is reflected: (i) in the software entities of the approach, that have different roles according to the distinct concerns we allocate on them; (ii) in the adoption of design views, which are the means to enforce separation of concerns at design level; (iii) and later, by commenting on the structure of the source code that we generate for the implementation entities of the approach.

3.2 Software entities

Our component oriented approach features three distinct software entities: the *component*, the *container* and the *connector*. The component is the only entity that appears at design level. Containers and connectors pertain to the implementation level.

The role of those entities is to address bounded and separate concerns: the component addresses exclusively functional and algorithmic concerns; the connector is used to address interactions concerns; the container is responsible for the realization of the extra-functional concerns, in particular regarding concurrency (tasking and synchronization), real-time and reconfiguration aspects.

We consider the container and the connector as the entities where extra-functional concerns shall be addressed; for this reason we contend that they are the entities where to realize all other extra-functional concerns that we do not address at the moment; for example dependability concerns.

In the following we describe these entities in detail.

Component. Probably, the most famous definition of "component" is by Szypersky [130]: "*A software component is a unit of composition with contractually specified interfaces and explicit*

context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

A more general definition is reported in [37]: "*A software component is a software building block that conforms to a component model. A Component Model defines standards for (i) properties that individual components must satisfy and (ii) methods, and possibly mechanisms, for composing components.*"

A component is the unit of composition of our approach. The whole software is built by creating assembly of components, deployed on an execution platform which takes care of their correct execution. As it was noted also by other authors, the "independent deployment of components" may not be true for many component-oriented approaches, and in fact, at the moment is not a core requirement of our approach.

A component provides a set of functional services and exposes them through an interface (called "provided interface"). For this reason the component is also an encapsulation unit, as the only way to interact with it is through the services of the provided interface. The component may need some services from other components or the environment in general in order to successfully provide its own services. All those required services are gathered in the required interface of the component. For this reason, the component is assembled with other components in order that its transitive closure can completely satisfy its required interface [31]. The component, in essence, stipulates a sort of contractual agreement by which, if all its functional needs (required services) as well as other possible assumptions on the environment or execution platform are satisfied, then it engages to provide its functional services.

As a distinguishing feature of the approach we advocate, components are pure functional units, that is they only contain functional code that specifies the sequential behaviour of the component. All extra-functional concerns are excluded from the component and are dealt by the other two software entities, the container and the connector, or, via them, by the execution platform.

Nevertheless, extra-functional properties are essential for the specification of the system. The central characteristics of our approach are that extra-functional concerns are stipulated exclusively by *declarative specification* in the design space through appropriate decoration of components and component interfaces; and the realization of extra-functional concerns proceeds by automated generation of the code for containers and connectors that implements them, after the overall feasibility in the extra-functional dimensions of interest has been sanctioned by (model-based) analysis. This part of the approach ensures the consistency between the specification of extra-functional concerns and their realization in the implementation.

Container. The container (see Fig. 3.1) is a software entity that can be regarded as a wrapper around the component and is responsible for the realization of all extra-functional properties that are specified for the component that it embeds.

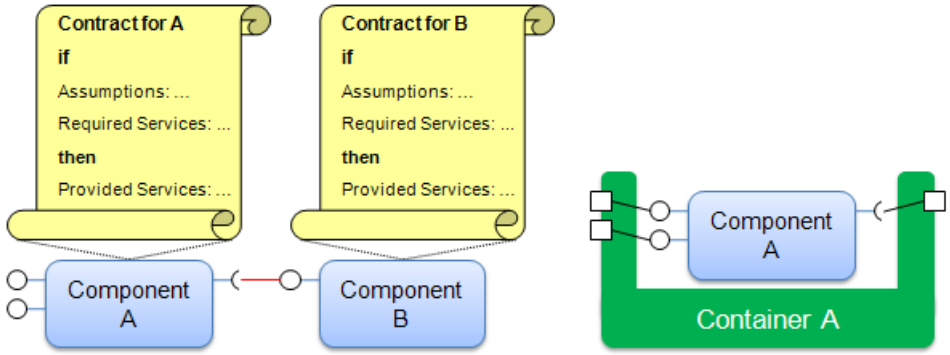

Figure 3.1: **On the left side**: Two components with their contractual interfaces and a component binding between the interfaces. **On the right side**: a container and its embedded component. The interface ”promotion” is shown.

The container exposes the same provided and required interfaces of the enclosed component, through an interface ”promotion” that creates delegation and subsumption relationships from the operations of the component to the equivalent operations on the container. In the general case, there is never direct communication to a component, since the communication is mediated by the enclosing container. Furthermore the container mediates the access of the component to executive services provided by the execution platform.

Connector. The connector [90] is the software entity responsible for the interaction between components, which actually is a mediated communication between containers. The role of the connector is to decouple interaction concerns from functional concerns. This means in practice that components are void of code that deals with interactions with other components.

A connector decouples the component from the other endpoint(s) of the communication. In this way, the functional code of a component can be specified independently of: (1) the component(s) it will be later bound to; (2) the cardinality of the communication; and (3) the location of the other parties. This is necessary as components are designed in isolation and their binding with other components is a later concern, or may vary in different reuse contexts.

The static nature of the target systems of the space domain reduces the variety of necessary connectors to a few basic kinds, which are required to perform function/procedure calls, remote message passing or data access (I/O operations on a file in the safeguard memory). This also means that we do not require an approach for the creation or composition of complex connectors [129]. More complex connector kinds may be necessary when certain guarantees on a remote communication are required (for example an ”exactly-once” semantics for remote communication) and to convert the underlying representation of data (for example from little endian to big endian). Another connector similar to the *Interface Adapter* in [68] is under consideration to

facilitate the binding between independently developed components; however, the use of that connector would be restricted to exclusively perform a match between the names of functional services (thus disallowing any mismatch in parameter types like in [68]).

Figure 3.2 depicts a connector that regulates the interaction between two containers. The figure also shows that there can never be direct connection of a component with the execution platform, as the connection is always mediated by the container.

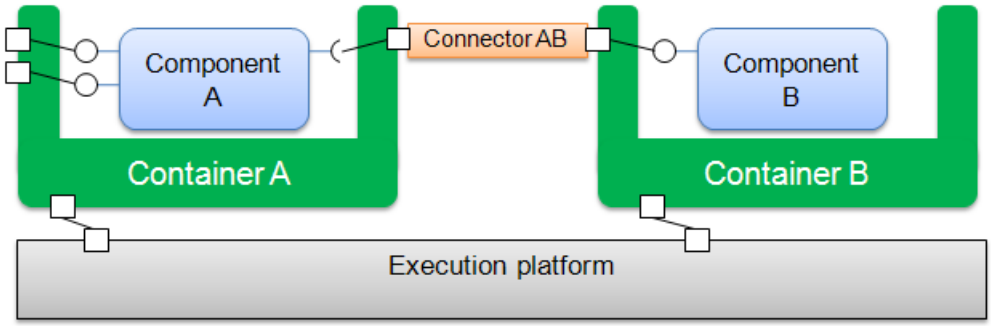

Figure 3.2: A connector that realizes the communication between two containers. The figure shows also the underlying execution platform.

3.2.1 Execution platform

With the generic term ”execution platform” we identify the middleware, the real-time operating system/kernel (RTOS / RTK), communication drivers and the board support package (BSP) for a given hardware platform. At this stage of the investigation for the software reference architecture, there is no attempt to investigate the architectural decomposition of the execution platform (hence we adopt an approach opposite to that of the AUTOSAR initiative [64, 49] of the automotive domain). For now, we consider the execution platform as a single monolithic block, and we just try to understand the different kind of services it is expected to provide to our design and implementation entities.

The concerns addressed by a given executive service determine the software entity allowed to use it. In fact we can classify those services according to their client entity:

- *services for containers*. They are used by the containers to enforce or monitor extra-functional properties. For example: tasking primitives, synchronization primitives, time-related primitives, timers.

- *services for connectors*. They are the implementation of communication means, constructs to address transparently physical distribution across processing units, libraries for translation of data encoding.

- *services for components*. They are (infrastructural) services intended for the functional service of components; typifying examples of such services include: access to the satellite on-board time for time-stamping data; management of context and recovery data. To use those services components do not access the execution platform directly: access is mediated by the corresponding container. This class of services must be represented at component level (as a sort of "special components") in order for user components to be able to use them as if they were regular components.

Note that the implementation of containers and connectors depends on the execution platform. In contrast to components, which only include sequential code and thus are independent from the execution platform, it is necessary to create specific implementation of containers and connectors for each execution platform of interest.

3.3 Design views

In this section we elaborate on the concept of *design view*, which is fundamental for the successful application of our approach. The section describes the concept of design view and the aspects it shall address. In section 3.8 instead, we report on which of those aspects we were able to implement in our solution and the technological problems and shortcomings we faced.

3.3.1 Software architecture and architectural description

In section 1.1 we described what a software architecture is and what are the concerns it addresses. There is an important distinction that we have to highlight at this point: the difference between the software architecture per se and its architectural description.

Borrowing again from ISO 42010/IEEE 1471:

"Every system has an architecture [...]. An architecture can be recorded by an architectural description [...]. This recommended practice distinguishes the architecture of a system, which is conceptual, from particular descriptions of that architecture, which are concrete products or artifacts."

Figure 3.3 describes these relationships. The various stakeholders involved in the procurement, definition or utilization of the system bring in the problem a set of concerns. The architectural description is the description of the architecture and is composed of a set of *views*, each of them being *"A representation of a whole system from the perspective of a related set of concerns"*.

Each view is the expression of a specific viewpoint, that is *"a specification of the conventions for constructing and using a view. A pattern or template from which to develop individual*

views by establishing the purposes and audience for a view and the techniques for its creation and analysis".

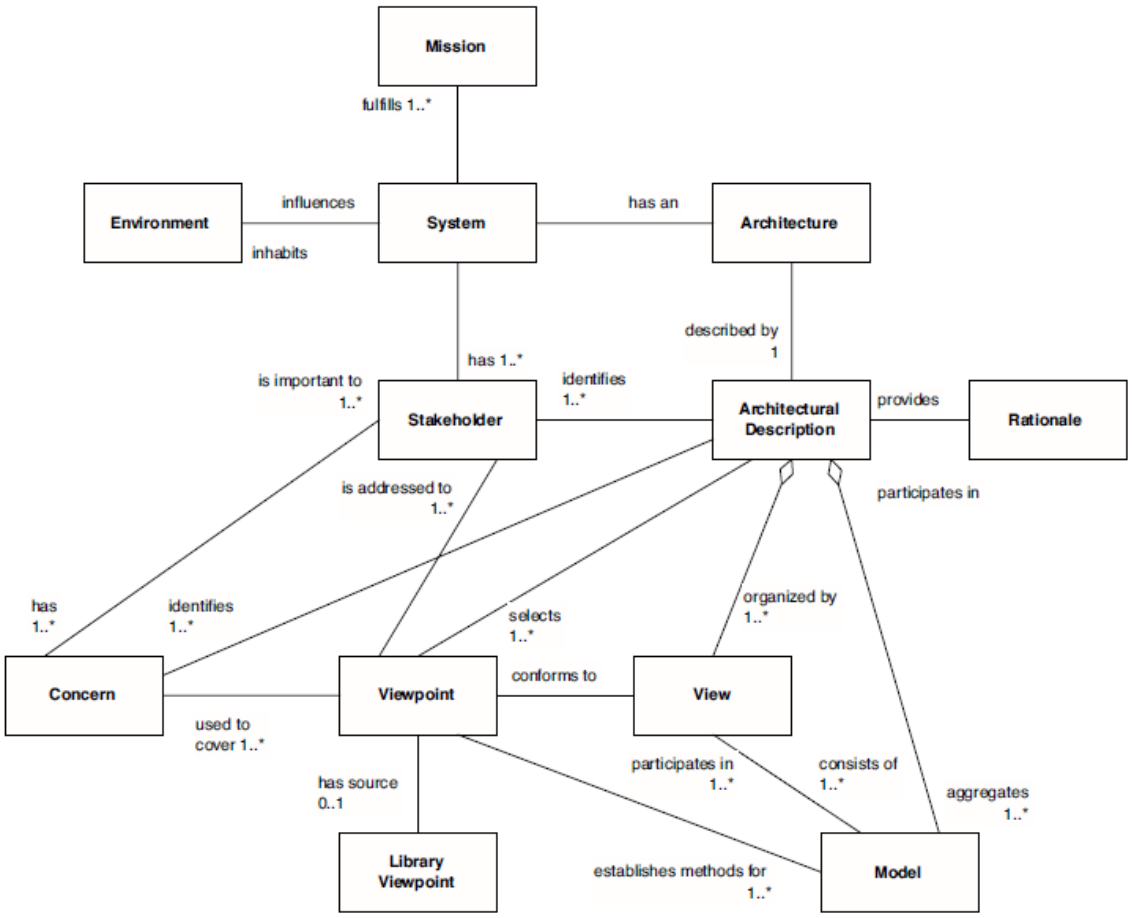

Figure 3.3: The conceptual model of an architecture and its architectural description as in the ISO 42010/IEEE 1471 standard. *Credits: IEEE*

We use these concepts as a base to our investigation. However, they have to be tailored to our context, which is a component-oriented approach a component-oriented approach supported by a design environment centered on Model-Driven Engineering (MDE) [120]. In the following we elaborate on how we specialize the ISO 42010/IEEE 1471 conceptual model to fit MDE in general and all our needs in particular.

3.3.2 Design views to enforce separation of concerns and C-by-C

As we noted in section 3.3.1, the ISO 42010 [76] standard stipulates that the *"architectural description of the system is organized into one or more constituents called views"*, and a view is a partial representation of a system from a particular viewpoint, which is the expression of some stakeholders' concerns.

If during the construction of a development approach we ratify that each view is the expression of a *single* concern, then views become effective and powerful means to enforce separation of concerns in the specification of the system. We will also see that design views can be used to apply C-by-C practices during the design stage.

In our approach we strive to adhere to ISO 42010 to the maximum possible extent. We further require that the user should operate exclusively at the level of the platform-independent model (PIM): complementary information on the target environment shall enable automated generation of the platform specific models (PSM) for analysis or implementation purposes via model transformation.

To make a view-based development effective, it is paramount that all the supported PIM views be consistently related to one another for all cross-cutting concerns. In general, two alternate strategies exist to meet this goal: 1) a *synthetic approach*, in which each view is modeled separately with one or more models which are later composed with the models resulting from the other views; 2) a *projective approach* in which the information that pertains to individual views is extracted from a single underlying model that describes the entire system.

We favour option 2), as we consider inter-view consistency easier to attain than inter-model consistency.

3.3.3 Engineering design views

Our approach addresses software development. Our interpretation of views is therefore narrowed down to the concerns and stakeholders of that context. In our vision in fact, design views shall fulfill two main goals: (i) the *presentation* of a system according to the viewpoint of given stakeholder; and (ii) the *provisions for the development* of the system availed to a given development actor.

The presentation of the system corresponds to the visualization of the current state of (a part of) the system under development. The visualization is typically in a graphical or tabular form that is able to present the user with all the information of interest. A design view selectively shows or hides entities of the model or restricts the visibility of the user on a given entity by granting visibility exclusively on a subset of its attributes.

For example, in Fig. 3.4-1 we depict two entities, *A* and *B*, which belong to distinct design views (α and β). In Fig. 3.4-2 we depict an entity *C* with three attributes (*m*, *n*, *o*). Attribute *m* is visible from view γ, while *n* and *o* are only shown in view δ.

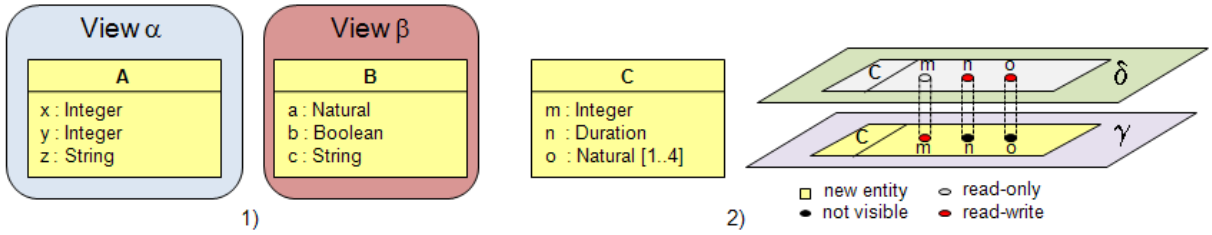

Figure 3.4: **1)** Entities A and B are visualized by distinct design views: α for A and β for B. **2)** Entity C is partially represented in view γ and fully represented (including attributes *n* and *o*) in view δ

The provisions for development instead provide the user with the means to create or modify the entities that appear in a design view. Returning back to Fig. 3.4-2, entity *C* can be created only in view γ; *C* is visualized in view δ, where it is a read-only entity (it cannot be modified or deleted). Attributes *n* and *o* are not visualized in view γ; they are visible and modifiable only in view δ.

It is evident that the two goals are just two aspects of the same problem, that is the allocation of visualization, creation and modification rights on design entities and their attributes to the various design views.

Those permissions can be notionally expressed for example as a table that enumerates the visualization and modification rights of a view with respect to design entities and their attributes. Table 3.1 is an example of such an implementation approach.

View α		**View β**	
Entity / Attribute	**Permission**	**Entity / Attribute**	**Permission**
A	*ro*	C	*ro*
B	*rw*	C.a	*ro*
C	*rw*	C.b	*rw*
C.a	*rw*	C.c	*rw*

Table 3.1: Example of permission table for design view α and β.

Design entities are represented as capital letters; an attribute *x* that belongs to an entity *X* is specified with the X.x notation.

A design entity may be simply visualized in a given view: in that case, no rights are given to the user of the view to modify the entity; we would denote that case with a *read-only* permission on the entity. Conversely, a design entity may be created or modified in a design view: in that case we would specify a *read-write* permission on that entity. From our current understanding,

we deem that it is not necessary to provide separate permissions for the creation and modification of an entity: however if we have to reconsider in the future our current stance, fortunately the modification would be trivial. Finally, if an entity need not be visualized in a given view, no table entry must be provided for that entity under that view.

The reasoning for attributes is very similar: for any given attribute, *read-only* permission allows its visualization in the context of the including design entity in the view of interest; *read-write* permission allows the user to set or modify the attribute; the absence of an entry for an attribute indicates that the attribute is neither visualized nor accessible in the view.

If an entity appears in an entry table for a design view without any other entry for at least one of its attributes, we assume that all the attributes of the entity default to that permission.

In keeping with the principle of separation of concerns, we contend that, in the general case, views shall not incur overlaps of responsibility for creation or modification of new modeling entities. While multiple views can have read rights over cross-cutting aspects, only a single view can have create/write rights on them. Checking that this is true when establishing the views for our approach is easy: we just need to inspect the permission table and assert that only a single view has *read-write* permissions on given modeling entity. This prescription implements C-by-C practice 3 (cf. section 3.1).

3.3.4 View-diagram relation

Diagrams are means to represent the design entities according to a graphical or tabular notation. The relationship between design views and diagrams is interesting.

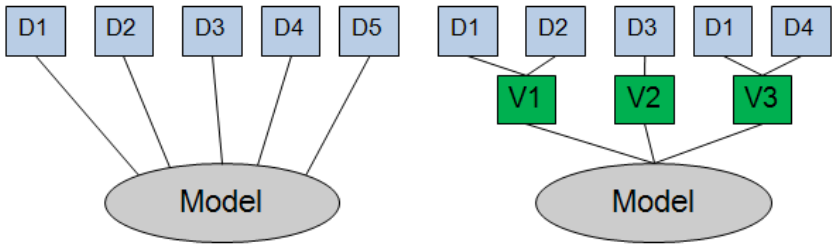

Figure 3.5: **a)** Classical relation between a model and the various diagrams; **b)** A diagram is expression of a design view, which focuses on a specific design concern.

On the left-hand side of figure 3.5, we depict the classical relationship between diagrams and the underlying model (as it is for example in a UML tool). On the right-hand side instead, we depict the relationship with the introduction of design views.

Diagrams are the graphical representation of (part of) a design view. All the entities that appear in a design view are depicted using one or more diagram *kinds* (for example a Class

diagram, a Composite Structure diagram, a table, etc...). The same diagram *kind* can be used by multiple views, even though the entities that will be depicted in the diagram shall conform with the permission rules defined for the view.

As highlighted in [78], we have to specialize the original conceptual model of architecture and architectural description of ISO 42010 (cf. figure 3.3), to tailor it for use in an MDE approach.

3.4 Design process

In figure 3.6 we recapitulate what we consider the main activities related to software design in a generic component-oriented approach.

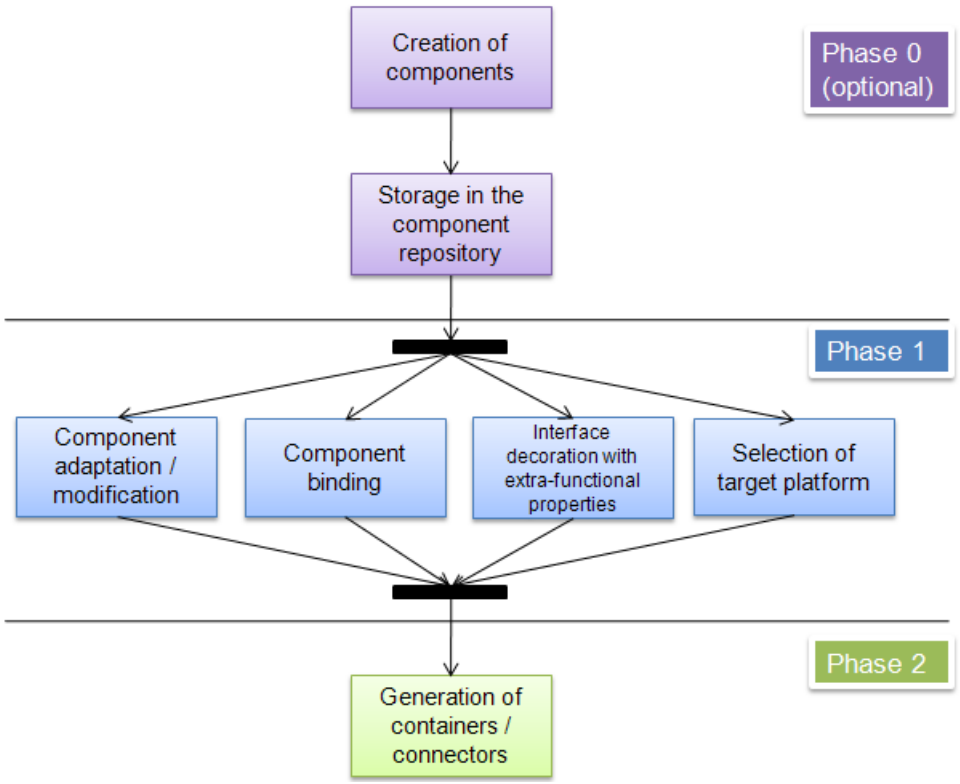

Figure 3.6: A generic component-oriented design process.

The figure distinguishes three conceptual phases. ”Phase 0” is solely centered on the design (and implementation) of *new* components. After the component is complete, it would be stored

in a *component repository*, which contains all the information necessary to restore and configure it in development projects.

”Phase 1” deals with the adaptation or modification of a component, the creation of assembly of components, the declarative specification of extra-functional attributes and the selection of the target platform.

”Phase 2” deals with the automatic generation of the implementation entities (i.e. containers and connectors) which are generated and according to the extra-functional attributes (after ratification of feasibility by analysis) and configured for the target platform.

”Phase 0” is fundamental in the initial period of adoption of the novel approach (as there are no already-developed components). However, since the whole approach aims at reusing components in different projects and contexts, it is possible, or better, desirable, that not all the components are created each time from scratch: if this phase were exercised less and less in subsequent projects, it would be a great recognition of the goodness of the approach as it would imply a conspicuous reuse of already developed components. We do not have however unrealistic expectations on the overall proportion of reused components: a conspicuous amount of them will still be *mission-specific*, hence created exclusively for the ad-hoc needs of a single mission.

3.5 Design entities and design steps

In this section we elaborate on the design and implementation entities that support our CBSE approach: 1) data types and interfaces; 2) component types; 3) component implementations; 4) component instances; 5) component bindings; 6) description of the hardware architecture and the target platforms; 7) containers; and 8) connectors.

Entities 1-6 belong to the design level of the problem and require intellectual contribution from the user; their specification occurs in the design space availed to the user, which is a platform-independent model.

Entities 7-8 belong to the implementation level of the problem. They are neither specified not explicitly represented in the design space and we seek for their automated generation. In fact by adopting a defined computational model and execution platform and deterministic transformation rules, it is possible to automatically generate containers and connectors from the information provided in the design space [24].

In the following we elaborate on the various design steps supported by our component model. As we elaborate later in section 3.6, those steps represent a specialization of the generic CBSE process outlined in figure 3.6.

The generic CBSE process implicitly recognizes a single organization responsible for the whole software development: the *software integrator*. Industrial Need 04 - ”Multi-team de-

velopment and product policy" and Industrial Need 07 - "Role of software suppliers" (cf. section 2.1) explicitly recognize another actor that may be involved in the development process: the *software supplier*. In the following we clarify which steps can be delegated by the software integrator to one or more software suppliers.

#01 - Data types Data types are the basic entities in our approach. The designer can define a set of data types such as primitive types, enumerations, ranged or constrained types, arrays or composite types (like the *struct* in C or the *record type* in Ada).

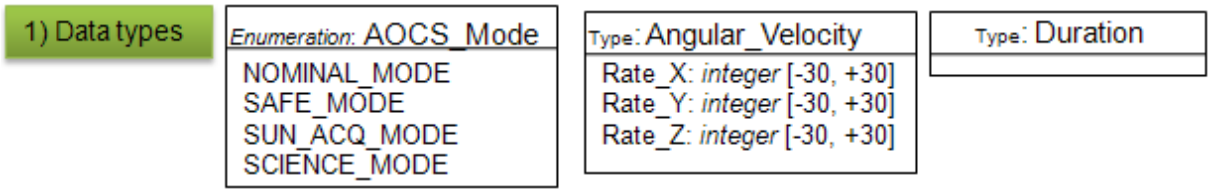

Figure 3.7: Data types.

#02 - Interfaces An interface is a set of one or more operations, with a defined operation signature, determined by an operation name and an ordered set of parameters, each one with a direction (*in*, *in out*, *out*) and a parameter type chosen between the defined types.

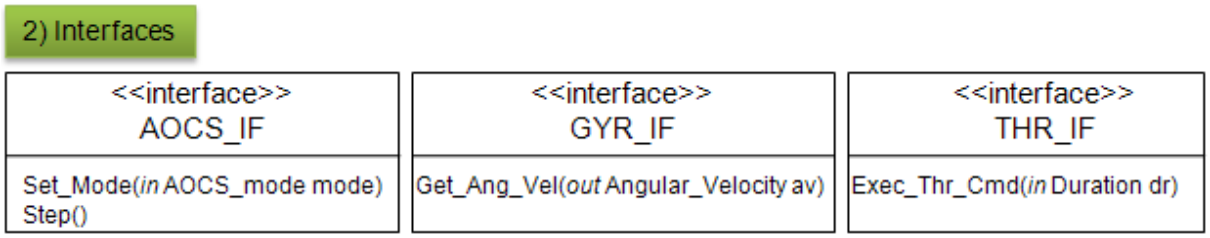

Figure 3.8: Interfaces.

#03 - Component types The component type is the design entity that forms the basis for a reusable software asset. A designer specifies a component type to provide a specification of the functional services of the component. The component type is designed in isolation, with no relationship with other components.

The component type specifies a list of provided interfaces and required interfaces (the services which have to be provided by other components), by referencing already-defined interfaces.

The component type also defines a set of attributes, which are typed parameters with a visibility modifier (*read-only*, *read-write*, *private*).

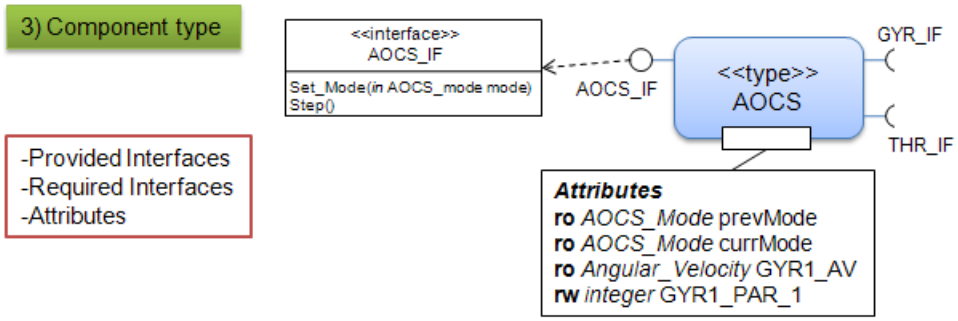

Figure 3.9: Component type.

From the list of attributes and their visibility modifiers we may automatically generate (possibly in a dedicated provided interface) a set of getter and setters operations. In particular: (i) for read-write attributes we generate a getter and a setter operation; (ii) for read-only attributes we generate a getter operation. No operations are generated for private attributes.

In keeping with the pure functional nature of components, no extra-functional attributes are specified on component types.

#04 - Component implementations. The designer proceeds in the design by creating a component implementation from a component type.

A component implementation fulfills two roles in our approach: (i) it is a concrete realization of a component type; and (ii) it is the subcontracting unit of the approach, whose realization can be delegated by the system integrator to a software supplier.

A component type may have several implementations (one more precise yet more computationally expensive, one more robust, etc...). Implementations may also differ for the implementation language.

A component implementation must implement all the functional services of its type, and includes the sequential/algorithmic code of those services and the necessary packaging information. The code is purely sequential code and shall be void of any tasking or timing constructs.

Despite the sequential nature of the code, an implementation may set specific *extra-functional constraints* to preserve the functional correctness of its behaviour. For example, a control law may work correctly only if executed within a range of frequencies, say either 8Hz or 10Hz. In that case, the functional code propagates some implicit extra-functional constraints and we need to explicit represent them in the component interface. and relate with the appropriate operation.

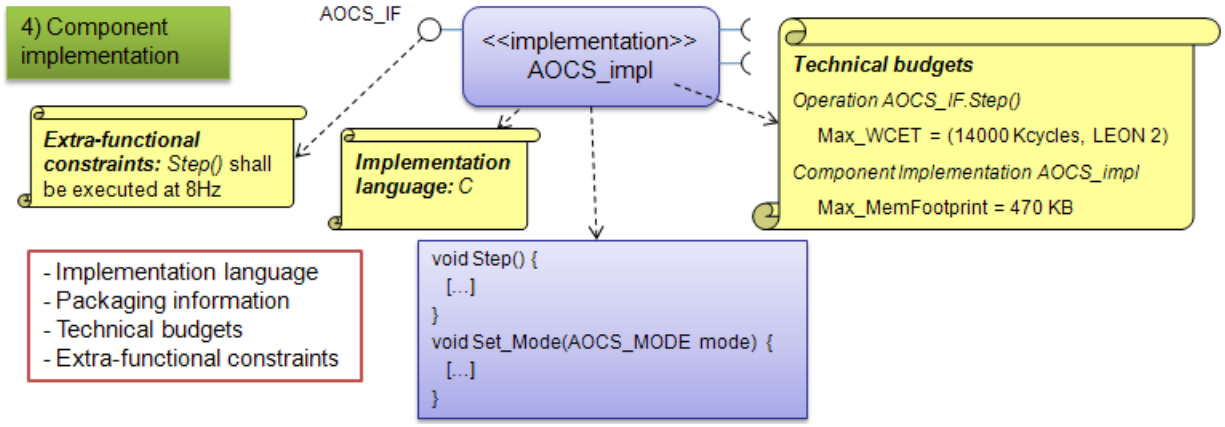

Figure 3.10: Component implementation.

Technical budgets on the execution time or memory footprint can be placed either on operations or on the whole component. The implementation of the component and its operation shall respect the allocated budget.

This is useful especially in the case a system integrator wants to subcontract part of the software. After establishing how the component functionally relates to the system (by establishing its needs at provisions at the component-type level), they can derive a component implementation, establish technical budget on it and then delegate the implementation to a software supplier.

In the latter case the software supplier is requested to address *exclusively* functional concerns (in particular the implementation of the sequential/algorithmic code), possibly under a defined budget envelope negotiated with the software integrator. We can see with the next steps that the specification of any extra-functional concern (*how* the software shall be executed) stays the responsibility of the software integrator and the implementation of extra-functional concerns stays the responsibility of the design environment.

Listing 3.1: Syntax in EBNF for technical budgets.

```
TechnicalBudgetDesc   = TechnicalBudget* ;
TechnicalBudget       = ComponentBudget | OperationBudget ;
ComponentBudget       = FootprintSizeBudget ;
OperationBudget       = WCETbudget ;
FootprintSizeBudget   = ComponentImplementationName, MaxMemFootprint ;
MaxMemFootprint       = "MaxMemFootprint", positiveNumber, unit ;
WCETbudget            = OperationName, MaxWCET ;
MaxWCET               = "Max_WCET", positiveNumber, unit, processorKind ;
```

Listing 3.1 represents the syntax for the kind of technical budget of interest to the stakeholders: the maximum WCET of an operation and the maximum footprint size of a component implementation.

#05 - Component instances and component bindings. A component instance is instantiated from a component implementation. There are no differences between a component implementation and one derived instance of it from the functional point of view.

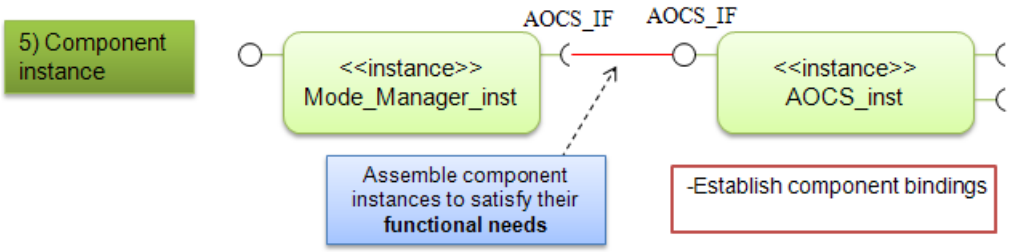

Figure 3.11: Component instance and component bindings.

A component instance serves three purposes: (i) it is the design entity that is subject to composition with other components to fulfill their functional needs; (ii) it is the deployment unit of the approach; (ii) it is the entity that is decorated with *extra-functional attributes*.

Purpose (i) is expression of functional concerns; (ii) is expression of deployment concerns; and (iii) of extra-functional concerns. In this step, we elaborate only on (i).

Component bindings are set by the user at design time between the required interface and a provided interface of component instances. The binding is subject to static type matching to ensure that the providing end fulfills the functional needs of the requiring end.

At this stage, component instances are not yet allocated to the processing units that execute them. An additional step is also required for the specification of extra-functional attributes.

#06 - Decoration with extra-functional attributes. After component instances have been created, the designer can initiate the decoration of the services of their provided interfaces with extra-functional attributes.

At this stage, the designer can specify *timing* and *synchronization* attributes. Firstly, the designer can establish the concurrent kind of the operation, by classifying it as either *immediate* or *deferred*. In the former case, the operation is executed by the flow of control on the side of the caller. In the latter case, the operation is executed by a dedicated flow of control.

An immediate operation is said *protected* if its execution requires to be protected with mutual exclusion from concurrent clients. Otherwise it is *unprotected*.

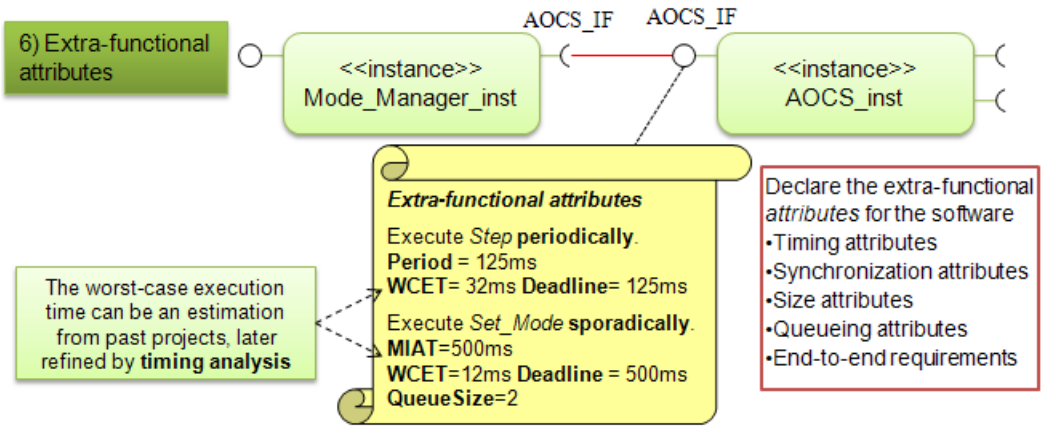

Figure 3.12: Decoration with extra-functional attributes.

It is evident that the choice of protecting the execution of an operation is done only after bindings between component instances are set; in fact it is necessary whenever any immediate (stateful) operation on a PI is requested by more than one client.

For a deferred operation, the designer must choose the release pattern, which can be periodic, sporadic or bursty.

- Periodic operations are executed by a dedicated flow of control released by the execution platform with a fixed period.
- Sporadic operations are executed by a dedicated flow of control which responds to a request posted via software invocation by another component or via hardware/software interrupt. Two subsequent releases of the operation are separated by a minimum timespan called minimum inter-arrival time (MIAT). Sporadic operations require the creation of a finite-size buffer for the incoming requests. The designer is expected to size the buffer by a specific attribute setting.
- Bursty operations, which are used to model dense releases of sporadic jobs possibly followed by long periods of inactivity. There is a maximum number of activations of a bursty operation in a bounded interval; the designer is expected to provide both of them as attributes, and the size of the buffer for incoming requests, similarly to sporadic operations.

For all deferred release patterns, the designer shall also specify the relative deadline for the completion of the operation.

For all operations, the designer at this stage can provide an estimation of their worst-case execution time (WCET), which is typically based on experience from previous projects. The value for the WCET can be then refined later with a value fed by timing analysis [143].

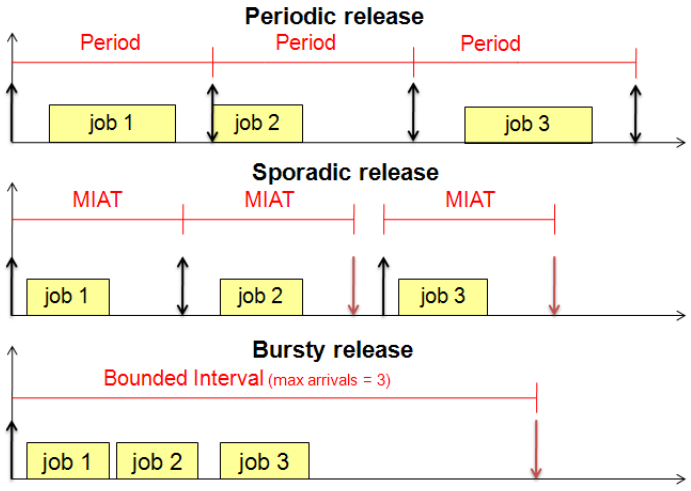

Figure 3.13: The periodic, sporadic and bursty release patterns. In the example, tasks have relative deadlines equal to the period/minimum inter-arrival time/bound interval. Black arrows pointed upward are job releases. Arrows pointed downward are deadlines (black colored if they coincide with a new job release).

If instance bindings have been set, other extra-functional attributes can be specified: (i) queuing attributes, e.g., queuing protocols and queue sizes; (ii) end-to-end timing requirements, e.g., end-to-end deadlines on a call chain across components (see the relevant paragraph).

The component model shall provide the user with means to define measurement units (and conversion factors between them). The definition of those units can then be factorized in a library reused across projects. Extra-functional attributes (listing 3.2) and technical budgets (listing 3.1) that represent a dimensioned attribute (e.g., a timespan, a memory size), require the specification of a value and a measurement unit.

With this approach, component types and implementation preserve their pure functional nature and the extra-functional attributes can be added at a later on component instances as a sort of superimposed layer.

Listing 3.2 represents the syntax for the current set of extra-functional attributes used to decorate interfaces of component instances.

Listing 3.2: Syntax in EBNF for extra-functional attributes attached to operations.

```
EFDescriptor     = operationName, concurrencyKind ;
concurrencyKind  = immediate | deferred ;
deferred         = cyclic | sporadic | bursty ;
immediate        = protected | unprotected ;
cyclic           = "cyclic",    period, WCET, [deadline], [offset] ;
sporadic         = "sporadic", MIAT,    WCET, [deadline], queueSize ;
bursty           = "bursty", boundedInterval, maxActivations, WCET,
                    deadline, queueSize ;
protected        = "protected",    WCET ;
unprotected      = "unprotected", WCET ;
WCET             = "WCET",      naturalNumber,   unit ;
period           = "period",    positiveNumber, unit ;
deadline         = "deadline", naturalNumber,   unit ;
offset           = "offset",    naturalNumber,   unit ;
MIAT             = "MIAT",      naturalNumber,   unit ;
boundedInterval  = "boundedInterval", positiveNumber, unit ;
maxActivations   = "maxActivations",  positiveNumber ;
queueSize        = "queueSize", positiveNumber ;
```

#07 - Hardware architecture and target platforms. The hardware architecture provides a description of the system hardware limited to the aspects related to communication, analysis and code generation purposes.

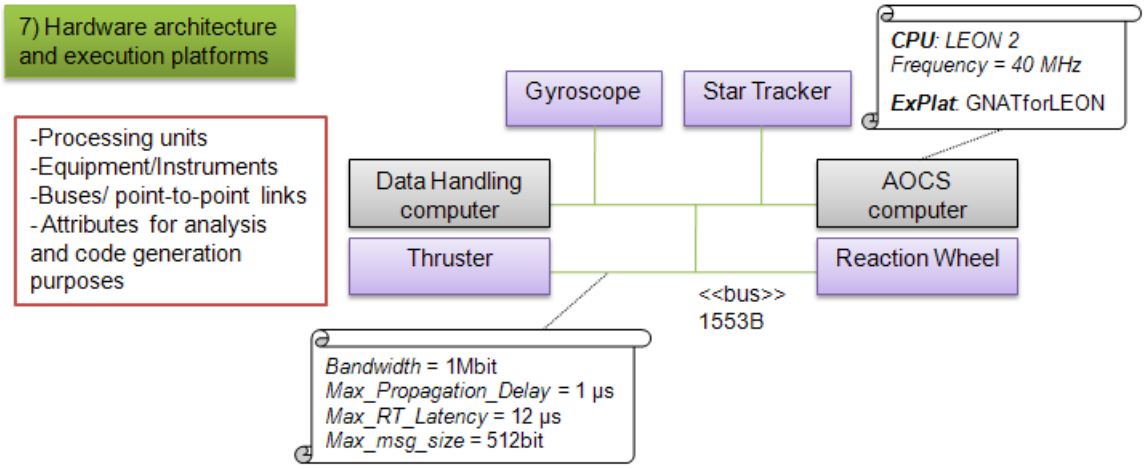

Figure 3.14: Hardware architecture

The following elements are described: (i) *processing units*, i.e., units that have general-purpose processing capability; (ii) *avionics equipment*, i.e., sensors, actuators, storage memo-

ries and remote terminals; (iii) the *interconnections* between the elements above, in terms of buses, point-to-point links, serial lines.

A description of the target platform for each processing units is also necessary.

This description, similarly to the entities of the hardware architecture consists in a set of attributes that are relevant for communication, analysis and code generation purpose.

For the moment we require the following set of attributes for the purpose of analysis:

- for processors: the *processor frequency*, which is used to re-scale WCET values expressed in processor cycles.
- for busses and point-to-point links: the *bandwidth*, the *maximum blocking* incurred by a message due to non-preemptability of lower-priority message transmission, the *minimum and maximum packet size*, the *minimum and maximum propagation delay*, the maximum time necessary for the bus arbiter/driver to prepare and send a message on the physical channel; and the maximum time to make it available to the receiver after reception at the destination end.

#08 - Component instance deployment. Once the hardware architecture has been defined, the last step to perform in the design space is the allocation of component instances to processing units.

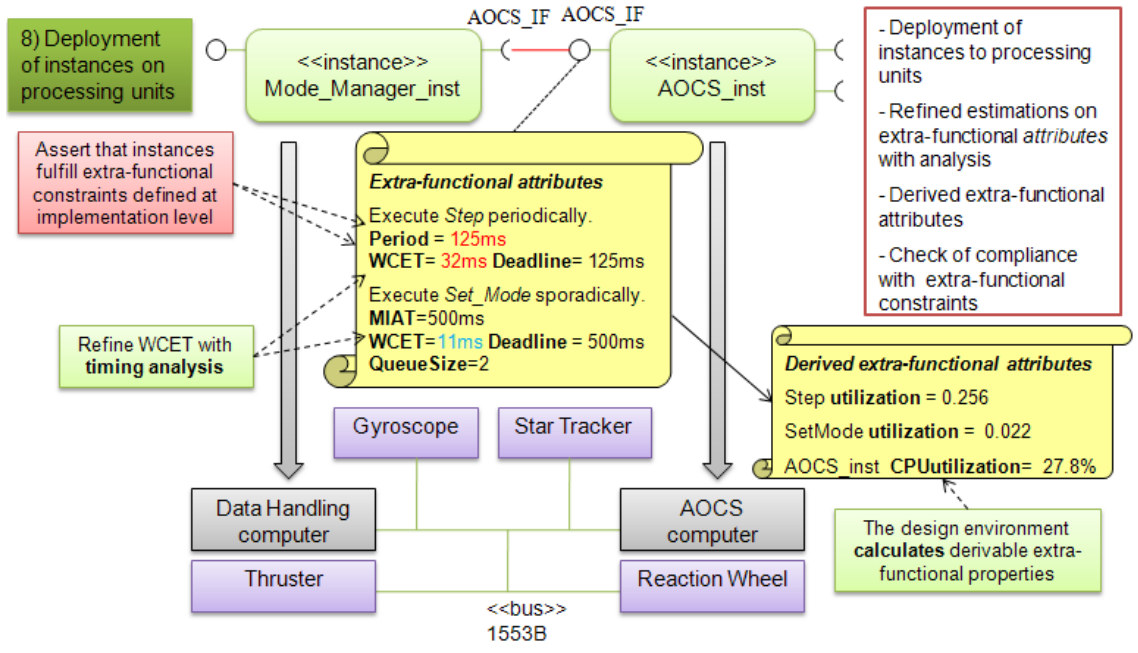

Figure 3.15: Deployment of component instances to processing units.

By knowing the execution platform for component instances it is possible to refine with analysis attributes that were only specified as estimates. One example is the WCET, that can be refined using specialized analysis tools.

In the general case, sizing attributes (footprint of the object code and stack size) depend on the execution platform the component is deployed on (and on the compiler in use). Therefore those attributes are defined and remain valid at instance level rather than implementation level. Alternatively, they can be propagated at component implementation level, by storing the attribute in a reusable descriptor that specifies the value of it on the target platforms of interest.

The design environment can then calculate derived extra-functional attributes, like the processor utilization, or (now that the frequency of the target processor is known) convert the WCET from processor cycles to a time unit (like milliseconds) to feed schedulability analysis.

In any case the design environment checks that the attributes defined at instance level are compatible with the applicable extra-functional constraints and technical budgets defined at implementation level.

In the example of figure 3.12 the design environment ensures that the frequency of operation *Step* matches the corresponding constraint (execution at 8Hz) in the component implementation and that the WCET is within the stipulated bound (cf. figure 3.9).

#09 - Model-based analysis The system model is subject to static analysis in the extra-functional dimensions of interest. For example, schedulability analysis verifies the feasibility of the timing attributes of interfaces with respect to the timing requirements [21].

The extraction of information from the user model (i.e., generation of intermediate models such as the PSM or the Schedulability Analysis Model) and generation of the input for the analysis tools shall be automatic and the results of the analysis shall be seamlessly propagated back to the design model as read-only attributes of the appropriate design entities. Section 3.8.2.3 describes this process in detail.

As the model transformation that generates the model representation for containers and connectors in the PSM (see Step #10) is an integral part of the analysis transformation chain, the transformation can add information about the cost in time and space of the containers and, in particular, of connectors. This is of utmost importance for an accurate analysis of the overhead introduced by the use of middleware services for remote communication, which ought to be included in the analysis model.

If the analysis singles out problems in the design, we want the results to be as informative as possible, so that the designer can use them to informedly change the design. The analysis can be iterated at will, with different iterations resulting from a simple change of attributes, until the system converges to a prediction deemed as satisfactory by the designer.

#10 - Generation of containers and connectors. When a component instance is allocated to a processing unit, we can generate the container within which the component is deployed on top of the execution platform of the processing unit. It is not necessary to set component bindings if we only need to create the container for a component in isolation. If component bindings have been set, then it is possible to also create the connectors for them.

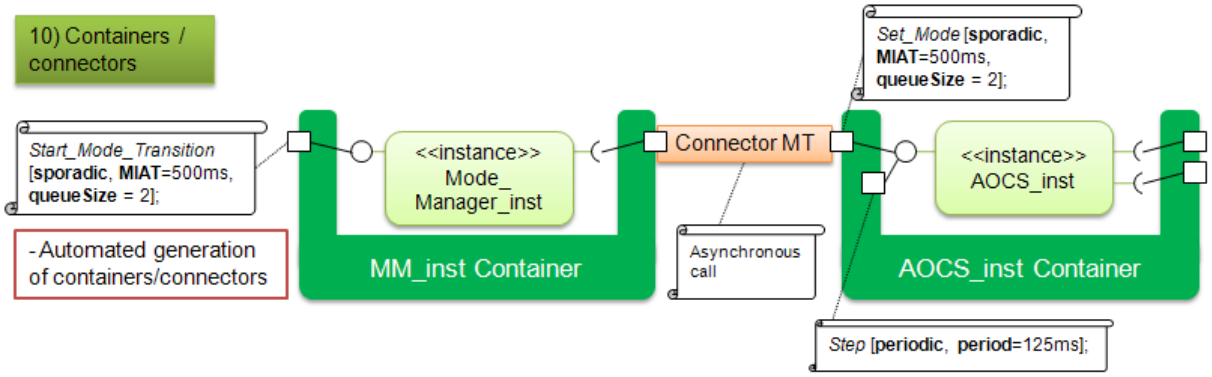

Figure 3.16: Automated generation of containers and connectors.

The internal structure of containers depends on the extra-functional attributes (now elevated to properties to be enforced) required for the components they may embed. In fact, for each allowable computational model, we need a small taxonomy of them. Interestingly, this permits to afford the development of default deterministic rules to automatically generate containers from the attributes set on the model.

The rules shall specify: (i) the structure of each container in terms of the interface exposed to the software system, its internal threads and its protected objects; (ii) the structure of each connector; (iii) how extra-functional attributes and deployment determine the creation of containers and connectors and how component instances and their operations are allocated on them.

As an example, a binding between two component instances deployed on the same processing unit can be resolved with a simple operation call; if one of the instances is deployed on another processing unit and all other attributes stay fixed, the call would necessary be resolved as a call to the middleware services for remote communication.

For every pair (computational model, execution platform), we can factor out the whole set of allowable containers and connectors in a library of code archetypes, which vastly simplifies automatic code generation [23] (see also section 3.8.2.4).

Interaction with the environment. Platform on-board software is embedded in nature and interacts with the environment by sampling information of interest through sensors (gyroscopes, Earth and sun sensors, star trackers, thermisters, ...) and controls the plant (the satellite) where

it is executed by using a set of actuators that take effect on the environment (thrusters, reaction wheels, magnetorquers, heaters, etc..)

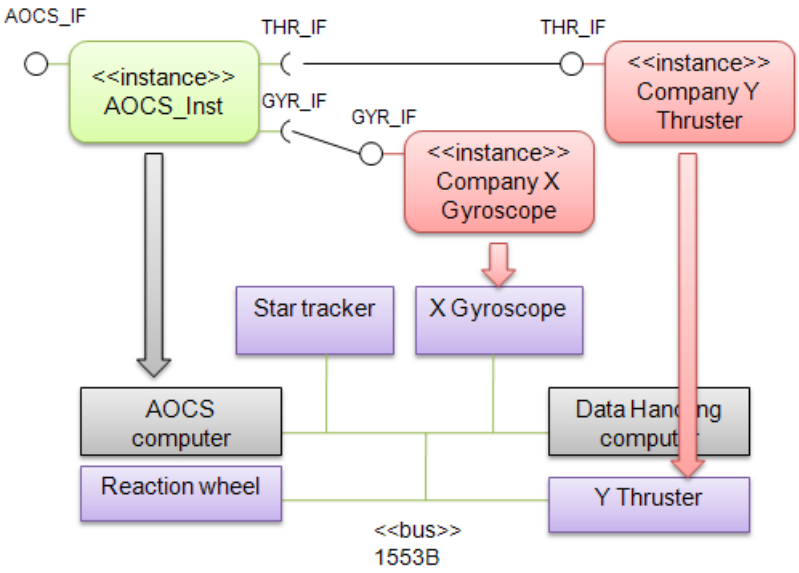

Figure 3.17: Relationship between software components and sensors and actuators. In the figure we also depict the deployment of those devices on their hardware counterpart.

At design level we have two needs: (i) represent those devices in the hardware architecture; (ii) relate software components to sensors and actuators.

For the latter, we need a component-level description of sensors and actuators in order to bind software components (thus at instance level) with the sensors and actuators they command.

For our component model we decided to represent devices in the same way as software components: the designer creates an interface for them, creates a component type, implementation and then instance for the device. The only difference with software components is that devices expose only provided interfaces and have no required interfaces, and their interfaces at instance level are not decorated with extra-functional attributes.

The distinction between component implementation and component instances facilitates the creation of multiple identical instances of the same device (irrespectively of redundancy concerns). For example, a satellite is typically equipped with 2-4 thrusters of the same type, which exercise their force in different directions.

The device representation is only for interaction purposes, as we intend to exclusively represent the software interface of the device and not its internals or generate driver code for it. Finally, the component-level representation of a device instance shall be deployed on its hardware counterpart in the hardware architecture.

Intra-component bindings. The description of component implementations according to the description of step #03 is completely black box. Unfortunately, an opaque description of the component hinders schedulability analysis.

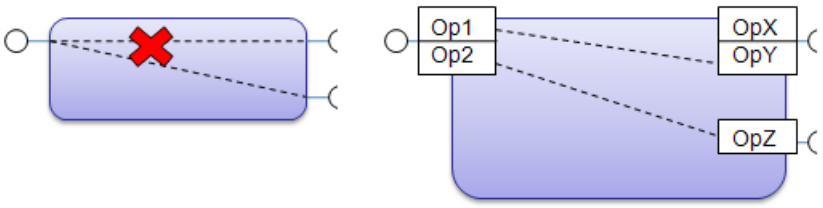

Figure 3.18: Intra-component bindings. A black box view of the component implementation hinders schedulability analysis. We need information on the RI needed by each PI (more precisely, the operations in the RI needed by each operation of a each PI.

For that reason, we need at least to know which required interfaces (RI) are requested by each provided interface (PI) of a component implementation, and, more precisely, which operations in the RI are called by each operation in the various PI of the component implementation (cf. figure 3.18). With this information it is possible to correctly understand the intra-component bindings between the PI and the RI side. This information is defined at implementation level, as it does not vary across different instances of the same component implementation.

Inter-component interactions. We must be able to capture the transitive call chain that originates at any given RI, as it is a fundamental input for schedulability analysis.

Bindings between instances represent bindings between interfaces of components. The intra-component bindings provide us the information on which operations of an RI are called; therefore, by following the bindings between component instances, we know which operations of a PI of another component are called in the remainder of the call chain.

In our approach, a call chain may span across components but it is guaranteed to be *finite* (it is represented as a tree with a leaf at every first occurrence of a deferred operation, i.e., with sporadic or bursty activation pattern, which is executed by its own thread of execution) and *acyclic* [31].

Furthermore, we need: (i) communication attributes, such as message sizes, to analyze the amount of transmitted data and enforce communication budget if applicable; (ii) the specification of end-to-end requirements on call chains across components.

The latter is particular useful for analysis purposes. Schedulability analysis (in the simplest forms of response-time analysis equations derived from [77]) can in fact provide response times only for single tasks. Conversely, the specification of end-to-end requirements can be used by a more expressive analysis framework (such as [103] and its derivatives [104, 60]) to provide

response time results on the completion of end-to-end chains of operations and optionally on single intermediate operations in the chain. This is possible because the analysis correctly interprets the end-to-end flow as a collaborative chain of activities across distinct tasks.

For the specification of those needs, we look favorably to representations similar to sequence diagrams, which seem sufficient to fulfill our needs.

In fig. 3.19, we use a UML sequence diagram and stereotypes of the UML MARTE profile for the specification of the message size and the end-to-end deadline of a chain of activities. The index i in the constraint $t1[i] - t0[i] \leq (125, ms)$ refers to the observation of the end of the activity within the *same* activation of the activity chain.

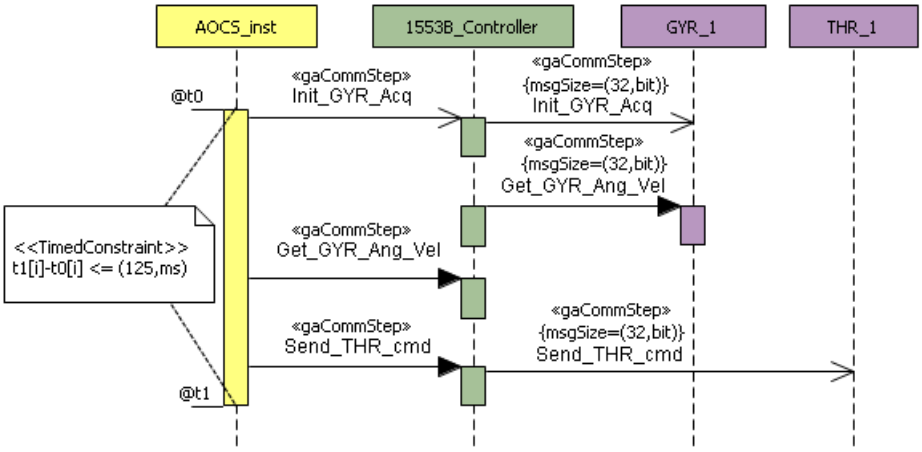

Figure 3.19: An example of specification of inter-component interactions, with specification of the size of transmitted messages and end-to-end requirements.

3.6 On the design flow and design views

As it is appears from the previous section, a component model does not only prescribe the syntactic rules to create design entities and how to relate them to one another, but – whether intentionally or implicitly, in any case inevitably – it also establishes a defined design flow.

The design flow comprises a series of steps that must be followed to create components, assemble them and ultimately produce the software system. It may also determine a set of precedence relationships between those steps.

This implies that the developers of the component model shall pay careful attention to ensuring that the design flow promoted by their approach is compatible with the development process in use with the concerned industrial domain.

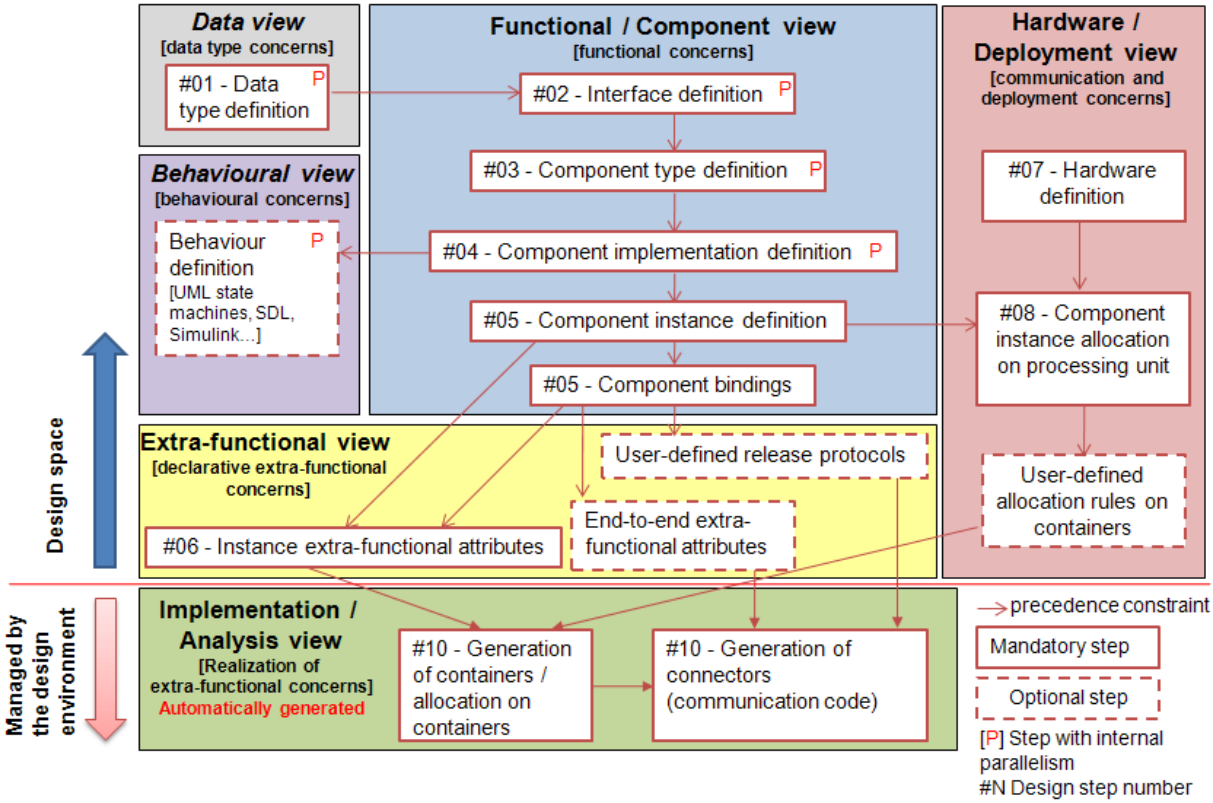

Figure 3.20: The design flow of our component model, including the design steps we described, and the design views used to enforce separation of concerns.

Figure 3.20 recapitulates the design flow for the component model we described and the precedence constraints it introduces, depicted as red arrows. In the same figure we also depict how we allocate the design steps we described to different views, and their related concerns.

The definition of Interfaces (Part of Step #02), component types (Step #03), component implementations (Step #04), component instances and instance bindings (Step #05) are allocated to a view that we termed *"Functional / Component view"* as those entities include only functional concerns.

Component instances still pertain to the "functional / component view". In order to decorate instances with extra-functional attributes (Step #06), we require the designer to explicitly transition to the "extra-functional view". In this manner, the modification rights of the latter view take effect: in the extra-functional view, the designer cannot create or modify entities, but only decorate the interfaces of instances with extra-functional attributes. The creation or modification of entities would require the designer to return back to the "functional / component

view".

In a UML context, the presence of "types" and "instances" (classifier and instance specification) in the same view would break the separation of concerns between the "static view" and the "dynamic view". Classifiers are in fact static entities that belong in the former view, instances are dynamic entities ("objects") created in the latter view. In our context instead, the inclusion of instances in the functional / component view is meaningful, as component instances are all statically defined at design time and no dynamic creation of them is permitted.

It is important to notice that with our approach no premature extra-functional choice is forcedly imposed on the component design. For example, we can compare our approach with provided interfaces to other approaches that rely on ports (see fig. 3.21).

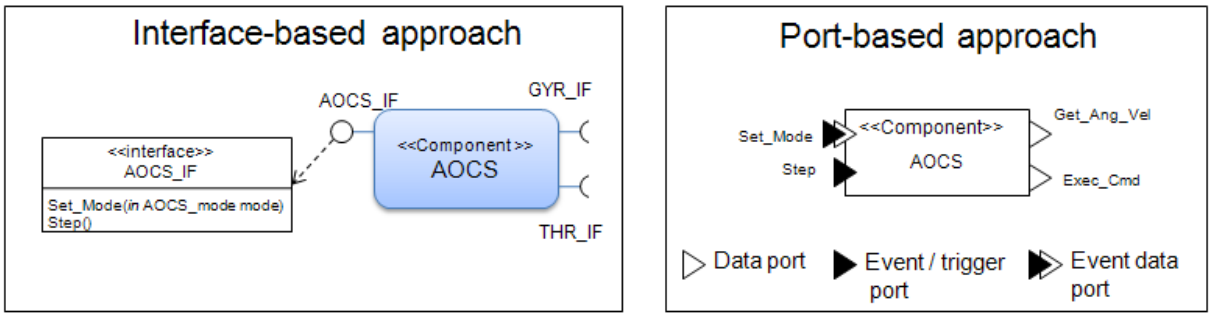

Figure 3.21: **On the left side:** A component which provides services as a set of provided interfaces. **On the right side:** A component which provides services as a set of ports.

When creating a component, the designer is forced to choose the kind of ports both at provided and required interface and port kind implicitly defines tasking semantics (for example an *in* event or event data port in [119] or a trigger port in [63] require a dedicated executer thread (or a thread selected from a thread pool), which is necessary to honor the extra-functional semantics of the port; a data port assumes that the service is executed by caller). Furthermore, the kind of an output port (event, data, event-data) always dictates the candidates for component binding: only those that have a port of the same kind. Therefore a functional concern (component binding) necessarily carries undesirable restrictions on the extra-functional form of the bound component.

The realization of our approach with a port-based component specification would require a notion of port void of explicit or implicit tasking semantics; the port would need then to be later refined according to the desired extra-functional semantics. In our case, all the extra-functional decisions are postponed to the appropriate view and no component requires any modification in case extra-functional attributes are changed; in contrast, a port-based approach would require the substitution of the port with a port of the appropriate kind.

The precise definition of data types (Step #01) shall be performed in a dedicated view were it is possible to define types independently of the underlying representation of the target platforms and selecting the encoding rules for their representation. We will not directly investigate the *data view* in the present investigation.

Currently we want to provide the designer with two choices for the definition of data types: (i) a support for the definition of types built in the component model incarnation, with an expressive power comparable to that of the Ada language (thus including at least: ranged types; constrained types; arrays and enumerations as true types; composite types); (ii) defer the specification of data types to an external source.

For the latter, the designer simply uses unique identifiers to refer to type definitions inside the component model. The type definition is currently in ASN.1 (Abstract Syntax Notation One) [72], permitting to interface with the data modeling provisions of the TASTE toolchain [38] by ESA.

The decoration of interfaces of component instances with extra-functional attributes requires the transition by the user to the *extra-functional view*.

At present, we directly support concurrency (tasking and synchronization) and real-time concerns. As more extra-functional dimensions are supported, most probably it will likely be simpler to split the extra-functional view in several view for their concerns; for example, a *real-time view*, a *dependability view*, etc..

The specification of the hardware topology and the description of the execution platform (Step #07), and the allocation of component instances to processing units (Step #08) are performed in the *hardware/deployment view*.

The generated containers and connectors (Step #10) realize the extra-functional attributes that were specified as declarative attributes in the extra-functional view. They thus belong to another view termed *implementation view*. In our approach we decided to make this view available exclusively in read-only mode. The reason is twofold and in line with our previous work [21]: firstly, the view is automatically generated, so the responsibility for the correct generation of it is not on the designer, but is ensured by model transformation; secondly, we do not allow manual modifications of the PSM. A similar reasoning applies also to all the necessary analysis views (for the generation of the SAM or other analysis-specific models).

Figure 3.20 highlights four optional steps: the modeling of the behaviour of a component implementation; the specification of end-to-end extra-functional requirements; user-defined rules for allocation on containers; and user-defined release protocols.

In our approach we support source or object code for component implementations directly developed in Ada or C/C++ (or generated with Matlab or Simulink), provided that complementary information to permit the analysis of interests is provided.

Alternatively, if the designer is interested, the implementation of components can be developed with model-based approaches (like UML state machines or SDL). In that case, the

concerns would be addressed in a dedicated view, termed *behavioural view*, which would be in charge of ensuring that the implementation is not violating the assumptions of the approach, in particular w.r.t. separation of concerns (for example, it shall actively forbid the inclusion of tasking and time-related constructs in the implementation code).

End-to-end extra-functional requirements are optional and are specified as described in paragraph ”Inter-component interactions”.

The allocation of components to containers is determined by the timing and concurrency attributes that decorate the operation on the provided interfaces. It is possible to define a set of standard, deterministic rules to automatically allocate components. There might be however situations in which the designer may want to directly control this allocation: for example they may decide to allocate several periodic operations on one and the same single-threaded container, in order to reduce the proliferation of threads, which is a common need for resource-constrained systems. We are planning to avail the designer means to influence the allocation to containers (in the example above, by proposing the allocation on the same container with suitable offsets among the release of the periodic operations). This flexibility shall however be subject to a preventive assessment by the design environment, which shall propose to the designer only a finite set of feasible solutions. The allocation is again performed automatically by the design environment.

User-defined release protocols would instead avail the designer a certain degree of flexibility in specifying a complex release protocol (in addition to the periodic, sporadic and bursty release patterns). We are thinking about availing a small closed language, similar to a state-machine specification, by which the designer can specify the desired release protocol. Again the realization of the release protocol (in the appropriate connector) would be ensured by automated code generation.

It is obviously important for both the feature described above that the underlying analysis framework is able to capture and intelligently consider them.

Our design process maps to the generic CBSE design process depicted in figure 3.6: component types and component implementations (just the object code or including the complete source code) are functional units that can be subject to (functional) reuse. They are packaging entities that can be stored in a component repository and later retrieved when necessary.

The remaining aspects of ”Step 1” in figure 3.6 are scattered among the functional view (for the adaptation/modification of components, and creation of component instances and instance bindings), the extra-functional view (for specification of extra-functional attributes), and the hardware/deployment view (for the description and selection of the target platform).

3.7 Inclusion of domain-specific concerns

3.7.1 General approach

The vast majority of component models, although possibly developed in the context of a specific application domain, seem to target needs that are somehow common to real-time embedded systems. Although the starting point of those investigations is typically an evaluation of the most important requirements for their application domain, and the resulting component model is more oriented to try to fulfill them, with few exceptions, there are no evident provisions that help to solve specific needs or recurrent problems of their application domain. Those approaches would qualify as *domain neutral*.

A *domain-specific* component model should instead be an approach specialized for a given real-time embedded application domain (e.g. space, telecommunications, civil avionics, etc.). In this respect, we therefore share the analysis made by Lau et al. [82], although our context and solution differ.

Support for ”domain-specific” concerns entails the support for all the concerns carried by that specific application domain. Without that support, the adoption of the component model as industrial baseline may become unattractive, as the extent of modification required to adapt the domain-neutral part to domain-specific needs may be significant, and may even undermine fully or in part the guarantees provided for by the original approach.

The ideal situation would be the development of a component model shared between different application domains, yet capable of expressing domain-specific aspects. The domain-neutral part of the approach would be used by all the domains. Each domain would then enable only the domain-specific part of interest (which includes language extensions, additional design views, specialized analysis and code generation) to complement the domain-neutral part.

The consequences of this vision are challenging at different levels: (i) from the methodological point of view, the solution to the domain-specific concerns shall not invalidate the core principles underlying the domain-neutral part; (ii) from the language point of view, the domain-specific concerns shall be included as modular parts of the language, and characterized by a low coupling with the domain-neutral part; (iii) from the tooling and design point of view, a domain-specific concern shall be presented in a separate design view, which is activated only when needed; (iv) from the analysis point of view, the narrowing of the target to a specific application domain makes it possible to specialize the analysis model with a fine-grained description of the relevant aspects of the target platform (for example, the precise delegation chain for remote message passing of the distribution middleware of choice) and to adopt refined analysis equations (for example, response time analysis specialized for the adoption of the Ravenscar Computational Model [140]).

3.7.2 Space-specific concerns

To support our argument, let us know elaborate on a set of space-specific concerns that shall be included in the software reference architecture. We then focus on a selection of them to discuss how we can include them in the approach and with what consequences.

The services specified in the Packet Utilization Standard (PUS) [45] are used for *commandability* and *observability* of the on-board software from ground. They specify the transmission and execution on-board of various kinds of operation requests and the expected response of the on-board software in terms of the acknowledgement, progress reports on the execution of the operation and the downlink of data to the ground receiver.

The ground segment (or an application on board) may send a telecommand (TC) to a telecommand receiver (termed ”Application Process”) to command the execution of certain operations on board. The on-board receiver replies by producing a response stored in one or more telemetry packets (TM) that are dispatched to ground.

The implementation of the PUS services is fundamental for the realization of on-board software as any in-flight communication with the satellite performed by the operational team is managed through them.

The PUS standard specifies 16 standard services, which are enumerated in table 3.2 together with a short description.

Table 3.2: List of the PUS services.

Service	**Service name**	**Short description**
1	*Telecommand verification service*	Verification of each distinct stage of execution of a telecommand packet from on-board acceptance to completion of execution
2	*Device command distribution service*	Upload of commands and register values for devices
3	*Housekeeping and diagnostic data reporting service*	Sampling and reporting of information not explicitly provided within the reports of another service
4	*Parameter statistics reporting service*	Report the maximum, minimum, mean and standard deviation values of on-board parameters during a time interval
5	*Event reporting service*	Report events of operational significance (failures, anomalies, autonomous actions, acknowledgment of non-anomalous events)

Table3.2 – continued from previous page

Service	Service name	Short description
6	*Memory management service*	Load, dump and check the contents of memory areas
7	*Not used*	
8	*Function management service*	Execution of operations not implemented as standard or mission-specific services whose execution can nevertheless be controlled from ground
9	*Time management service*	Downlink of the satellite time reference
10	*Not used*	
11	*On-board operations scheduling service*	Load and conditional release of a set of time-triggered telecommands stored in a dedicated on-board schedule (to perform operations without coverage of ground stations)
12	*On-board monitoring service*	Monitoring of the on-board parameters with respect to a configurable check list and report of monitoring violations
13	*Large data transfer service*	Transfer of large data units in controlled manner (to manage data that does not fit the maximum telemetry packet size)
14	*Packet forwarding control service*	Control rules to enable or disable the dispatch to ground of telemetry packets generated on board
15	*On-board storage and retrieval service*	Rules to store telemetry packets in on-board storages and capabilities to retrieve them via ground commands (used in time intervals without coverage of ground stations)
16	*Not used*	
17	*Test service*	Activation of test functions defined on board
18	*On-board operations procedure service*	Storage and execution on board of ”operation procedures” (a single telecommand from ground can start the release of a preloaded set of telecommands executed on-board)
19	*Event-action service*	Definition of a set of actions executed autonomously on board when a given event is detected.

Each service has a set of sub-services which take the form of specific requests via TC or reports via TM; some of them are mandatory for any PUS implementation, while the rest is optional. Additionally, each mission adds its own mission-specific services or specializes standard services with mission-specific service sub-services.

Let us look at service 15 - *On-board storage and retrieval service* - as an illustrative example. The service deals with the configuration rules to store the telemetry packets in on-board storages and how to downlink them via ground commands. The service is typically used when ground coverage is not available. Figure 3.22 depicts the use of the service.

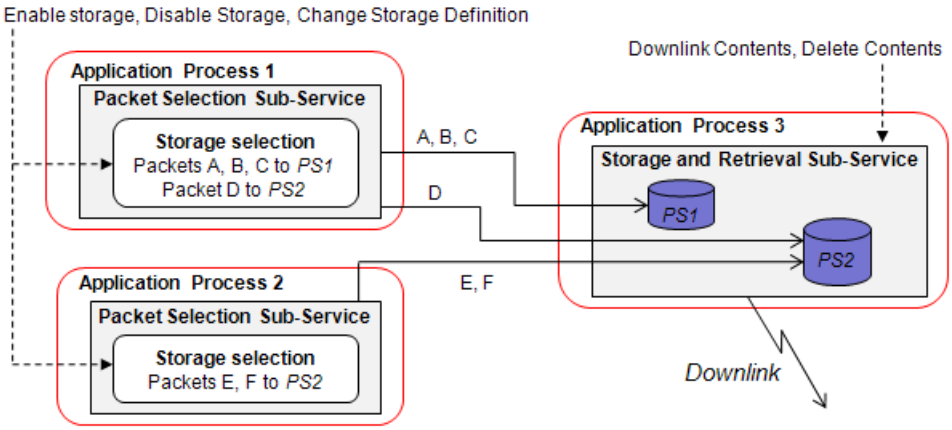

Figure 3.22: PUS service 15 - On-board storage and retrieval service.

Table 3.3 instead enumerates all its mandatory and optional sub-services.

Table 3.3: PUS service 15 - On-board storage and retrieval service. ST is the service subtype. The service reports in the right column are sent via telemetry as a response to the service request on the left column. Service subtypes in italic are optional.

PUS Service 15 - On-board storage and retrieval service

ST	Service requests	ST	Service reports
Packet selection sub-service			
1	Enable Storage in Packet Stores		
2	Disable Storage in Packet Stores		
3	*Add Packets to Storage Selection Definition*		

Table3.3 – continued from previous page

ST	Service requests	ST	Service reports
4	*Remove Packets from Storage Selection Definition*		
5	*Report Storage Selection Definition*	6	*Storage Selection Definition Report*
Storage and retrieval sub-service			
7	*Downlink Packet Store Contents for Packet Range*	8	*Packet Store Contents Report*
9	*Downlink Packet Store Contents for Time Period*	(8)	
10	*Delete Packet Stores Contents up to Specified Packets*		
11	*Delete Packet Stores Contents up to Specified Storage Time*		
12	*Report Catalogues for Selected Packet Stores*	13	*Packet Store Catalogue Report*

We want to capture these PUS concerns in our component-oriented approach.
Telecommands and telemetry are just structured packets of bits (see figure 3.23).

Packet Header (48 Bits)							Packet Data Field (Variable)			
Packet ID				Packet Sequence Control		Packet Length	Data Field Header (Optional) (see Note 1)	Application Data	Spare	Packet Error Control (see Note 2)
Version Number (=0)	Type (=1)	Data Field Header Flag	Application Process ID	Sequence Flags	Sequence Count					
3	1	1	11	2	14					
16				16		16	Variable	Variable	Variable	16

Figure 3.23: The structure of a telecommand packet [45]. *Credits: European Cooperation for Space Standardization (ECSS)*

Hence, they have an abstraction level that is too low to be usefully represented at component level. The design space of the component model shall not be polluted by low-level implementation entities like tasks, semaphores, and such like. For the same reason, introducing TC and

TM would break the abstraction level of the design space; in comparison with the OSI reference model [74], they would address some limited concerns of the network and transport layers.

We have three objectives for the positioning of PUS services in our approach:

1. at *design level*: to provide a representation or means to configure the use of PUS services at a level of abstraction that is appropriate for the design space of our component model (thus abstracting from TC and TM);

2. at *implementation level*: to deal with TC and TM exclusively at the level of containers and connectors, using the information that is derived from the design model. We do not want that components (and their code) are polluted by TC and TM.

3. in general, to understand which services are *infrastructural* in nature and try to treat them as transparently as possible. An example is service 1 - Telecommand verification service. We contend that the management of TC reception and verification and the generation of TM progress reports shall simply be integrated in how containers are generated, without any required intervention of the user from the design space.

A final objective with which we permeated the investigation is the strive to recognize whether the PUS service in question may be regarded as a space-specific instance of a more general problem, which can be of interest to other domains. If that is the case, it would be worth to (at least partially) elevating the concern to *domain neutral* at least as regards the modeling and design space. That would provide a design-level treatment of the concern common to other domains, even in the case that some residual domain-specific configuration of it are necessary. The implementation solution instead will always be dependent on the application domain and target platform.

The concerns we outlined can be broken down in a series of questions that we used as guideline to our investigation.

- Does the service require a new dedicated design entity in order to be represented at design level? (for example a ”special component” like the component-level representation of devices, i.e. just for interaction purposes)

- Shall we add (domain-specific) attributes to already existing model entities so as to configure the service? In which design view? In which diagram?

- How does the implementation of the service impact code generation of containers and connectors? For example how it is possible to dispatch telemetry or add PUS-specific data structures on a container?

- Does the service impact the execution platform? What mechanisms and provisions shall be added and/or which modified?

- Does the service describe a need or concern that is relevant for other domains too? Is it useful to factorize it in the domain-neutral part of the approach?

Finally, we provide two furhter examples of PUS services and discuss how we plan to include them in our component-oriented approach.

Service 12 – *On-board monitoring service* – makes it possible to monitor a set of on-board parameters w.r.t. a set of ”check definitions”. There is a monitoring list on board to which it is possible to add or delete monitored parameters, enable or disable the monitoring of some parameters, change the monitoring frequency, define or modify the check definitions.

A check definition can be: a ”limit-check” which ascertains if the parameter is within a range of minimum and maximum values; a ”delta-check”, which controls that the difference from the previous sampling of the parameter is within a range of allowed variation; an ”expected-value check” which determines that the parameter has a defined value.

When a monitoring check fails, the service raises an event as specified by the applicable check definition.

First of all we single out the entities of our approach that are the subject to this service: they are the attributes of components at instance level. We therefore avail the means to associate a set of check definitions to each component instance attribute (see figure 3.24).

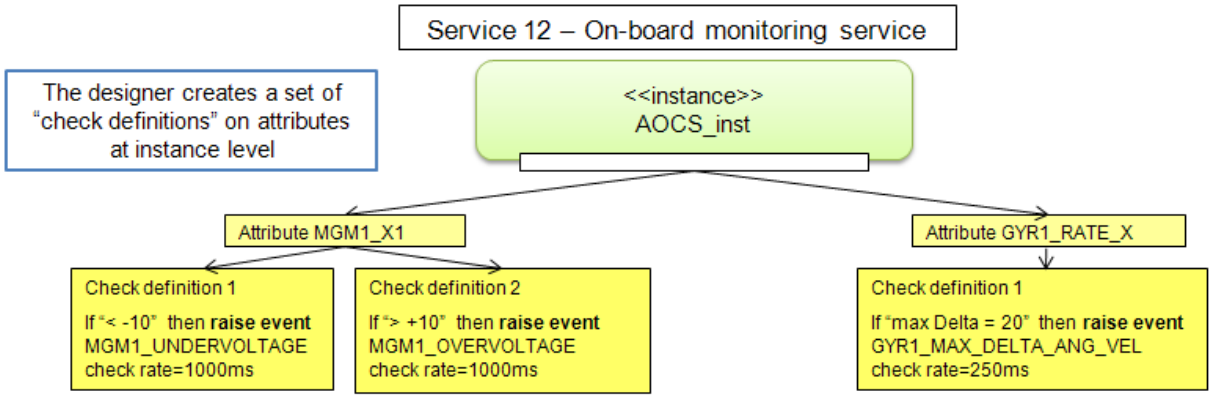

Figure 3.24: Our approach to the realization of PUS service 12 - On-board monitoring service.

In the example, the AOCS instance has an attribute related to the voltage of a magnetometer with two associated check definitions. If the value for the voltage falls below or raises above certain values, a given event is raised. There is then a check associated to the angular velocity on the x axis for one of the gyroscopes: if two subsequent measurements differ of more than the allowed value, the specified event is raised. Check definitions also specify the rate of the relevant monitoring.

Up to now, no need to provide in our component model a first-class notion of ”event” was elicited (at least explicitly). It is not really feasible to express this concept with the entities described up to now for our component model. An event fundamentally differs from a required interface because it does not represent a functional need that must be satisfied by the component in order to operate properly. It is not even a functional service of a component, such as a provided interface.

We think that the concept of ”event” can be useful also in other domains, in accord with the strategy outlined before. We therefore considered equipping the component model with a first-class entity to define events, and the syntactic means to specify the events that can be raised by a component and received by one or more components registered for the event. A similar idea has been realized in LwCCM [96]. This ”event” entity would be domain-neutral, hence would have a specification, allocation to views and representation common among all domains. A space-specific counterpart would then refine the notion with all the necessary attributes for PUS events.

Our second and last example is PUS service 8 - Function management service. The service is simple to describe, much harder to realize.

The execution of an operation on board may be requested remotely by the ground segment. The PUS standard expects that this is realized by equipping that operation with a ”FunctionID”; the ground segment can then send a telecommand of service 8 to request the execution of the operation with a defined FuntionID.

We realize this service in our approach as depicted in figure 3.25. At design level, the user simply tags an operation of a provided interface (at instance level), enabling its visibility from ground via PUS service 8. The design environment then ensures to assign automatically or according to configuration rules – in any case transparently – all the additional attributes necessary for the correct execution of the service (in this case the FunctionID and another attribute called APID, which is used to identify on-board receivers).

Furthermore, code generation inserts in the container all the parts necessary to accept incoming telecommands for service 8 (TC(8,1)), the code to decode the TC and its parameters (which are the actual parameters to use in the call of the operation of the component), verify the well-formedness of the TC and acknowledge its correct or incorrect reception by sending the appropriate telemetry packet to the TC sender (TM(1,1) or TM(1,2)), performing the software call to the operation of the component, and then inform the TC sender via telemetry about the progress of execution of the operation with various sub-services of the telecommand verification service: TM(1,3) to signal the start of execution of the operation, TM(1,5) to signal the percentage of completion of the operation, and finally TM(1,7) to signal the successful completion of the operation. Failures can be notified with complementary sub-services: TM(1,4), TM(1,6) and TM(1,8) respectively.

Standard libraries that perform those actions already exist; for example OBOSS [139]. How-

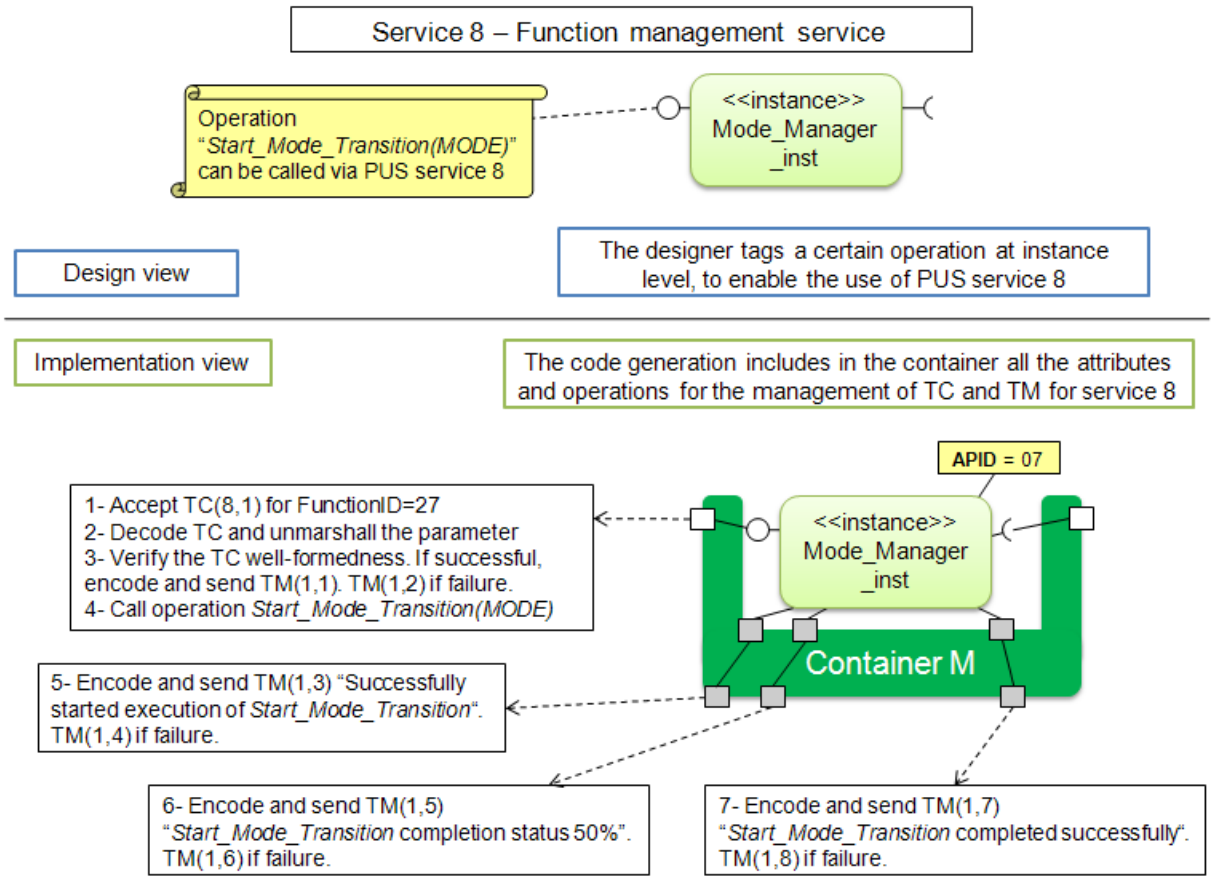

Figure 3.25: Our approach to the realization of PUS service 8 - Function management service.

ever, their approach is code-centric. The advantage of our approach is that we are dealing with these concerns directly at the design level, hence increasing the abstraction level and treating these concerns as transparently as possible. The automated code generation ensures that the implementation is consistent with the specification.

The final concern regarding the sound inclusion of PUS services in our approach is the support for schedulability analysis. Even if we did not describe all the services in detail, the reader may have understood that the implementation of these services, albeit confined to the container/connector part of the architecture, pervades the on-board software and may impact the timing behavior considerably.

This situation is similar to the adoption of a middleware for distribution transparency. The middleware in fact may introduce non-negiglible overhead due to the delegation chains that carry out the end-to-end communication, and this arguably shall be accurately represented in

the input for schedulability analysis.

As a conclusion, we think that the number and heterogeneity of the PUS services is a convincing argument to support our claim that domain-specific concerns (meaning, expression of a single application domain) are fundamental for the industrial adoption of an approach without incurring costly modifications. Additionally, we think that the ability to recognize and separate at the various level of the problem the domain-neutral and domain-specific concerns is important as it permits to factorize only the aspects common to multiple domains and selectively activate only the domain-specific aspects of interest. This arguably avails a simpler language to the designer, in contrast to a language that proposes to the user all domain-neutral and domain-specific aspects without any means to filter among the latter. The net result of this strategy is focused specification, which in turn may lead to increased productivity.

At the current stage of the investigation, we analyzed in detail a selected subset of the PUS services (service 2, 5, 8, 12, 19) to confirm the feasibility of our approach, and we deem that the strategy we propose to address domain-specific concerns is in general feasible.

3.8 Realization

The realization of our component model immediately at the beginning has an important hurdle to overcome: the selection of the design language to specify components in the design space. We discuss this issue and show that there are two paths we can follow: the design of a domain-specific language (DSL) or the adoption of a standard mainstream language.

We elaborate on the consequences of one choice or the other, by highlighting how the subsequent intellectual and technological effort is devoted to quite different aspects of the problem.

We then proceed with the narration of the path we chose for this PhD thesis: the development of a DSL. We report on the main design choices regarding the construction of an MDE design environment supporting our component model in aspects such as the metamodel, the creation of the design editor, the support and strategy for model-based analysis and code generation. As a conclusion we relate on the future work and the main challenges ahead of us.

3.8.1 Selection of the design language

A fundamental distinction must be drawn between what is the *component model* and what is the *design language* that is used to create components.

The design of the component model (component features like ports, support for interfaces, attributes, how components relate one another, deployment and extra-functional concerns, etc.) precedes and is disjoint from the specification language that is used to design the components in the user space.

When it comes to defining the design language for the component, there always is at least one default solution: to create a domain-specific language (DSL).

Alternatively, the developer of the component model may try to express the component model using a standard modeling language for real-time embedded systems. Example languages include the UML MARTE profile [99] or AADL [119].

Choosing a standard language as the design language of the component model can earn strategic advantages: the standard status of the language can encourage tool vendors to invest in the creation of design environments for it. In that case, the component model would leverage the tooling, thereby dispensing with the burden of developing an ad-hoc design environment from scratch. This also permits to invest effort and focus on the value added of the approach rather than on the entry level of the tooling (model comparison, model synchronization and serialization, model versioning, etc...).

Furthermore, it is quite likely that the developers of tools for specialized analysis (e.g., schedulability analysis, fault-tree analysis, etc.) converge over time to a common reference language that permits them to feed their tools with the information extracted directly from standard representations of the system model at hand. In this case, the component model developers are no longer required to create syntactic and semantic mapping to the input formalisms of each of the needed or desired analysis tools.

Of course, the component model developers must be able to fully express all the features and traits of the component model with the standard language, whether directly or using its extension mechanisms. as for example with UML stereotypes or AADL property sets.

When one tries to reconcile with the pre-existing structure of the modeling language, two situations may occur. The chosen language is able to fully express the component model, because evidently its expressive power covers a superset of the needed features. Or else, the chosen language is not expressive enough to fully represent all that the component model can articulate and thus the mapping fails. The situation is depicted in figure 3.26.

Furthermore, if a suitable (i.e. expressive enough) standard mainstream language is adopted, it has to be tailored to express *solely* the concerns of the component model. The best strategy to achieve this objective is – in accord with the C-by-C development practices outlined in section 3.1 – to single out the appropriate subset of the language and enforce it with the means availed by the language itself (hence, forbidding the use of some modeling entities, or constraining their use, or the relationships with other entities).

In UML 2.x, it is possible to use the profile mechanism, which permits to forbid the use of metaclasses, to introduce new modeling elements (based on the existing metaclasses) or and permits the specification of modeling constraints specified with the *Object Constraining Language* (OCL) [100], to determine the well-formedness of the model according to given criteria.

AADL offers only the *property set* mechanism for extension, so that it is possible to add

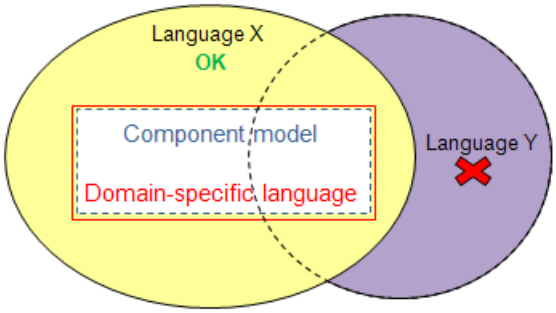

Figure 3.26: Relationship between component model and the design language to express it. A component model has always a corresponding domain-specific language able to fully express it. Language X can express a superset of the necessary concepts, and it is able to fully express the component model. That is not true, for language Y, which lacks the expressive power for all the concepts of interest.

typed members to already existing entities, but it is not possible to specify new modeling entities (not even through specialization mechanisms) and the standard does not offer mechanisms to create and enforce modeling constraints, although some research in that direction have been made [56].

Finally, choosing a mainstream language forces to specify the system with the same diagram support (if any) specified by the language: creating diagram with a focused or more simple graphical notation is not possible.

The DSL approach instead naturally mirrors the component model, and for this reason is in principle the best solution. Unfortunately, the main drawback is that typically the metamodel for the DSL, the whole design environment and all transformations to extract information for analysis tools has to be created from scratch.

For what concerns this PhD thesis we decided to opt for the development of a DSL, which is naturally in line with the founding principles of the approach and facilitates fast iterations for the improvement and validation of the approach. In the next section we detail the activities for the development of the DSL and of the design environment.

3.8.2 Development of the design environment

3.8.2.1 Definition of the domain-specific language

In the context of this PhD thesis we head our investigation toward the realization of a domain-specific language.

We think that creating an incarnation of the component model as DSL in this early stages

of the realization offers the flexibility to explore the best way to express the value added of the component model, and facilitates fast prototyping of ideas to solve small parts of the quest incrementally.

When instead one chooses to base on a mainstream language, a considerable part of the intellectual effort is dedicated to reconciling the component model to the pre-existent (and in some aspects immutable) structure of the metamodel of the language, which is the base for that language to express the component model. Then we must tailor the language to express only the needs of the component model. Furthermore, if the adopted language also has a graphical representation, the component model has to be fully and solely expressed using the diagram support provided by the adopted language.

Our design language is defined as a domain-specific metamodel (DSM) specified in *ecore*, an implementation of Meta Object Facility (MOF) [75] (more precisely of its subset "Essential MOF"). The core part of the language is considerably inspired by the "kernel" package of the UML metamodel [101] (cf. figure 3.27).

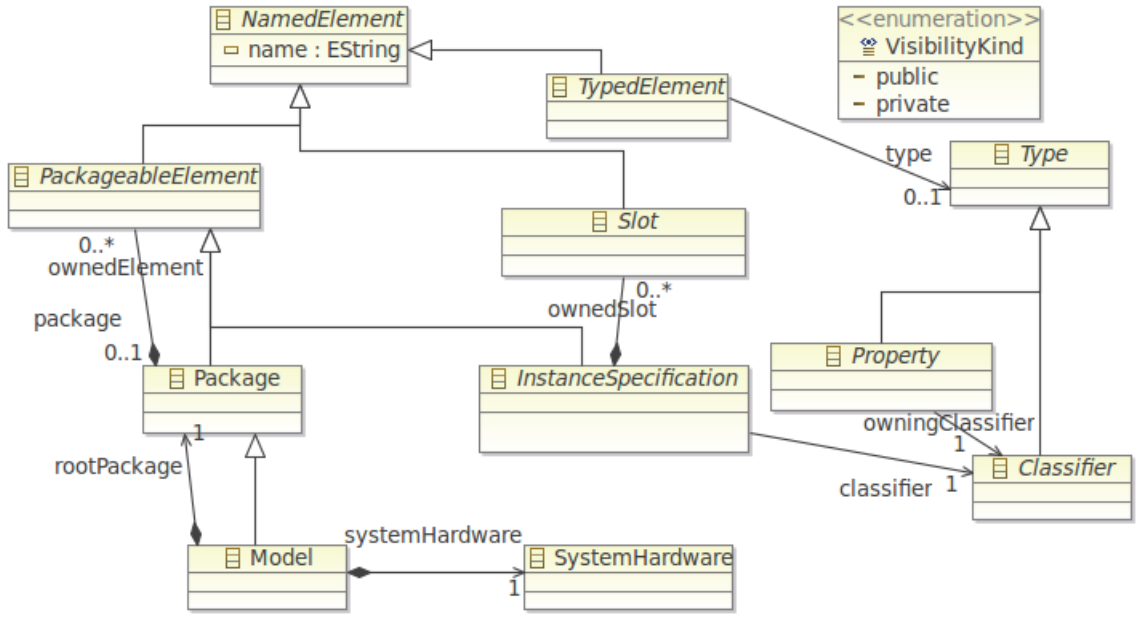

Figure 3.27: The core part of our domain-specific metamodel.

Nevertheless, we had the freedom to depart from the UML metamodel whenever we could express the intended structure of our component model in simpler or better way. In motivated cases, we chose to avoid the creation of metaclasses (especially the counterpart of many abstract metaclasses that in UML have a structural and factorization role in the metamodel) and establish direct relationships between metaclasses. In this manner we avoided the need to frequently

”specialize” generic relationships á-la-UML. This is typically done in a UML profile with the introduction a-posteriori of constraints on the validity of the relationship that is originally between abstract or metaclasses that are too high in the hierarchy to meaningfully and completely represent the entity of interest.

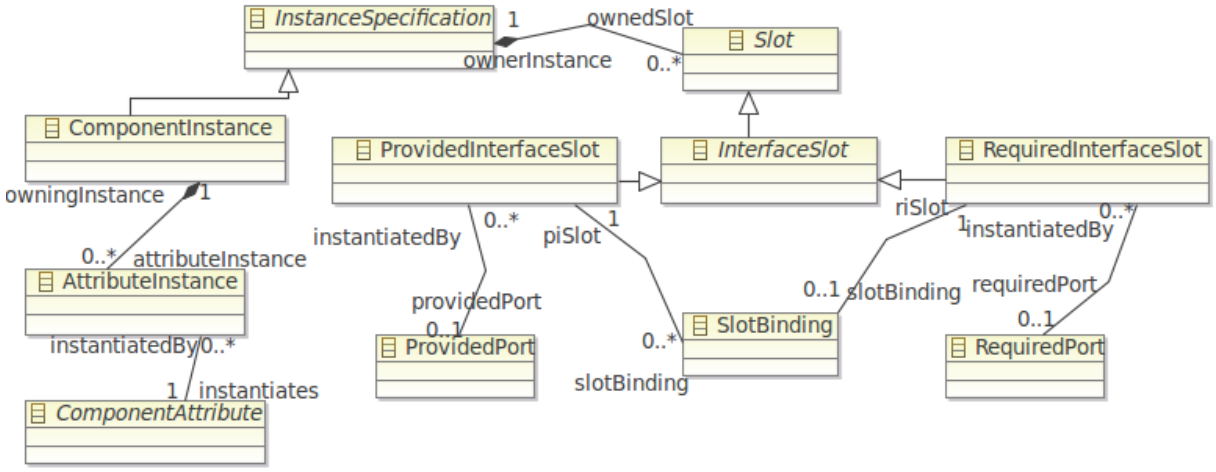

Figure 3.28: Metaclasses related to component instances and slots (PI and RI at instance level)

Figure 3.28 depicts the portion of our metamodel devoted to component instances, component attributes (at instance level), and slots (i.e. provided and required interfaces at instance level). The reader may note that we have retained the UML parlance for the constructs at this level.

3.8.2.2 Creation of a prototypal design editor

The creation of a design editor for the component model we discussed can be considered as a secondary activity of a PhD. In fact, even if the contribution of the editor per se to the intellectual value of the work is lower than the foundational aspects, its role can be considered quite strategic for two reasons:

1. firstly, it has a confirmatory role, as it is a tangible evidence that the approach for the component model is technically *feasible*. When the design editor is later augmented with support for model-based schedulability analysis and code generation, to represent the complete design environment, it can instead be considered as evidence of the methodological feasibility of the overall approach.

2. secondly, it is very valuable in aiding the *validation* of the approach. In fact, it is possible to design and realize a complete case study and disseminate that proof-of-concept

demonstrator to the industrial stakeholders, so as to receive feedback on the suitability and goodness of fit of the various aspects of the approach and an evaluation of the coverage of the technical requirements (hence industrial needs) that originated our investigation.

In our discussion in section 3.8.1 we maintained that the most adverse consequence when choosing the DSL approach is the need to create the design editor from scratch.

Until a few years ago, when tool developers wanted to use the Eclipse platform[1] as their development baseline, the only option for the construction of an editor was the EMF/GMF framework[2]. EMF/GMF is a mature technology; however, especially due to the programmatic nature of GMF, the effort needed for the creation of a basic graphic editor is very high; the refinement up to a product that can be adopted by industry, possibly an untenable effort.

Fortunately, we were able to access and experiment with a relatively new technology called Obeo Designer[3]. Obeo Designer is a closed-source commercial framework for the creation of editors; it leverages the standard EMF-generated editor. The general idea behind Obeo Designer is that the association between a metaclass of a metamodel and its graphical representation shall be declarative, using a simple tree editor; and this is done using some basic yet customizable graphical templates. All these templates can be customized in their shapes, colors, icons. The shift from the programmatic approach of GMF to such a declarative approach greatly decreases the effort for realization of an editor (insertion of Java actions may still be necessary, but only for complex tasks).

The main interest in the framework however, owes to its support for design views (called ”viewpoints”) as first-class entities. It is possible to create distinct design views, each of them comprising various diagram types in accord with our ideas of section 3.3.4. For each view, it is possible to create the most suitable representation for the design entities that we want to represent, and the tools (palette items) for the creation, modification or deletion of them.

A designer can in fact: (i) associate a graphical representation to entities of a view using a set of predefined and customizable shape templates; (ii) establish a set of layers for each diagram. Each layer contains a subset of the entities of the diagram and they can be activated or deactivated by the user; (iii) specify the tools for creation of entities in the design palette; the tools are bound to a layer and thus are available only if their layer is active; (iv) specify the actions to carry out when modifications or deletion of entities or entity attributes are performed; (v) specify rules to verify the well-formedness of the entities created by the user (perform model ”validation” in Eclipse parlance); the rules are either valid for all views (i.e. checked on the model) or for single views.

Unfortunately, it is not instead possible to customize per view the visible or modifiable attributes of the design entities (”properties” in Eclipse parlance). The editor visualizes in a

[1]Eclipse platform – http://www.eclipse.org

[2]Eclipse modeling framework – http://www.eclipse.org/modeling

[3]Obeo – Obeo Designer 4.6 – http://www.obeo.fr/pages/obeo-designer/en

panel their full list, according to the visibility rules specified for the underlying EMF editor, hence valid once and for all for the whole generated editor.

There are therefore some missing items of our wish list: the fine-grained management of visibility and read-write permissions on entities and *their attributes* (only the visibility of a complete entity in the diagram is currently managed); and view-dependent presentation of attributes in the ”property view” of Eclipse.

The custom visualization of Eclipse properties is being addressed by a project currently in incubation phase: the Extended Editing Framework (EEF)[4]. An inclusion of the capabilities of EEF in Obeo Designer (favoured by the fact that Obeo is the EEF project lead) would fulfill this uncovered need.

At the current state of implementation, we created a basic, yet complete editor to perform the user-level design steps (#01 - #08) described in section 3.5 (cf. table 3.3).

We also created basic diagram support for the specification of intra-component bindings and we are currently investigating the support for the specification of inter-component interactions, using a diagram style similar to sequence diagrams.

Table 3.4: Status of the implementation of the prototype editor. Where the entry reads Y, the feature is already implemented. Where it reads N, the feature is under development. Where it reads P, the feature is partially implemented.

Feature	**Status**
Interfaces and Data Types	Y
Component types	Y
Component implementations	Y
Component instances	Y
Component bindings	Y
Hardware architecture	Y
Component instance deployment	Y
Extra-functional constraints (component-implementation level)	N
Synchronization, timing and sizing attributes at instance level	Y
Component-level representation of devices	Y
Intra-component bindings	Y
Inter-component interactions	P
Support for design views	Y
Space-specific hardware (with attributes for analysis and code generation)	P
PUS services	P

[4]http://www.eclipse.org/modeling/emft/?project=eef#eef

The editor supports the *functional*, *hardware/deployment*, and *extra-functional* views, as described in figure 3.20.

Furthermore, we are experimenting on the implementation – at language and editor level – of PUS service 5, and service 8 and 12 as described in section 3.7.

The remaining design steps – model-based analysis (Step #09) and automated code generation for containers and connectors (Step #10) – are going to be integrated in the editor, in order to be performed seamlessly from the design environment. In the following sections we report on the status of their design and realization.

3.8.2.3 Model-based schedulability analysis

For model-based schedulability analysis, we decided to postpone the implementation. We do not want to neglect this part of the investigation and, on the contrary, we deem it as qualifying and of utmost importance.

However, fortunately this is the aspect of the overall problem which we are accustomed the most, and where we have the maximum confidence about its feasibility. We will in fact base on the same strategy devised and proven effective by previous work of the author in the ASSERT project [106, 21].

The general strategy is depicted in figure 3.29.

The problem encompasses two aspects:

1. to ensure that the information specified at design level (i.e. components and their declarative extra-functional attributes) is sufficient to extract or derive all the information needed for the generation of a Schedulability Analysis Model (SAM).

2. to engineer the process for the generation of the SAM, the extraction of the information to feed an external analysis tool and to propagate the results of the analysis back to the appropriate entities of the SAM and then to the user model.

The SAM comprises the representation of the implementation entities (i.e. containers and connectors in our CBSE approach) that are generated to realize the declared extra-functional attributes (in this case, time-, tasking- and synchronization-related attributes). The feasibility of the transformation is ensured by deterministic rules dictated by the adopted computational model.

Since our approach is centered on the adoption of a computational model (we target a finite set of implementation entities of known structure and timing behaviour), this transformation

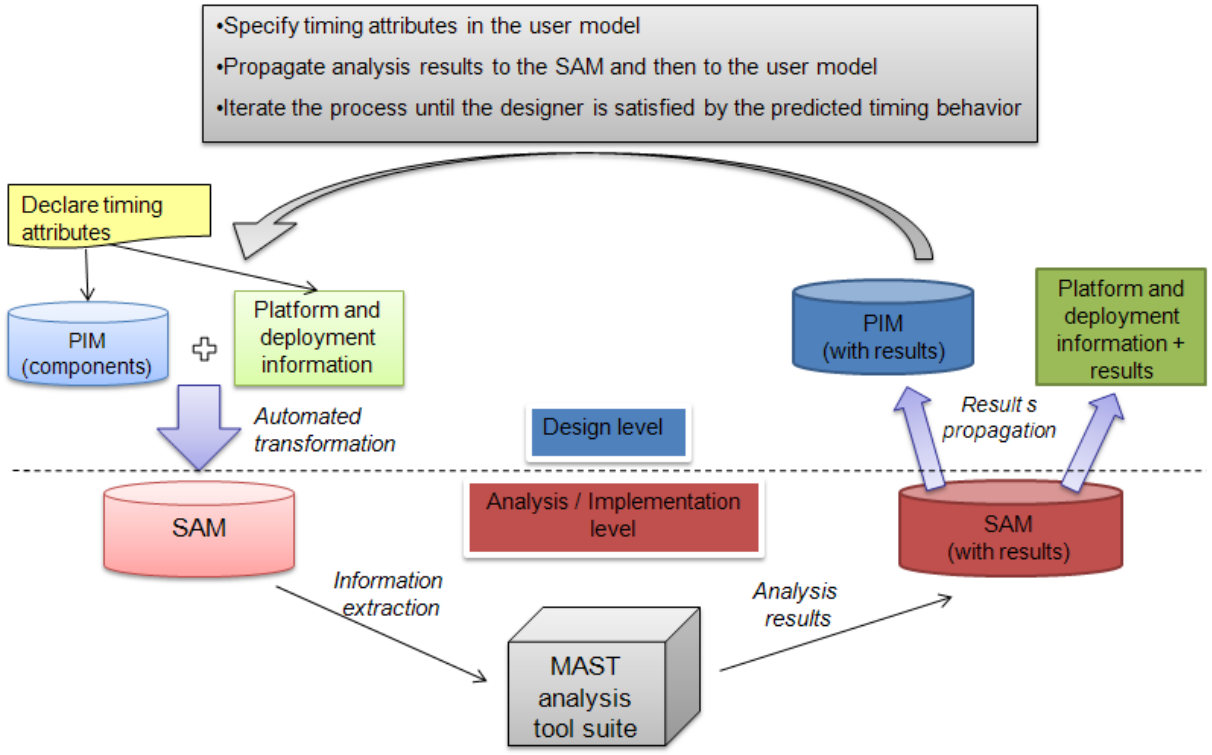

Figure 3.29: Model-based schedulability analysis, with back propagation of the analysis results into the user model.

has the same input as the transformation for the generation of the PSM for code generation; this permits us to factorize the effort.

We use MAST [133] for schedulability analysis. We ensure that the user (hence the generated PSM) is able to specify all the end-to-end information needed to take benefit of the most advanced analysis equations implemented therein (i.e. offset-based analysis with precedence constraints [104]).

3.8.2.4 Code archetypes and code generation

As in ASSERT, our code generation targets the Ada Ravenscar Profile [26] and uses a set of property-preserving archetypes [93] we already adopted in that project.

The task structure that we use as a base for the code mapping approach takes inspiration from HRT-HOOD [28], from which we also inherit the terms OBCS and OPCS.

Early work on code generation from HRT-HOOD to Ada was described in [42] and [23].

An important additional goal of our archetypes is to achieve complete separation between

the algorithmic/sequential code of the system (hence in our approach, the source code of a component implementation) and the code that manages concurrency and real-time.

The achievement of this goal confirms at implementation level and at run time our claim that it is possible to develop the algorithmic contents of the software (that is, the component behaviour) independently of tasking and real-time issues. The latter are exclusively derived from the extra-functional attributes declaratively specified on component instances and completely managed by the code generated by the design environment.

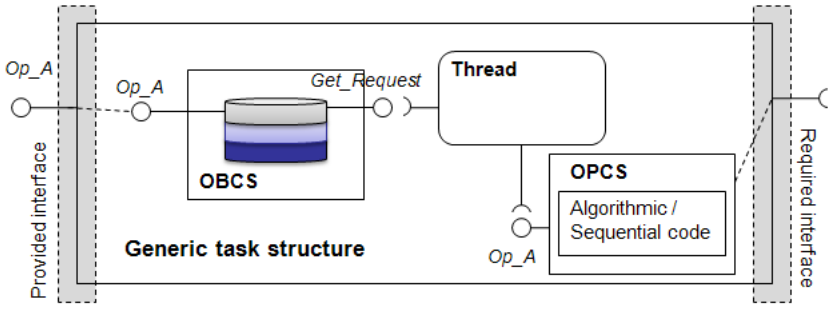

Figure 3.30: Generic task structure

Our goal is achieved by encapsulating the sequential algorithmic code in a dedicated task structure. Figure 3.30 depicts the generic structure of our task archetypes.

The sequential code is enclosed in a structure that we term the Operational Control Structure (OPCS). The code is executed by a Thread, which represents a distinct flow of control of the system. The task structure may be optionally equipped with an Object Control Structure (OBCS). The OBCS represents a synchronization agent for the task; we use it mainly for sporadic tasks. The OBCS consists of a protected object that stores incoming requests for services to be executed by the Thread. The interface exposed by the OBCS to the Thread is inspired by the *Adapter* pattern [55]. As multiple clients may independently require services to be executed by that Thread, the operations that post execution requests in the OBCS are protected. Upon each release, the Thread fetches one of those requests (FIFO ordering is the default) and then executes the sequential code, stored in the OPCS, which corresponds to the request.

Our entities encapsulate their internal structure and expose to the external world just an interface that matches the signature of the operations embedded in the OPCS. The different concerns dealt with by each such entity are separately allocated to its internal constituents: the sequential behaviour is handled by the OPCS; tasking and execution concerns by the Thread; interaction with concurrent clients and handling of execution requests are handled by the OBCS.

We can then recognize that the OPCS is the place where the component implementation is to be allocated; the Thread and the provided and required interface of the task structure realize the concerns of the container; the bindings between an RI and the PI of another task structure

and the OBCS of the task on the called side realize the concerns of the connector, the latter, as the enforcer of release protocols in place at the receiver side.

The use of Ada 2005 for the realization of containers and connectors is very convenient for the development of heterogeneous systems, and facilitates our job to support component implementations developed in different languages. In fact, the Ada 2005 reference manual defines standard bindings to other languages (cf. Annex B in [73]). Among the supported languages, C and C++ are of direct interest for components developed for the satellite platform software. Needless to say, components developed in Ada are immediately supported.

In the following we provide some additional details on the structure of the three constituents of the task structure.

OPCS The OPCS is an incarnation of the *Inversion of Control* design pattern[5] (a.k.a. *Dependency Injection*), which is a popular practice in lightweight container frameworks in the Java community (e.g. Spring). The connectors used by the OPCS to call operations of its RI are not created by the OPCS, but are simply provided with setter operations at system start up (see line 9 of listing 3.3 and line 6 of listing 3.4).

The use of the pattern warrants separation between the configuration of the OPCS (the connectors it uses) and the use of the OPCS; re-configuration of the connected end of the component bindings is thus easy, as it only require to change the connector availed to the OPCS. If we need to change the properties of the connector itself (e.g. a change of the communication semantics from best effort to at most once, or introduction of multiplexing), again this is simply a matter of substituting and re-setting the connector associated to the OPCS.

Listing 3.3: Application of the Inversion of Control pattern in the specification of an OPCS.

```
1  -- in producer.ads
2  package Producer is
3
4     type Producer_FC is new Controlled with private;
5     type Producer_FC_Ref is access all Producer_FC'Class;
6     -- [code omitted]
7
8     -- Called on initialization of the OPCS
9     procedure Set_x(This : in out Producer_FC; c : in Consumer.Consumer_FC_Ref);
10
11    -- Operation of the OPCS
12    procedure Produce(This : in out Producer_FC);
13
14    private
15
16     type Producer_FC is new Controlled with record
17        x : Consumer.Consumer_FC_Ref;
18     end record;
19
20 end Producer;
```

[5] http://www.martinfowler.com/articles/injection.html

Listing 3.4: Application of the Inversion of Control pattern in the body of an OPCS.

```
1  -- in producer.adb
2  package body Producer is
3
4     -- [code omitted]
5
6     procedure Set_x(This : in out Producer_FC; c : in Consumer.Consumer_FC_Ref) is
7     begin
8        This.x := c;
9     end Set_x;
10
11    procedure Produce (This : in out Producer_FC) is
12    begin
13       -- do useful stuff
14       This.x.Consume([Parameters]);  -- The actual call is performed on the connector
15                                      -- passed upon initialization of the OPCS
16    end Produce;
17
18 end Producer;
```

Thread: cyclic release pattern Our code archetype for cyclic tasks allows the creation of a cyclic task by instantiating an Ada generic package, passing the operation that it has to execute periodically.

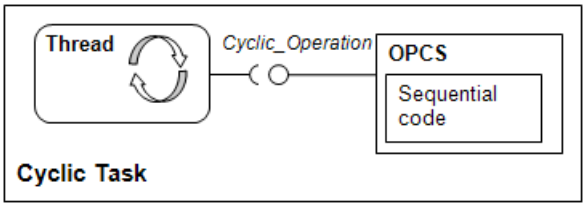

Figure 3.31: Structure of a cyclic task

The archetype is quite simple. In fact, we only need to create a task type that cyclically executes a given operation with a fixed period.

The specification of the task defines the task type for the cyclic thread. Each thread is instantiated with a statically assigned priority and a period which stays fixed throughout the whole lifetime of the thread. The task type is created inside a generic package, which is used to factorize the code archetype and make it generic on the cyclic operation.

Listing 3.5: Archetype for a cyclic task (specification)

```
1  with System; use System;
2  with Ada.Real_Time; use Ada.Real_Time;
3
4  generic
5     with procedure Cyclic_Operation;
6  package Cyclic_Task is
7
8     task type Thread_T
9        (Thread_Priority : Priority;
10        Period : Positive) is
11       pragma Priority (Thread_Priority);
12    end Thread_T;
13 end Cyclic_Task;
```

The Ada body is specified as follows:

Listing 3.6: Archetype for a cyclic task (body)

```
1  with Ada.Real_Time;
2  with System_Time;
3
4  package body Cyclic_Task is
5
6      task body Thread_T is
7        use Ada.Real_Time;
8        Next_Time : Time := System_Time.System_Start_Time;
9        begin
10          loop
11             delay until Next_Time;
12             Cyclic_Operation;
13             Next_Time := Next_Time + Milliseconds (Period);
14          end loop;
15    end Thread_T;
16 end Cyclic_Task;
```

The body of the task is composed by an infinite loop. Just after elaboration, the task enters the loop and is immediately suspended until a system-wide start time (*System_Start_Time*). This initial suspension is used to synchronize all the tasks that are to execute in phase and let them have the first release at the same absolute time.

When resuming from the suspension (which notionally coincides with the release of the task), the task contends for the processor and executes the *Cyclic_Operation* specified at the moment of the instantiation of its generic package. Then it calculates the next time it has to be released (*Next_Time*) and as first instruction of the subsequent loop, it issues a request for absolute suspension until the next multiple of its period.

The code archetype is simple to understand, yet a few comments are in order.

Firstly, the reader should note that the *Cyclic_Operation* is parameterless. That is not much of a surprise, as it is consistent with the very nature of cyclic operations which are not requested explicitly by any software client.

Secondly, this version of the cyclic task assumes that all tasks are initially released at the same time (*System_Start_Time*). Support for a task-specific offset (phase) is easy to implement: we just need to specify an additional Offset parameter on task instantiation, which is then added to *System_Start_Time* to determine the time of the first release of the task. The periodic release of the task will then assume the desired phase with respect to the synchronized release of the tasks with no offset.

Finally, we must stress that the use of *absolute time* and thus of the construct **delay until** (as opposed to *relative* time and the construct **delay**) is essential to prevent the actual time of the periodic release to drift.

The latter characteristic of the archetype, combined with the use of a Ravenscar-compliant execution platform ensures provision 1 for property preservation (cf. sec. 2.3.3): ”enforcement of the timing properties that are to stay constant during execution”.

Our archetype should cater for the two remaining provisions for property preservation: ”monitoring of the timing properties that are inherently variable in so far as they are influenced or determined by system execution”; and ”trapping and handling of the violations of asserted properties”.

In the following we show how it is possible to extend the archetype for the monitoring of the worst-case execution time of a task, in accord with the latter provisions.

Ada 2005 provides a simple yet efficient mechanism to monitor the execution time of tasks, to detect WCET overruns and to react in a timely fashion to a violation event: *execution-time timers*.

Every individual task can be attached to a timer that monitors the CPU time that is consumed by the task. At each task activation the timer is set to expire whenever the task exceeds its allotted CPU time (which is meant to be its WCET). In the event of a WCET overrun, a `Timer_Handler` protected procedure is immediately executed. As per an implementation advice of the Ada reference manual, the procedure is typically executed in the context of the interrupt service routine that acknowledges the clock interrupt.

Listing 3.7: Cyclic task with monitoring of WCET overrruns.

```
1  -- [code omitted]
2  loop
3
4     delay until Next_Time;
5     Set_Handler (WCET_Timer,
6                  Milliseconds (WCET),
7                  WCET_Violation_Handler);
8     Cyclic_Operation;
9     Next_Time := Next_Time + Milliseconds (Period);
10
11 end loop;
12 -- [code omitted]
```

In listing 3.7, we show how we can modify the loop of our cyclic task to use an execution-time timer. The handler *WCET_Violation_Handler* is fired when WCET overruns are detected.

The handler can be coded to notify the fault to the fault authority, which will take appropriate measures to remedy to the timing fault according to its severity, frequency and criticality of the affected system function [93].

Additional goals Our goal is to refine those code archetypes to lift some of their current limitations and introduce some important improvements. The areas we are investigating are:

1. to allow the change of the period or MIAT of a task;

2. to support a more flexible allocation of operations to tasks, by enabling the execution of multiple periodic operations by the same task executor (under the condition of fulfilling their periodic nature);

3. to support configurable release protocols at the OBCS;

4. to support the implementation of the infrastructural part of the PUS services.

Items 2 and 3 are the implementation-level realization of two needs that we described in section 3.6, i.e. user-defined allocation of components to containers and user-defined task release protocols.

3.8.2.5 Component packaging

In the description of the component model we identified the component implementation as the packaging unit of the approach.

In our current implementation we do not have support for packaging yet. The provisions required to the component repository would be: (i) storage of the source/object code of a component implementation; (ii) extraction from the user model of the information relative to the component implementation (and its type) and storage of them in the component implementation; (iii) extraction of the model information from the component repository and inclusion in the user model under development.

The latter case would be used in the case of reuse of a component implementation. The realization of this packaging concerns was postponed for the sake of priorities and also because it can be considered mainly a technological problem.

3.9 An alternative implementation using the UML MARTE profile

In the timeframe of this PhD we were also involved in another initiative which aims at the realization of goals comparable to those that triggered the investigation narrated so far: the CHESS project. The project was kicked-off in early 2009 and is due to complete at the beginning of 2012.

CHESS (*Composition with Guarantees for High-Integrity Embedded Component Assembly*) is an international project which aims at the design of a component-based methodology for the development high-integrity real-time systems. CHESS targets primarily the space, telecommunication and railway domains; the applicability of the methodology to the automotive domain is also being investigated. Industrial partners of the various domain are involved in the project, and they are useful to obtain feedback on the approach during its development.

A sizeable set of the goals of CHESS are comparable to those that drove the investigation narrated in this PhD thesis. We strove to keep aligned the methodological framework of CHESS with that of the present investigation. Indeed, CHESS is based on the same overall approach described in section 2.3. We also were able to promote in CHESS the component model described in this chapter.

CHESS elected UML as the baseline for its design language, named CHESS Modeling Language (CHESS-ML). More precisely, CHESS-ML is an extension of a subset of the UML MARTE profile [99].

The project team considered the creation of a new domain-specific language and the construction of a complete development platform from scratch too onerous; evidently, the greater complexity of implementation of C-by-C practices and separation of concerns and the need to reconcile the intended component-based approach with the (in part immutable) structure of the UML metamodel were considered less risky.

MDT-Papyrus[6] – extended with CHESS-specific plugin – was chosen as the development platform. MDT-Papyrus is based on the Eclipse platform which relies on the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF). MDT-Papyrus is becoming the reference Eclipse-based editor for UML models. It supports the definition of UML profiles, and already includes an implementation of the MARTE profile.

3.9.1 Main challenges and comparison with the DSL approach

Among the various challenges to be faced by the CHESS project, the most interesting among those we are personally involved include:

[6] http://www.eclipse.org/modeling/mdt/papyrus/

- the creation of CHESS-ML, the language for the specification of components;
- the implementation of design views;

In the following we briefly comment on them.

Creation of the design language. The CHESS team performed an initial iteration for the creation of CHESS-ML. At the present project stage, the design language is only partially able to express the component model described in this thesis.

For this reason, we do not consider it very interesting to describe the language in detail and provide a full mapping to the entities of the component model; we prefer to highlight in general terms the criteria that we adopted for the creation of the CHESS profile.

As it was anticipated, CHESS-ML is based on an extension of a subset of the UML MARTE profile. The structure of the MARTE profile is modular: the profile is further divided in a set of subprofiles that represent different aspects addressed by the profile (e.g. HLAM for high-level application modeling, SAM for schedulability analysis, PAM for performance analysis, etc...). In principle one could decide to trim the profile definition and retain only the aspects of interest. In practice however, the intricate dependencies of the profile and the inevitable dependencies with the base packages (e.g. VSL Value specification language, NFP non-functional properties, etc...) make the tailoring at the profile level inconvenient. A considerable part of the language tailoring is therefore enacted with OCL constraints that statically complement the metamodel structure and on-the-fly checks. We discuss them in the next paragraph.

For what concerns the language itself, we want the user to operate exclusively at a PIM level (which features the same amount of views as presented for the DSL approach, plus a dedicated view for dependability concerns). Even in this case, we want the implementation and analysis model to be automatically generated by the design environment (cf. Figure 3.29).

The design language was thus engineered to offer to the user exclusively high-abstraction entities from UML (interfaces, classes, components, etc...); high-level constructs from the GCM and MARTE_Foundations subprofile of MARTE (such as $<< ClientServerPort >>$ and $<< assign >>$) and extensions of UML components via dedicated stereotypes ($<< ComponentType >>$, $<< ComponentImplementation >>$). Lower-level or analysis entities (such as $<< SwSchedulableResource >>$, $<< SaAnalysisContext >>$, $<< SaEndToEndFlow >>$) are included only in the automatically generated PSM model. Entities were extended (by use of stereotyped specializations) when they did not provide all the attributes we need for model transformation, analysis or code generation.

Implementation of design views. The implementation of design views in the CHESS context is quite challenging and the picture more complicated than explained in section 3.3 for two reasons: (i) MDT-Papyrus does not provide the primitive support we need for the realization of

design views; (ii) as a consequence, the notion of view had to be expressed at the level of the UML language.

UML can be extended with the profile mechanism, which permits: (i) to forbid the use of modeling entities, thereby singling out a language subset; (ii) to introduce new design entities by definition of *stereotypes*, a sort of "tag" that specializes an already existent metaclass; (iii) to apply additional constraints, expressed in OCL (Object Constraint Language) [100], to determine the well-formedness of a model against specific criteria.

In CHESS we adopt an approach similar to SysML [102], which tried to provide a basic separation and allocation of design entities to views (see fig. 3.32).

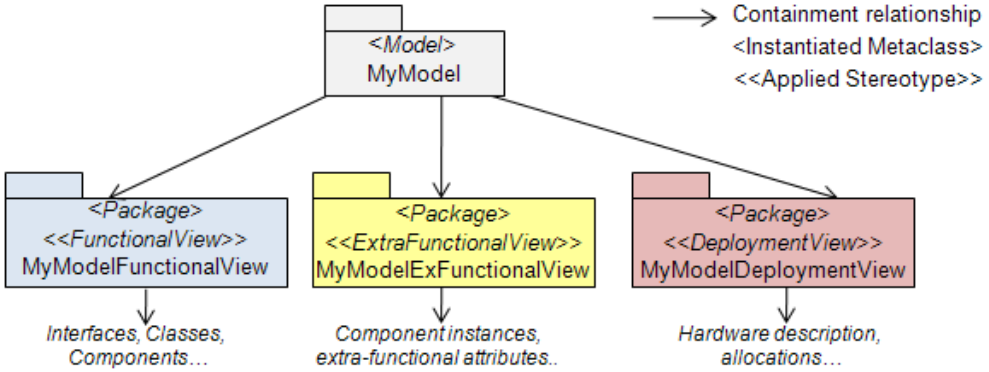

Figure 3.32: Realization of design views in UML as packages with an applied stereotype.

When creating a new CHESS model, the model structure is initialized by creating a set of stereotyped packages that represent all the design views of interest. When we create a new design entity (an instance of a metaclass), we place it in the package subtree of a view.

A design view is realized as an entity of the CHESS design language itself and it is instantiated on each model. The concept of view is therefore unnecessarily coupled with the underlying organization of the model.

An implementation of a design view needs the encoding of:

1. view identification and permissions: its name and identifier and all the contained entities (UML elements, their associations and features) which constitute the concern(s) it address;

2. relationship between diagrams and their owning view;

3. the integrity constraints: the rules which assure the well-formedness of the model and are enforced or checked by the view;

The design view is then complemented with a set of automation capabilities the goal of which is twofold:

1. aid the user by automatically creating complex design entities with a single action (with a tool provided in a dedicated palette);

2. warrant for inter-view consistency. For example it may be necessary to import in the current view entities defined in another (with new visualization and modification rights) and refine them (e.g. adding internal elements or setting references). The automation capabilities can deliver the user from the burden of those actions and responsibility of ensuring the correctness of cross-view references.

To date, the CHESS team, of which the author is a member, created a set of plug-in for MDT-Papyrus which: (1) automatically creates the desired stereotyped packages to initialize design views upon model creation; (2) provide an internal representation of design views; (3) engineer the desired relationship between views and diagrams; (4) implement a preliminary set of integrity constraints; (5) provide specialized palettes with the intent of including only the model entities that belongs in CHESS-ML and exclude non-admissible UML/MARTE entities.

For what concerns the implementation of integrity constraints, they are of two kinds: (i) OCL constraints that complement the CHESS profile; and (ii) on-the-fly checks.

The first kind of checks are specified to complement the metamodel of the CHESS language; unfortunately the compliance of a model to the constraints is evaluated only when the user explicitly requires the model verification. The constraint is evaluated through the standard EMF validation framework. The drawback of this solution is that the designer may introduce errors in the design which would be detected only at a later time, whenever they decide to perform a model verification. That would incur potentially costly backtracks in the design.

For that reason, we perform the most critical checks *on-the-fly* (e.g. when we establishing a binding between a required and a provided interface, we check if the interface at the provided side is compatible with the interface at the required side.) The on-the-fly constraint verifier is partly based on the EMF Model Transaction listener and is implemented in Java code.

The CHESS team was able to engineer the relationship between diagrams and their owning view in the underlying MDT-Papyrus infrastructure. That is quite interesting as it is possible to internally recognize the view in which the designer is working. Hence on-the-fly checks can be allocated to views, as we advocated in 3.3, and only the checks of the current view are verified, thus reducing the computational cost of the verification framework.

Discussion. The main difference between our implementation of the component model as DSL, and the on-going implementation in CHESS consists in the different areas where the intellectual and engineering effort is placed.

As regards the DSL, the main intellectual effort is the creation of the metamodel; the engineering effort is mainly the creation of the development environment. As we have described, in our case the effort to be put in the latter was greatly mitigated by using a framework designed for the specification of modeling editors.

As for CHESS-ML, the main intellectual effort is to express the component model with the UML/MARTE profile language: from the syntactic point of view, the challenge is reconciling the component model with the partly immutable structure of the UML/MARTE metamodel and the fixed set of UML diagrams; from the semantics point of view, the challenge is to express the intended semantics of component model entities (with particular regard to extra-functional dimensions) with existing entities, possibly with semantics defined by the UML and MARTE standards. Another concern is the complementary provision of OCL constraints to tailor CHESS-ML. The main engineering effort consists in the implementation of on-the-fly checks and engineering of design views. The latter mainly because views have to be realized with language entities – hence at the wrong modeling level – and for a considerable extent directly in the design editor.

In both cases, design views are not fully implemented as we need. In the DSL implementation, design views are technologically realized as first-class entities in the modeling framework we use. Unfortunately we still lack in that framework the association of attributes (as opposed to complete entities) to design views, and fine-grained read-write permissions and property presentation according to the current view. In CHESS, we currently support design views as stereotyped packages. As a shortcoming, the position of an entity in the model determines the view to which it belongs. We have designed the support for fine-grained read-write permissions and association of on-the-fly checks to views; but implementation has not started yet.

At the time of this writing the team of MDT-Papyrus is working on some interesting extension to their platform. The work is centered on engineering *layers* for diagrams. Each layer contains a set of entities (normally allowed by the diagram) and the specification of the visibility of their attributes as well as rules to customize the presentation of them. Layers can be activated or deactivated for a diagram and the final representation depends on the contents of the stacked layers that are active.

This new feature is planned for future versions of MDT-Papyrus and will certainly simplify the engineering of views as we described. It is however not a full-fledged implementation of views, and confirms the need of a CHESS-specific implementation of rules for read-write permissions, and the framework for view-dependent constraint enforcement.

In both the DSL and CHESS contexts, the work on design views relies on a technological solution. Therefore it is inherently non-portable. The problem is that there is no concept of design views in the mainstream modeling structure, which is based exclusively on the model/meta-model/meta-meta-model hierarchy. The problem would be solved only with a theoretical framework for the creation of design views that complements that hierarchy. Design

views would be first-class entities, conceptually specified in a structure similar to that of a metamodel, that would be able to reference entities of the meta-model level. Reference frameworks for the creation of design editor (an equivalent of the current EMF/GMF) could use that information to automatically create editors with design views, customized with visibility and write-permissions according to their description.

As regards model-based schedulability analysis, there are no significant differences between the two approaches. One of the input sources for analysis information is a SAM model specified with the DSL, the other is a SAM model specified with MARTE and CHESS-specific extensions.

For code archetypes, we plan to use exactly the same archetypes. Code generation will differ only in the extraction of information from the PSM used as input.

Chapter 4

Evaluation

In this chapter we provide some elements for an evaluation of the work described in this thesis and the proposed component model in particular.

In section 4.1 we describe a case study for the use of the approach: the re-engineering of EagleEye, a small-sized Earth observation mission used as test case by ESA in their simulation framework for on-board software.

In section 4.2 we recapitulate the technical requirements that have to be fulfilled by the software reference architecture. We summarize which elements of our investigation fulfill each requirement. We also comment on possible remaining work that is necessary for the complete fulfillment of the requirement.

In section 4.3 we summarize the feedback we already received by ESA and other stakeholders on the approach in general and on the component model in particular.

4.1 Case study: re-engineering the EagleEye Earth observation mission

”EagleEye” was a very small, yet representative Earth observation mission that was commissioned as a conceptual test case for ESA’s Virtual Spacecraft Reference Facility (VSRF). The VSRF is a simulation framework with the goal of providing a representative environment for the test of on-board software. EagleEye was designed and developed in the period 2003-2005.

EagleEye is a small Earth observation satellite (250 Kg) operating on a Sun-synchronous orbit. It carries a single payload (called ”GoldenEye”) which consists in a high-resolution imaging camera with off-nadir capabilities (the camera pointing is not restricted to the field of view immediately below the satellite). The payload is commanded by the main on-board computer (OBC) via commands sent on the 1553B command bus. The OBC is equipped with

an ERC32 radiation-hardened processor[1], elaborating 17 MIPS at a frequency of 24 MHz.

The imaging data are stored on a solid state mass memory (100 GB) via a high-speed link.

The satellite is equipped with an S-band antenna for the reception of telecommands (TC) from ground and the downlink of telemetry (TM). The uplink speed is 2 KBytes/sec, the downlink speed is 4 KBytes/sec. An X-band antenna is used to downlink images stored on board when ground coverage is available, with a maximum capability of 100 Mb/sec.

In figure 4.1 we depict a simplified version of the system architecture of the satellite.

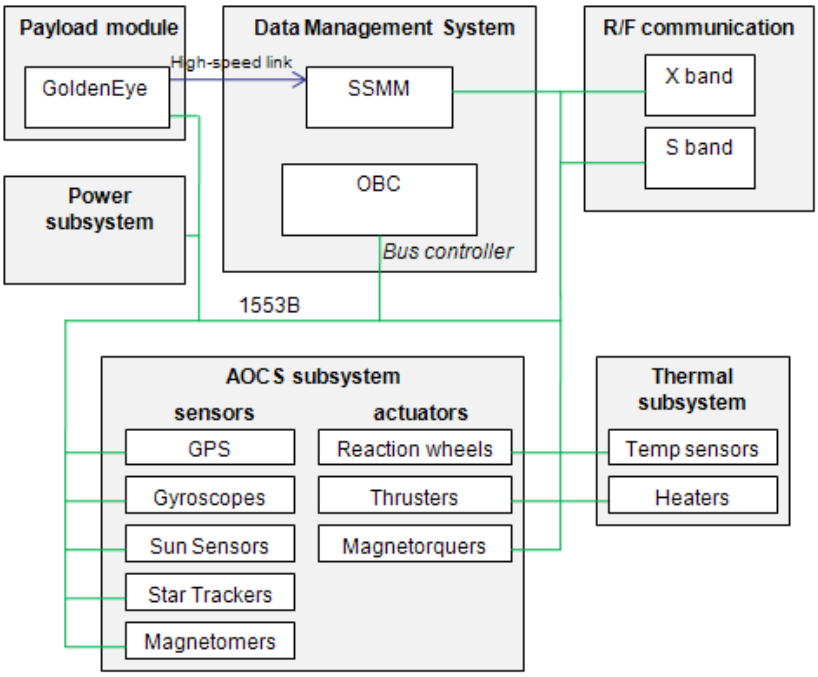

Figure 4.1: System architecture of the EagleEye satellite

The main on-board computer (OBC) is part of the Data Management System.

A MIL-STD-1553B bus connects the OBC to all the other spacecraft subsystems: the AOCS subsystem, the thermal subsystem, the power subsystem, the R/F communication subsystem and the payload module. The sensors of the AOCS subsystem comprise: 3 star trackers in hot redundancy, 2 gyroscopes in cold redundancy, 2 sun sensors in hot redundancy, 2 magnetometers in hot redundancy, and 2 GPS receivers in cold redundancy. The actuators of the same subsystem comprise: 3 magnetorquer bars in cold redundancy, 4 reaction wheels used in hot redundancy,

[1]The ERC32 is the previous generation of SPARC processors adopted for platform modules in ESA missions. It was the reference processor in the years of development of EagleEye.

and 4 thrusters (4.5N of thrust each). In figure 4.1 we do not depict the internals of the power subsystem and redundant equipment for the sake of simplicity.

The 1553B is a master-slave bus. There is only a single bus controller at any one time for the whole bus; in our case, the bus controller is the OBC. All other connected ends are slaves (termed ”remote terminals” or RT). An RT does not send messages on the bus autonomously; it only replies to requests from the bus controller.

The platform software comprises three main applications: the AOCS software, the thermal control software and the power control software. In figure 4.2 we depict a high-level view of the on-board software.

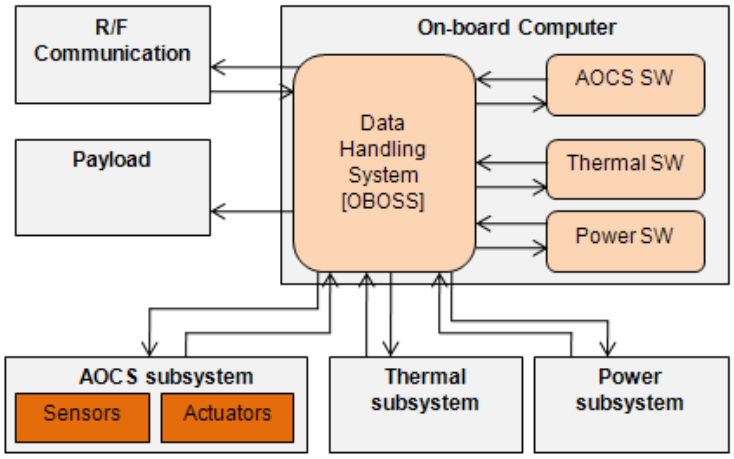

Figure 4.2: High-level view of the Eagle Eye on-board software

The three applications need to communicate with their controlled subsystems, in particular to acquire data from sensors and send commands to actuators. Those communications are carried out by sending commands on the 1553B bus.

There is neither a separate payload software, nor a separate payload computer in this (”relatively”) simple case study; control of the payload is directly embedded in the AOCS software.

All the communications (hence, setting and getting of parameters) are mediated by the Data Handling System. OBOSS [139] is used as the data handling framework. Each software application that sends or receives telecommands or telemetry (termed ”Application Process”) instantiates and in part configures by specialization the implementation of the PUS services (cf. section 3.7.2) provided by OBOSS.

Unfortunately this code-centric approach flattens the design to the same low level of abstraction of the data handling system. In fact the interfaces of the software subsystems (as those of the sensors and actuators) are expressed at a low abstraction level, that is in terms of PUS services and PUS packets exchanged with the data handling system. For example the Interface

Control Document (ICD) of the AOCS software contains a long list of parameters required by the AOCS software (expressed in the form of TM packets received from sensors, and partly actuators) and a corresponding list of parameters sent by the AOCS software to actuators (expressed in the form of TC).

We decided to use EagleEye as a case study to investigate the applicability of the component model described herein and to assess the current status of realization of our design environment. This activity will also help us to understand if we are able to increase the abstraction level of the design process.

Hardware specification. The first activity that we perform is the creation of the model representation of the hardware architecture of EagleEye.

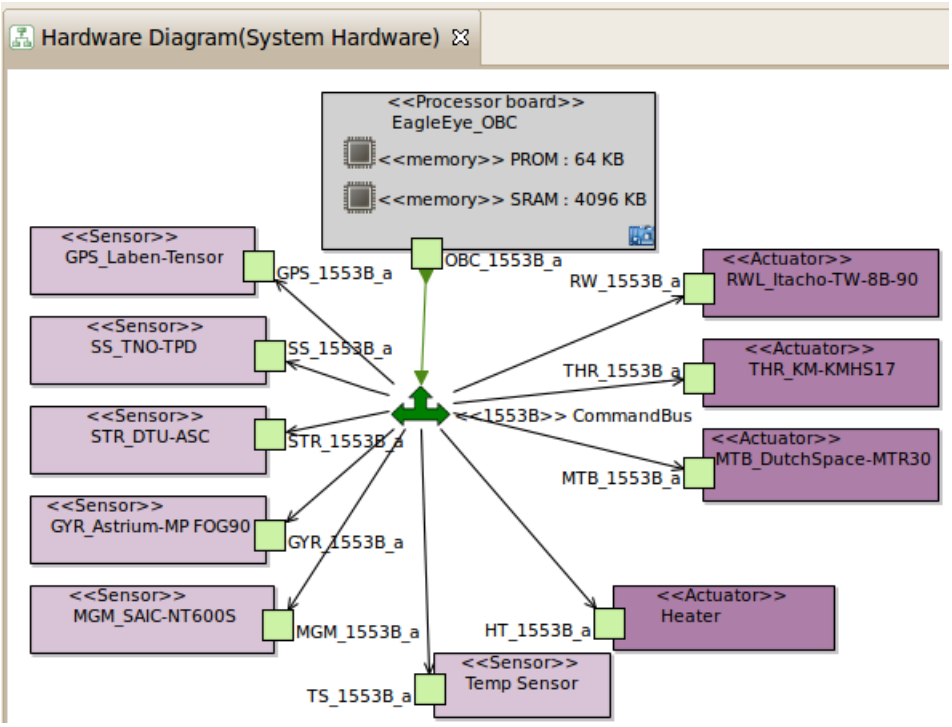

Figure 4.3: Hardware diagram for a subset of the EagleEye system architecture.

In figure 4.3 we depict a representation of a subset of the hardware architecture; the reader should keep in mind that this is done to keep the example as compact as possible, so as to facilitate the visualization of figures in this thesis, rather than for representation problems.

The figure depicts the on-board computer connected through a 1553B bus to the sensors and actuators of the AOCS and thermal subsystems (cf. 4.1). The on-board computer is equipped with a PROM of 64KB that stores the boot software and 4MB of SRAM, used by the on-board

software. The processor is not depicted in the figure but is defined in the attributes of the processor board.

The design environment checks that only a single bus controller has been defined for the bus (port OBC_1553B_a in our example); the remaining connection ports are considered 1553B remote terminals.

Functional specification. We then proceed with the definition of the data types and interfaces. In figure 4.4, we depict a subset of the interfaces that we define for EagleEye.

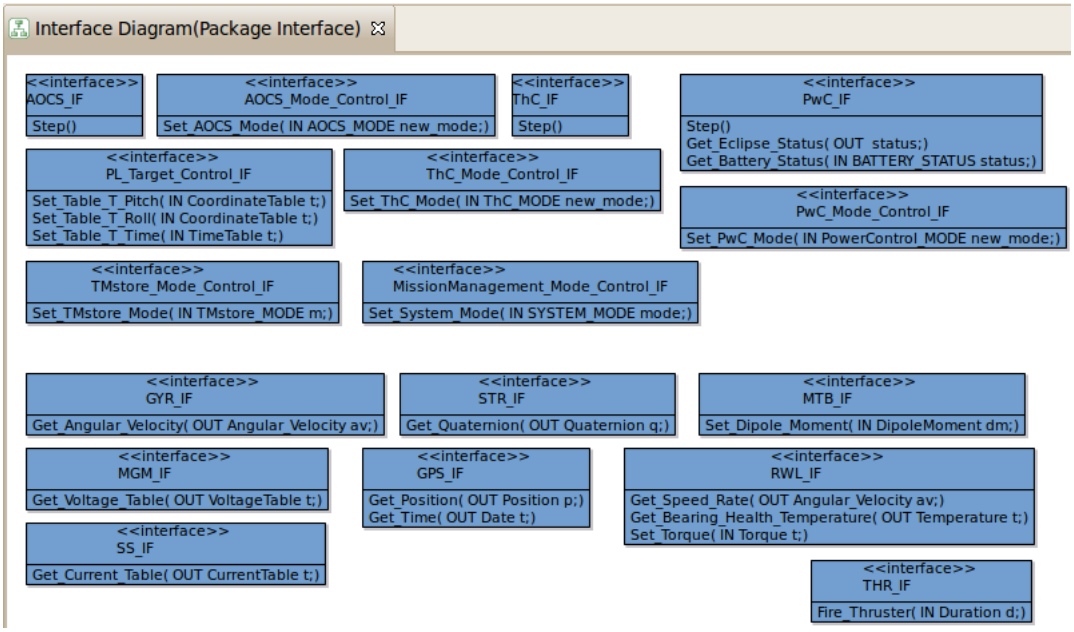

Figure 4.4: Interface diagram for a subset of the EagleEye interfaces.

On the top, there are nine interfaces related to software components: *AOCS_IF* and *AOCS_Mode_Control_IF* which enlist the services to operate the AOCS and to control its operational mode; *PL_Target_Control_IF* to control the activities of the payload; *ThC_IF* and *ThC_Mode_Control_IF*, to operate and control the operational mode of the Thermal Control; *PwC_IF* and *PwC_Mode_Control_IF*, to operate and control the operational mode of the Power Control; *TMstore_IF* for the management of telemetry; *MissionManagement_Mode_Control_IF* for the management of operational modes of the system.

On the bottom, there are five interfaces related to sensors and three related to actuators (related to the AOCS subsystem). For simplicity, we omit the definition of interfaces for sensors and actuators related to the Thermal and Power subsystems and the control interfaces for all those equipments (to power on or off the equipment and perform other equipment checks or tuning).

In figure 4.5 we depict the component types defined for the AOCS and the sensors and actuators of the AOCS subsystem.

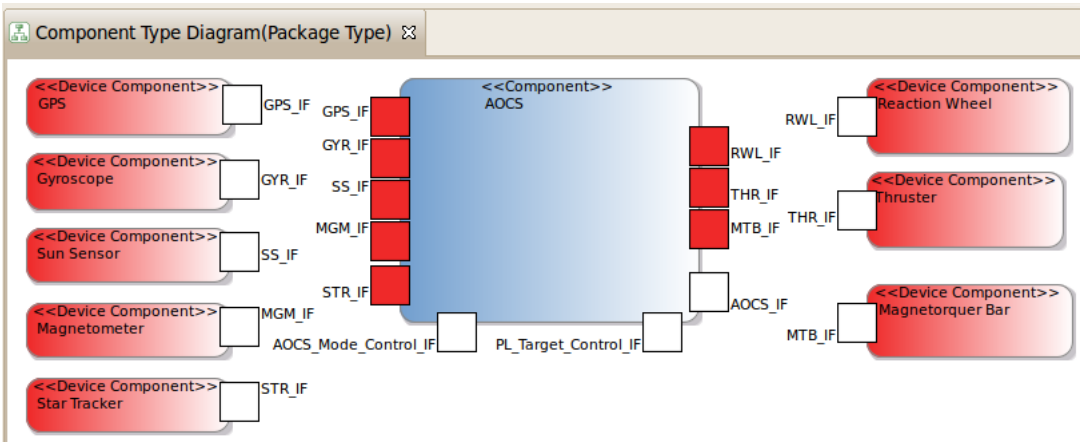

Figure 4.5: Component type diagram for the EagleEye AOCS subsystem.

The AOCS component type provides three interfaces (depicted in white color): *AOCS_IF*, *AOCS_Mode_Control_IF* and *PL_Target_Control_IF* (as in this satellite, the payload is directly managed through the AOCS). The AOCS exposes eight required interfaces (colored in red), five to gather from sensors the input data to perform its computations, three to send commands to the actuators to control the satellite plant.

We then define one component type for each sensor and actuator. Sensors are depicted on the left of the AOCS, actuators on the right. Each sensor or actuator provides the corresponding interface that we defined in figure 4.4. In the case we should also support the control interfaces for these devices, each component type would simply expose another provided interface.

Adding provided or required interfaces is easy. It is sufficient to select a tool in a dedicated palette and click on the component type of interest; a pop-up menu shows the list of all the already defined interfaces and the designer can select one ore more interfaces to add either as provided or required interfaces.

The implementation level for our model is identical to the type level. We simply derive a single implementation for every component type. There would be then one AOCS component implementation, one Gyroscope component implementation, and so on. In a real development process, the implementation level can be used to create different implementations of the same component to host different version of the source code; and to declare technical budget for implementations, possibly delegated to subcontractors.

In figure 4.6 we represent the instantiation of the AOCS and all its related devices.

A component instance is automatically created with a set of PI and RI inherited from its implementation. At this level, required interfaces can be bound with compatible provided in-

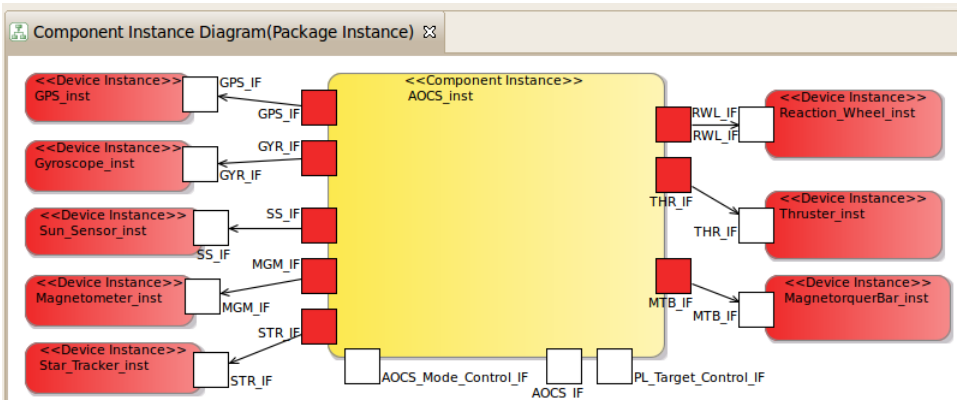

Figure 4.6: Component instance diagram for the EagleEye AOCS subsystem.

terfaces. In our example, we bind all the RI of the AOCS with the corresponding PI in device instances. The design environment actively ensures (via static type checking) that only compatible interfaces are connected.

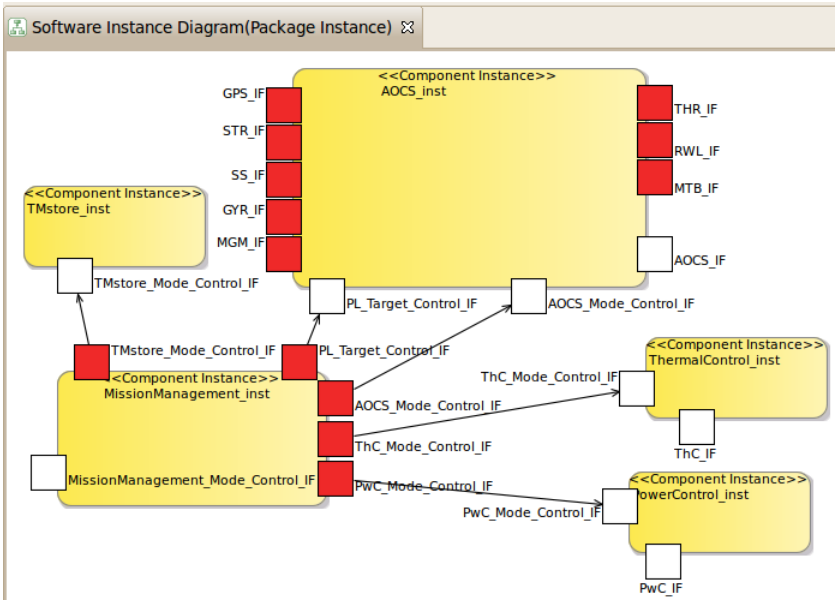

Figure 4.7: Component instance diagram for the software components of EagleEye. Visualization of device instances is selectively deactivated.

In figure 4.7, we instead show all the *software* component instances of our case study: a mission manager, which is connected to the ”Mode_Control” PI of the three main applications (the

AOCS, the thermal control and the power control), to a component for the storage of telemetry data, and to the PI for the configuration of the payload's targets.

Intra-component bindings. Let us see how we can specify intra-component bindings. Bindings are set on a component implementation and are valid for all its instances. In figure 4.8, we show the bindings for operation *Set_System_Mode* of the component implementation of the MissionManagement.

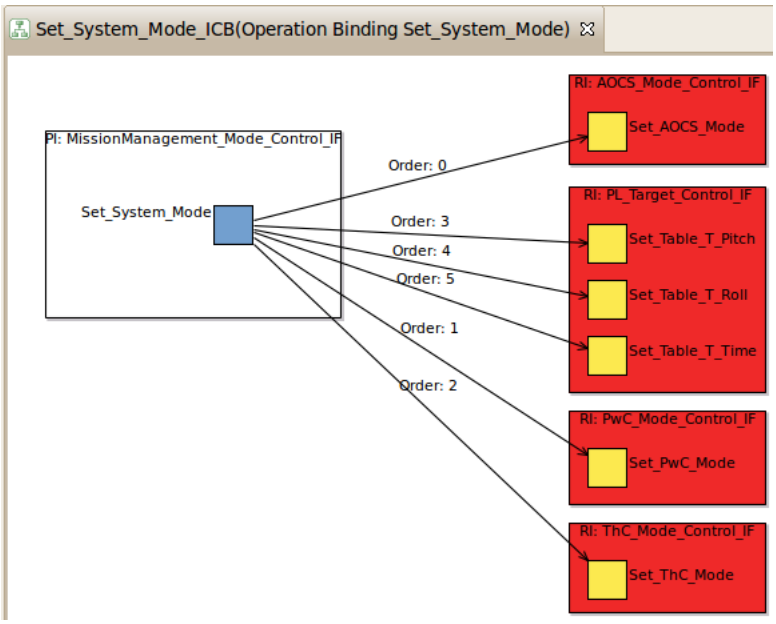

Figure 4.8: Intra-component bindings for operation *Set_System_Mode*.

As the diagram shows, we can specify which operation of which RI is called by an operation on a PI, and in which order. In the depicted scenario, our operation of interest happens to call all the operations of the RI of its component implementation. It first calls *Set_AOCS_Mode*, then *Set_PwC_Mode*, and then *Set_ThC_Mode*, to set new operational modes of the three applications. Finally it calls all the operations in the RI *PL_Target_Control_IF*, to set new target acquisitions for the payload.

Instance deployment. After instances have been defined we can proceed with two activities: deployment of instances and decoration with extra-functional attributes. Let's have a look to deployment first. Deployment of component instances is performed using a simple table in which for every component instance we specify the processor board on which it is deployed on.

We decided to use a simple table (figure 4.9) for deployment instead of a graphical diagram (like in UML2), so as to reduce the graphical clutter resulting from the presence of tens of

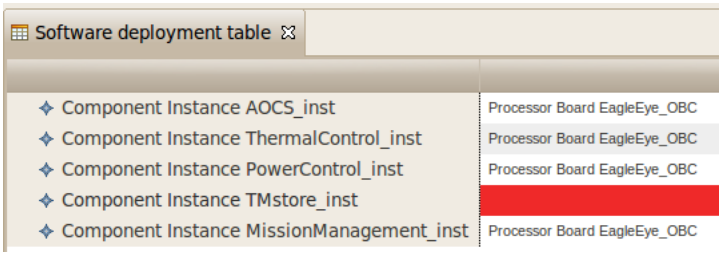

Figure 4.9: Deployment table for the EagleEye component instances.

instances to be deployed typically on one to four processor boards (typically two, excluding redundancy). The table outlines in red the instances that have not been deployed yet. When the designer verifies the model in the deployment view, the list of instances not deployed on any processor board are reported in the ”problem view” of Eclipse as the result of the ”model validation”.

Figure 4.10 shows the table for the deployment of device instances to hardware devices.

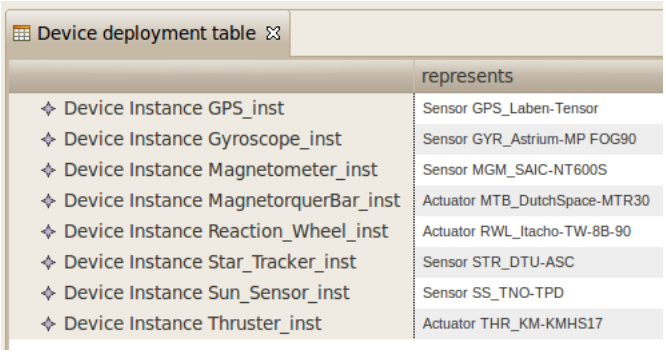

Figure 4.10: Deployment table of the EagleEye device instances on hardware devices.

The reader may remember that we needed a representation of devices at component level in order to relate software components to devices to determine the functional relationships and for the generation of communication code (cf. section 3.5). We then have to deploy each device instance on its real counterpart, which has been defined in the hardware architecture.

The dichotomy with type (and implementation) and instances is particularly useful for the modeling of redundant devices. In fact, it is sufficient to create only a single type (and implementation) for each device kind, and then as many device instances as required. Each instance will then be deployed on the appropriate hardware device counterpart.

Concurrency and real-time attributes and schedulability analysis. We can now define the basic concurrency and real-time attributes on the component instances. We then show how it is possible to perform schedulability analysis of the system, based on this information. Single processor and distributed analysis is supported using the MAST schedulability analysis tool [133]. We will not present instead the analysis of the 1553B schedule and bandwidth, as at the moment we only have partial support for them in the design environment.

When the extra-functional view is activated in the editor, then it is possible to right-click on a provided interface of a component instance and create a new table for the specification of extra-functional attributes. The table adds an "extra-functional descriptor" to the provided interface that stores the annotations of the extra-functional attributes for each operation of a PI. Figure 4.11 shows the extra-functional descriptor for the *AOCS_IF* PI.

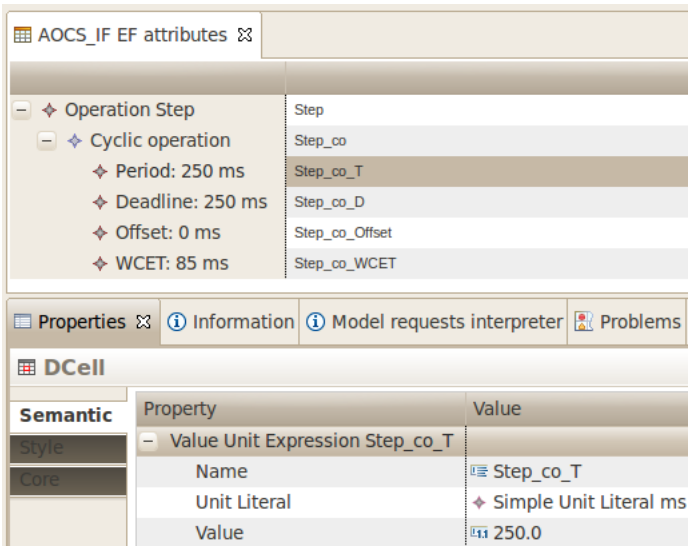

Figure 4.11: Extra-functional descriptor for the *AOCS_IF* provided interface in *AOCS_inst*.

The interface contains a single operation *Step*, that is properly represented in the table and is the entry point of the main AOCS control law. According to the EagleEye documentation, we declare this operation as a cyclic operation, with a period of 250ms, an offset of 0 (the phase with respect to the synchronous release of all tasks at system start time) and a deadline equal to the period. The control law that is implemented has in fact a frequency of 4Hz.

It is possible to similarly specify the concurrency and real-time attributes for the operation *Set_AOCS_Mode()*, relative to the provided interface *AOCS_Mode_Control_IF*. The operation was declared sporadic, with a MIAT of 1000ms and a deadline of 250ms.

In figure 4.12 instead we show the extra-functional attributes assigned to the operations of the provided interface *PL_Target_Control_IF* of the AOCS instance.

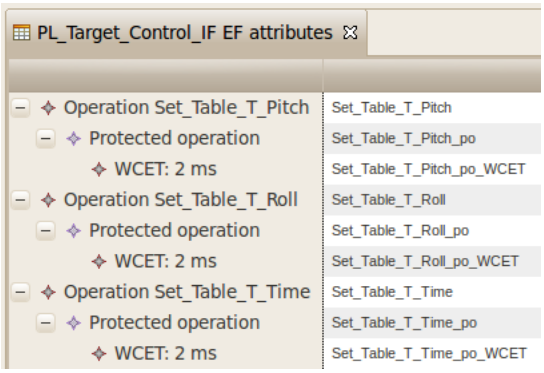

Figure 4.12: Extra-functional descriptor for the *PL_Target_Control_IF* provided interface in *AOCS_inst*.

The interface is used to configure a set of new target acquisitions by the imaging camera of the payload; this is done by uploading separately a set of coordinates for the pointing of the camera and a set of time instants when the images shall be captured. We specify that all the three operations are protected operations in order to prevent race conditions due to the concurrent reading and writing of the information on the targets.

We can link the operations of the interface *PL_Mode_Control_IF* to the internal state of the component on which they operate. We also create another corresponding interface with equivalent getter operations on the same internal state; and we decide that it is available (for some reasons) only internally to the AOCS component. For the moment we have to specify all these conditions manually, but we plan to engineer their automated configuration, when realizing the full automated support for getter and setter operations of component attributes as described in section 3.5. Finally we declare that operation *Step* of *AOCS_inst* is calling the operations of that getter interface.

In table 4.1 we show the complete set of concurrency and real-time attributes set on operations of each instance PI. The worst-case execution time shown in the figure is the own cost of the operation.

The call chain of an operation may span across components and ends at the first occurrence of an operation marked as deferred (whether sporadic or bursty), as those operations have their dedicated executor thread. Conversely, the execution of immediate operations (protected or unprotected) is performed in the context of the calling thread.

For this reason, the execution cost of a PI operation must include the execution cost of RI that are bound to a PI with immediate operations. In our example, this applies to operation *Step* of *AOCS_inst* and *Set_System_Mode* of *MissionManagement_inst*. The additional execution costs amount to 6ms for *Set_System_Mode* (*Set_Table_T_Pitch* + *Set_Table_T_Roll* +

Set_Table_T_Time) and 3ms for *Step* (*Get_Table_T_Pitch* + *Get_Table_T_Roll* + *Get_Table_T_Time*) and they are reported in table 4.1 in square brackets.

Table 4.1: List of concurrency and real-time attributes for EagleEye. T: period/MIAT D: relative deadline C: WCET *p*: assigned priority

Instance	**Operation**	**Concurrency kind**	**T**	**D**	**C**	***p***
MissionManagement_inst	Set_System_Mode	Sporadic	1000	250	10 [+6]	8
AOCS_inst	Set_AOCS_Mode	Sporadic	1000	250	25	7
AOCS_inst	Step	Cyclic	250	250	85 [+3]	6
ThermalControl_inst	Set_ThC_Mode	Sporadic	1000	250	15	5
ThermalControl_inst	Step	Cyclic	1000	1000	180	4
PowerControl_inst	Set_PwC_Mode	Sporadic	1000	1000	20	3
PowerControl_inst	Step	Cyclic	1000	1000	150	2
TMstore_inst	Set_TMstore_Mode	Sporadic	1000	1000	20	1
AOCS_inst	Set_Table_T_Pitch	Protected	-	-	2	-
AOCS_inst	Set_Table_T_Roll	Protected	-	-	2	-
AOCS_inst	Set_Table_T_Time	Protected	-	-	2	-
AOCS_inst	Get_Table_T_Pitch	Protected	-	-	1	-
AOCS_inst	Get_Table_T_Roll	Protected	-	-	1	-
AOCS_inst	Get_Table_T_Time	Protected	-	-	1	-

An automated transformation to produce a Ravenscar-compliant PSM generates the analysis/implementation view of figure 4.13. The transformation creates the containers, with the necessary cyclic or sporadic tasks and protected objects (shared resources with an access protocol) to warrant the realization of the extra-functional attributes. It also assigns *priorities* to tasks and calculates the *ceiling priority* for protected objects similarly as described in [21].

As regards the implementation entities produced for the *AOCS_inst* component instance, we can notice that other component instances (such as *MissionManagement_inst*) can access only the interfaces present on the *AOCS_inst* description. The parameter-less operation marked as cyclic is assigned to a dedicated cyclic task and cannot be called for obvious reasons: it is periodically triggered by the execution platform. The getter interface for the payload control is available and called only internally (by operation *Step*). For all generated containers, the call to one of their interfaces is either encoded in a request descriptor posted at the OBCS of the

relevant sporadic task or delegated to the interface of a protected object, where the required concurrent semantics is applied.

No case of unprotected operations is present in this example. In that case, the call would be simply executed by the thread of the caller without overimposing any concurrent semantics.

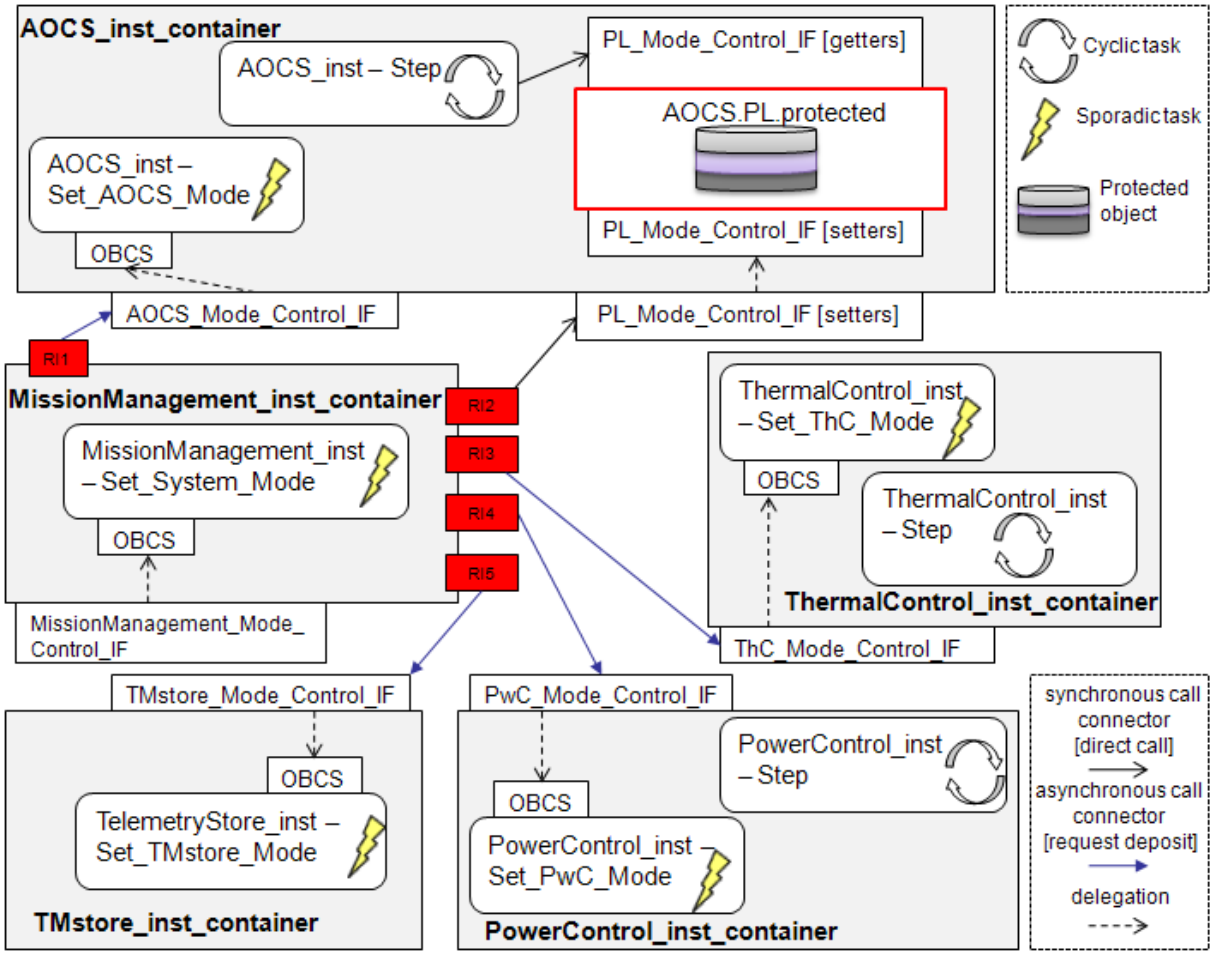

Figure 4.13: Analysis / Implementation view of EagleEye.

The kind of the connectors that bind RI to PI are determined by the automated generation to match the concurrency semantics at the *provided* side.

In this example (as all instances are deployed on the same processor), bindings to protected operations are transformed in direct synchronous calls; bindings to sporadic operations are transformed in asynchronous calls which deposit a request of execution in to the OBCS of the sporadic task). In the Ravenscar Computational Model, OBCS and protected objects are

equipped with the Immediate Ceiling Protocol (ICP), a derivative of Baker's Stack Resource Policy (SRP) [9], which features a worst-case blocking equivalent to the Priority Ceiling Protocol (PCP) [124]. The ICP statically guarantees absence of deadlock; minimizes the blocking time incurred by higher-priority tasks due to resource contention by lower-priority task; and ensures that a task can be blocked at most once during a single activation, and if blocking occurs, it is prior to the first execution of the task during the activation.

Information to perform schedulability analysis with MAST can be extracted from the analysis/implementation view. At the present stage we created manually the input file for MAST; it will be created automatically when the whole design environment is complete.

Table 4.2 shows the results of the schedulability analysis for the worst-case execution scenario: the mission manager issues a mode change request to all the other applications (AOCS, Thermal, Power, Telemetry store), triggering their sporadic tasks, and configures a new set of targets for the payload. The applications are still required to complete their pending (periodic) jobs.

Table 4.2: List of results of schedulability analysis. WCRT: worst-case response time B: worst-case blocking time

Instance	**Operation**	**WCRT**	**B**	**Schedulable**	**Ceiling priority**
MissionManagement_inst	Set_System_Mode	17	1	Y	-
AOCS_inst	Set_AOCS_Mode	42	1	Y	-
AOCS_inst	Step	129	0	Y	-
ThermalControl_inst	Set_ThC_Mode	144	0	Y	-
ThermalControl_inst	Step	412	0	Y	-
PowerControl_inst	Set_PwC_Mode	432	0	Y	-
PowerControl_inst	Step	670	0	Y	-
TMstore_inst	Set_TMstore_Mode	690	0	Y	-
AOCS_inst	Set_Table_T_Pitch	-	-	-	8
AOCS_inst	Set_Table_T_Roll	-	-	-	8
AOCS_inst	Set_Table_T_Time	-	-	-	8
AOCS_inst	Get_Table_T_Pitch	-	-	-	8
AOCS_inst	Get_Table_T_Roll	-	-	-	8
AOCS_inst	Get_Table_T_Time	-	-	-	8

The table shows the worst-case response time for each task, and its maximum blocking time incurred due to the use of synchronization protocols. We show in the table also the ceiling priority at which the protected operations are executed. For the sake of simplicity, we omit from the schedulability analysis the OBCS of sporadic tasks. However, OBCS are protected objects, hence the posting of requests in an OBCS (with a protected operation) contributes to the calculation of the maximum blocking time for tasks. Therefore, they should be included in the generated analysis scenario. As we anticipated in section 3.5, detailed knowledge of containers and connectors makes it possible to include information about their execution overhead in the analysis model, so that the analysis accurately reflects the software entities executed at run time.

The results of the analysis will then be propagated to the analysis model and then to the user model, where they can be presented to the designer as attributes of the design entities (component instances, processors, etc...), similarly to the extra-functional descriptor shown in figure 4.11. It is also possible to create a complete table that comprises all the results, so it is easier to check the fulfillment of all the timing requirements and use the table to generate project documentation.

Space-specific concerns In figure 4.14 we show our initial support for PUS concerns.

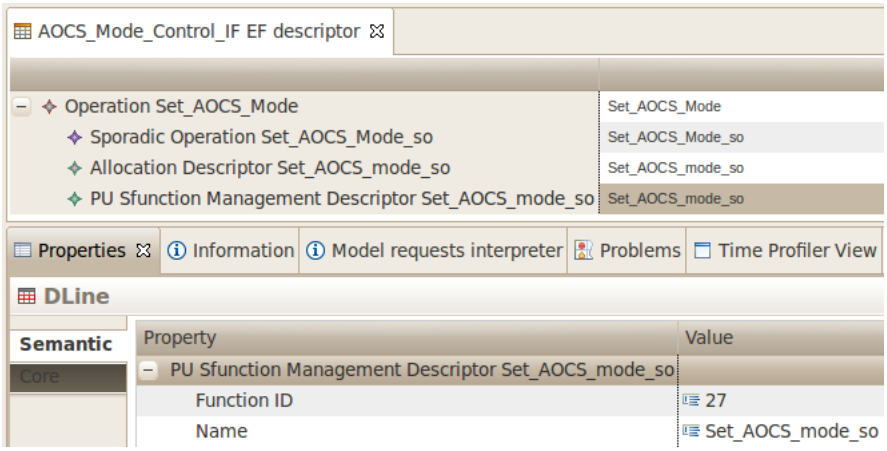

Figure 4.14: Extra-functional descriptor that allows to call an operation via PUS service 8.

When the ”PUS view” is activated it is possible to annotate entities and attributes in diagrams or tables with space-specific concerns. As an example, the figure represents operation *Set_AOCS_mode* of the provided interface *AOCS_Mode_Control*, that was marked as sporadic operation. The PUS view allow the creation of an additional descriptor by which we mark the operation to denote that it is possible to call it via PUS service 8 (cf. the example in section 3.7.2). The code generation can use this information to equip the *container* for the compo-

nent with the additional features (in the interface and in the body of the container) required to accept and elaborate incoming TCs of that service type.

Structure of the original source code. The original EagleEye source code consists mainly in the Data Handling System (OBOSS), including the mission-specific adaptations and specialization that are necessary to use the framework, and the C code for the AOCS, as generated from a Simulink model. Up to now, we were not able to obtain the source code for the remaining smaller applications (the Thermal Control, the Power Control, the mission management and telemetry store). The code is executed on a simulator of the ERC32 processor (called TSIM). A simulator called EuroSIM is programmed with a model of the environment and exchanges data with the on-board software via a shared memory area (in practice, it updates a state vector with calculations on the status of sensors and actuators data).

Discussion. The main goal of the example was to show the different approach to development that is fostered by our component model, even if we do not have yet a complete realization of the design environment; and to reason on the advantages and current shortcomings of our approach. At the present stage of realization, we have an almost complete prototypal design editor for the domain-neutral part of the approach: the only significant feature still lacking is the specification of intra- and inter-component bindings via activity and sequence diagrams. Integration of support for space-specific concerns like PUS services or dedicated hardware in the metamodel and in the editor is under progress. At the moment no code generation or analysis capabilities have been implemented yet. As we explained in sections 3.8.2.3 and 3.8.2.4, for the former, we are busy refining the ASSERT code archetypes to lift some of their current limitations and introduce some important improvements; for the latter, we have the maximum confidence of feasibility, as we base on the same strategy devised and proven effective by previous work of the author in the ASSERT project [106, 21].

We cannot therefore comment on those aspects.

Nevertheless, we can recapitulate some other aspects of the modeled example.

First of all, we are not modeling a new case study, that is developing new software components with the approach; we chose the more ambitious task of trying to model an existing ”legacy”, code-centric system with our approach. The possibility to include legacy code is one of the most critical obstacles to the uptake of a new development approach. Legacy code would be used at the beginning to mitigate the start-up cost and try to fill components without creating them completely from scratch. However, legacy code can break the consistency of the approach, all the more so when strict separation of concerns is paramount. The major obstacle is that the code for the AOCS code at our disposal is a single monolithic C file which represent the main control law. This forces the modeling of a single big component for the whole AOCS. We cannot confirm our capability of inclusion of C code generated from Simulink – of course,

void of any tasking and time-related construct – without the implementation of code generation patterns. However, as we already said, that effort has been demonstrated feasible in conditions similar to ours during the follow-up of the ASSERT project, and with the TASTE toolchain of ESA [38]. Therefore we consider the task labour-intensive, but certainly feasible.

The first advantage of our approach is the separation between functional and extra-functional concerns. Sequential code embedded in component implementations can be reused under different concurrency and real-time requirements. We do not expect that component implementations that host control laws can very often take advantage of this capability as their control law typically require execution with the same frequency. The capability is quite interesting instead for event-driven components (in EagleEye, the mission management and telemetry store). Reuse independently of extra-functional concerns in this context means also reuse of implementation independently of the PUS mapping. For example, every component could be reused with a different mapping of component interface operations or attributes to PUS services, with a simple re-instantiation and different specification of the new PUS mapping. This is possible because we confined the realization of PUS concerns to the container and connector.

Secondly, the Data Handling System (which belongs to a lower abstraction and architectural layer) does not need to be explicitly represented in the design. The designer can then focus in modeling components and their intended communication with them. The designer can then rely on the code generation for the insertion of the communication code, via the traversed communication channels and with the appropriate PUS services.

Finally, the other major advantage of using our approach, is that the model – not surprisingly for a MDE approach – becomes the center of development and the source for all design and development information. This is so important, for example, for the generation of documentation for components. For example, an Interface Control Document which enlists the parameters of a component instance that are exposed in the TM is simply a by-product of code generation, with the guarantee that it is always aligned with the model and hence with the implementation.

The major shortcoming in our current approach is the lack of support for hierarchical components, which would be helpful to tame the complexity of design and decompose components in smaller child components.

We report on our preliminary ideas on this topic in the future work section (5.2).

4.2 Coverage of technical requirements

The work described in this thesis is a first iteration on the design and realization of the software reference architecture described in chapter 2. In section 2.2 we enlisted the high-level requirements originated in the scope of this investigation. When we presented our overall approach, we claimed that it has the potential of fulfilling all those high-level requirements.

In section 2.2.1 we reported on the importance of those requirements and the expected timeframe for their fulfillment. In this section instead we comment on the following: (i) whether our investigation successfully covered all the mandatory requirements for the software reference architecture and which of the optional requirements; (ii) a recapitulation that describes which elements of our approach contribute to the fulfillment of the requirement; (iii) what is the realization status of the elements that contribute to the fulfillment of the requirement.

Table 4.3 is a summary of the coverage and realization status of our implementation activities. For each requirement, we mark if it was addressed during the present iteration of the investigation; what is the status of the realization of the elements of the approach that contribute to its fulfillment; if the implementation is necessary to considered the requirement fulfilled for the present investigation. The last assertion means that there are some requirements for which it is necessary to provide a full implementation in order to consider them fulfilled; for others it is sufficient just the implementation of a part or no implementation is really necessary, as the requirement was already demonstrated feasible in the scope of the ASSERT project. An example of this case is HR-05: Static analyzability: in the present context we really need to ensure that we are able to extract from the user model all the necessary information to feed analysis and code generation; the implementation of model-based analysis and code generation instead were already experimented in ASSERT and thus can be considered feasible also in this context. Their realization is just a matter of time and labour.

Table 4.3: The high-level requirements of the investigation and their fulfillment status. We highlight in red the mandatory requirements that are partially or not addressed in the investigation related to this PhD thesis.

ID	**High-level requirement**	**Covered**	**Implemented**	**Implementation required**
HR-01	*Separation of concerns*	Yes	Yes	Yes
HR-02	*Software reuse*	Yes	Yes [Design]	Yes [Design]
HR-03	*Reuse of V&V products*	Deferred	No	Yes
HR-04	*Component-based approach*	Yes	Yes	Yes
HR-05	*Static analyzability*	Yes	Yes [Design] No [Analysis, Implementation]	Yes [Design] No [Analysis, Implementation]
HR-06	*Property preservation*	Yes	Implemented but not integrated	Yes
HR-07	*Late accommodation of modifications in the software*	Yes	Partially	No

Table 4.3 – continued from previous page

ID	High-level requirement	Covered	Implemented	Implementation required
HR-08	*Hardware/Software independence*	Yes	[Yes] (RTOS)	No
HR-09	*Support for heterogeneous software*	Yes	Yes	No
HR-10	*Provision of mechanisms for FDIR*	Partially	Partially	Yes
HR-11	*Software observability*	Partially	Partially	Yes
HR-12	*Software update at run time*	Deferred	No	Yes

A detailed description for each requirement follows.

High-level Requirement 01 - Separation of concerns. In our approach we achieve separation of concerns at two levels:

1. for component specification, as the definition of our component model is primitively equipped with a set of design views. Each design view is a partial representation of the system according to a single concern.

2. at implementation, with a careful allocation of concerns to distinct software entities: components, containers and connectors.

In particular, we ensure that components are pure functional units: their specification and source code do not carry implicitly or explicitly extra-functional concerns.

The code archetypes that we adopt preserve that nature in the implementation.

High-level Requirement 02 - Software reuse. Our approach supports immediate reuse of components under different extra-functional concerns. This enables reuse of functional contents across projects of the same organization.

Containers and connectors depend on: (i) the computational model they contribute to enforce; (ii) the specific execution platform on which they leverage.

To enable reuse of components across different execution platforms it is thus necessary to create containers and connectors specialized for the target execution platforms of interest, and the implementation of a new chain of model transformations for their code generation. This

additional activity has to be performed only once and for all for every additional execution platform, and enables the potential reuse of a component by different software prime contractors.

To compile our code archetypes we use the GNATforLEON 2.1.0 cross-compilation chain to target the LEON 2 processor. In the scope of the CHESS project, we will use the ObjectAda compiler by Atego (formerly Aonix).

High-level Requirement 03 - Reuse of V&V products. This requirement is not currently covered by our investigation. As already anticipated in section 4.3, in accord with ESA we postponed the investigation on this requirement, as it requires an ad-hoc investigation on the methodological, technological and management support to enable reuse of qualification proofs (as complementary to software reuse) and the possible normative support to accept those proofs at the customer side (i.e. ESA).

High-level Requirement 04 - Component-based approach. The main part of this investigation was dedicated to the definition of a component model for the software reference architecture. The component model was developed so as to be compatible with the fulfillment of the other applicable high-level requirements.

High-level Requirement 05 - Static analyzability. Static analyzability of the software produced with our component-oriented approach is guaranteed by a combination of factors.

At design level, the component model shall provide all the necessary support, in terms of entities, attributes and their semantics, to produce design models that are amenable to static analysis.

At implementation level, as components encompass exclusively functional concerns, they are void of any tasking or time-related construct. We have thus the guarantee that a component does not comprise constructs that explicitly or implicitly spawn threads or use suspension primitives, hence impacting schedulability analysis.

The extra-functional attributes specified on component instances drive the automated generation of containers and connectors. Those two software entities are implemented to realize the adopted computational model; in our case the Ravenscar Computational Model (RCM). For these reasons, all the generated tasking and time-related constructs are known to the design environment, which can thus create a schedulability analysis model that corresponds to the final aspect of the implementation.

High-level Requirement 06 - Property preservation. In our approach, property preservation is achieved with the use of property-preserving code archetypes [93] for the generation of containers and connectors; and the use of an execution platform that conforms to the adopted

computational model. The implementation: (i) enforces controllable (constant) properties; (ii) monitors properties that depend on system execution (e.g. WCET, deadline fulfillment); (iii) provides mechanisms to notify and react to violation of those properties.

High-level Requirement 07 - Late accommodation of modifications in the software. The use of Model-Driven Engineering helps to tame the negative effects of design iterations; with the support by MDE, they can be used to gradually refine the design, by having confirming assurance of feasibility by analysis results. Our approach uses three different component entities: *types* to express the functional boundaries of a component; *implementations* for (i) the realization of a component, (ii) to establish its technical budgets – so that it can be considered as a subcontracting unit – and (iii) to annotate the extra-functional constraints emerging from the algorithmic code; *instances*, to relate to other components for the fulfillment of functional needs and declaration of extra-functional properties. This separation, together with the iterative capabilities of early model-based analysis can help mitigate the need and impact of late modification required to the software.

High-level Requirement 08 - Hardware/Software independence. Hardware/Software independence is realized in the lower layers of the execution platform, in the Board Support Package of the RTOS and the drivers for the communication channels of interest. The target execution platform for our prototype demonstrator, GNATforLEON 2.1.0/ORK+ targets the LEON2 processor and provides drivers for the following communication channels: the MIL-STD-1553B bus, which is typically used as the main command bus of the satellite platform; the SpaceWire point-to-point link, which is used for high-speed data transfer, such as from a satellite payload to a mass memory; and the UART interface of the LEON2 processor board.

High-level Requirement 09 - Support for heterogeneous software. Our approach supports components (i.e. component implementations) developed in various programming languages: Ada 95/2005, C and C++ are supported. This is possible because: (i) components are independent from the execution platform; (ii) connectors (written in Ada 2005 in our implementation) can support the communication between heterogeneous languages.

For what concerns data types, Ada 2005 compilers natively provide packages that reflect the base types of C/C++ for the target implementation.

Alternatively, it is possible to manage all data transfers between components through the use of a ”pivotal” language, ASN.1 (Abstract Syntax Notation One) [72], which is a language-independent notation to describe data structures and their binary encoding. ASN.1 was used for example in a track of the ASSERT project to support communication to software developed in SDL or SCADE [22]. In the TASTE toolchain [38] of the European Space Agency, communication to software generated by Simulink is supported with the same mechanism.

High-level Requirement 10 - Provision of mechanisms for Fault Detection Isolation and Recovery. The realization of some of the PUS services, namely service 2 (Device command distribution service), service 5 (Event reporting service), service 12 (On-board monitoring service), service 19 (Event-action service), provides interesting mechanisms to use for the realization of a FDIR policy. The reader may have noticed that this subset of services is subject to our initial investigation on the implementation of space-specific concerns, as outlined in section 3.7.2.

Additionally, this requirement would require the realization of concerns related to dependability, such as the creation of (model-based) design patterns for redundancy and equipment management.

However, a detailed investigation of dependability concerns and integration of them in the component model was outside the scope of this thesis. For this reason, we need to consider the requirement only partially covered.

Nevertheless, we can spin in ideas and results from the CHESS project, thanks to the convergence on the core component model and the overall methodology. In fact in CHESS, a dedicated team is currently investigating the inclusion in the approach of: Fault-tree analysis (FTA); Failure modes and effects analysis (FMEA); Failure mode, effects, and criticality analysis (FMECA); and State-based, Wide Data-flow, Call-graph and Failure propagation analysis.

High-level Requirement 11 - Software observability [Optional requirement]. For the moment, the realization of some of the PUS services, namely service 3 (Housekeeping and diagnostic data reporting service), service 4 (Parameter statistics reporting service) and service 5 (Event reporting service) will contribute to the fulfillment of this requirement. Possible other provisions are not planned for the first iteration on the software reference architecture and are deferred to subsequent iterations.

High-level Requirement 12 - Software update at run time [Optional requirement]. In this first iteration on the software reference architecture we did not address explicitly software update at run time.

The three areas where software update would be interesting are: (i) update of component implementations; (ii) update of component bindings; (iii) modification of the extra-functional attributes.

Update of component implementations can be necessary either to correct bugs in the algorithmic code, or to provide an improved (e.g. more optimized) version of the component services, or to implement software patches to hardware faults that occur during the lifetime of the satellite.

The modular structure of our code archetypes (cf. figure 3.30) greatly facilitates this job. In fact it is sufficient to temporary inhibit the execution of the Thread (by enqueueing it at the

OBCS) to safely replace the contents of the OPCS, which represents the component implementation.

Update of component bindings can be supported by careful implementation of connectors and taking advantage of object-oriented constructs of the implementation language.

Modification of extra-functional attributes instead impact containers and connectors. Services for the modification of the period or MIAT for a task or for the modification of the monitored properties (for example the WCET) can be engineered and incorporated in the Thread. For the moment we do not consider more radical changes (such as a complete change of the concurrency type of a container, e.g. from sporadic to protected or viceversa); in fact we consider the necessity of those changes as unlikely.

As a conclusion, even though we did not explicitly address this requirement, we do not think we will find stopping problems when we are going to address and implement it in future iterations of the software reference architecture.

4.3 Feedback from stakeholders

The most important feedback on the overall approach that we promote remains that of the ASSERT project. In that project, the two software primes reimplemented some sizeable proprietary subsystem of their reference systems. One of the results of those experiments sanctioned that in particular separation of functional and extra-functional concerns was both attractive and desirable.

As we said during the presentation of our work, the definition of the component model already received some scrutiny and approval by the SAVOIR-FAIRE working group. Some of the stakeholders were already skilled in component-oriented approaches and found the core ideas of the approach valid. Unfortunately, up to now, there was no time and a structured occasion to have more detailed feedback from the industrial stakeholders, in particular on the implementation activities.

The ESA staff members of the ”Software Systems Engineering” section with which we have collaborated in the last few years gave some preliminary positive feedback on the current status of implementation of the component model; and they decided to fund the author for the next year’s activities toward: (i) the inclusion of hierarchical components in the component model; (ii) integration of code generation; (iii) integration of model-based schedulability analysis; (iv) additional support for PUS services in the component model; (v) completion of the design editor.

In the forthcoming months, the status of the implementation of the component model (i.e. metamodel, design editor and code archetypes) will be provided to two software suppliers. They will start to use the component model in the context of the ESA-funded *COrDeT-2* project

(Component-Oriented Development Techniques), which kicked-off in September 2010, and will provide feedback by assessing the capabilities of the component model on a case study.

Chapter 5

Conclusions and future work

In the last few years we were involved in an initiative of the European Space Agency for the investigation and realization of a software reference architecture for the development of software in future satellite missions.

The work that was carried out in the scope of this PhD thesis started with a thorough investigation on the industrial needs set as strategic goals by ESA and the other major stakeholders of the European space domain for future space missions.

The industrial needs were gathered with several interviews with ESA staff members and discussed and refined in several meetings of SAVOIR-FAIRE, a working group comprised of staff members of ESA, national space agencies, software prime contractors and software suppliers. The comprehension and verbalization of these requirements was fundamental to better understand the complete picture from the point of view of the various stakeholders.

The analysis of the industrial needs led to the decision of deriving a set of high-level requirements, which were used as a baseline for the development of a novel component-oriented approach to the development of on-board software. The derivation of requirements was particularly profitable, especially thanks to the visits of the author to the premises of the main software primes, EADS Astrium (Toulouse) and Thales Alenia Space (Cannes). The focused discussions had with the major players of the domain greatly facilitated faster discussion and exchange of ideas, and permitted rapid correction and evolution of ideas thanks to the feedback we received.

The high-level requirements were later submitted to SAVOIR-FAIRE, where, after discussion and corrections, they received the approval of the represented stakeholders.

After analysis of the requirements by SAVOIR-FAIRE, the working group concluded that the best solution to fulfill all of them was the definition and adoption of a software reference architecture.

A software reference architecture is a single, agreed and common solution adopted by the stakeholders of the domain for the definition of the software architecture of the systems sub-

ject of the investigation (the platform software of Earth observation, science and deep space missions).

That decision had the clear advantage of permitting to base our investigation on the promising results of past experience and in particular on the methodological asset investigated during the ASSERT project.

In fact we based our approach on four central constituents: (i) a component model, for the definition of software systems as a composition of reusable software units characterized by known functional attributes and decorated with a known and finite range of extra-functional attributes; (ii) a computational model, to relate components and their extra-functional attributes to a framework of analysis equations, so as to make the component description statically analyzable; (iii) a programming model, which consists in a subset of a programming language coupled with a set of suitable code archetypes that are able to warrant that the implementation of the design entities reflects the analysis assumptions and semantics as required by the computational model; (iv) a conforming execution platform, which is able to host and execute components in compliance with the computational model and warrants the preservation of the system and component properties asserted by analysis.

The component model is fundamental to attain *composability*, which is achieved when the properties asserted on components in isolation are preserved upon component composition and deployment on the target. The component model together with the computational model are fundamental to attaining *compositionality*, that is the capability of calculating the properties of the system or of an assembly of components as a function of exclusively the externally visible properties of its constituting components. Our approach is also permeated with *property preservation*, which warrants that the properties asserted on the design specification are conveyed in the implementation and are preserved at run time.

The approach provides what we termed *composition with guarantees*, a crucial property that can be regarded as a form of composability and compositionality that can be assured by static analysis, guaranteed throughout implementation, and actively preserved at run time.

We also included in the approach the use of Model-Driven Engineering and the provisions for domain-specific aspects, as we regard them as fundamental for the industrial applicability of the approach.

Constituents (ii), (iii) and (iv) were already investigated by the author in the scope of the ASSERT project. The core of the PhD was devoted to the definition and realization of a component model to complete the approach, and the formulation of the software reference architecture.

The component model is centered on a number of distinctive features.

Separation of concerns. We permeate the approach with separation of concerns, in particular between functional and extra-functional concerns. In our approach components are pure functional units, that are void of any extra-functional concern. This implies that their source code is void of any tasking and time-related language construct.

Our approach requires a very specific interpretation of the software entities: components include only functional and algorithmic concerns; interaction and communication concerns are addressed by the connector; the remaining extra-functional concerns are addressed by the container.

Separation of concerns is obtained: (i) at implementation level, by carefully allocating the concerns to the three entities of our approach: the component, the container and the connector; (ii) at the specification level, by ensuring that the design language and the the design flow (the design steps to be followed for the creation of components, cf. figure 3.20) are not inadvertently or purposely forcing to introduce extra-functional concerns on components; (iii) during the design process in general, by adopting design views.

Compatibility with ISO 42010 and use of design views ISO/IEC 42010 [76] (which adopted IEEE 1471), describes recommended practices for the architectural description of software-intensive systems.

A component-based approach addresses scores of aspects related to the creation and description of an architecture. In particular the component model addresses the concerns of a set of stakeholders. The component model supports the various viewpoints of those stakeholders by providing a set of design views, each of them presenting a partial representation of the system under development.

One distinguishing feature of our approach is that we define the design views together with the definition of our component model and we attribute each design step of our design flow to a precise view, so as to identify the design stage in which the step can be performed and the responsibility of the action.

As views are expression of a single or of a small, related set of concerns, they become effective means to enforce separation of concerns throughout the design process.

Support for model-based analysis. The computational model guarantees compositionality in the time dimension; compositionality is guaranteed in the implementation and at run time by the careful and conforming selection of the programming model and execution platform. We ensured that a user model can contain exclusively entities that – for to their semantics and extra-functional attributes – can be analyzed according to the analysis equations related to our computational model of choice, that is the Ravenscar Computational Model [27].

We worked to ensure that the design environment is able to automatically extract or derive all information for analysis from the user model. What we also want is the ability to feed complex analysis equations and back propagate results directly in the design space as attributes of the user-level model entities, with the highest degree of automation and the minimum effort for the user.

Integration of domain-specific concerns. In our component model we worked toward the inclusion of domain-specific concerns. In fact, we contend that in order to aim at industrial

adoption, a component model shall include provisions specialized for the application domain of interest to complement those for the realization of embedded real-time systems in general.

Without those provisions, industrial adoption of a component model may result unattractive, as the adaptation of the domain-neutral component model to the specific application needs of the domain may lead to an onerous effort which may even undermine in part or in total the guarantees provided by the original approach. In section 3.7.2 we provided extensive examples related to the space domain to sustain our claim.

In our vision the best approach is the development of a domain-neutral component model complemented by a set of domain-specific extensions for a set of domain of interest. The domain-neutral part of the approach is shared by all domains, which would then ”activate” only the domain-specific extension of interest. Domain-specific provisions include language extensions, additional design views for the controlled specification of the domain-specific concerns, and specialized analysis and code generation.

Other aspects worth mentioning are:

Containers and connectors. In our approach, containers and connectors are not represented in the platform-independent design space, as they are implementation entities that belong to a platform-specific model that it is automatically generated via model transformation according to a set of deterministic rules. The transformation is driven by the extra-functional attributes that are used to decorate components. Containers and connectors are the software entities that *realize* in the implementation those *declared* extra-functional attributes.

Alternative approaches may require the specification of containers and connectors by the designer in the design space. In this way we also dispense with the problem of choosing the formalism and the diagram kind to represent them.

Code generation and functional / algorithmic specification. In our approach we aim at complete generation of code addressing extra-functional concerns.

On the contrary, for what concerns algorithmic code, we do not support in our infrastructure the generation of code, mainly due to the multitude of formalisms and technologies necessary to cover all the domain needs.

We prefer to acknowledge the heterogeneity, permit the designer to develop the functional / algorithmic code by hand or in their favorite environment (Matlab, Simulink, SDL,...) and provide mechanisms to include the functional code in a system designed with our component-oriented approach.

Of course, the code shall not break the assumptions that are valid for component implementations in general, i.e. it shall be void of tasking and time-related constructs.

We started the realization of the component model described in this thesis and we decided to create an incarnation centered on a domain-specific language, specified as an *ecore* domain-specific metamodel.

At the time of this writing we almost completed the development of a design editor for the specification of components and support for the functional, extra-functional and hardware / deployment view described in section 3.6.

We decided to adopt the property-preserving code archetypes used in the ASSERT project as a baseline for our approach. We could have already started the development of code generation for them, but we preferred to defer it to work on a number of improvements to the archetypes which include: (i) support for the change of period or MIAT of tasks; (ii) support for more flexible allocation of operations to tasks; (iii) support for user-defined release protocols; (iv) support for PUS services at container and connector level.

Finally, for the sake of priorities, we decided to postpone the integration of model-based schedulability analysis with the design editor (see section 3.8.2.3). The methodology for its realization was already investigated and implemented by the author during the ASSERT project. For this reason, this is the part of the overall approach for which we have the maximum confidence and we foresee no uncharted methodological or technological challenges in its realization.

In the timeframe of our PhD we also worked in the scope of the CHESS projects. CHESS aims at the development of a multi-domain component-oriented approach targeting the space, telecommunication and railway domains. We strove to keep aligned the overall methodology adopted by CHESS and its component model with what was investigated in the scope of this thesis.

CHESS decided to adopt an extension of the UML MARTE profile as the design language. This enabled us to compare our work on the DSL with the implementation of the same component model as realized in CHESS. We outlined the main differences between the two approaches with respect to the engineering of the design language. We also compared the effort required for the realization of design views, highlighting the distance of the current implementations with our requirements for design views.

5.1 Scientific Publications

In this section we summarize the publications that originated from the various topics presented in this PhD thesis.

- A subset of the most representative industrial needs and high-level requirements presented in section 2.1 and 2.2 were described in [109] (*SEAA'2010*). Unfortunately, page space limitations did not permit us to convey in that paper the complete picture with respect to those topics.

- The four central constituents of the software reference architecture (cf. section 2.3) were described in [108] (*RTCSA'2009*).

- The founding principles of our component-oriented approach – separation of concerns and correctness-by-construction – (cf. section 3.1), and how we permeate our approach with them, were described in several of our publications: [108] (*RTCSA'2009*), [105] (*HoPES'2010*) and [109] (*SEAA'2010*).

- The definition of our component model (cf. section 3.2 and 3.5) was described in [109] (*SEAA'2010*) and, with less details in [110] (*CRTS'2010*).

- The topic of *design views* was the subject of [105] (*HoPES'2010*), in which we focused on a description of their preliminary implementation in the CHESS project. We are writing a more detailed paper on the topic, with additional considerations w.r.t. what we elaborated on in section 3.3.

- Property preservation and how to warrant it from design to implementation and at run time was the main topic of [93] (*Ada-Europe'2010*).

- Finally, the strategy we adopted for model-based schedulability analysis (cf. section 3.8.2.3), that we inherited from ASSERT, was described in [21] (*ECRTS'2008*) and [31] (*IEEE Transactions on Industrial Informatics'2010*).

The complete list of publications in the timespan of this PhD can be found in Appendix A.

5.2 Future work

In the remainder of the year 2011 we will work on four main topics to improve the component model and the overall software reference architecture: *hierarchical components*; *support for scenario-based analysis*; *support for reuse of products of V&V activities* and *software updates at run time*.

The lack of hierarchical components is probably the most significant shortcoming of our current approach. Instead of integrating them directly in the original definition of the component model we decided to postpone their introduction in the component model to the first revision. The main reason of the postponement is that during the meetings of the SAVOIR-FAIRE working group it was apparent that all stakeholders required a notion of hierarchy in the component model. However, related requirements were not clear or worse, contradictory.

In particular there was no consistent vision among the stakeholders on the requirements concerning the following aspects related to hierarchical components: (i) specification of extra-functional attributes on parent and child components; (ii) black-box view of parent and child components and relationship with schedulability analysis; (iii) subcontracting of hierarchical components (What does it mean to subcontract a parent or a child component? What are the consequences?).

From initial discussions, it seems that hierarchical components would be used mainly to tame the complexity of design and (indirectly) mitigate the cluttering of entities in the design editor, a presentation problem common to all MDE design environments.

Therefore it seems that hierarchical *decomposition* would be favored with respect to hierarchical *composition*. That is not really a surprise, as the space domain has a long tradition and a market structure that favor top-down development.

Nevertheless we want an approach to hierarchical components that is able to support both top-down and bottom-up development; the latter may become important when *reusing* components rather than developing them from scratch.

Our preliminary proposal consists in providing a hierarchical model that supports potentially infinite levels of hierarchy in the functional dimension (hence at the level of component types and their corresponding implementation). Parent components delegate one or more of their provided interfaces to a child component. A child component either promotes its required interfaces or binds them to another child component at the same level.

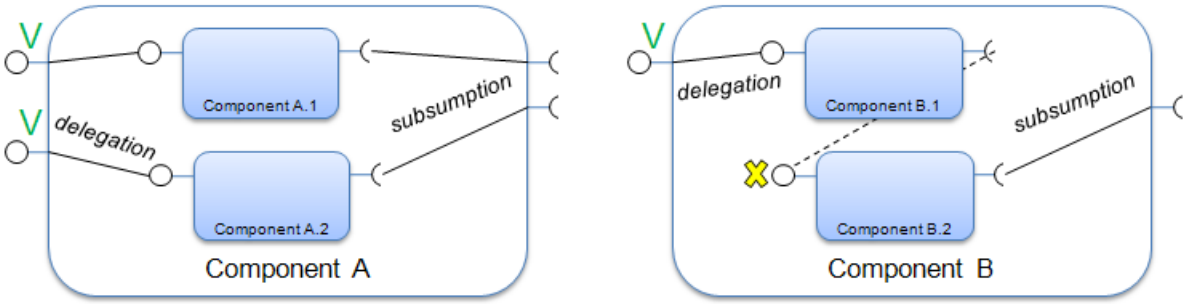

Figure 5.1: Proposal for hierarchical components.

We depict our proposal in figure 5.1. When establishing hierarchical components, the only interfaces that it is possible to decorate later (at instance level) with extra-functional attributes are those at the top-most parent level. With this constraint, any provided interface of child components either: (i) represents a delegation from a parent interface and therefore will indirectly ”share” the same extra-functional attributes at instance level (as depicted in the left side of the figure); or (ii) does not represent a delegation from a parent interface and therefore it cannot

be decorated later with extra-functional attributes; as a consequence all its operations shall be considered passive (as depicted in the right side of the figure).

Simple (or worse simple-minded) support for schedulability analysis is unfortunately not enough for real-scale systems. The problem used to be with the pessimism incurred by the *analysis equations*, but this is no longer so thanks to the constant advancement of the theory. The problem rather consists in the generation of the worst-case scenario, single or multiple that they may be. In fact, an automated support for the extraction of analysis information can generate only the *theoretical* worst-case scenario. We however know that this *theoretical* worst-case scenario may never happen in particular due to some combination of logical conditions in the system that the extraction tool was not able to detect.

We would like to investigate the conditions by which the designer can provide the necessary information to prune infeasible worst-case scenario. We think that the best framework to realize this provision is the support for scenario-based analysis. The designer should be able to: (i) relax the formulation of the worst-case scenario based on their knowledge of the system; (ii) create several analysis scenarios – at least one per operational mode – based on various loads and conditions of the system.

A few other approaches already experimented the realization of scenario-based analysis in a component-oriented approach; for example, follow-up activities to the development of the ROBOCOP component model [17].

As for the activities related to the reuse of outputs of the V&V activities, we are establishing with ESA the best way forward. However, the complexity of the topic in the various axes – methodological, technological and normative – most probably will require its investigation in a dedicated project activity.

Finally, we will work on the mechanisms at implementation level to enable software updates at run time. However a more thorough reasoning on the topic would require to address *architecture evolution* in general [91], in order to establish and control from the design specification the evolution of the software during its whole lifecycle.

Appendix A

List of scientific publications

The author contributed to the following scientific publications during the timeframe of his PhD:

- M.Bordin and M. Panunzio and T. Vardanega: "Fitting Schedulability Analysis Theory into Model-Driven Engineering". *Proc. of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*, July 2008 [21]. An article on the approach to model-based schedulability analysis devised and realized in the ASSERT project, which is adopted in our current investigation.

- M. Bordin and M. Panunzio and S. Puri: "Rapid Model-Driven Prototyping and Verification for High-Integrity Real-Time Systems". *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. September 2008 [19]. An article of the tool demonstration track to show the potential of the ASSERT tool prototype, which includes model-based schedulability analysis and code generation according to the same founding principles of our current investigation.

- M. Bordin and M. Panunzio and C. Santamaria and T. Vardanega: "A Reinterpretation of Patterns to Increase the Expressive Power of Model-Driven Engineering". *Proc. of the 1st International Workshop on Model Based Architecting and Construction of Embedded Systems*, held in conjunction with "*The ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08)*", September 2008 [20]. An article on an investigation which aimed to increase the expressive power availed to the designer in the MDE design space while ensuring that the automated generation of the platform-specific models are compliant to an underlying computational model.

- M. Panunzio and T. Vardanega: "On Component-Based Development and High-Integrity Real-Time Systems". *Proc. of the 15th IEEE International Conference on Embedded and*

Real-Time Computing Systems and Applications (RTCSA'09), August 2009 [108]. An article which describes the 4-ingredients approach on which is based our investigation.

- D. Cancila and R. Passerone and T. Vardanega and M. Panunzio: "Toward Correctness in the Specification and Handling of Nonfunctional Attributes of High-Integrity Real-Time Embedded Systems". *IEEE Transactions on Industrial Informatics*, Volume 6, No. 2, May 2010 [31]. The main contribution of the author to the paper is the approach to model-based schedulability analysis that is adopted in our current investigation.

- M. Panunzio and C. Santamaria and A. Zovi and T. Vardanega: "Correctness by Construction and Separation of Concerns in a MDE Design Infrastructure". *Proc. of the 1st Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems*, held in conjunction with the *6th European Conference on Modelling Foundations and Applications (ECMFA'10)*, June 2010 [105]. An article which reports on how we plan to realize a design environment supporting C-by-C and separation of concerns in the scope of the CHESS project, in accord with the principles described in this thesis. The article underlines the instrumental role of design views to achieve the latter goal. The article then reports on the realization of a design editor for the component model of this PhD thesis, using an extension of the MARTE profile as design language and MDT-Papyrus as the base platform.

- E. Mezzetti and M. Panunzio and T. Vardanega: "Preservation of Timing Properties with the Ada Ravenscar Profile". *Proc. of the 15th International Conference on Reliable Software Technologies - Ada-Europe 2010*, June 2010 [93]. An article which describes how the Ada Ravenscar profile and a set of suitable code archetypes can be used as a programming model on top of a conforming execution platform to preserve at run time the attributes used as input for schedulability analysis; it also describes a set of mechanisms and policies to react to run-time violations of those attributes.

- E. Mezzetti and M. Panunzio and T. Vardanega: "Bounding the Effects of Resource Access Protocols on Cache Behavior". *Proc. of the 10th International Workshop on Worst-Case Execution-Time Analysis (WCET'10)*, July 2010 [92]. An article, unrelated to the main contents of this thesis, which presents a comparison between a set of well-known resource access protocols for uniprocessor systems with respect to their impact on the cache behaviour, and provides a bound to this effect for the Priority Inheritance Protocol [124], the Priority Ceiling Prototol [124] and the Immediate Ceiling Protocol, a derivative of Baker's Stack Resource Policy [9].

- M. Panunzio and T. Vardanega: "A Component Model for On-board Software Applications". *Proc. of the 36th Euromicro Conference on Software Engineering and Advanced*

Applications (SEAA'10), September 2010 [109]. An article which recapitulates the main industrial needs and technical requirements elicited and derived during our investigation and describes our component model.

- M. Panunzio and T. Vardanega: "Common Pitfalls and Misconceptions of Component-Oriented Approaches for Real-Time Embedded Systems: Lessons Learned and Solutions". *Proc. of the 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'10)*, held in conjunction with *The 31st IEEE Real-Time Systems Symposium (RTSS'10)*, November 2010 [110]. An article which recapitulates the pitfalls and misconceptions we found scrutinizing other component-based approaches, which can compromise their adoption and how we avoid them in our component model.

Bibliography

[1] Aeronautical Radio, Incorporated. ARINC Specification 653-2 : Avionics Application Software Standard Interface - Part 1 Required Services, 2005.

[2] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE Approach to Component-based Development of Vehicular Systems. *Journal of Systems and Software*, 80(5):655–667, 2007.

[3] M. Aldea Rivas and M. González Harbour. MaRTE OS: an Ada Kernel for Real-Time Embedded Applications. In *Reliable Software Technologies - Ada-Europe*, pages 305–316, 2001.

[4] A. Aldini and M. Bernardo. On the usability of process algebra: An architectural view. *Theoretical Computer Science*, 335(2-3):281–329, 2005.

[5] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Proc. of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, 2003.

[6] S. Angelov, P. Grefen, and D. Greefhorst. A Classification of Software Reference Architectures: Analyzing Their Success and Effectiveness. In *Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, WICSA/ECSA*, 2009.

[7] ASSERT: Automated proof-based System and Software Engineering for Real-Time systems. FP6 IST-004033 2004-2008 `http://www.assert-project.net`.

[8] J. C. M. Baeten. A Brief History of Process Algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[9] T. P. Baker. Stack-based Scheduling for Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

[10] T. P. Baker and A. C. Shaw. The Cyclic Executive Model and Ada. In *Proc. of the 9th IEEE Real-Time Systems Symposium*, pages 120–129, 1988.

[11] M. Barbacci, P. Clements, A. Lattanze, L. Northrop, and W. Wood. Using the Architecture Tradeoff Analysis Method (ATAM) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study. Technical report, SEI, Carnegie Mellon University, 2003.

[12] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[13] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, 2006.

[14] I. J. Bate. *Scheduling and Timing Analysis for Safety Critical Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, November 1998.

[15] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The Synchronous Languages 12 Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[16] E. Bini, G. C. Buttazzo, and G. Lipari. Speed Modulation in Energy-Aware Real-Time Systems. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, pages 3–10, 2005.

[17] Y. Bondarau. *Design-Time Performance Analysis of Component-Based Real-Time Systems*. PhD thesis, Technische Universiteit Eindhoven, December 2009.

[18] E. Bondarev, M. R. V. Chaudron, and P. H. N. de With. A Process for Resolving Performance Trade-Offs in Component-Based Architectures. In *Proc. of the 9th International Symposium on Component-Based Software Engineering*, pages 254–269, 2006.

[19] M. Bordin, M. Panunzio, and S. Puri. Rapid Model-Driven Prototyping and Verification for High-Integrity Real-Time Systems. In *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[20] M. Bordin, M. Panunzio, C. Santamaria, and T. Vardanega. A Reinterpretation of Patterns to Increase the Expressive Power of Model-driven Engineering Approaches. In *1st International Workshop on Model Based Architecting and Construction of Embedded Systems*, 2008.

[21] M. Bordin, M. Panunzio, and T. Vardanega. Fitting Schedulability Analysis Theory into Model-Driven Engineering. In *Proc. of the 20th Euromicro Conference on Real-Time Systems*, 2008.

[22] M. Bordin, T. Tsiodras, and M. Perrotin. Experience in the Integration of Heterogeneous Models in the Model-driven Engineering of High-Integrity Systems. In *Proc. of the 13th International Conference on Reliable Software Technologies - Ada-Europe*, pages 171–184, 2008.

[23] M. Bordin and T. Vardanega. Automated Model-Based Generation of Ravenscar-Compliant Source Code. In *Proc. of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[24] M. Bordin and T. Vardanega. Correctness by Construction for High-Integrity Real-Time Systems: a Metamodel-driven Approach. In *Proc. of the 12th International Conference on Reliable Software Technologies - Ada-Europe*, 2007.

[25] D. Box. *Essential COM. Object Technology Series*. Addison-Wesley, 1997.

[26] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Reliable Software Technologies - Ada Europe*, 1998.

[27] A. Burns, B. Dobbing, and T. Vardanega. Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems. *Technical Report YCS-2003-348, University of York*, 2003.

[28] A. Burns and A. J. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier, 1995.

[29] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.

[30] J. L. Campos, J. J. Gutiérrez, and M. González Harbour. Interchangeable Scheduling Policies in Real-Time Middleware for Distribution. In *Proc. of the 11th International Conference on Reliable Software Technologies - Ada-Europe*, pages 227–240, 2006.

[31] D. Cancila, R. Passerone, T. Vardanega, and M. Panunzio. Toward Correctness in the Specification and Handling of Nonfunctional Attributes of High-Integrity Real-Time Embedded Systems. *IEEE Transactions on Industrial Informatics*, 6(2):181–194, 2010.

[32] E. Canuto. Drag-free and attitude control for the GOCE satellite. *Automatica, A Journal of IFAC, the International Federation of Automatic Control*, 44(7):1766–1780, July 2008.

[33] J. Carlson, J. Feljan, J. Mäki-Turja, and M. Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *Proc. of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, 2010.

[34] J. Carlson, J. Haåkansson, and P. Pettersson. SaveCCM: An Analysable Component Model for Real-Time Systems. In *Proc. of the Second Workshop on Formal Aspects of Components Software*, pages 627–635. Electronic Notes in Theoretical Computer Science, Elsevier, 2005.

[35] T. Chantem, X. S. Hu, and M. D. Lemmon. Generalized Elastic Scheduling. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 236–245, 2006.

[36] R. Chapman. Correctness by Construction: a Manifesto for High Integrity Software. In *ACM International Conference Proceeding Series; Vol. 162*, 2006.

[37] M. Chaudron and I. Crnkovic. *Component-based software engineering, chapter 18 in H. van Vliet, Software Engineering: Principles and Practice*. Wiley, 2008.

[38] E. Conquet, M. Perrotin, P. Dissaux, T. Tsiodras, and J. Hugues. The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software. In *Proc. of Embedded Real Time Software and Systems (ERTS)*, 2010.

[39] I. Crnkovic, M. Larsson, and O. Preiss. Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. In *WADS*, pages 257–278, 2004.

[40] D. Dvorak (editor). NASA Study on Flight Software Complexity. Technical report, Commissioned by the NASA Office of Chief Engineer, 2009.

[41] R. I. Davis and A. Burns. Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proc. of the 26th IEEE Real-Time Systems Symposium*, pages 389–398, 2005.

[42] J. A. de la Puente, A. Alonso, and A. Alvarez. Mapping HRT-HOOD Designs to Ada 95 Hierarchical Libraries. In *Proc. of the International Conference on Reliable Software Technologies - Ada-Europe*, pages 78–88, 1996.

[43] E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859 – 866, 1972. ISSN 0001-0782.

[44] E. W. Dijkstra. On the role of scientific thought. In E. W. Dijkstra, editor, *Selected writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag New York, Inc., 1982. ISBN 0-387-90652-5.

[45] European Cooperation for Space Standardization (ECSS). Space Engineering - Ground systems and operations - Telemetry and telecommand packet utilization, 2003. ECSS-E-70-41A.

[46] European Cooperation for Space Standardization (ECSS). Space engineering - Software, 2009. ECSS-E-ST-40C.

[47] European Cooperation for Space Standardization (ECSS). Space product assurance - Dependability, 2009. ECSS-Q-ST-30C.

[48] European Cooperation for Space Standardization (ECSS). Space product assurance - Software product assurance, 2009. ECSS-Q-ST-80C.

[49] H. Fennel and S. Bunzel et al. Achievements and exploitation of the AUTOSAR development partnership. Technical report, AUTOSAR Partnership, October 2006.

[50] D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing*. 2003.

[51] Framework for Real-time Embedded Systems based on COntRacts – Final Project Report, 2009. `http://www.frescor.org/index.php?page=publications`.

[52] G. Douglass Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[53] Gaisler Research. LEON2 Processor User's Manual - XST Edition. Version 1.0.23, May 2004.

[54] B. Gallagher. Using the Architecture Tradeoff Analysis Method to Evaluate a Reference Architecture: A Case Study. Technical report, SEI, Carnegie Mellon University, 2000.

[55] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison - Wesley, 1995.

[56] O. Gilles and J. Hugues. Validating Requirements at Model Level. In *Proc. of the 4th workshop on Model-Oriented Engineering*, 2008.

[57] M. González Harbour, J. Gutierrez, J. Palencia, and J. Drake. MAST: Modeling and Analysis Suite for Real-Time Applications. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, 2001.

[58] J. B. Goodenough and L. Sha. The priority ceiling protocol: a method for minimizing the blocking of high priority Ada tasks. In *Proc. of the 2nd International Workshop on Real-time Ada Issues*, pages 20–31, 1988.

[59] G. Gößler and J. Sifakis. Composition for Component-Based Modeling. In *First International Symposium on Formal Methods for Components and Objects*, pages 443–466, 2002.

[60] J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and M. González Harbour. Schedulability analysis of distributed hard real-time systems with multiple-event synchronization. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 15–24, 2000.

[61] A. Hall and R. Chapman. Software Engineering - Correctness by Construction. Technical Report Issue 1.1, Praxis High-Integrity Systems, 2004.

[62] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck. The Rubus component model for resource constrained real-time systems. In *Proc. of the Third International Symposium on Industrial Embedded Systems*, pages 177–183, 2008.

[63] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren. SaveCCM - A Component Model for Safety-Critical Real-Time Systems. In *Proc. of the 30th Euromicro Conference*, pages 627–635, 2004.

[64] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Mat, K. Nishikawa, and T. Scharnhorst. AUTOSAR, An industry-wide initiative to manage the complexity of emerging Automotive E/E-Architectures. In *Convergence 2004, Proceedings of the International Congress on Transportation Electronics*, 2004.

[65] T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 40(10):32–40, 2007.

[66] S. Hissam, J. Hudak, J. Ivers, and M. K. et al. Predictable Assembly of Substation Automation Systems: An Experiment Report – Second Edition. Technical Report CMU/SEI-2002-TR-031, Software Engineering Institute, Carnegie Mellon University, 2003.

[67] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[68] Hyun Gi Min and Si Won Choi and Soo Dong Kim. Using Smart Connectors to Resolve Partial Matching Problems in COTS Component Acquisition. In *Proc. of the 7th Int. Symposium on Component-Based Software Engineering*, pages 40–47, 2004.

[69] IEEE. Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000, 2000.

[70] IEEE. IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support. IEEE Std 1003.13, 2003.

[71] International Telecommunication Union. Series Z: Languages and General Software Aspects for Telecommunication Systems - Formal description techniques (FDT) - Specification and Description Language (SDL), November 2007. Recommendation Z.100 - `http://www.itu.int/rec/T-REC-Z.100-200711-I/en`.

[72] International Telecomunication Union. Information technology Abstract Syntax Notation One (ASN.1): Specification of basic notation, 2008. ITU-T Recommendation X.680.

[73] ISO SC22/WG9. Ada Reference Manual. Language and Standard Libraries. Consolidated Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Amendment 1, 2005.

[74] ISO/IEC. Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model. ISO/IEC 7498-1, 1996.

[75] ISO/IEC. Information technology – Meta Object Facility (MOF). ISO/IEC 19502, 2005.

[76] ISO/IEC/(IEEE). Systems and Software engineering - Recomended practice for architectural description of software-intensive systems. ISO/IEC 42010 (IEEE Std) 1471-2000, 2007.

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

[78] E. Jouenne and V. Normand. Tailoring IEEE 1471 for MDE Support. In *UML Satellite Activities*, pages 163–174, 2004.

[79] W. Kozaczynski and G. Booch. Component-Based Software Engineering. *IEEE Software*, 15(5), 1998.

[80] P. Kruchten. *The Rational Unified Process: An Introduction – 2nd Edition*. Addison-Wesley, 2000.

[81] J. Kwon, A. Wellings, and S. King. Ravenscar-Java: a high integrity profile for real-time Java. In *Proc. of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140, 2002.

[82] K.-K. Lau and F. M. Taweel. Domain-Specific Software Component Models. In *Proc. of the 12th International Symposium on Component-Based Software Engineering*, pages 19–35, 2009.

[83] C.-F. Lin. *Modern Navigation, Guidance, And Control Processing*. Prentice Hall, 1991. ISBN: 978-0135962305.

[84] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[85] P. López Martínez, J. M. Drake, P. Pacheco, and J. L. Medina. Ada-CCM: Component-based Technology for Distributed Real-Time Systems. In *Proc. of the 11th International Symposium on Component-Based Software Engineering*, 2008.

[86] P. López Martínez, J. M. Drake, P. Pacheco, and J. L. Medina. An Ada 2005 Technology for Distributed and Real-Time Component-Based Applications. In *Proc. of the 13th International Conference on Reliable Software Technologies - Ada-Europe*, pages 254–267, 2008.

[87] M. J. Morgan. Integrated Modular Avionics for Next Generation Commercial Airplanes. *IEEE Aerospace and Electronic Systems Magazine*, 6(8), August 1991.

[88] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., 1999.

[89] J. M. Martínez and M. González Harbour. RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet. In *Reliable Software Technologies - Ada-Europe*, 2005.

[90] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proc. of the 22nd International Conference on Software Engineering*, pages 178–187, 2000.

[91] T. Mens, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43:42–48, 2010.

[92] E. Mezzetti, M. Panunzio, and T. Vardanega. Bounding the Effects of Resource Access Protocols on Cache Behavior. In *Proc. of the 10th International Workshop on Worst-Case Execution-Time Analysis*, 2010.

[93] E. Mezzetti, M. Panunzio, and T. Vardanega. Preservation of Timing Properties with the Ada Ravenscar Profile. In *Proc. of the 15th International Conference on Reliable Software Technologies - Ada-Europe*, 2010.

[94] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proc. of the 7th Int. Symposium on Component Based Software Engineering*, pages 146–161, 2004.

[95] J. Muskens, M. Chaudron, and J. J. Lukkien. A Component Framework for Consumer Electronics Middleware. In C. Atkinson et al., editor, *Component-Based Software Development for Embedded Systems*, volume 3778 of *Lecture Notes in Computer Science*, pages 164–184. Springer Berlin / Heidelberg, 2005.

[96] Object Management Group. CORBA Component Model, v4.0, April 2006. `http://www.omg.org/technology/documents/formal/components.htm`.

[97] Object Management Group. Deployment and Configuration of Component-based Distributed Applications Specification, v.4.0, April 2006. `http://www.omg.org/spec/DEPL/4.0/`.

[98] Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP) - Version 3.1, January 2008. `http://www.omg.org/spec/CORBA/3.1/`.

[99] Object Management Group. *UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE)*. 2009. version 1.0 `http://www.omg.org/spec/MARTE/1.0/`.

[100] Object Management Group. *Object Constraint Language*. 2010. version 2.2 `http://www.omg.org/spec/OCL/2.2/`.

[101] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure - version 2.3, 2010. `http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`.

[102] OMG. SysML specification – version 1.2, 2010. `http://www.omg.org/spec/SysML/1.2/`.

[103] J. C. Palencia and M. González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proc. of the 19th IEEE Real-Time Systems Symposium*, 1998.

[104] J. C. Palencia and M. González Harbour. Exploiting Precedence Relations in the Schedulability Analysis of Distributed Real-Time Systems. In *Proc. of the 20th IEEE Real-Time Systems Symposium*, 1999.

[105] M. Panunzio, C. Santamaria, A. Zovi, and T. Vardanega. Correctness by Construction and Separation of Concerns in a MDE Design Infrastructure. In *1st Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems*, 2010.

[106] M. Panunzio and T. Vardanega. A Metamodel-driven Process Featuring Advanced Model-based Timing Analysis. In *Proc. of the 12th International Conference on Reliable Software Technologies - Ada-Europe*, 2007.

[107] M. Panunzio and T. Vardanega. An Approach to the Timing Analysis of Hierarchical Systems. In *Proc. of the 13th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[108] M. Panunzio and T. Vardanega. On Component-Based Development and High-Integrity Real-Time Systems. In *Proc. of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[109] M. Panunzio and T. Vardanega. A Component Model for On-board Software Applications. In *Proc. of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 57–64, 2010.

[110] M. Panunzio and T. Vardanega. Common pitfalls and misconceptions of component-oriented approaches for real-time embedded systems: lessons learned and solutions. In *Proc. of the 3rd Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2010.

[111] L. Pautet and S. Tardieu. GLADE: A Framework for Building Large Object-Oriented Real-Time Distributed Systems. In *Proc. of the 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, pages 244–251, 2000.

[112] S. Priya Marimuthu and S. Chakraborty. A Framework for Compositional and Hierarchical Real-Time Scheduling. In *Proc. of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.

[113] J. A. Pulido, J. A. de la Puente, J. Hugues, M. Bordin, and T. Vardanega. Ada 2005 Code Patterns for Metamodel-based Code Generation. *Ada Letters*, XXVII(2), 2007.

[114] J. A. Pulido, S. Urueña, J. Zamorano, T. Vardanega, and J. A. de la Puente. Hierarchical Scheduling with Ada 2005. In *Proc. of the 11th International Conference on Reliable Software Technologies - Ada-Europe*, 2006.

[115] Radio Technical Commission for Aeronautics (RTCA). Software Considerations in Airborne Systems and Equipment Certification - DO-178B, 1992.

[116] W. Royce. Managing the Development of Large Software Systems. In *Proc. of IEEE Wescon*, pages 1–9, 1970.

[117] N. Rozanski and E. Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley Professional, 2005.

[118] J. Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Technical Report DOT/FAA/AR-99/58, Computer Science Laboratory, SRI International, March 2000.

[119] SAE International. Architecture Analysis and Design Language (AADL). `http://www.aadl.info`.

[120] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[121] B. Selic and L. Motus. Using models in real-time software design. *IEEE Control Systems Magazine*, 23(3):31–42, 2003.

[122] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic. Integration of Extra-Functional Properties in Component Models. In *Proc. of 12th International Symposium on Component-Based Software Engineering*, pages 173–190, 2009.

[123] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[124] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *Proc. of the 7th IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[125] I. Shin and I. Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, 2003.

[126] I. Shin and I. Lee. Compositional Real-Time Scheduling Framework with Periodic Model. *Transactions on Embedded Computing Systems*, 7(3):1–39, 2008.

[127] J. Sifakis. A Framework for Component-based Construction Extended Abstract. In *Proc. of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, pages 293–300, 2005.

[128] Software Engineering Institute (editor). Defining Software Architecture - Modern, Classic, and Bibliographic Definitions, 2010. SEI - Carnegie Mellon `http://www.sei.cmu.edu/architecture/start/definitions.cfm`.

[129] B. Spitznagel and D. Garlan. A Compositional Approach for Constructing Connectors. In *Working IEEE/IFIP Conference on Software Architecture*, pages 148–157, 2001.

[130] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. 2nd ed. Addison-Wesley Professional, Boston, 2002.

[131] The Java Community Process. JSR 270: Java SE 6 Release Contents. Java Specification Request 270, December 2006.

[132] The Real-Time for Java Expert Group (RTJEG). JSR-000001 Real-time Specification for Java - (Maintenance Release 2), 2006. `http://www.rtsj.org`.

[133] Universidad de Cantabria. MAST: Modeling and Analysis Suite and Tools. `http://mast.unican.es`.

[134] Universidad Politécnica de Madrid. GNATforLEON cross-compilation system.

[135] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.

[136] T. Vardanega. Development of On-Board Embedded Real-Time Systems: An Engineering Approach. Technical Report ESA STR-260, European Space Agency, 1999.

[137] T. Vardanega. A Property-Preserving Reuse-Geared Approach to Model-Driven Development. In *Proc. of the 12th IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, 2006.

[138] T. Vardanega. Property Preservation and Composition with Guarantees: From ASSERT to CHESS. In *Proc. of the 12th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 125 –132, 2009.

[139] T. Vardanega and G. Caspersen. Engineering software reuse for on-board embedded real-time systems. *Software – Practice and Experience*, 32(3):233–264, 2002.

[140] T. Vardanega, J. Zamorano, and J. A. de la Puente. On the Dynamic Semantics and the Timing Behavior of Ravenscar Kernels. *Real-Time Systems*, 29:59–89, 2005.

[141] K. Wallnau and J. Ivers. Snapshot of CCL: A Language for Predictable Assembly. Technical Report CMU/SEI-2003-TN-025, Software Engineering Institute, Carnegie Mellon University, 2003.

[142] K. C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.

[143] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G.Bernat, C. Ferdinand, R.Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, G. Staschulat, and P. Stenströem. The worst-case execution time problem: overview of methods and survey of tools. *IEEE Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[144] R. Witwer. Developing the 777 Airplane Information Management System (AIMS): a View from Program Start to One Year of Service. *IEEE Transactions on Aerospace and Electronic Systems*, 33(2):637–641, 1997.

[145] J. Zamorano, J. F. Ruiz, and J. A. de la Puente. Implementing Ada.Real_Time.Clock and Absolute Delays in Real-Time Kernels. In *Proc. of the 6th International Conference on Reliable Software Technologies - Ada-Europe*, pages 317–327, 2001.