PhD-FSTC-2017-20

Dissertation

Presented on 13/03/2017 in Luxembourg
to obtain the degree of

Docteur de l'Université du Luxembourg
en Informatique
&
Docteur de l'Université de Toulouse
en Informatique
Délivré par : *l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)*

by

Guillaume Brau
Born on 28/07/1988 in Tarbes (France)

Integration of the Analysis of Non-Functional Properties in Model-Driven Engineering for Embedded Systems

Dissertation defence committee

Dr. Pierre Kelsen, Chairman (*Président*)
Professor, Université du Luxembourg

Dr. Pierre de Saqui-Sannes, Vice Chairman (*Président suppléant*)
Professeur des Universités, *Institut Supérieur de l'Aéronautique et de l'Espace*

Dr. Emmanuel Grolleau, *Rapporteur*
Professeur des Universités, *École Nationale Supérieure de Mécanique et d'Aérotechnique*

Dr. Frank Singhoff, *Rapporteur*
Professeur des Universités, *Université de Bretagne Occidentale*

Dr. Nicolas Navet, Dissertation Co-Supervisor (*Co-directeur de thèse*)
Associate professor, Université du Luxembourg

Dr. Jérôme Hugues, Dissertation Co-Supervisor (*Co-directeur de thèse*)
Maître de conférences, *Institut Supérieur de l'Aéronautique et de l'Espace*

Guillaume Brau

Integration of the Analysis of Non-Functional Properties in Model-Driven Engineering for Embedded Systems

General Concepts and Application to the Timing Analysis of Architectural Models

---

Intégration de l'analyse de propriétés non-fonctionnelles dans l'Ingénierie Dirigée par les Modèles pour les systèmes embarqués

Concepts généraux et application à l'analyse temporelle de modèles architecturaux

Version 2.0 (official version)

Abstract

The development of embedded systems is a complex and critical task, especially because of the non-functional requirements. In fact, embedded systems have to fulfill a set of non-functional properties dictated by their environment, expressed for example in terms of timing, dependability, security, or other performance criteria. In safety-critical applications for instance (e.g. an airplane), missing a non-functional requirement can have severe consequences, e.g. loss of life, personal injury, equipment damage, environmental disaster, etc.

Models and analyses are valuable asset to design complex embedded systems. Modeling enables to represent the system properties, whereas analysis makes it possible to evaluate them. Yet, modeling and analysis techniques have been historically investigated separately in software/systems engineering. On the one hand, Model-Driven Engineering uses domain-specific models as primary artifacts to develop a system. On the other hand, mathematically-based analysis techniques such as real-time scheduling analysis, model-checking, dependability analysis, etc. makes it possible to analyze the diverse non-functional properties of computer systems. Thus, a major contribution to improve the development of embedded systems, and the main objective of this thesis, will be to integrate models, as defined by the basic principles of Model-Driven Engineering (i.e. the triad model, metamodel, model transformation), with mathematical-funded analysis approaches to analyze the non-functional properties of embedded systems. This thesis aims at providing a general and coherent view on this problem by investigating two fundamental questions:

- How to apply an analysis on a model? (technical issue)
- How to manage the analysis process? (methodological issue)

This thesis advances several important concepts regarding the integration issue.

First of all, we revisit the way model transformations are done to accommodate specific analysis engines. Arguing that an analysis is less based on a particular model syntax than specific data, we promote query mechanisms called accessor to analyze the non-functional properties of a system at design time. These accessors enable to extract data from a model and then analyze them. Expected benefit is that analysis can be integrated to any kind of model as soon as an implementation of accessors to model internals is provided.

Next, we aim at formalizing the analysis execution. We show that an analysis is basically a program with preconditions and postconditions. The preconditions are the properties to hold true on an input model to successfully execute the analysis. Postconditions are the properties guaranteed on the model after the analysis execution. With preconditions and postconditions, an analysis is complete and sound.

Lastly, we abstract away from the execution aspect through the notion of contract. A contract completely defines the interfaces of an analysis in terms of processed data and properties. Inputs/Outputs (I/O) describe input and output data. Assumptions/Guarantees (A/G) describe input and output properties. SAT methods can then be used to automatically reason about these interfaces, and provide greater automation support: which analysis can be applied on a given model? Which are the analyses that meet a given goal? Are there analyses to be combined? Are there interference between analyses? Etc.

We evaluate different implementation of these concepts using multiple languages including general-purpose programming languages (Python), constraint languages (REAL), and specification languages (Alloy).

We investigate and apply these concepts for the timing analysis of architectural models. We illustrate the capabilities of our approach to deal with concrete systems coming from the aerospace: a drone, an exploratory robot and a flight management system. In particular, we demonstrate that accessors enable to apply real-time scheduling analyses onto different kinds of architectural models, e.g. written with the industry standard AADL (Architecture Analysis and Design Language) or the new time-triggered language CPAL (Cyber-Physical Action Language). In addition, contracts make it possible to automate complex analysis procedures and, to some extent, to mechanize the design process itself.

Keywords: Embedded Systems, Model-Driven Engineering, Analysis, Real-Time Scheduling, Contracts, Architecture Description Languages.

Acknowledgments

(Remerciements)

Contents

Chapter I

Introduction

Abstract

In this introduction chapter, we first present the context of the works and our research motivations. Next, we state the problems that we aim to address in this context. After that, we introduce the contributions which are provided in this thesis. We also detail the work hypotheses that fix the limits of these contributions. Lastly, we describe the organization of this manuscript.

I.1 Context and motivations

Software systems have become an integral part of our daily life, be it for work or entertainment through Personal Computers or laptops, for transportation in automobiles, trains or airplanes, to communicate via mobile networks or the Internet, but also for healthcare, energy management, economics and many other applications.

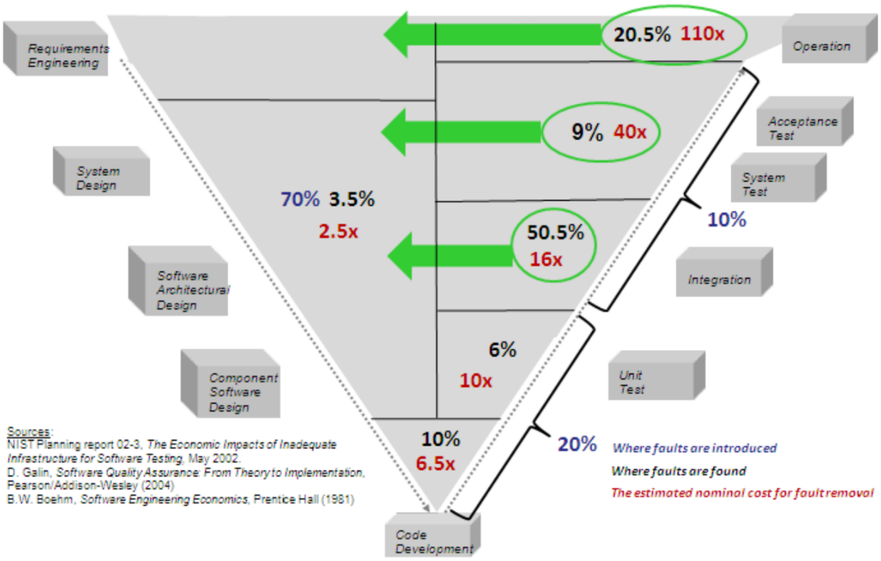
I.1.1 Non-functional requirements in embedded systems

An embedded system is a particular kind of computer system. Embedded systems consist of hardware, software, and an environment to interact with. In particular, embedded systems have to fulfill the non-functional requirements dictated by the environment, expressed for example in terms of timing, dependability, security, or other performance criteria. Embedded systems can be found in many application areas, especially in safety-critical applications such as aeronautics, space or automotive. In safety-critical applications, missing a non-functional requirement can have severe consequences, e.g. loss of life, personal injury, equipment damage, environmental harm, etc.

With ever increasing functionalities and growing complexity, embedded systems oblige not only to innovate in terms of technologies (e.g. IMA or TTA architectures, real-time computer networks, multi/many core systems, mixed criticality systems, etc.) but also to provide techniques and tools to develop them. In this thesis, we study state-of-the-art methods and tools to develop embedded systems.

I.1.2 Development process: towards analysis co-design

A system life-cycle is typically broken down in five main stages which are requirements engineering, design, implementation, Verification & Validation and, finally, operation. Several studies notice that the distance between design and V&V activities in current development processes results in costly regressions and reworks (see Figure I.1 for an example with the V-model).



Sources:
NIST Planning report 02-3, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, May 2002.
D. Galin, *Software Quality Assurance: From Theory to Implementation*, Pearson/Addison-Wesley (2004)
B.W. Boehm, *Software Engineering Economics*, Prentice Hall (1981)

Figure I.1: Benefits of Early Fault Discovery (taken from [1]). *In current development approaches, a majority of faults is introduced at early-stages while those faults are discovered late in the development process, according to Feiler et al. 70% of faults have their origins at design time while 80% of them are discovered after the implementation phase.*

Novel development approaches, such as Model-Driven Engineering [2] or Virtual Integration [1], shift from a system/test to a model/analysis paradigm. The watchword for these approaches could be “model, validate, then build”[1]. The core idea is to describe the system through many different *models*, possibly integrate the viewpoints, and verify/validate the system *at design time*. Then, the system can be manually built or (semi-)automatically generated from models. With this approach, the design process consists of a set of modeling and analysis steps: models are used to define the system from high-level models to low-level models and code, whereas analyses are applied on such models to gradually validate or invalidate the design choices. The system is “correct-by-design”.

In this thesis, we explore a Model-Driven Engineering approach that systematically combines models and analyses to develop embedded systems, emphasizing especially on the design phase.

[1] originally “integrate then build” in [1]

I.1.3 Coupling models and analyses

Modeling and analysis are dual activities to comprehend any system, be it to explain the Solar System, understand a social system, architect a house, or design a computer system. Modeling enables to represent a system, whereas analysis makes it possible to dissect this system.

In embedded systems engineering, modeling and analysis solutions have been investigated separately. On the one hand, Model-Driven Engineering is an engineering approach that focuses on domain-specific models so as to develop software systems. On the other hand, mathematical-funded analysis approaches such as real-time scheduling analysis, *model-checking*, dependability analysis, etc. make it possible to analyze the diverse non-functional properties of embedded systems.

A major contribution to improve the development of embedded systems, and the main objective of this thesis, will be to integrate models, as defined by the basic principles of Model-Driven Engineering (i.e. the triad model, metamodel, model transformation), with mathematical-funded analysis approaches to analyze the non-functional properties of embedded systems. In this thesis, we concentrate on architectural modeling through Architecture Description Languages, and real-time scheduling analysis. We explain the problem in greater detail and subsequent research lines in the next sections.

I.2 Problem statement

The integration of models and analyses raises two fundamental questions:

- How to apply an analysis on a model? (technical issue)
- How to manage the analysis process? (methodological issue)

I.2.1 How to apply an analysis on a model?

Modeling and analysis features are usually provided as part of distinct tools:

1. languages such as AADL [3], EAST-ADL (combined with AUTOSAR) [4, 5], or SysML and MARTE UML profiles [6, 7] provide standardized notations for modeling system architectures;
2. analytical frameworks for Verification & Validation activities targeting real-time scheduling tools [8, 9], model-checkers [10, 11], etc.

An approach commonly used to connect the toolsets, known as model transformation, is to translate a model used for design into a model used for analysis. For example, see [12] for a survey on model transformations to analyze the non-functional properties of AADL models (i.e. behavior, schedulability, timing and dependability).

In that context, one can either implement a comprehensive model transformation (e.g. metamodeling under the MOF standard [13], in the Eclipse Modeling Framework [14], transformation with a dedicated language such as ATL [15]); or more probably relies on an *ad hoc* transformation chain to deal with the design and analysis models under different technical spaces. Yet, we note two main drawbacks with this approach:

(1) one must define a multiplicity of transformations attached to specific tools,

(2) in the current state of the art, ensuring the correctness of model transformations is yet an unsolved problem (see works by Amrani [16] on this topic).

Depending on its complexity, we note that the analysis can be operated directly from the modeling tool using query languages (e.g. OCL). Thus, a first research direction in this thesis will be to further explore and compare means to analyze the non-function properties of a system from architectural models.

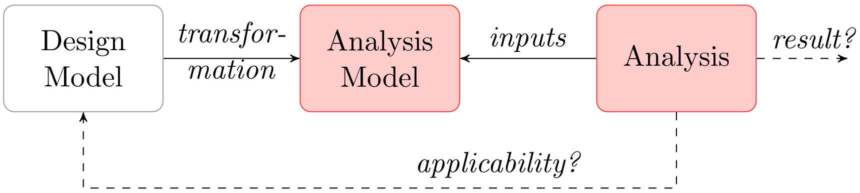


Figure I.2: Analysis in Model-Based Systems Engineering is based on a transformation process. *Design and analysis features are part of distinct tools: (1) a model used for design in a first tool is translated into a model used for analysis in a third-party tool; (2) the analysis in the third-party tool is then applied on its own model. This approach does not address the validity of the transformation: is the analysis applicable on the model which is considered? Upstream, the analysis result is not handled: what is the meaning of the analysis result?*

I.2.2 How to manage the analysis process?

From technical analysis solutions, engineers must be able to manage the analysis process. These are some questions faced by an engineer when applying an analysis:

When to apply the analysis? On the one hand, an analysis is carried out according to a precise analytical model, e.g. a task model. On the other hand, an analysis fulfills a particular objective, e.g. it provides a result about the schedulability of the system. The correct application of an analysis is thus a consistent association between a model, an analysis and an objective: the model in input must comply with the analysis expectations (data required by the analysis, respect of the assumptions made by the analysis, etc.), and the analysis must meet the objectives of the engineer in output.

What to do with the analysis result? Carrying out an analysis is not a dead-end. Firstly, an analysis may report on an engineering goal: performances, timing, safety, etc. Secondly, elementary analysis results may be combined to build wider results; or must be computed in a precise order to be sound.

To answer these questions, we must investigate a more systematic approach that will enable to manage analysis activities at design time. This approach must be supported by MDE tools alongside modeling languages and analysis engines.

In the next section, we explain the research lines explored in this thesis, and introduce our contributions to tackle the aforementioned issues. We also explain the hypotheses that delimit our works.

I.3 Lines of research and contributions

From the problem statement, we explore three complementary lines of research:

R1: exploration of means to analyze the non-functional properties of a system from its models,

R2: investigation of the semantics of an analysis and reasoning on the analysis process.

R3: practical application of these concepts and experimentation through case studies.

R1 and **R2** target conceptual and practical solutions for the problems stated above. **R3** is more application-oriented and seeks to evaluate the benefits of combining models and analyses for engineering real embedded systems. The remainder of the section sums up the four contributions of this thesis with respect to these lines of research.

I.3.1 Technical integration through model query

C1: model query focuses on the technical issue behind the analysis of a model. We tackle the problem from a different standpoint compared to related works that emphasize on model transformation. Arguing that an analysis is less based on a particular model syntax than specific *data*, we promote query mechanisms called *accessor* to analyze the non-functional properties of a system at design time. These accessors enable to extract data from a model and then analyze them. An expected benefit is that an analysis can be integrated to any kind of model as soon as an implementation of accessors to model internals is provided. Another advantage is that an analysis could be easily implemented by using a general-purpose programming language (e.g. Python) instead of relying on specific analysis engines.

I.3.2 Semantic integration and decision through contracts

C2: semantics of an analysis. In a second time, we aim at formalizing the semantics of an analysis. We show that an analysis is basically a program with preconditions and postconditions (i.e. like a Floyd-Hoare triple). The preconditions are the properties to hold true on an input model to successfully execute the analysis. Postconditions are the properties guaranteed on the model after the analysis execution. We show that a full analysis, including preconditions and postconditions, can be implemented through above-mentioned accessors.

C3: contract-driven analysis. We extend the previous contribution through the notion of contract, semantically equivalent to a Floyd-Hoare triple. A contract completely defines the interfaces of an analysis in terms of processed data and properties. Inputs/Outputs (I/O) describe input and output data. Assumptions/Guarantees (A/G) describe input and output properties. SAT methods can then be used to automatically reason about these interfaces, and answer complex questions about the analysis process: *which analysis can be applied on a given model? Which are the analyses that meet a given goal? Are there analyses to be combined? Are there interferences between analyses?* Etc. In practice, contracts can be defined with the help of a specification language such as Alloy, and evaluated through associated SAT solvers.

I.3.3 Proof-of-concept analysis and orchestration tool

C4: proof-of-concept tool. As an example of application, we propose a proof-of-concept tool that enables not only to analyze architectural models but also to orchestrate the analysis process. This tool implements several functions, each one implementing a part of the concepts introduced earlier. In particular, our tool provides accessors towards AADL and CPAL models, various real-time scheduling analyses programmed in Python, and an orchestration module based on Alloy. We illustrate the capabilities of such a tool on various case studies coming from the aerospace. Through these case studies, we show that our tool enables not only to automate the analysis process at design time but also to enhance the design process by systematically combining models and analyses.

I.4 Work hypotheses

The three following hypotheses fix the limits of our contributions. These hypotheses may be relaxed in future works.

H1: embedded systems. We concentrate on embedded systems [17]. Embedded systems are computer systems that present two special features: (1) they consist of hardware, software and an environment; (2) they have to meet *non-functional properties* dictated by the environment.

H2: design through architectural description languages. We focus on early design phases, especially the architectural design stage. For this purpose, we study two particular Architecture Description Languages: the Architecture Analysis and Design Language (AADL) [18], an industry standardized language to describe the architecture of real-time embedded systems, and the Cyber-Physical Action Language (CPAL), a new language for the model-driven development and real-time execution of Cyber-Physical Systems (CPS) [19].

H3: real-time properties. We concentrate on real-time properties. A real-time system is a system for which the "the correctness depends not only on the logical result of the computation but also on the time at which the results are produced"

[20]. Worst-Case Execution Times (WCET), Worst-Case Response Times (WCRT) and Worst-Case Traversal Times (WCTT) are examples of real-time constraints to be analyzed. For this purpose, we emphasize on a particular kind of analytical methods called real-time scheduling analyses [21].

I.5 Thesis organization

This thesis is organized into nine chapters. The core chapters are split into two subsequent parts: concepts and application.

Part 1 (Concepts) presents both the concepts preceding our works and the concepts contributed in this thesis.

> **Chapter II (Background)** introduces the necessary background concepts related to embedded systems, model-driven engineering and model-based analysis. In particular, we present two Architecture Description Languages (ADL) used in this thesis, namely the Architecture Description Language (AADL) and the Cyber-Physical Action Language (CPAL). We also introduce the important concepts of real-time scheduling analysis.
>
> **Chapter III (Model query)** deals with model query. It presents query mechanisms, called accessors, to analyze the non-functional properties of a system from architectural models. This chapter explains the rationale behind model query and presents an implementation of accessors through a dedicated Application Programming Interface.
>
> **Chapter IV (Semantics of an analysis)** focuses on the analysis, especially its semantics. This chapter firstly shows that a full analysis consists of preconditions, the analysis itself, and postconditions. Then, we evaluate several implementation means, including both specialized constraint languages and more generic accessors.
>
> **Chapter V (Contract-driven analysis)** explores the notion of contract. Contracts specify the interface of an analysis in terms of processed data and properties, and allow for automatic reasoning on analysis interfaces. In a proof-of-concept, we show that contracts can be defined with the help of a specification language such as Alloy, and evaluated through associated SAT solvers. In this way, we are able to systematize the analysis activities at design time.

Part 2 (Application) presents an implementation of these concepts through a tool prototype and experiments these concepts on various case studies.

> **Chapter VI (Tool prototype)** describes a tool prototype that implements the various concepts introduced in the first part of the thesis. This proof-of-concept tool implements several functions so as to automate analysis activities at design time. In particular, our tool implements accessors towards AADL and CPAL

models, analyses programmed in Python, and an orchestration module based on Alloy.

Chapter VII (Case studies) presents several case studies. We use the prototype tool presented in the previous chapter to experiment a design workflow that combines architectural models and analyses. We describe three cases studies: an open-source drone named *Paparazzi*, the *Mars Pathfinder* exploratory robot, a *Flight Management System*. Through these case studies, we show that the analysis is an integral part of the design process.

This dissertation finishes with a general conclusion and some perspectives.

Chapter VIII (Conclusion) recaps the content of this thesis and summarizes the main results.

Chapter IX (Perspectives) sketches potential improvements, extensions and research directions to continue the work initiated in this thesis.

Part 1

Concepts

Chapter II

Background

Abstract

This chapter presents the general concepts that are necessary to comprehend the issue tackled in this thesis and proposed contributions regarding methods and tools to develop real-time embedded systems. We firstly present the special features of embedded systems in Section II.1. In particular, a major problem related to embedded systems is to cope with non-functional properties, e.g. real-time, safety or security. We consider two complementary approaches to that end. On the one hand (Section II.2), Model-Driven Engineering (MDE) is a generative engineering approach based on the triad model, metamodel, model transformation. At the core of MDE, Domain-Specific Modeling Languages enable to form models, especially through Architecture Description Languages (ADL) during early stages of design. We present two particular ADLs: the Architecture Analysis and Design Language (AADL) and the Cyber-Physical Action Language (CPAL). On the other hand (Section II.3), we focus on model-based analyses, that is to say approaches that apply mathematical reasoning from an analytical representation of the system to check some non-functional properties. We concentrate on real-time scheduling analysis. Real-time scheduling analysis determines whether a task system meet some timing constraints or not (e.g. deadlines). In Section II.4, we discuss the link between MDE and analysis that founded the motivation of our works. We finally conclude this chapter in Section II.5.

II.1 Embedded systems

This thesis deals with the modeling and analysis of embedded systems at large. An embedded system is a specific kind of computer system.

Definition 1 (Embedded system)**.** *An embedded system is an engineering artifact involving computation that is subject to physical constraints. Embedded systems consist of hardware, software, and an environment. [22]*

In particular, an embedded system possesses the following core features [22, 23, 17, 24, 25, 26]:

- it is made up of a combination of hardware and software components,

- it is designed to perform a fixed function, specific to an application,
- it interacts with the external physical world and has to meet the constraints dictated by the environment,
- it is part of a larger system.

II.1.1 Hardware and software architecture

An embedded system combines hardware and software components in order to carry out a fixed function, specific to the application. At the highest level, we can represent the major elements of an embedded system with the layered model in Figure II.1.

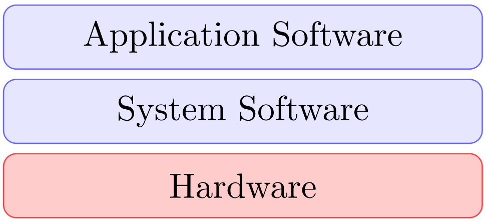


Figure II.1: Embedded systems model (after [23]).

The hardware layer contains the physical components provided by an embedded board. Hardware typically consists of processors, memories, data storage, input/output devices, communication networks, etc. The system and application software layers contain the software being executed by the embedded system. The system software layer provides abstractions between the hardware and application software such as device drivers, operating systems or middlewares. The application software layer finally contains application-specific software that run on top of the system software layer. With that architecture, the application can be programmed through the various services provided by the system software layer, without interfacing directly with the physical components.

II.1.2 Non-functional constraints

Embedded systems have to meet specific non-functional constraints [24, 27]. We briefly present some of these constraints in the next paragraphs.

Small size, low weight. Embedded systems are physically located in larger systems. Therefore, they may have to fit into a restricted place between electrical or mechanical components, for instance Electronic Control Units (ECU) in cars. Weight may also be critical, for example for fuel economy or when it impacts the dynamics of the embedding vehicle (aircraft, spacecrafts, small-sized vehicles such as drones), or simply for ergonomics (portable equipment such as laptops).

Real-time operation. Embedded systems continuously interact with the external physical world. Real-time, which is the physical time in the environment of the system, is an integral part of embedded systems [25, 17].

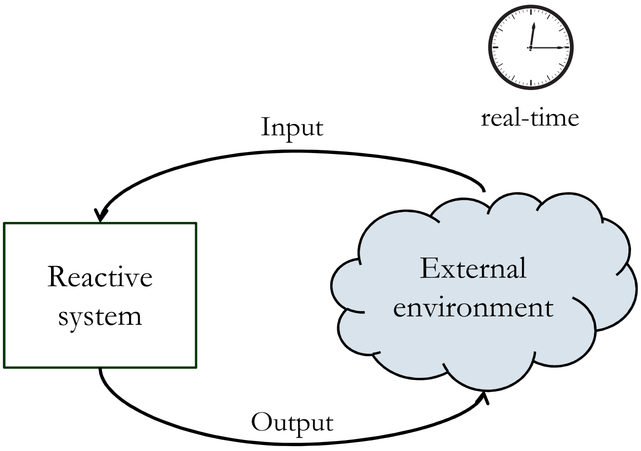


Figure II.2: Interaction between an embedded systems and the external physical world.

Definition 2 (Real-time system)**.** *A real-time system is a system for which the correctness depends not only on the logical result of the computation but also on the time at which the results are produced [20].*

More precisely, the computer system must *react* in constraint time to external events, in order to keep control of the external process [28]. Typically, tasks executed by the computer have deadlines, which is the time by which the task must be completed. More generally, embedded systems can have to fulfill many different kinds of temporal constraints, not just deadlines: a task must be executed no earlier than a precise time; a task must be executed strictly periodically, or can accept a jitter; a task may be required to be executed after another task; etc.

Control/command systems or process control systems are typical examples of reactive/real-time systems. We can further classify real-time systems according to their criticality [29]. For example, we distinguish between *hard* [30, 31], *soft* [32] and *mixed-criticality* [33, 34] real-time systems. Violating a temporal constraint in a *hard* real-time system can have catastrophic consequences. Systems to pilot and aircraft, to control a critical chemical process, or to monitor health of a patient are some examples of hard real-time systems.

Safe and reliable. Embedded systems can be used in applications where delivering the correct service is vital to achieve the mission or ensure the safety of the public or the environment. Those systems are referred to as mission- or safety-critical systems [35]. A failure of the system (caused for instance by a real-time fault or a hardware fault) can have catastrophic consequences: loss of life, personal injury, equipment damage, environmental damage, etc. A life-support system in an intensive care unit is an example of safety-critical systems. We can mention aircraft flight control or nuclear systems as other examples.

Safety-critical embedded systems must be *dependable*. Dependability is "the ability to deliver a service that can justifiably be trusted" (Avizienis et al. [36]). Dependability is an integrating concept that encompasses numerous attributes such as safety – the absence of catastrophic consequences on the user(s) and the environment –, reliability – the continuity of correct service –, availability – readiness for correct service –, and so on.

Other performance constraints. Embedded systems may have to cope with a wide range of performance constraints (i.e. performance measures) [23, 26]: power consumption, processor throughput, various memory usage, network bandwidth, etc.

II.1.3 Development process

The development of embedded systems is a complex and critical task. It is hence based on systematic activities as part of a development process. Each activity produces a different result (requirements document, design models, programs, etc.) with the goal to produce the right system.

There exist plenty of development processes that lead to the production of a system. There are fundamental activities which are common to all processes: *requirements*, *design*, *implementation*, *Verification* & *Validation* and *operations* [37].

Requirements fully express the functionality that the system must provide, and constraints on its operation. This activity is also called requirement engineering. We distinguish the functional requirements which are the basic functions that the system must provide ("*what* the system must do") from the non-functional requirements which are the constraints that the system must fulfill to correctly behave in its operational environment ("*how* the system performs a specific function"). Real-time operation, low power-consumption, dependability or security are examples of non-functional requirements. Requirement activities output a *requirements document.*

Design defines all the aspects of the system which are necessary to meet the requirements, including software and hardware concerns. A system design describes for example the subsystems, the components of the (sub)systems, the interfaces between components, the data used in the system, the algorithms, the protocols, etc. The design process may involve the production of several *models* of the system at different levels of abstraction.

Implementation realizes the system design with all the required material: hardware, programs, configuration files, etc. The implementation phase results in an *executable system.*

Verification & Validation ensure that the system meets the functional and non-functional requirements. We distinguish two main approaches:

- *analyses* that are carried out on system models such as the requirements document, design models and the program source code,
- *tests* which are conducted on the product system.

Analyses can be performed at all stages of the process as they operate on a representation of the system. Conversely, testing is only applicable at late design stages, when the product system is available.

Operations represent the last phase of the development process. At this stage, the system has been delivered and is operating in its environment. Operations may involve extra activities such as correction of undetected errors, product improvements, enhancement with new requirements, etc.

Process models represent system processes. For instance, Figure II.3 depicts the aforementioned activities as separate process phases in the classic waterfall model [38]. In this process, the phases/activities are "cascaded". In theory, a phase can only start if the previous one has finished. In practice, the process progress is rarely linear and may involve several iterations over preceding and succeeding steps. Other process models such as the V-model, the spiral model or the iterative model organize these tasks in different ways [39, 37].

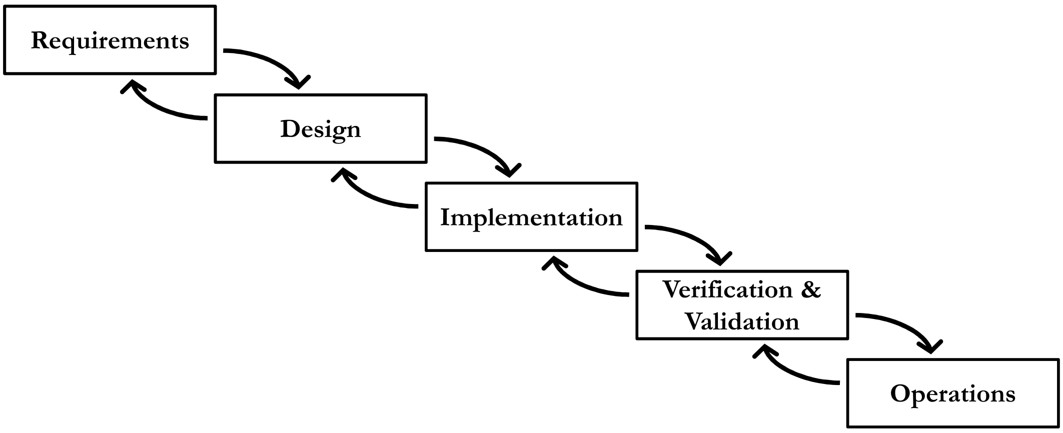


Figure II.3: Waterfall development process model (after [38]).

II.2 Model-Driven Engineering

Model-Driven Engineering (MDE) is a paradigm which considers models as primary artifacts to develop software systems.

Definition 3 (Model-Driven Development)**.** *Model-Driven Engineering describes software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations [2]*

II.2.1 What is a model?

The watchword of Model-Driven Engineering is "everything is a model" [40]. The literature proposes plenty of definitions of the notion of *model*. We retain the following definition in the context of this thesis.

Definition 4 (Model)**.** *A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [41].*

Therefore, a model is an abstraction of a subject system. We can possibly represent a system with various models related with each other, e.g. as many different points of view, e.g. see [42, 43].

The next definition emphasizes that a model must be written with a language. This language might be plain English, a programming language, or a dedicated modeling language called a Domain-Specific Language [44, 45].

Definition 5 (Model (language))**.** *A model is a description of (part of) a system written in a well-defined language. [46].*

*A well-defined language is a language with well-defined form (*syntax*), and meaning (*semantics*), which is suitable for automated interpretation by a computer [46].*

Figure II.5 depicts the relationships between a model, the system it represents, and the language in which it is written.

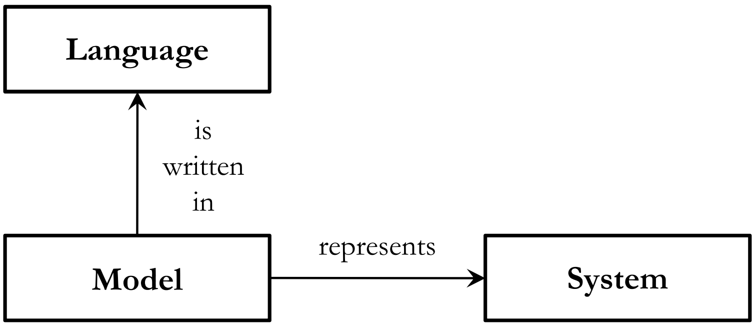


Figure II.4: Relationships between a model, a subject system, and a language (after [46]).

General-Purpose Modeling vs. Domain-Specific Modeling. We usually distinguish between general-purpose modeling languages (GPML) and domain-specific modeling languages (DSML). Contrary to general-purpose modeling languages which provide universal concepts (e.g. the UML [47]), domain-specific modeling languages are specialized languages which focus on a particular domain [44, 45]. Domain-specific modeling languages directly capture the high-level concepts of a subject domain. Thereby DSMLs improve the efficiency of models as they are easier to understand and learn for a domain expert, but also more easily transformable, analyzable, etc. Of course, the use of a DSML is restricted to a specific domain, meaning that many DSMLs are necessary to cover all the aspects of a system. We discuss the definition of DSMLs through metamodels in the MDE context hereinafter.

II.2.2 Notions of metamodeling

The mechanism to define a language in Model-Driven Engineering is called *metamodeling.*

What is a language? Any language, be it considered in linguistic or in computer sciences, consists of a *syntax* and a *semantics*. The syntax refers to the representation of a language, the elements that form the language – words, sentences, boxes, diagrams, etc. –, while the semantics deals with the meaning of this language [48].

Definition 6 (Language)**.** *A language is a tuple* $\{S, Sem\}$ *with* S *is the syntax of the language and* Sem *is the semantics [46, 49].*

In the context of modeling languages in particular [48, 49]:

- the abstract syntax, manipulated by a computer, defines the structure of the language, that is the concepts of the language and the relationships between them,
- the concrete syntax, manipulated by the end-user, describes a specific – human-readable – representation of these concepts with a textual or graphical formalism,
- the semantics of a modeling language is defined through a semantic domain, and a mapping of the syntactic elements to the semantic domain. There are several ways to describe the semantics of a language, among which *operational semantics* or *denotational semantics* to define the *behavioral* semantics of a domain-specific modeling language.

Figure II.5 represents the relationship between those concepts.

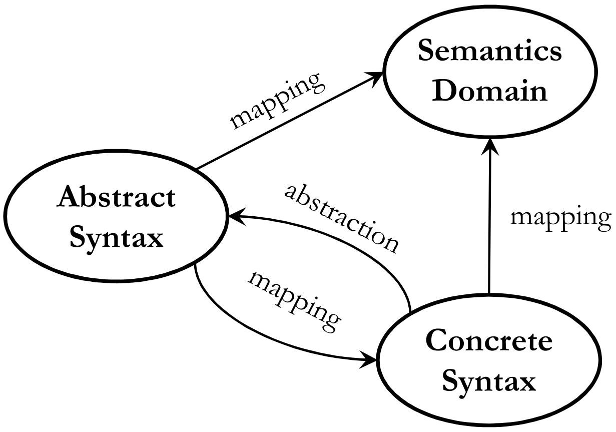


Figure II.5: Components of a language (after [49])

Metamodel. Naturally, in MDE, modeling languages are themselves defined by specific models, called *metamodels*.

Metamodels enable to structure models by defining the abstract syntax of the modeling language. A metamodel precisely define the elements that can be used in a language together with their relationships, and complete the structural description with the well-formedness rules that must be respected by the conforming models [50].

Yet, Kleppe notices that a metamodel is any model that is part of a language specification, not only defining the abstract syntax of the language but also the concrete syntax or the semantic domain [51].

Definition 7 (Metamodel)**.** *A metamodel is a model that defines the language for expressing a model.[13]*

A metamodel itself must be written in a well-defined language. We call metalanguage this specific language used to describe modeling languages. Figure II.6 shows the metamodeling approach. Because a metalanguage is itself a language, it should be defined by a metamodel, called meta-metamodel, written in another metalanguage. To limit the number of abstractions, the meta-metamodel must be able to describe itself.This phenomenon is known as the meta-circularity property of meta-metamodels.

Examples of metalanguages include MOF and EMOF standards by the OMG [13], Eclipse Ecore implementation of EMOF [14] or Kermeta [52].

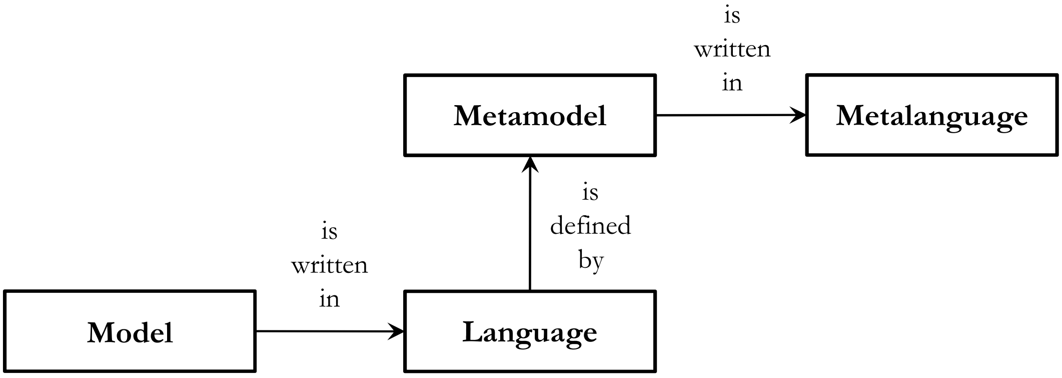


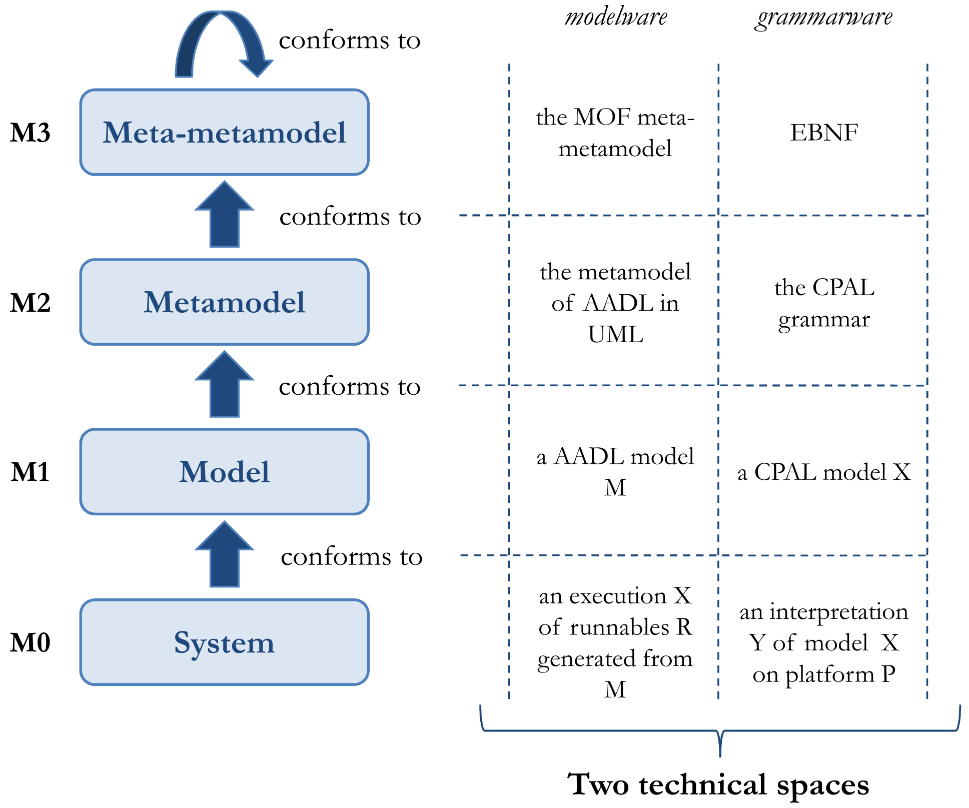
Figure II.6: Metamodels define models (after [46])

The four modeling layers. Therefore, the models can be represented in four layers [46, 40] as shown in Figure II.7. A model at a level *conforms to* the model at the upper level.

The $M0$ layer, the instances in the real world, corresponds to the running system. The $M1$ layer contains models. A model represents the system with a language. Metamodels at the $M2$ layer defines the modeling language used by $M1$. The $M3$ layer finally defines the meta-metamodel that describe the metalanguage. The meta-metamodel is defined in terms of itself.

Every different metamodeling architecture defines a *technical space* [53, 54]. The left part of Figure II.7 shows two examples of metamodeling architectures used in this thesis, as part of different technical spaces. The first architecture contains elements of the AADL language in a *modelware*, defined from the MOF meta-metamodel. The second architecture depicts elements of the CPAL language in a *grammarware*, based on the Extended Backus-Naur Form (EBNF). Examples of model instances in the real world are execution of C/ARINC653 runnables generated from a AADL model, or CPAL files interpreted on top of a Raspberry Pi platform.

Definition 8 (Technical space)**.** *A technical space is a set of tools and techniques attached to a pyramid of metamodels which is defined by a family of similar (meta-)metamodels [55].*



	modelware	*grammarware*
M3	the MOF meta-metamodel	EBNF
M2	the metamodel of AADL in UML	the CPAL grammar
M1	a AADL model M	a CPAL model X
M0	an execution X of runnables R generated from M	an interpretation Y of model X on platform P

Figure II.7: A particular metamodeling pyramid defines a specific technical space.

II.2.3 Notions of model transformation

Model transformation is the third pillar of Model-Driven Engineering. It automates various manipulations of models. Model refinement (vs. abstraction), synthesis/code generation (vs. reverse engineering), translation or analysis are some intents behind a model transformation [56, 16, 57].

Definition 9 (Model transformation)**.** *A model transformation is the automatic generation of a target model from a source model, according to a transformation definition [46].*

Figure II.8 represents the elements that participate in a model transformation [46, 16]:

- an input model, written in a source language, is transformed into an output model, written in a target language, by executing a transformation definition,
- a transformation definition, written in a transformation language, describes how a model in a source language can be transformed into a model in a target language,

- source, target and transformation languages are defined in terms of a metalanguage.

Notice that a model transformation is a function between abstract syntax and/or concrete syntaxes [51, 50]. Guaranteeing the semantics of model transformations is the subject of dedicated researches, e.g. see [16].

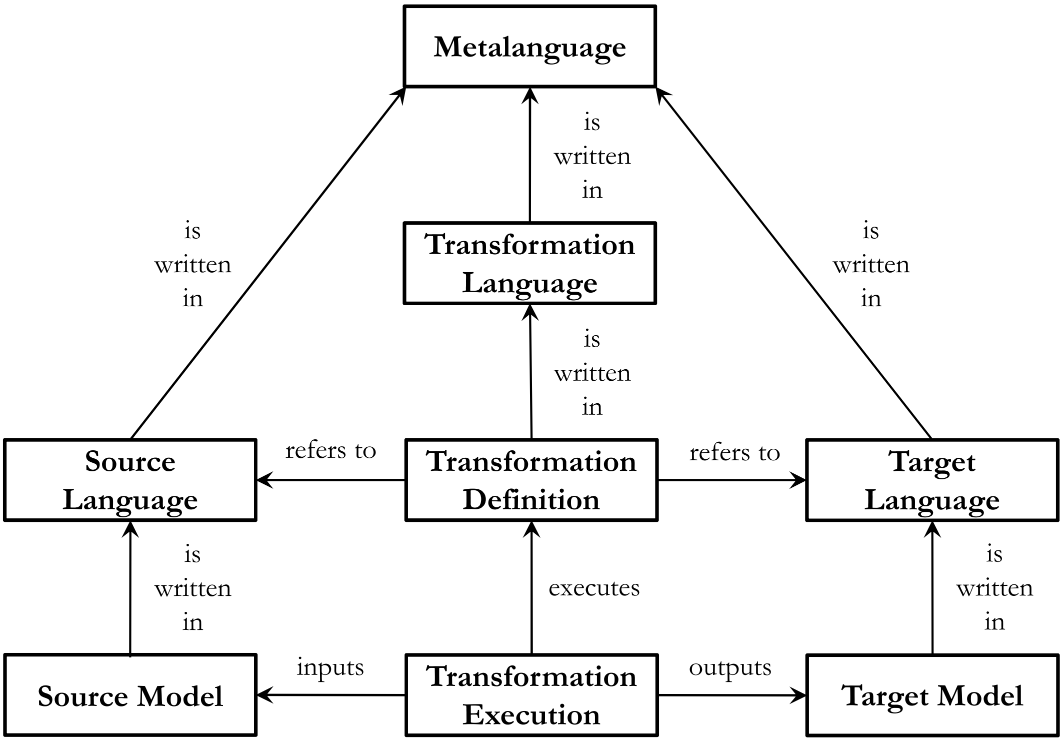


Figure II.8: Components involved in a model transformation (after [16]).

Model transformation can be classified following many different criteria, e.g. see [58, 59, 56]. A common classification of model transformations considers the source and target languages [58]. *Endogenous* transformation refers to models expressed in the same language. At the opposite, *exogenous* transformations are defined on different languages. Transformations can be further classified by considering the abstraction level of the source and target models [58]. A *horizontal* transformation is a transformation that considers source and target models at the same level of abstraction. A *vertical* transformation considers source and target models at different abstraction levels. Czarnecki and Helsen [59] propose another classification to distinguish between *model-to-text* and *model-to-code* transformation approaches. Kleppe [60] proposes a taxonomy of transformations based on the elements of a language, e.g. Kleppe defines *in-place* transformation, *view transformations* or *structure transformations* according to the transformation between abstract syntaxes and/or concrete syntaxes. More recently, Amrani et al. [56] proposed a classification of model transformation based on an intent catalog.

There exist many transformations languages, based on different approaches [59]. We can mention programming-based approaches that associate an internal model representation to an API in order to manipulate models directly (e.g. based on JMI or EMF [50]), or approaches based on dedicated model transformation languages such as ATL (Atlas Transformation Language) [15], Kermeta [52] or QVT (Query/View/-Transform) [61, 62], the OMG standard language to specify model transformations.

II.2.4 Case study: Architecture Description Languages

In this thesis, we concentrate on a specific kind of domain-specific language called Architecture Description Language (ADL) [63, 64].

Architecture Description Languages capture both the static structure of a system and its behavior. ADL are especially useful during the preliminary – architectural – design stage. For example, Figure II.9 shows the positioning of ADLs in the waterfall model.

Definition 10 (Architecture Description Language)**.** *An architecture description language is a formal language that can be used to represent the architecture of a software-intensive system. By architecture, we mean the components that comprise a system, the behavioral specifications for those components, and the patterns and mechanisms for interactions among them. [63]*

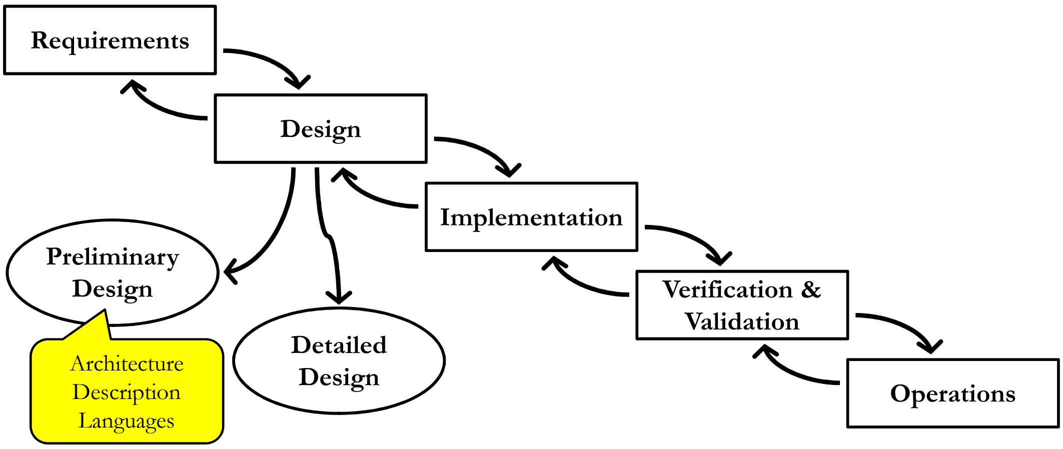


Figure II.9: Positioning of Architecture Description Languages in the (waterfall) development process.

Numerous Architecture Description Languages exist. In this thesis, we study two particular ADLs: the Architecture Analysis and Design Language (AADL), an SAE International standard [3], and the Cyber-Physical Action Language (CPAL), a new language inspired by the synchronous programming approach [19]. We briefly present these languages thereafter.

II.2.4.A AADL: the Architecture Analysis and Design Language

AADL at a glance. The Architecture Analysis and Design Language (AADL) is an ADL dedicated to "the specification, analysis, automated integration and code generation of real-time performance-critical (timing, safety, schedulability, fault tolerant, security, etc.) distributed computer systems"[1]. AADL is an SAE International

[1] http://www.aadl.info/ accessed September 2016

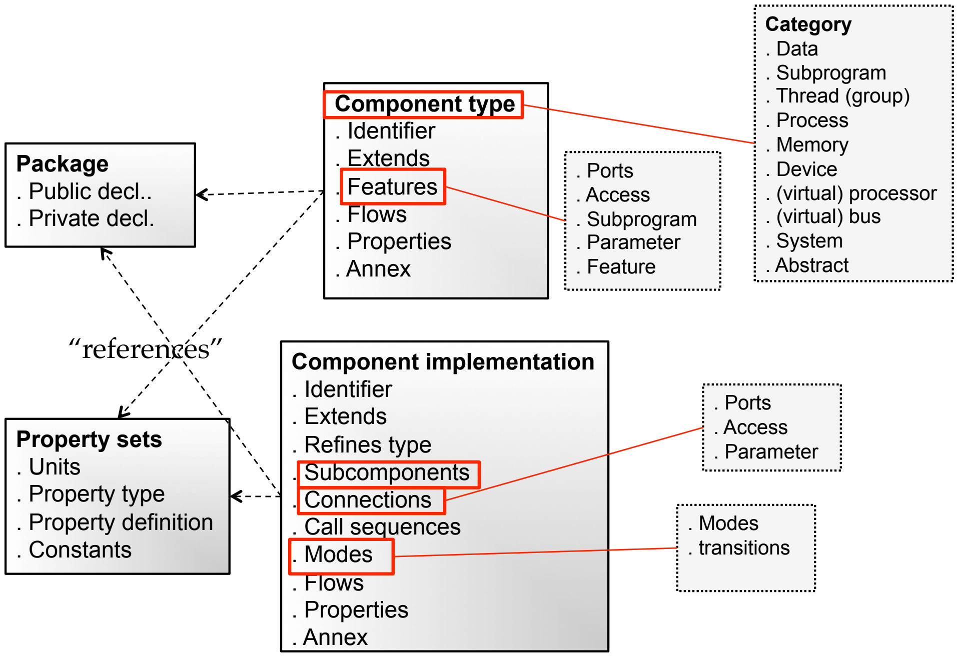


Figure II.10: Simplified meta-model of AADL (taken from [67])

standard [3]. AADL originates from the former MetaH language [65, 66] and has been improved and revised several times[2].

AADL is a textual language first, but also has a graphical representation [18]. It represents both the static and dynamic architecture of a system:

- the static architecture consists in a hierarchy of interacting software and hardware components,
- the dynamic architecture describes operational modes, connection configurations, fault tolerant configurations, behaviors of individual components, etc.

AADL focuses on the definition of clear components interfaces, and separates the implementations from these interfaces. From the separate description of these components, one can build an assembly of components that represent the full system. To take into account the multiple interactions between components, the AADL defines different patterns: subcomponent, connection and binding.

An AADL model can incorporate non-architectural elements: non-functional properties (execution time, memory footprint, . . .), behavioral or fault descriptions. Hence it is possible to use AADL as a backbone to describe all the aspects of a system.

Figure II.10 depicts the main concepts of AADL. Let us review these elements in more detail.

Components. An AADL description is made of *components*. Each component category describes well-identified elements of the actual architecture, using the same

[2] AADLv2.1 is the latest version to date, from September 2012. AADLv2.2 and AADLv3 are in the planning stage.

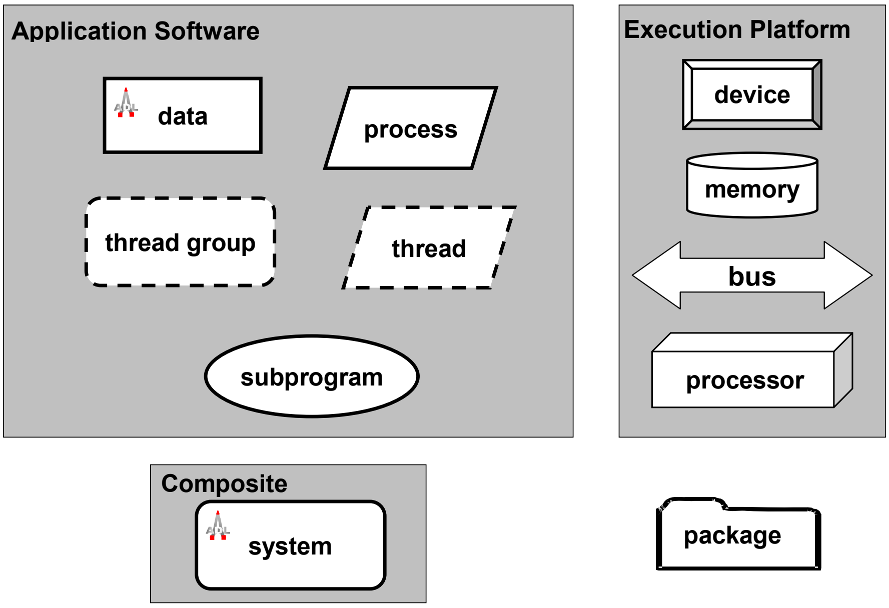


Figure II.11: Graphical representation of the main AADL components (taken from [18])

vocabulary of system or software engineering. The AADL standard defines three categories of components:

- application software components: `data`, `thread`, `thread group`, `subprogram` and `process`,
- execution platform components: `memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`,
- composite components (`system`) or imprecise (`abstract`).

Figure II.11 shows the graphical concrete syntax of the different kinds of components.

A component is to be declared in two parts: the *component type* and the *component implementation*. The interface of a component is called the *component type*. A component type firstly defines the external interface in terms of *features*. Features can be ports, subprograms or data accesses depending on the communication scheme. In addition, a components type defines *properties*. Properties are typed attributes that specify constraints or characteristics that apply to the elements of the architecture such as clock frequency of a processor, execution time of a thread, bandwidth of a bus. Some standard properties are defined, e.g. for timing aspects; but it is possible to define new properties for different analysis (e.g. to define particular security policies). Each type is optionally attached with one or several *implementations*. Implementation describes the internal structure of the components: subcomponents, their connections, behavioral specification, source code, etc. They can also refine non-functional properties.

```

-- Data
data a_data
properties
  Source_Data_Size => 4 Bytes;
end a_data;

-- Subprograms
subprogram Produce_Spg
features
  output_parameter : out parameter a_data;
properties
  Source_Language => (C);
  Source_Text     => ("foo.c");
  Compute_Execution_Time => 150 ms . . 200 ms ;
end Produce_Spg;

-- Threads
thread Producer
features
  out_data : out event data port a_data;
properties
  Dispatch_Protocol => Periodic;
  Period => 500 ms;
end Producer;

thread implementation Producer.Impl
calls
  call_subprogram : { the_subprogram : subprogram Produce_Spg;
      };
connections
  parameter the_subprogram.output_parameter -> out_data ;
end Producer.Impl;

thread Consumer
features
  in_data : in event data port a_data;
properties
  -- Omitted
end Consumer;

thread implementation Consumer.Impl
  -- Omitted
end Producer.Impl;

-- Process
process pc
end pc;

process implementation pc.Impl
subcomponents
  Prod : thread Producer.Impl;
  Cons   : thread Consumer.Impl
connections
  c1 : port Prod.out_data -> Cons.in_data;
end pc.Impl;
```

Listing II.1: Producer/consumer software elements in AADL.

Listing II.1 illustrates these concepts on a producer/consumer example. For instance, a specific `thread Producer` type is declared at line 19. The component type defines an output port to connect with another component, together with the main real-time properties to describe the timing behavior of that type of thread. The implementation at line 27 specifies subprogram calls to carry out this thread. The subprogram type declared at line 9 references the actual source code of the program within its properties.

Component declarations have to be instantiated into subcomponents of other components in order to form the system architecture. For example, in Listing II.1, the producer/consumer process at line 49 has two subcomponents, i.e. a producer thread `Prod` and a consumer thread `Cons`. At the top-level, a system contains all the component instances. Most components can have subcomponents, so that an AADL description is hierarchical. A complete AADL description must provide a top-most level system that will contain certain kinds of components (*processor*, *process*, *bus*, *device*, *abstract* and *memory*), thus providing the root of the architecture tree. The architecture in itself is the instantiation of this system, which is called the *root system*. Listing II.2 depicts the Producer/consumer root system. At line 6, the system implementation consists of the process and underlying software elements defined in Listing II.1, the processor to schedule and execute the threads of the bound process, and the memory to store the data.

```
1   -- System
2
3   system Producer_Consumer
4   end Producer_Consumer;
5
6   system implementation Producer_Consumer.Impl
7   subcomponents
8     the_process : process pc.Impl;
9     the_processor : processor rm_processor.Impl;
10    the_memory : memory ram_mem;
11
12  properties
13    Actual_Processor_Binding => (reference (the_processor))
          applies to the_process;
14    Actual_Processor_Binding => (reference (the_memory)) applies
          to the_process;
15
16  end Producer_Consumer.Impl;
```

Listing II.2: Producer/consumer system in AADL

Component interactions. Components use their features to interact in many different ways:

- *Connections*: the most common communication way uses `ports`, connecting an `out` port of a component to an `in` port of another. AADL defines three types of ports to transfer data, events (control), or both: *data ports*, *event ports* and *event data ports*. For example, the `pc` process in Listing II.1 connects the `Prod` and `Cons` threads (line 54) through their ports. `access` to data, buses or subprograms, or `parameters` passed into and out of a subprogram are other examples of connections between components. Connections represent logical flows (e.g. control or data flow) between components through their features,

- *Calls* to subprograms in a threads or another subprogram, as done in the `Producer` thread (line 31 in Listing II.1),
- *Bindings* map application software components to execution platform components. For example, a process is bound to a processor to specify that this specific process must be executed by this specific processor (line 13 in Listing II.2).

Annex and property sets. In addition to the core language, AADL proposes several user-defined extension mechanisms through property sets and annex sublanguages [68]:

- Property sets allow one to define custom properties to extend standard ones. This is the path taken by the "Data modeling annex document" that allows one to model precisely data types to be manipulated, or the "ARINC653 annex document" that defines patterns for modeling ARINC653 systems,
- AADL annex sublanguages offer the possibility to attach additional considerations to an AADL component like behavioral specification. They bind a domain-specific language to components.

These extensions mechanisms are of particular interest to address project-specific concerns such as modeling electric power consumption, modeling precise performances of buses, or error modeling. The combination of core and user-defined extensions make it possible to customize architecture models and support specialized analysis.

Analysis and code generation. AADL initial requirement document mentions analysis as a key objective. AADL models can be analyzed with a large set of analysis theories and tools[3]: real-time analysis with scheduling theory (e.g. Cheddar [8], MAST [9] or MoSaRT [69, 70] tools), real-time process algebra [71], real-time calculus [72] or network calculus [73]; behavioral analysis through mappings to formal methods and associated model-checkers based on Petri Nets [74] or other formalism like FIACRE/TINA [75, 76], RT-Maude [77], UPPAAL [10, 78], BIP [79, 80], CADP [81, 82], etc.; dependability assessment from the Error Model Annex, like the COMPASS project [83] or ADAPT [84, 85]; security verification [86, 87]; etc.

In addition, AADL allows for code generation. For example, Ocarina [88] implements Ada and C code generators for a wide variety of regular real-time platforms (RT-POSIX, FreeRTOS, Vxworks, RTEMS, Xenomai) and avionic platforms (ARINC653); or model transformations to synchronous programs in SIGNAL [89] or LUSTRE [90], or to the hardware description language SystemC [91].

Related languages. We can mention UML-based languages SysML [6] and MARTE [7] or EAST-ADL [4] among the languages providing concepts and abstractions similar to AADL, as stated in [92, 93, 94].

[3] An updated list of supporting tools, projects and papers is available at `http://www.aadl.info`.

MARTE (Modeling and Analysis of Real-Time and Embedded Systems) is a UML profile dedicated to the modeling and analysis of real-time and embedded systems. It relies on domain-specific extensions of the general UML to bring concepts to model real-time and embedded applications. These extensions focus on non-functional elements of real-time applications. These elements may be defined to support modeling, analysis, or both. For instance, Optimum [95] clarifies usage of MARTE concepts for schedulability analysis, or [96, 97] use MARTE for dependability assessment.

EAST-ADL (Electronics Architecture and Software Technology - Architecture Description Language) [4] is an Architecture Description Language for automotive embedded systems, developed in several European research projects. It is based on concepts from UML, SysML and AADL, but adapted for automotive needs and compliance with AUTOSAR [5]. EAST-ADL has been designed to complement AUTOSAR with descriptions at higher levels of abstractions: vehicle features, functions, requirements, variability, software components, hardware components and communications. EAST-ADL models can be analyzed. For instance, Chen et al. [98, 99] discuss analysis of EAST-ADL models, focusing on model checking using SPIN, safety analysis using Hip-Hops and some timing analyses.

We reviewed and compared these languages in more detail in a paper [100].

II.2.4.B CPAL: the Cyber-Physical Action Language

CPAL (Cyber-Physical Action Language) is a language to model, simulate, verify and program Cyber-Physical Systems (CPS) [19, 101]. The language in itself is inspired by the synchronous programming approach [102, 103] and time-triggered languages such as Giotto [104]. The syntax of CPAL is close to the syntax of the C language but provides concepts specific to embedded systems with a formal execution semantics. In addition, CPAL is a real-time execution engine. CPAL models are *interpreted* with the guarantee that a model will have the same behavior in simulation mode on a workstation and in real-time mode on any embedded board. CPAL is jointly developed at the University of Luxembourg and by the company RTaW since 2011.

Functional architecture in CPAL. CPAL enables to represent the functional architecture of the system. The functional architecture consists of the set of functions, the activation scheme and the data flow between the functions. In addition, a CPAL model describes the functional behavior of the functions, that is the code of the function itself.

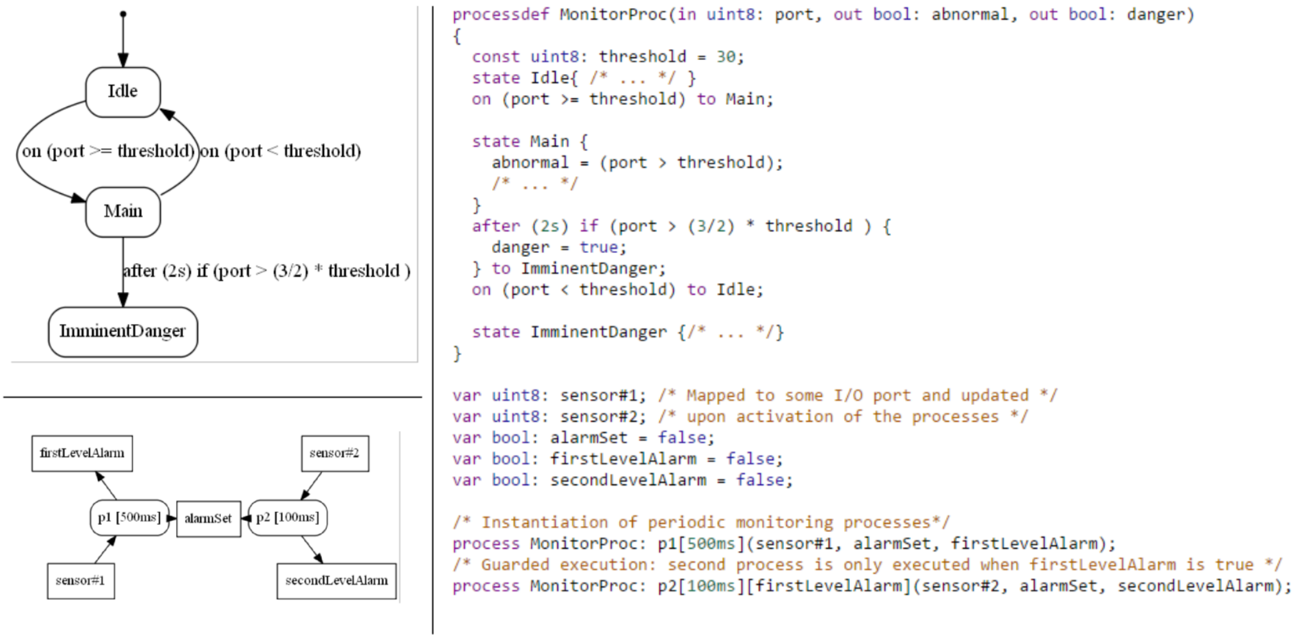
Processes are the core entities of a CPAL model. Processes have their own dynamics: they are activated at a specified rate or when a specific condition is fulfilled. CPAL processes are equivalent to the concepts of tasks, runnables or threads in other domains-specific modeling languages. A process is firstly defined with a list of parameters completed with the code of the function itself. One of several instances of the process can then be created in the CPAL program.

Finite-State Machines describe the logic of a process based on the semantics of Mode-Automata [105]. Each process embeds a FSM. The simplest version of a process consists in a single state that is executed repeatedly. For instance, FSMs can be used to describe the different running mode of a system. CPAL implements the following semantics for FSM: execute a possible transition first and then execute the current state of the FSM.

Communication inter-processes is done via process arguments passed through `in` and `out` *ports*. The argument can be either a *global variable* or a *communication channel* that is a `stack` or a `queue`. The main difference is that a global variable is passed by *value* to a process, meaning that the processes will work on copied data, while a channel is a *reference* to the actual data. Communication channels are more efficient in terms of speed and memory compared to communication by global variables. In addition, communication channels provide more powerful data buffering mechanisms. Queues and stacks respectively implement FIFO and LIFO buffering.

Real-time is an integral part of CPAL with precise activation models and scheduling policies. Process activation are specified through specific process parameters, including periods and, possibly, offsets or specific activation conditions. Processes are then scheduled according to a scheduling algorithm. FIFO (First-In First-Out), Non-Preemptive Earliest Deadline First (NPEDF) and Non-Preemptive Fixed Priority (NPFP) are scheduling policies available in CPAL.

Figure II.12 illustrates the main constructs of CPAL through a monitoring process example. The CPAL program defines a monitoring process which signals an abnormal behavior and, possibly, raises an alarm after a while when a value measured from a sensor exceeds a threshold. The first level alarm is to be confirmed from another sensor by a second process executed at a higher rate. If it is the case, a second level alarm is set. The CPAL program describes all the functional, logical and real-time aspects.



```
processdef MonitorProc(in uint8: port, out bool: abnormal, out bool: danger)
{
  const uint8: threshold = 30;
  state Idle{ /* ... */ }
  on (port >= threshold) to Main;

  state Main {
    abnormal = (port > threshold);
    /* ... */
  }
  after (2s) if (port > (3/2) * threshold ) {
    danger = true;
  } to ImminentDanger;
  on (port < threshold) to Idle;

  state ImminentDanger {/* ... */}
}

var uint8: sensor#1; /* Mapped to some I/O port and updated */
var uint8: sensor#2; /* upon activation of the processes */
var bool: alarmSet = false;
var bool: firstLevelAlarm = false;
var bool: secondLevelAlarm = false;

/* Instantiation of periodic monitoring processes*/
process MonitorProc: p1[500ms](sensor#1, alarmSet, firstLevelAlarm);
/* Guarded execution: second process is only executed when firstLevelAlarm is true */
process MonitorProc: p2[100ms][firstLevelAlarm](sensor#2, alarmSet, secondLevelAlarm);
```

Figure II.12: A monitoring process in CPAL (from [19]).

Analysis and execution of CPAL programs. The second objective of CPAL right after modeling is to make it possible to evaluate and execute CPS. For this purpose, the CPAL core language is completed with analysis-specific language constructs called annotations, and an interpreter.

Annotations describe the non-functional properties of a system in great detail. Timing annotations for instance, defined in a dedicated `@cpal:time` block, specifies the timing behavior of the CPAL program. CPAL provides execution-time annotations (e.g. varying execution-time or WCET) and scheduling annotations (e.g. interarrival times, jitters, scheduling parameters such as process priorities, deadlines, etc.). For example Listing II.3, depicts a CPAL model that includes several execution time annotations within process states, e.g. `@cpal:time {State1.execution_time = 15ms;}`. According to these annotations, the execution time of a state is static (for instance at lines 6 or line 12) or dynamic (for example if it depends on a condition at line 24 or line 26). Execution time could be equally expressed at the process level, thus applying to all potential states of a process.

```
 1
 2  processdef Varying_Execution_Time()
 3  {
 4    state State1 {
 5      @cpal:time {
 6        State1.execution_time = 15ms;
 7      }
 8    }
 9    on (true) to State2;
10
11    state State2 {
12      @cpal:time {
13        State2.execution_time = 35ms;
14      }
15    }
16    on (true) to State1;
17  }
18
19  processdef Conditional_Execution_Time()
20  {
21    state Main {
22      @cpal:time {
23        if (uint16.rand_uniform(0,2)==0) {
24          Main.execution_time = 1ms;
25        } else {
26          Main.execution_time = 15ms;
27        }
28      }
29    }
30  }
31
32  process Constant_Execution_Time: p1[70ms]();
33  process Conditional_Execution_Time: p2[200ms]();
```

Listing II.3: CPAL program with timing annotations.

The *interpreter* enables to execute CPAL models. The interpreter runs either in *simulation* mode or in *real-time* mode. An execution in simulation mode is as fast as possible, meaning that the interpreter makes optimistic assumptions and the program is not granted access to the hardware. For instance, the code executes in zero-time except if timing annotations are provided in the code. Real-time mode enables to actually execute the model on a platform, with access to the hardware.

The interpreter does not consider optimistic assumptions but real execution, e.g. code execution-time depends on the frequency of the processor, use of I/O devices, etc. Table II.1 summarizes the platforms currently supported by the interpreter.

Platform	Supported execution mode	Access to HW?
Windows 32/64bit	Simulation	✗
Embedded Windows 32/64bit	Real-time and Simulation	✗
Linux 64bit	Simulation	✗
Embedded Linux 64bit	Real-time and Simulation	✓
Mac OS X	Simulation	✗
Freescale FRDM-K64F	Real-Time	✓
Raspberry Pi	Real-time and Simulation	✓

Table II.1: Platforms supported by the CPAL interpreter.

CPAL provides different types of analyses based on the annotations and/or the interpreter:

- simulation of the timing behavior of the system in a dedicated mode,
- mechanisms to measure the WCETs in real-time mode on a specific target,
- schedulability analysis using timing annotations, e.g. see [106].

For example, Figure II.13 represents a simulation of the CPAL program in Listing II.3, as displayed in the CPAL-Editor. The vertical bars represent the processes activation based on the periods, whereas the widths of the bars depict the execution times according to the execution time annotations. The processes are scheduled according to a FIFO policy, while the execution time depends on the state of the process defined in the FSM.

Related languages. Synchronous dataflow models provide an intermediate level of abstraction between a low-level program and a high-level architecture model such as AADL. More importantly, models with synchronous languages like LUSTRE [107], SIGNAL [108], Esterel [109] or Prelude [110, 111] have a formal execution semantics (i.e. the synchronous semantics). The synchronous approach is based on strong mathematical foundations and naturally suits formal design and verification of reactive systems, e.g. see [112, 113].

Giotto is a time-triggered architecture language [104]. A Giotto model depicts the software architecture of a system with both the functional and timing aspects. At its core, Giotto provides a formal execution semantics (i.e. the Giotto semantics).

Navet et al. [19] outline some links between synchronous architecture languages and CPAL. They explain that CPAL is a lighter and easier to learn programming language compared to synchronous programming languages, while being equally able to guarantee the necessary timing predictability of the application. The authors

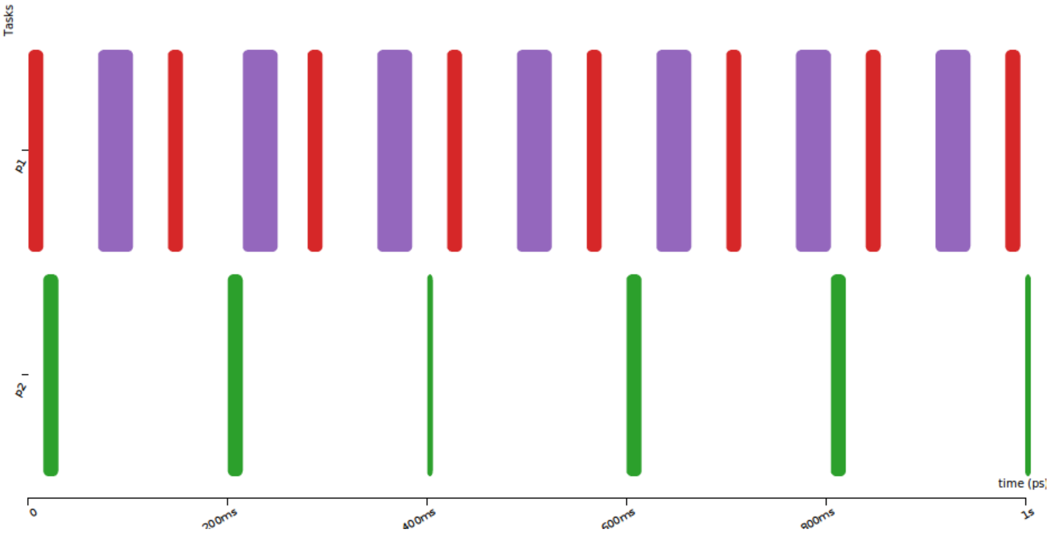


Figure II.13: Gantt diagram representing the execution of the processes defined in Listing II.3.

highlight bridges with the higher-level languages Prelude and Giotto. Yet, they note that those languages are neither programming languages to define the functional behavior of the tasks, nor an execution platform.

Works like [90, 114, 115] show overlaps between high-level ADLs and synchronous ADLs. For instance, the authors in [90] translate a subpart of an AADL model to a LUSTRE program; and evaluate AADL models with tools available for synchronous programs. Henzinger et al. [104] noted some bridges between Giotto and MetaH, the ancestor of AADL. In particular, Giotto captures some aspects of MetaH (e.g. real-time tasks and communications) in an abstract and formal way. In the context of this thesis, we present in Chapter VII a case study that combines CPAL to AADL to fully model an avionic system.

II.3 Model-based analysis

Analysis and verification is an important aspect of the design of embedded systems. These activities aim to check that the system will meet the non-functional properties at run-time. In that context, models are valuable assets to investigate a system design, answering questions in place of the real system.

II.3.1 Main analysis approaches

We can cite three main analysis approaches that are fully or partly based on abstract models:

> **Simulation** consists in a virtual execution of a given system according to a model of the system and a simulation environment [90]. Simulation approaches are able to deal with large systems. Yet, a simulation is generally unable to enumerate all potential system's states and execute all possible

Analysis Approach	Supported activity	Analysis Support	Scope of results
Simulation	Design	Simulation model	Non-exhaustive
Model-checking	Design	Formal model	Exhaustive
Analytical methods	Design	Analytical model	Deterministic
Tests	Verification	Test model + System	Non-exhaustive

Table II.2: Some special features of usual model-based analysis approaches.

scenarios. Therefore, a lengthy simulation time (the amount of time provided to the simulator to explore system's states) may be necessary to compute precise simulation results, but does not guarantee that these results are complete.

Model-checking is a formal approach to automatically verify finite-state software or hardware systems [116]. Model-checking considers a formal model (e.g. Petri nets [117], timed automata [118], etc.) and properties to verify, expressed in a logical formula. An algorithm explores all possible states of the model and determines whether given properties hold or not. A major impediment of model-checking is known as the state space explosion problem that results in impracticable models.

Analytical methods are *ad hoc* mathematical-based approaches which do not belong to the aforementioned analysis approaches. These approaches consider an analytical model that is to be analyzed through an algorithm to answer a given question about the system. For example, feasibility tests determine whether real-time tasks will met their deadlines according to a given scheduling algorithm [21]. Feasibility tests are based on a task model and consist of equations to verify and algorithms that implement them. Another example is Network Calculus, a mathematical approach that reasons in terms of flows and servers to compute worst case performances of networks [119]. Network Calculus tools implement algorithms based on the min-plus algebra [120] to analyze such models, e.g. [121].

Tests operate on the product program. More precisely, "testing is the process of executing a program with the intent of finding errors" [122]. Testing consists in executing the system according to test cases in order to verify that the system conforms to its specification. Testing cannot guarantee the absence of all errors. A major issue in testing is hence to maximize the detection of errors through efficient testing methods and effective test cases, e.g. Model-Based Testing [123].

Table II.2 summarizes some key features of these analysis approaches. In particular, the approaches differ with respect to the supported activities (e.g. design vs. verification), the analysis support (product system vs. analysis model) and the scope of the result (e.g. completeness, determinism).

In this thesis, we concentrate on analytical methods. We are especially interested in the analysis of real-time properties. We study two analysis approaches for this purpose: scheduling analysis and network calculus.

II.3.2 Case study: real-time task scheduling analysis

A real-time system is made up of a set of tasks which much be executed on one or more processors and possibly share some resources. The tasks must be executed such that the temporal constraints required by the environment are met. The scheduler is the component in charge of building up an execution order (i.e. a schedule) that fulfills the temporal constraints with available resources. We firstly review the basic concepts of real-time task and scheduling. We then introduce some analytical approaches to analyze real-time scheduling.

II.3.2.A Real-time task model

Real-time tasks are the basic entities of a real-time system. A task is a logical unit of computation in a processor [124], that is a set of program instructions that are to be executed by a processor. Tasks may be also referred to as processes or threads in other contexts. A task job is a specific instance of task execution.

A task τ_i can be characterized by temporal parameters. Table II.3 summarizes some common task parameters.

Parameter	Notation	Note
computation time (or capacity)	C_i	
relative deadline	D_i	
period or minimum inter-release time	T_i	
offset	O_i	
jitter	J_i	
priority (if applicable)	P_i	
release time	$r_{i,j}$	periodic task: $r_{i,j} = O_i + (j-1) \cdot T_i$
start time	$s_{i,j}$	$s_{i,j} \geq r_{i,j}$
finish time	$f_{i,j}$	
absolute deadline	$d_{i,j}$	periodic task: $d_{i,j} = r_{i,j} + D_i$
response time	$R_{i,j}$	$R_{i,j} = f_{i,j} - r_{i,j}$, a valid schedules requires that $\forall \tau_i \in \mathcal{T}$, $max_{\forall j}(R_{i,j}) \leq D_i$

Table II.3: Usual real-time task parameters.

According to the occurrence of jobs, we usually distinguish between *periodic*, *aperiodic* and *sporadic* tasks. Jobs in a periodic task are released in a regular basis and are separated by a constant interval of time called the period. Sporadic tasks occur irregularly but can be characterized by a minimum inter-release time between consecutive jobs. Aperiodic tasks occur at unknown times. For example, Figure II.14 represents a periodic task execution with a Gantt diagram.

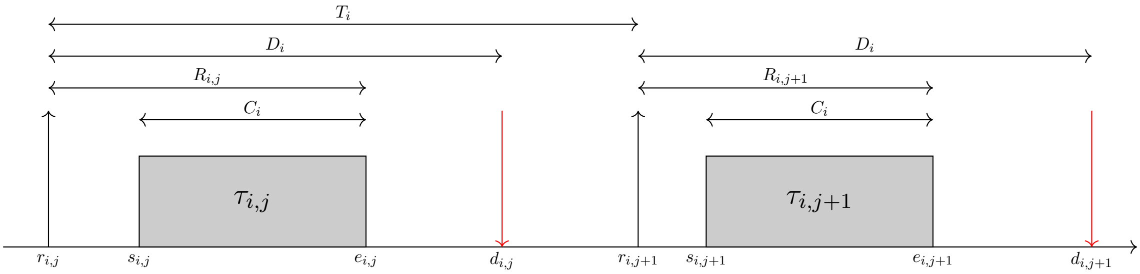


Figure II.14: Representation of a real-time periodic task with a Gantt diagram. *For a task τ_i: T_i the period, C_i the computation time and D_i the relative deadline. $\tau_{i,j}$ denotes the j^{th} job of a task i: $r_{i,j}$ is the release time, $s_{i,j}$ the start time, $e_{i,j}$ the completion time, $d_{i,j}$ the absolute deadline. A system is schedulable if $\forall \tau_i \in \mathcal{T}$, $\forall R_{i,j}$ the response time respects $R_{i,j} \leq d_{i,j}$.*

II.3.2.B Scheduling

The objective of real-time scheduling is to define an execution order of the tasks that fulfill the timing constraints with available resources. A scheduling takes account of a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, a set of processor $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ to execute the tasks and, possibly, a set of shared resources $\mathcal{R} = \{R_1, R_2, \ldots, R_s\}$.

Scheduling algorithm. Numerous scheduling algorithms have been proposed in the literature. A scheduling algorithm provides a schedule of tasks, that is, at any time, assigns ready task jobs to available processors and, if necessary, shared resources. A real-time scheduling algorithm aims to provide a schedule that meets all the timing constraints.

We do not provide a complete taxonomy of scheduling algorithms (see for example [125]). Yet, we can distinguish the scheduling algorithms mentioned in this thesis between:

- *monoprocessor* scheduling (or uniprocessor scheduling) if the system has only one processor versus *multiprocessor* scheduling otherwise,
- *off-line* scheduling (or static scheduling) where the schedule is specified prior to run-time in opposition to *on-line* scheduling (or dynamic scheduling) where the schedule is calculated during the execution of the system,
- *preemptive* scheduling if the algorithm is able to suspend a task execution, to execute a higher priority task, and then resume the execution of the first task; and *non-preemptive* scheduling whether a task cannot be interrupted until its execution is completed,
- *priority-driven* algorithms that assign a *fixed* or *dynamic* priority to tasks (i.e. Fixed Task Priority), respectively jobs (i.e. Fixed Job Priority, Dynamic Priority), and schedules at any time the task, resp. job, with the highest-priority,

- *independent tasks* scheduling that considers task sets with no precedence relationships and no shared resources; and *dependent tasks* scheduling that must take account of precedence constraints, critical shared resources, or both.

Rate Monotonic (RM), Deadline Monotonic (DM) and Earliest-Deadline First (EDF) are among the most popular real-time scheduling algorithms. Table II.4 summarizes some features of the algorithms mentioned in this thesis with respect to the classification discussed earlier.

Figure II.15 represents a schedule produced by the Deadline Monotonic algorithm. The Deadline Monotonic algorithm assigns a fixed priority to each task τ_i according to its relative deadline D_i. The task with the lowest relative deadline is assigned the highest priority. $D_1 \geq D_2 \geq D_3$ so $P_1 \geq P_2 \geq P_3$. Thereby, the scheduler plans, at each time, the task with the highest priority. The scheduling algorithm is able to preempt a task to allocate the processor to a task which has an higher priority. For example, τ_3 is preempted at time 5 to execute the highest priority task τ_1, and then resumes at the completion of τ_1 at time 6.

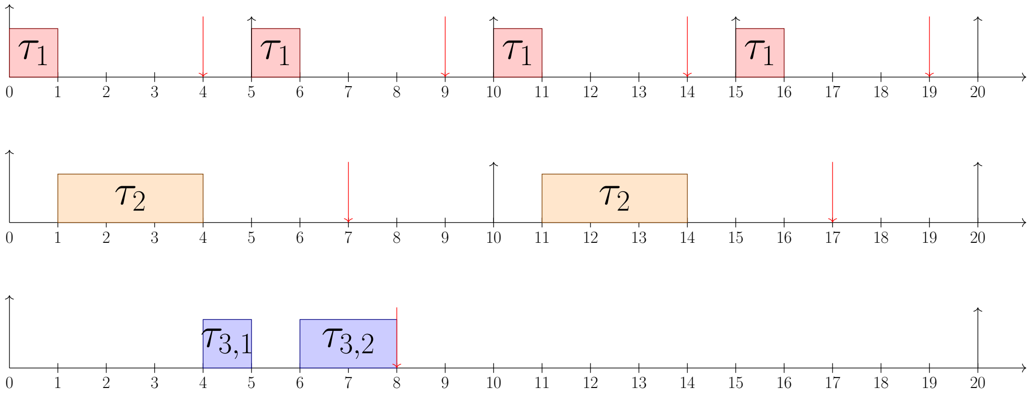


Figure II.15: An example of schedule produced by the Deadline Monotonic algorithm. τ_1: $C_1 = 1$, $T_1 = 5$, $D1 = 4$; τ_2 : $C_2 = 3$, $T_2 = 10$, $D_2 = 7$; τ_3 : $C_3 = 3$, $T_3 = 20$, $D_3 = 8$

II.3.2.C Scheduling analysis

Scheduling analysis aims to determine whether the scheduling algorithm will produce a schedule that will meet the timing constraints at run-time.

Schedulability and feasibility. According to Davis and Burns [126]:

- a task set is *schedulable* according to a given scheduling algorithm if the schedule produced by this algorithm satisfies all the deadlines.
- a task set is *feasible* if it exists any scheduling algorithm that makes it schedulable.

Scheduling algorithm	Hardware architecture	Preemption	Scheduling policy	Type of execution	Dependency of tasks
First In, First Out (FIFO)	uniprocessor	non-preemptive	executes jobs in the same order of job arrival	on-line	ignored
Fixed Priority (FP)	uniprocessor	preemptive	priority-driven, Fixed Task Priority	on-line	ignored
Rate Monotonic (RM)	uniprocessor	preemptive	priority-driven, Fixed Task Priority according to periods	on-line	ignored
Deadline Monotonic (DM)	uniprocessor	preemptive	priority-driven, Fixed Task Priority according to relative deadlines	on-line	ignored
Earliest-Deadline First (EDF)	uniprocessor	preemptive	priority-driven, Fixed Job Priority according to absolute deadlines	on-line	ignored
Non-Preemptive Fixed Priority (NPFP)	uniprocessor	non-preemptive	priority-driven, Fixed Task Priority	on-line	ignored
Non-Preemptive Earliest-Deadline First (NPEDF)	uniprocessor	non-preemptive	priority-driven, Fixed Job Priority according to absolute deadlines	on-line	ignored

Table II.4: Characteristics of some scheduling algorithms used in this thesis.

Schedulability analyses , which can also be referred to as feasibility analyses or feasibility/schedulability tests, are analytical methods based on the real-time scheduling theory to state if a task set is schedulable according to a given scheduling algorithm [21]. We usually distinguish between:

- *exact* tests that provide a *sufficient and necessary condition* with respect to the scheduling of a set of tasks, hence allowing to state with certainty whether the task set is schedulable or not; and
- *approximate* tests only provide a *sufficient* condition, telling only if the task set is schedulable as soon as the test succeeds (and providing no conclusion when the test fails).

There exist plenty of schedulability analyses. These analytical techniques evaluate different performance metrics. For example:

- *utilization-based tests* evaluate the *processor utilization factor* to determine the feasibility of a task set. That is, such tests checks that the fraction of processor time used to execute the task set does not exceed the theoretical bound admissible by a given scheduling algorithm, e.g. see [127, 128],
- *response-time analysis* calculates the *worst-case response time* of each task. A necessary and sufficient schedulability test is then to check that the worst-case response times are lower than the relative deadlines, e.g. see [129, 130],
- other analyses may consider the *processor demand* criterion [21], etc.

Liu and Layland [127] proposed for example an exact schedulability test for EDF based on the processor utilization. They firstly defined the processor utilization factor of a set of n periodic tasks by:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{II.1}$$

Liu and Layland then proved that a set of n periodic tasks with $P_i = D_i$ is schedulable according to the deadline driven scheduling algorithm EDF if and only if:

$$U \leq 1 \tag{II.2}$$

Many schedulability tests have been proposed so far, targeting the numerous task models and scheduling algorithms proposed in the literature, or improving many aspects of the tests (e.g. scope of the result, pessimism, computational complexity, etc.) [21]. We do not discuss the evolution of real-time scheduling analysis in greater depth. Sha et al. [21], Davis and Burns [126] and Stigge and Yi [131], for example, have provided good surveys on the matter. Yet, we will be required to review some evolution of task models and associated analyses in the context of Chapter III and Chapter IV. Furthermore, we use various schedulability analyses throughout this manuscript to illustrate and put into practice the concepts presented in this thesis.

II.4 Discussion

In this thesis, we emphasize on models to develop embedded systems. We firstly review two approaches that consider model as first-class artifacts: model-based engineering and model-driven engineering. Secondly, we discuss the link between models and analysis that founded the motivation of our works.

II.4.1 Model-Driven Engineering or Model-Based Engineering?

Models are valuable assets to design embedded systems. Yet, the use of models for system design has been explored in different directions: model-based software-systems engineering, model-driven engineering, model-driven architecture, etc. These different terminologies actually overlap.

Model-*based* is the wider denomination. It denotes such approaches that use models as the central artifact to support various activities in relation to *engineering systems*, e.g. design only, development that target the creation of the system, or engineering when considering the whole system lifecycle. According to the INCOSE MBSE initiative [132], "model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases". MBSE operates a shift from a document-based to a model-based approach to enhance systems engineering. Yet, MBSE is more a precept than (for the moment) a systemic approach (see the roadmap proposed by the INCOSE initiative [132]). MBSE promotes methodologies, processes, methods, tools and environment that use models for the engineering of complex systems. For instance, SysML is a language devoted to model-based systems engineering targeting specification, analysis, design, verification and validation of complex systems. For further examples see a review by Estefan [133].

Model-*driven* engineering is a slight different view, with stronger bases. The motto of MDE is "everything is a model". As stated in Definition 3, MDE is firstly a software development *approach* that partly or totally generates a *software system* from models. MDE is secondly an *architecture* to that end, based on the triad model, metamodel, model transformation. For instance, Model-Driven Architecture (MDA) is a particular implementation of MDE with a set of OMG standards like MOF (Meta Object Facility), UML (Unified Modeling Language), XMI (XML Metadata Interchange) or OCL (Object Constraint Language).

Let us illustrate the difference between MBSE and MDE through AADL. AADL supports an architecture-centric model-based engineering approach [18]. MBSE with AADL is supported by a tool platform called OSATE [134]. This tool platform includes a model editor, analysis tools and generators. MBSE through AADL must fully define a methodology, that is processes, methods, and tools, to develop a system. MDE with AADL emphasize less on the methodological aspects but must address the models "around" AADL, that is the definition of the AADL language through metamodels in Ecore [14], definition of the couplings between models and analyses, or definition of transformations between Platform Independent Models (e.g. from AADL models to analysis-specific models) or to code.

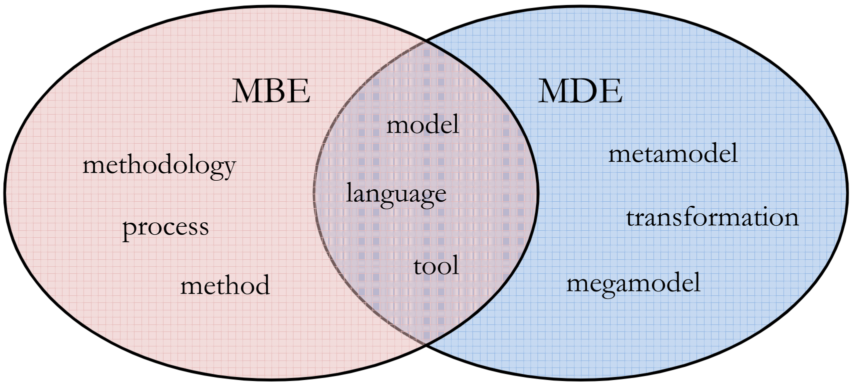


Figure II.16: Intersection between Model-Driven Engineering and Model-Based Engineering.

MBSE and MDE are still under exploration at the present time. If numerous core concepts have been established so far, with application in many tools, it is neither possible to give a complete map of MBSE and/or MDE yet, nor to define clearly the border between MBSE and MDE (there are many overlaps between the two visions). For example, *megamodeling* is an initiative to define a theory about MDE concepts through a dedicated model called a megamodel [135]. On the other hand, the definition of a MBSE theory is part of the roadmap defined by the INCOSE MBSE initiative [132].

The works presented in this thesis actually occur in the two contexts: MBE as we emphasize on models at large to develop embedded systems, and MDE as we reuse the fundamental concepts of model, metamodel (language) and model transformation.

II.4.2 Link between ADLs and analysis

Analysis tools are based on specific models that implement the analytical models. Therefore, numerous works seek to analyze architectural models by bridging the gap between architectural models and analysis-specific models. These works, referring more or less explicitly to the principles of MDE, typically implement a model transformation that translates an architectural model into a tool-specific model used for analysis.

For example with AADL, model transformations have been implemented towards terminal tools or intermediate frameworks: real-time specific languages Cheddar ADL and MAST models with the OCARINA tool suite [136, 137] or MoSaRT [70]; transformations exist to map AADL models to model-checker UPPAAL [78], TINA via FIACRE [75, 76] or CADP via LNT [82]; ADAPT to dependability analyses [85]; etc. A more exhaustive list of analysis tools and transformations applicable to AADL models is available in a survey [12].

We review the link between ADLs and analysis in greater detail in Chapter III.

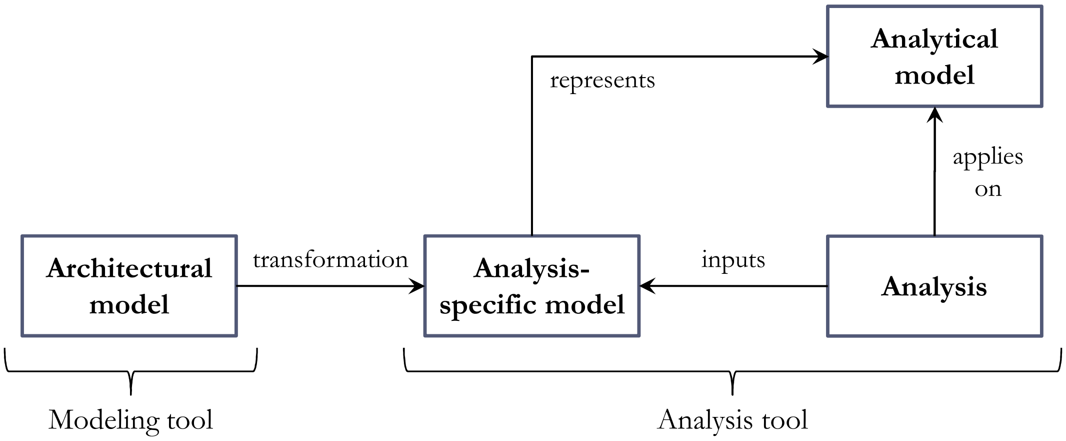


Figure II.17: Link between Architecture Description Languages and model-based analysis.

II.4.3 Design process: Design vs. Modeling vs. Analysis

Design, modeling and analysis are concepts closely intertwined. As discussed previously, modeling is the activity that consists in representing a system. As stated by France and Rumpe [2], "models are created to serve particular purposes, for example, to present a human understandable description of some aspect of a system or to present information in a form that can be mechanically analyzed". Analysis hence represents the other side of the coin. Analysis is "a careful study of something to learn about its parts, what they do, and how they are related to each other; an explanation of the nature and meaning of something"[4]. Analysis helps to understand a system through dissection of its model. Design is finally the process of creating the system from models and analyses. As represented in Figure II.18, the creation (design) of embedded systems is based on an iterative process of modeling and analysis.

If the use of models for the design of embedded systems is now better defined by the Model-Driven Development, the use of analysis is less clear. In practice, analysis remains considered as a side activity, if not ignored. Some solutions exist through model transformations as discussed earlier or with integrated frameworks, for example with well-known MATLAB/Simulink [138] or SCADE [113] in industry, the Ptolemy project in academia [139], or AADL-based frameworks OSATE [134], MASIW [140], ASIIST [141], etc. Yet, these solutions are incomplete. Integrated frameworks hardcode models and analyses in a same environment, with the key advantage of providing a solid integration of these artifacts. Nevertheless, they do not always provide the way to use them in the design process. Another shortcoming is that modeling and analysis capabilities are *de facto* restricted to a specific and closed environment. The modeling and analysis capabilities can be extended through model transformations, as discussed earlier. Yet, these model transformations, beyond the intrinsic problem of their implementation (treated in Chapter III), do not give attention to the semantics of the analysis (tackled in Chapter IV and Chapter V). The problem of defining exhaustively the design process goes far beyond the scope of this thesis.

[4] according to `http://www.merriam-webster.com/`

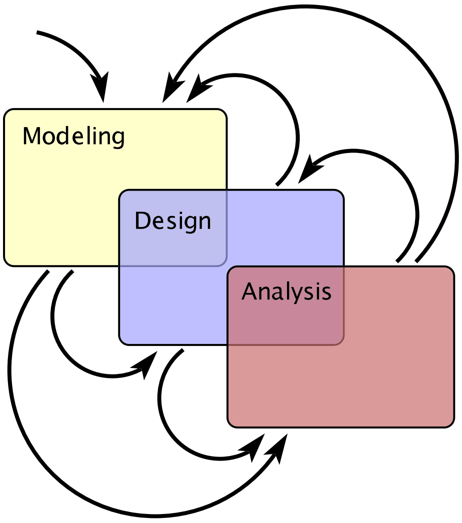


Figure II.18: Modeling and analysis is the design process (taken from Lee and Seshia [17]). *Designing an embedded systems involves several modeling and analysis iterations.*

II.5 Synthesis and conclusion

This chapter reviewed methods and tools to develop real-time embedded systems. We firstly underlined two special features of embedded systems: hardware/software architectures and non-functional constraints. We discussed in particular the crucial role of models to develop complex embedded systems with strong quality constraints. In essence, a model represents some aspect of a system and enables to analyze it.

We presented two methods based on models to cope with the constraints of embedded systems development: model-driven engineering and model-based analyses.

Model-Driven Engineering is a development approach that partly of totally generate a software system from models. We reviewed the core concepts of MDE: models, metamodels, and model transformations. We presented a particular kind of domain-specific language: Architecture Description Languages. An ADL captures the static and dynamic architecture of a system in initial design phases. This architecture model can then be used to automatically, semi-automatically or manually derive an actual system. We presented two ADL used in this thesis in more detail: the Architecture Analysis and Design Language (AADL), an SAE International standard, and the Cyber-Physical Action Language (CPAL), a new language inspired by the synchronous programming approach.

Model-based analyses are mathematical-based approaches applied on analytical models to check that the system will meet the non-functional properties at run-time. We mentioned simulation, model-checking or analytical methods as examples of model-based analyses. In this thesis, we concentrate on analytical methods, especially real-time task scheduling analyses that determine whether a task system meet some temporal constraints (e.g. deadlines) or not. We presented the important concepts of real-time scheduling (analysis) employed in this thesis.

In the last part of this chapter, we emphasized the link between design – MDE – and analysis – mathematical-based methods – through models. This founded the motivation of our work: by fully supporting the coupling between modeling and analysis, we may greatly enhance the design of high-quality embedded systems. The link between modeling and analysis has been explored in different ways by the research community, e.g. trough a model transformation from an architectural model to an analysis-specific model, or with "all-in-one" frameworks. Yet, these solutions are incomplete. Integrated frameworks narrow the scope of modeling and analysis to a specific and closed environment. Model transformations, beyond the intrinsic problem of correctly implementing them, do not give attention to the semantic aspects of the surrounding analysis. This "one-step" process results in practice in a cul-de-sac for the designer: *is the transformation correct? Is the analysis applicable? What is the meaning of the result? How to consider analysis results in the design process?* And so on.

In the next chapters, we study both the technical-transformation (in Chapter III) and semantic aspects (in Chapter IV and Chapter V) of the model/analysis integration problem. We implement the proposed concepts in a prototype of tool (presented in Chapter VI) and apply them so as to design various embedded systems (case studies in Chapter VII).

Chapter III

Model query

Abstract

This chapter deals with query mechanisms, called accessors, to analyze the non-functional properties of a system at design time. In Section III.1, we present the rationale behind model query. In particular, we review the analysis elements in detail – analysis algorithms and data structures – and show how these elements are linked with the notions of models and metamodels. In Section III.2, we present several data structures that can be used for the analysis of real-time systems. Section III.3 presents a first implementation of query mechanisms in Python. We finally end this chapter with a discussion about transformation-based analysis approaches (Section III.4) and a conclusion (Section III.5).

III.1 Rationale behind model query

In this section, we firstly identify the basic elements that exist in any analysis. In particular, we show how these elements are linked to the notions of models and metamodels. Then, we explain the notion of model query. Finally, we propose to implement model queries through a dedicated Application Programming Interface.

III.1.1 Identifying the analysis elements

Analysis algorithm and data structure. An analysis is nothing more than a particular program. An analysis is thus made of two parts: *data structures* to represent and organize the data, and *algorithms* to process them, and gain information from them. Paraphrasing Wirth [142], we could say:

"Data Structure + Algorithm = Analysis Program"

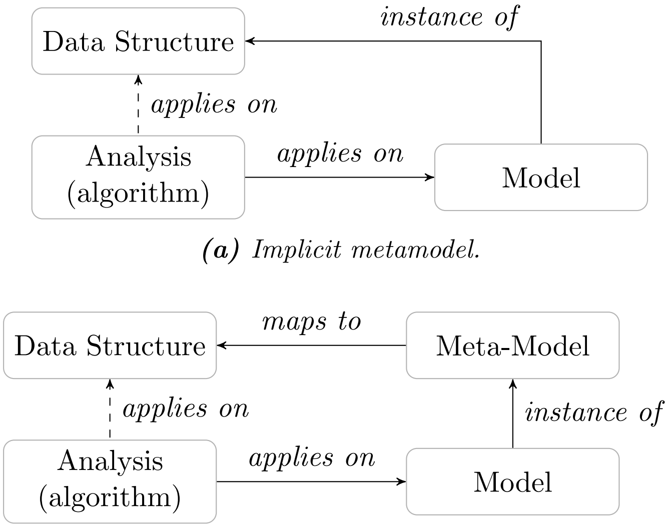
For instance, a real-time scheduling analysis consists of data structures to describe real-time workloads at different levels of abstraction, e.g. with a simple periodic task models or with more exhaustive graph-based models [131]; and algorithms to compute performance metrics from those data structures such as the processor utilization factor, task response times, etc.

Link with models and metamodels. We distinguish between the analysis space and the modeling space. In Figure III.1a, the data structures that are part of an analysis can be represented to the user in a model. Figure III.1b clearly shows the metamodel that defines the model. Thus, a relation must exist between the analysis data structure and the metamodel: there should be a mapping between the analysis data structures are the model concepts defined in the metamodel.

Let us finally note that what we previously referred to as "design-specific" or "analysis-specific" models only differ in the abstraction gap that separates analysis structures from model concepts. In fact, an "analysis-specific" model represents concepts for a particular analysis problem (for example, concepts of the real-time scheduling theory in MoSaRT and Cheddar ADL), whereas a "design-specific" model provides more general concepts to fully design a system (for example, general concepts of `system`, `process` and `bus` in AADL). In any case, analysis data are present in a model, appearing more or less explicitly to the analyst.

In conclusion, before applying an analysis on a model, one must:

1. at design time,
   (a) clearly define the data structures that are required by the analysis,
   (b) define the model concepts which maps those data structures,
2. at run time, query the model, i.e. request and analyze the data from the model, by taking into account (1a) and (1b).



(a) *Implicit metamodel.*

(b) *Explicit metamodel.*

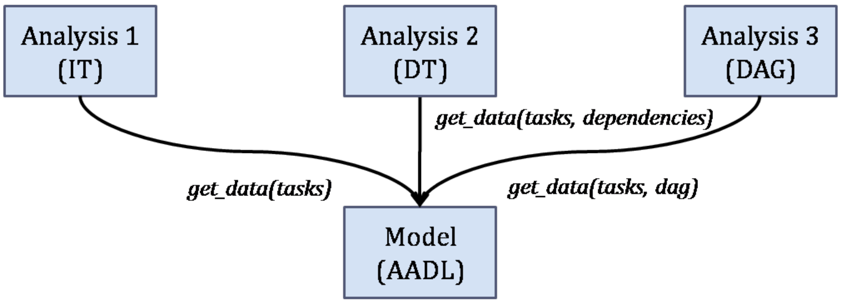
Figure III.1: Elements involved in an analysis and their relationships. *Analysis algorithms and data structures on the one hand, models and metamodels on the other hand are involved in the analysis process. The analysis of a model instance assumes a* mapping *to the analysis data structure, possibly via a metamodel.*

III.1.2 Model query

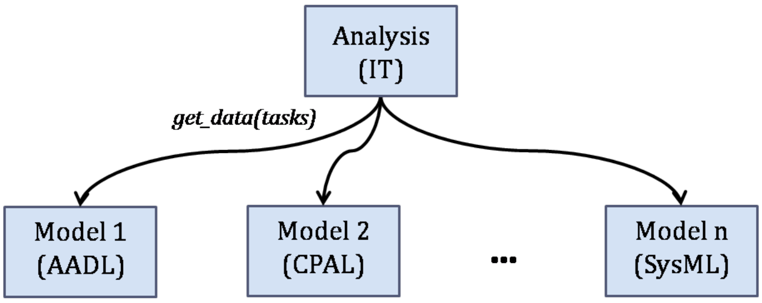
In the same way as SQL queries enable to retrieve information from databases [143], or Xquery for querying data from XML documents [144], *accessors* make it possible to extract data from domain-specific models.

Definition 11 (Accessor)**.** *An accessor is a function that gives back a typed data from a model according to the type of data structure passed as an argument, i.e.* $data = acc(data\_structure\_type)$.

Figure III.2 depicts two use cases. In Figure III.2a, an $Analysis_1$ that considers Independent Tasks (IT) retrieves a `task set` from an AADL model. Other analyses may extract different data structures from that model, e.g. a `dependency graph` to analyze Dependent Tasks ($Analysis_2$, DT) or a `directed acyclic graph` to assess tasks with non-deterministic behaviors ($Analysis_3$, DAG). In Figure III.2b an analysis can extract the same data structure from many models (e.g. AADL, CPAL, etc.). In conclusion, with accessors, many analyses can analyze many data structures from many models.



(a) *Use case: a model can be queried by many analyses. Three analyses query different type of data structures (e.g.* `tasks` *and their* `dependencies`*) on a AADL model via the* `get_data` *method.*



(b) *Use case: an analysis can query many models. An analysis query the same* `task` *data structure from many models, e.g. AADL, CPAL and SysML models.*

Figure III.2: Two use cases of data queries over design models.

In thesis, we implemented accessors in a specific Application Programming Interface (API). This API makes it possible to extract data from any architectural model, and manipulate them in an analysis program.

III.1.3 Implementation through an Application Programming interface

We propose to implement accessors through a dedicated Application Programming Interface (API). In Figure III.3, the Data Access API operates on top of various architectural models. This API is to be implemented in two parts:

1. definition of the data structures that can be used by the analyses,
2. implementation of the accessors to retrieve the data from the models. For this purpose, one must explore the model instances with the help of tool-specific APIs.

In Section III.3, we present an implementation of the Data Access API in Python. Accessors have been implemented towards AADL and CPAL models.

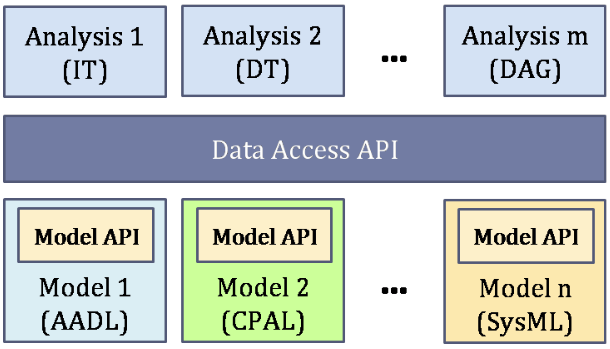


Figure III.3: Proposed Application Programming Interface. *The Data Access API defines and implements query methods to retrieve analysis data from various models, using the low-level APIs to manipulate the model instances in AADL, CPAL or SysML.*

Taking advantage of stakeholder expertise. Design and analysis activities are usually carried out by different *stakeholders*: (1) designers who define the models and (2) analysts who concentrate on the study of the model data. The stakeholders have their own *expertise*: definition and use of models on the one hand, definition of data structures and analytical reasoning on the other hand.

In Figure III.3, we break up the application in three components: analyses on the one hand, models on the other hand, Data Access API as the interface between them. This approach brings several advantages:

1. *separation of concerns, independence*: the components are independent (e.g. the API separates the analysis of data structures from the manipulation of these data in models), the stakeholders can concentrate on the subject that they understand the best,
2. *collaboration*: the collaboration between designers and analysts is eased by the definition of clear a API that consist of data structures and methods to access them in the models,

3. *reliability*: the stakeholders can negotiate and implement the interfaces based on their own expertise, i.e. definition of the data structures by the analysts, communication of the data structures to the designers, and implementation of the accessors to model internals by the model designers.

III.2 Data structures for the analysis of real-time systems

Task models are used to formally describe and analyze real-time workloads. In this section, we investigate some important data structures proposed by the real-time research community, which have been surveyed for example in [131], [21] or [126]. We only focus on preemptive uniprocessor systems. We present each data structure as follows. First, we shortly present the theoretical model. Secondly, we describe the data structure with a UML class diagram. Lastly, we provide an example of representation with a concrete syntax, in Python (programming language), AADL or CPAL (architecture description languages).

III.2.1 The basic periodic task model and its extensions

Real-time scheduling analysis grew up from the periodic task model. Since then, this model has been extended many times.

III.2.1.A The periodic tasks model

Theoretical model. The periodic task model has been introduced by Liu and Layland in 1973 [127]. It is based on the concept of task to realize a function, i.e. each function to be performed is associated with one or more tasks.

A task $\tau = (T, C)$ is characterized by a period T and a computation time C (or an upper bound of the computation time called $WCET$ for Worst-Case Execution Time). T and C are positive integers, i.e. $T \in \mathbb{N}$ and $C \in \mathbb{N}$.

In addition, the model specifies a processor to execute the tasks and a scheduling policy to decide the scheduling of the set of tasks on the processor, e.g. Rate Monotonic (RM).

Data structure. Figure III.4 depicts the definition of the data structure of the periodic tasks model with a class diagram. The elements of the theoretical model are represented with various classes: `Task`, `Processor` and `SchedulingPolicy` are the basic elements of the model. The attributes of the classes describe the elements properties, e.g. a `PeriodicTask` has a `name`, a `period` and a `worst_case_execution_time`. The relationships between the elements are also defined, e.g. a classes association denotes that the `SchedulingPolicy` is defined *for* a `TasksSet` *over* a `Processor`.

Concrete syntax. Listings III.1 and III.2 represent a task with two different concrete syntaxes. The first representation uses the Python programming language. The second representation uses the AADL language. The mapping between the

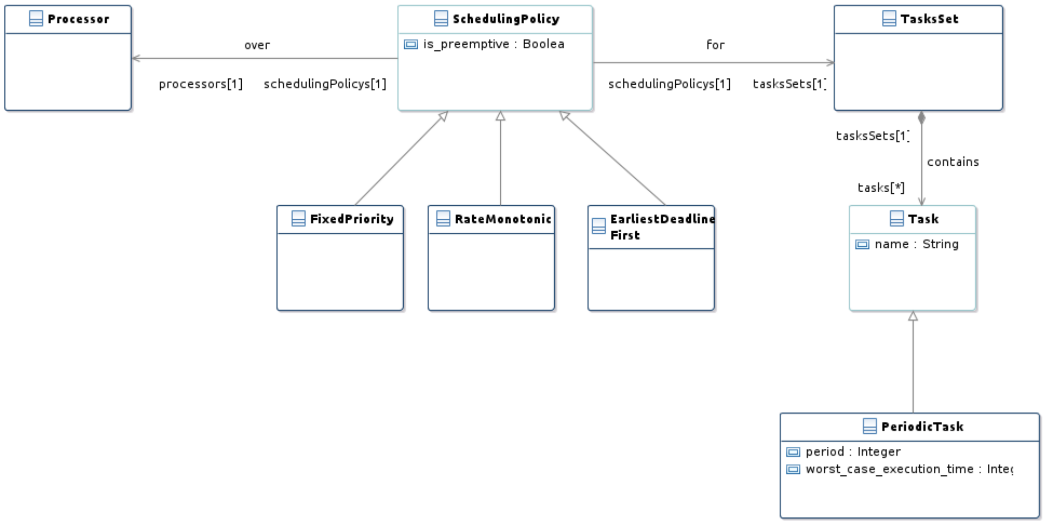


Figure III.4: Data structure of a periodic tasks model represented with a class diagram. *Task, Processor and SchedulingPolicy are the basic elements of the model*

elements of the data structure in Figure III.4 and the elements of the metamodels is given in Table III.1.

Data Structure	Concrete syntax	
	Python	AADL
class: Task	class: Task (Data_Struct)	Component Type: Thread
attribute: name	attribute: name	Component_Identifier
attribute: period	attribute: period	Thread_Properties: Period
attribute: best case execution time	attribute: best case execution time	Thread_Properties: Compute_Execution_Time
attribute: worst case execution time	attribute: worst case execution time	Thread_Properties: Compute_Execution_Time

Table III.1: Periodic model: mapping between the element of the data structure and the elements of the AADL and Python metamodels.

III.2.1.B Later developments

Theoretical model. The periodic task model has been later generalized with the *sporadic* task model and *multiframe* models. These models allow representing such task sets which can have non-regular release times, worst-case execution times and deadlines.

In the sporadic task model [145], task jobs are not released periodically but have to respect a Minimum Inter-release Time (MIT) T. In addition, the model considers an explicit deadline D ($D \in \mathbb{N}$) to the task definition: $\tau = (T, C, D)$.

The *Multiframe* model [146] and the *Generalized MultiFrame (GMF)* model [147] are able to express k jobs of different types, e.g. a task in the generalized multiframe model involves a triple $\tau = (\mathbf{T}, \mathbf{C}, \mathbf{D})$ with three vectors to describe k potentiality of frames:

- $\mathbf{T} = (T_0, T_1, \ldots, T_{k-1})$ are the minimum inter-release times,

```

" A simple class to represent a task "

class Task(Data_Struct):
   name='A_Task'
   " Timing values in milliseconds "
   period=20
   best_case_execution_time=0
   worst_case_execution_time=10
```

Listing (III.1) *Represented with Python concrete syntax.*

```
thread A_Task
   properties
   Dispatch_Protocol => Periodic;
   Period => 20 ms;
   Compute_Execution_Time => 0 ms .. 10 ms;
end A_Task;
```

Listing (III.2) *Represented with AADL concrete syntax.*

Figure III.5: Example of a task model.

- $\mathbf{C} = (C_0, C_1, \ldots, C_{k-1})$ are the worst-case execution times,
- $\mathbf{D} = (D_0, D_1, \ldots, D_{k-1})$ are the deadlines.

Data structure. The tasks in a model can be defined in different ways. In the class diagram depicted in Figure III.6, a `Task` can be a `PeriodicTask`, a `SporadicTask` or a `GeneralizedMultiFrame` task.

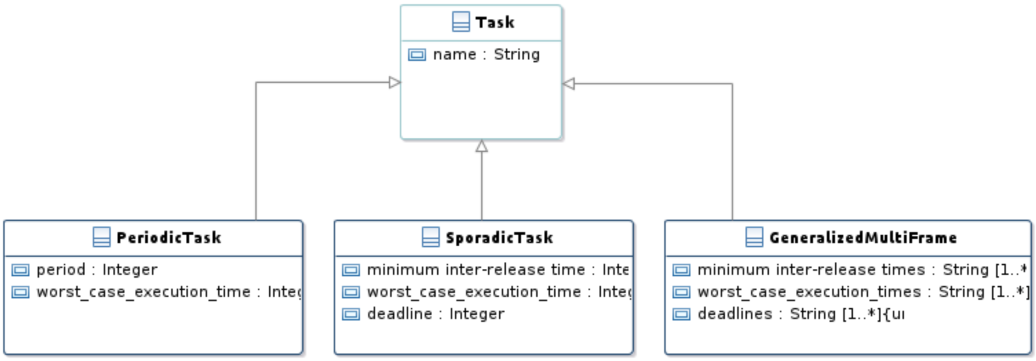

Task
name : String
PeriodicTask
period : Integer
worst_case_execution_time : Integ
SporadicTask
minimum inter-release time : Inte
worst_case_execution_time : Integ
deadline : Integer
GeneralizedMultiFrame
minimum inter-release times : String [1..*
worst_case_execution_times : String [1..*]
deadlines : String [1..*]{uı

Figure III.6: Data structure to represent several type of tasks with a class diagram. *Only the changing part is represented: a `Task` can be implemented with a `PeriodicTask`, a `SporadicTask` or a `GeneralizedMultiFrame` task.*

III.2.2 Graph-based tasks models

Graphs are among the more expressive data structures to characterize real-time workloads. We illustrate two cases of utilization: dependent tasks and tasks with non-deterministic behaviors.

III.2.2.A Dependency graph.

Theoretical model. The periodic task model and its generalizations discussed in the previous subsection assume independent tasks. In real systems, the tasks can be dependent in many situations, e.g. when sharing resources such as buffers, network buses or other hardware devices. We can use a graph G=(V,E) to represent the dependencies between the tasks:

- V are vertices, each vertex is a task of the model $V \subseteq \mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$,
- $E \in V \times V$ are edges, representing a dependency between tasks.

In that case, the resources must be accessed in a mutually exclusive manner. In order to cope with synchronization problems such as priority inversion in fixed-priority preemptive systems, concurrency control protocol have been introduced, e.g. Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP) [148].

Data structure. The data structure of the dependent task model is represented on with a class diagram Figure III.7. New elements are introduced:

- dependency graph: a `Dependency` can be associated with a `Task`,
- shared data: a `Dependency` can involve a `SharedData`; `SharedData` can be further defined, e.g. `Buffer`, `Bus`, etc.
- protocol: several concurrency control protocols (`ConcurrencyProtocol` class) can be used to manage the resources, e.g. `PriorityInheritance` or `Priority Ceiling` protocols.

Concrete syntax. Figure III.8 depicts four tasks with the CPAL graphical syntax. Three tasks `Task1`, `Task2` and `Task3`, represented with rounded rectangles with periods within brackets, use a shared data named `aSharedData`, represented with a simple rectangle. Arrows depicts access modes: read or write. The last `Task4` is independent.

III.2.2.B Directed acyclic graphs

Theoretical model Task models based on Directed Acyclic Graph (DAG) can be used to represent such tasks with non deterministic behaviors [131] – non deterministic inter-release times, worst-case execution times and deadlines. In a DAG structure $G = (V, E)$:

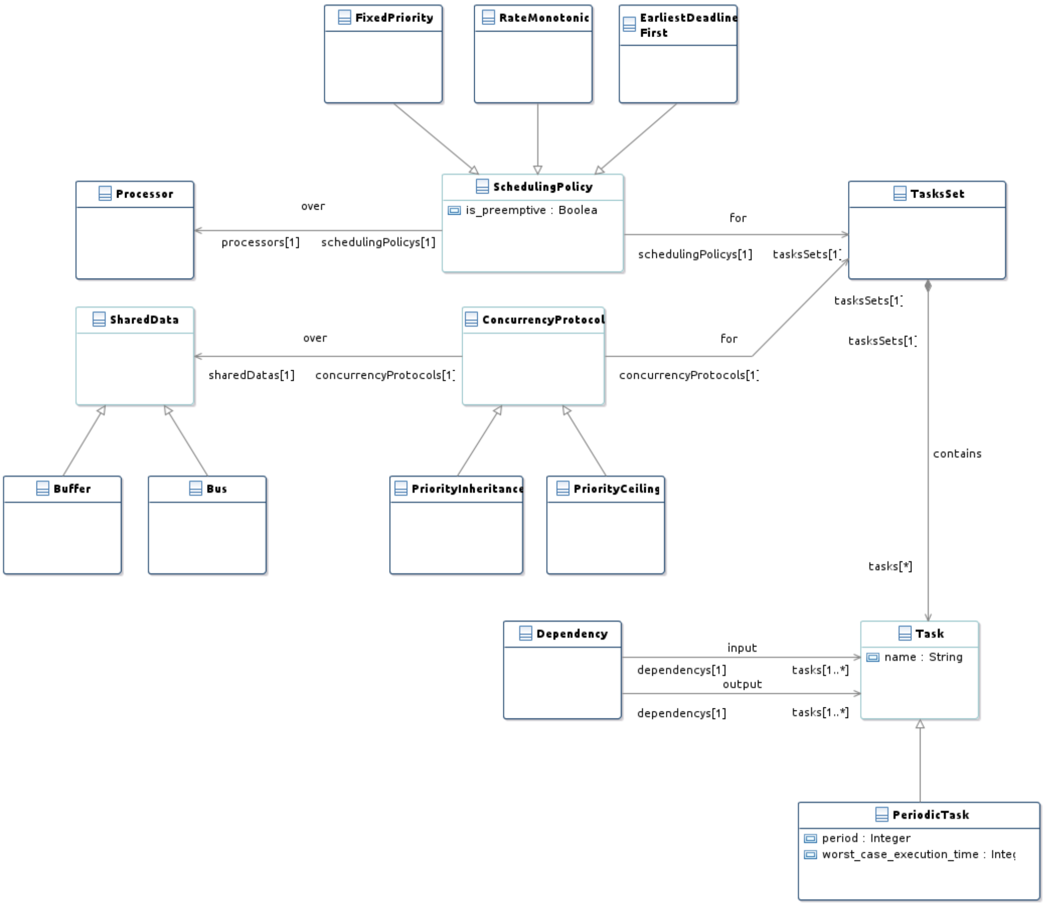


Figure III.7: Data structure to represent dependent tasks with a class diagram. *A graph can be used to denote the dependencies between the tasks, i.e. a* `Dependency` *can be associated with a* `Task`*. Access to* `SharedData` *(e.g.* `Buffer` *or* `Bus`*) involves a concurrency control protocol (*`ConcurrencyProtocol` *class).*

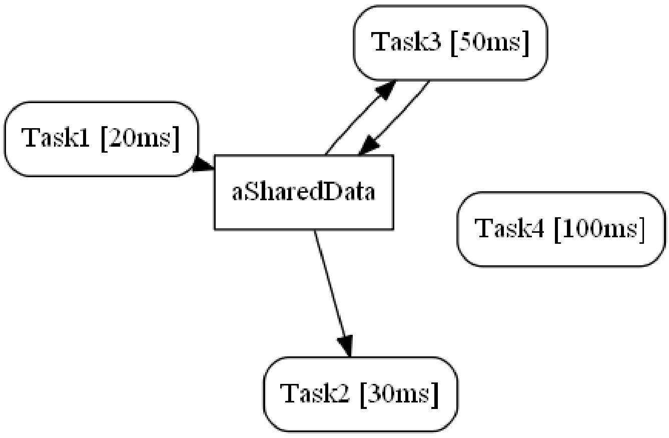


Figure III.8: Dependent tasks represented in the CPAL graphical syntax. *Four tasks are represented with rounded rectangles with periods within brackets. Three tasks use a shared data represented with a simple rectangle. Arrows depicts access modes: read or write. The fourth task is independent.*

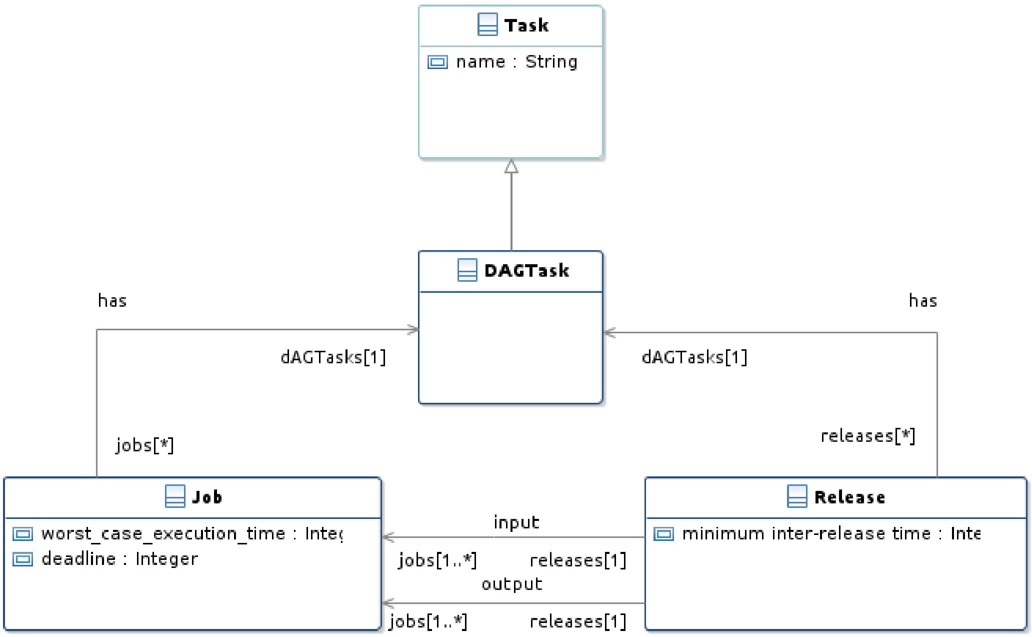


Figure III.9: Data structure to represent DAG tasks with a class diagram. *Only the changing part is represented: in a DAG task a graph is used to represent jobs and release times, here a `DAGTask` is defined via `Job` classes that can be associated with `Release` classes.*

- V are vertices, with each vertex $v \in V$ represents the release of a job,
- $E \in V \times V$ are edges, and each edge $(v, v') \in E$ represents the inter-release separation.

In addition, labels are assigned to the edges and vertices:

- a pair $< e(v), d(v) >$ is associated to a each vertex to denote job execution times and deadlines,
- a value $t(v, v')$ is associated to each edge (v, v') to denote the minimum inter-release times.

Data structure. Figure III.9 gives the definition of a DAG Task data structure with a class diagram. A DAG task consists of several jobs and release times, hence the class `DAGTask` has a composition relationship with `Job` and `Release` classes. In the DAG task model, a release has input and output jobs, hence the class `Release` has two associations `input` and `output` pointing to the class `Job`. The `Job` and `Release` are further described by the attributes `worst_case_execution_time` and `deadline`, and the attribute `minimum_inter-release_time` respectively.

III.3 Implementation of the Python API

Accessors enable to retrieve data structures from architectural models, and then to analyze them. In the previous section, we presented various real-time task models and associated data structures. In this section, we show an implementation of accessors in Python. Accessors have been implemented towards AADL and CPAL models.

Figure III.10 represents the application layers implemented in our prototype. The implementation of the prototype is based on the Python programming language. In addition, we may use dedicated resources for low-level model manipulations, e.g. we use the OCARINA [88] tool-suite to parse AADL models.

III.3.1 Data Structure, Data Model and Accessors

The data model is the centerpiece of the data access API. It contains the data that are to be used by the analysis at run time. The data model is based on data structures which are class-oriented implementations of the analysis data structures presented in Section III.2.

At run time, an analysis uses the data model to: (1) get the data to process and (2) store the result of this processing. Access to the data is implemented in two parts:

1. *data model*: different procedures to access the data structures and maintain such data structures up-to-date,
2. *low-level accessors* to retrieve data from the domain-specific models if necessary.

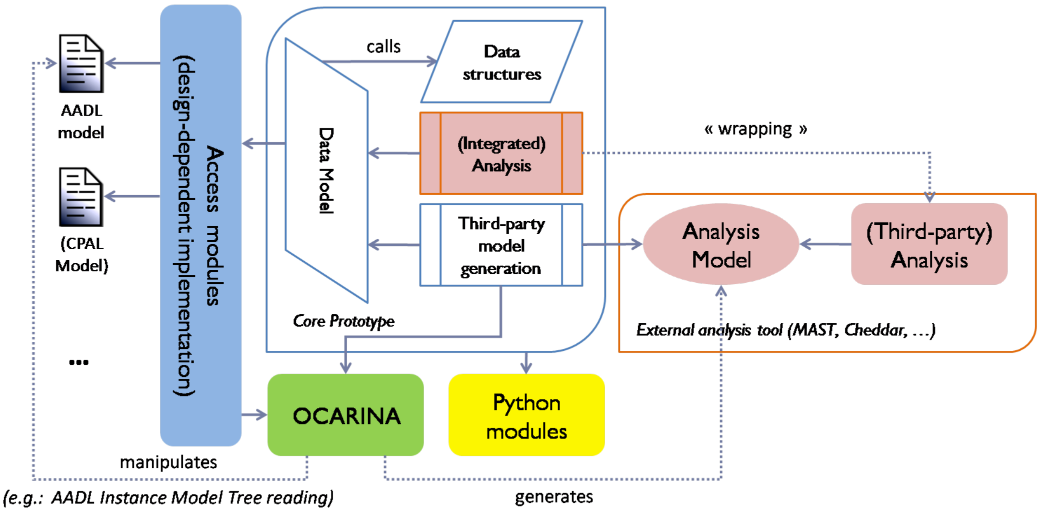


Figure III.10: Prototype modules overview. *The prototype is based on core modules to implement the analysis, an access module to retrieve data from the design models and several backend modules. The analysis can be externalized to a third-party tool via model generations.*

For instance, Figure III.11 describes the procedure to get a data structure from the data model. If the required data structure is not present in the data model, the data model must retrieve such data structure from the domain-specific model. For this purpose, it uses the sub-procedure `Get Data Structure from Design Model`.

We must implement the low-level accessors so as to extract data from the domain-specific models. Such accessors are specific to the target models. For example, we may need to explore the AADL Instance Tree (AIT) in order to retrieve data about real-time tasks from a AADL model. We do not discuss such operations in greater detail in this chapter. See Chapter VI for a thorough presentation of the prototype (Section VI.2.2 for accessors).

III.3.2 Analysis

The analysis module implements the analysis algorithm. The analysis algorithm consists of the basic sequence: (1) load the data model, (2) process the data model, and (3) update the data model with analysis results.

As an example, feasibility analysis of a set of independent tasks scheduled with Rate Monotonic can simply be implemented with the Python language as shown in Listing III.3.

The data model used for the analysis is loaded at the beginning: `self.list_of_tasks=data_model.getList` It consists of a `list_of_tasks` is made up of `Task` instances. The analysis computes the processor utilization factor (`_utilization_factor`) for the list of tasks before comparing this factor against the theoretical bound (`_utilization_factor<=_test_bound`).

A data structure representing the schedulability of the tasks set (`Schedulability` class) is created and initialized with the analysis result. The data model is finally up-

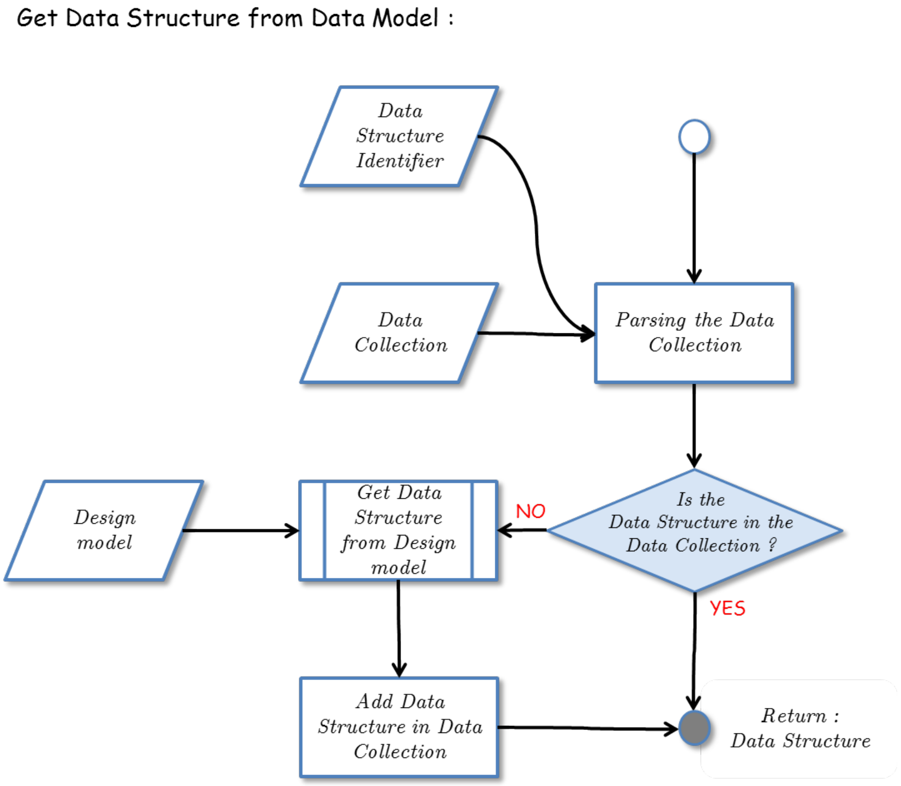

Get Data Structure from Data Model :
Data Structure Identifier
Data Collection
Parsing the Data Collection
Design model
Get Data Structure from Design model
NO
Is the Data Structure in the Data Collection ?
YES
Add Data Structure in Data Collection
Return : Data Structure

Figure III.11: Process Flowchart depicting the procedure to get a data structure from the data model. *If necessary, the data structure is accessed in the design model via the sub-process* `Get Data Structure from Design Model`*.*

```python
def analysis(self, data_model):
   print "--> Analysis execution : RMA"
   #input: required data entity
   self.list_of_tasks=data_model.getListOfTasks()
   #analysis implementation
   _utilization_factor=0.0
   for task in self.list_of_tasks:
      _utilization_factor=_utilization_factor+task.
          execution_time/task.period
   _tasks_nbr=float(len(self.list_of_tasks))
   _test_bound=_tasks_nbr*(2.0**(1.0/_tasks_nbr)-1.0)
   if _utilization_factor<=_test_bound:
      _Sched="OK"
   else:
      _Sched="NOK (NAP)"
   #output: provided data entity
   data_model.update(Data_Struct("SCHEDULABILITY_TEST", [
       Schedulability(_Sched, [])]))
```

Listing III.3: Feasibility analysis implemented with Python. *Example of implementation of a feasibility analysis.*

dated with the analysis result: `data_model.update(Data_Struct("SCHEDULABILITY_TEST", [Schedulability(_Sched, [])]))`.

III.4 Discussion

Query mechanisms enable to analyze the non-functional properties of a system from one of its models. This section first discusses related works based on model transformation, and then compares model transformations with model queries. We show that the two issues are actually orthogonal.

III.4.1 Related works

Model transformation. Modeling and analysis activities are usually based on distinct tools which use their own models. An approach commonly used to connect the toolsets is hence to translate a model used for design into a model used for analysis as represented in Figure III.12.

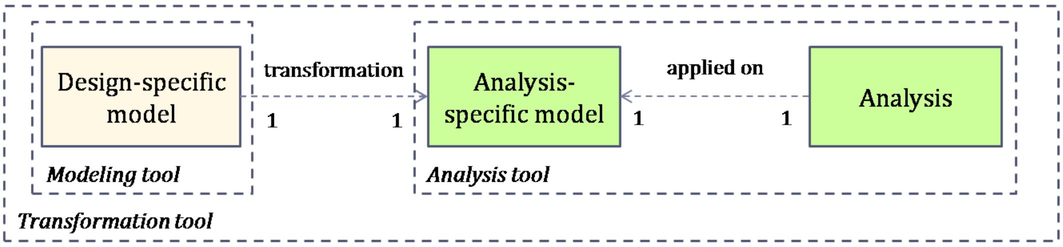


Figure III.12: Analysis based on a model transformation. *Design and analysis features are part of distinct tools: (1) a model used for design in a first tool is translated into a model used for analysis in a third-party tool; (2) the analysis in the third-party tool is then applied on its own model.*

Numerous transformations have been defined to connect analysis tools to AADL models. For example, transformations have been implemented to translate AADL models into Cheddar ADL and MAST models with the OCARINA tool suite [136, 88], transformations chains exist towards UPPAAL [78], TINA [75, 76] or CADP model-checkers [82],etc. A more exhaustive list of transformations applied to AADL models is available in a survey [12]. Yet, we note several limitations with a transformation-based analysis approach:

- *How to define the transformation?* One can either implement a comprehensive model transformation (e.g. metamodeling under the MOF standard [13] in the Eclipse Modeling Framework [14], transformation with a dedicated language such as ATL [15]), or more often relies on an *ad hoc* transformation chain to deal with the design and analysis models under different technical spaces (i.e. tools).

- *How many transformations are necessary?* A transformation is defined in terms of a couple of models, themselves being part of particular tools as represented in Figure III.12. Thus, it is necessary to define a multiplicity of models attached to specific tools/models.

- *Is the defined transformation correct?* An important challenge is to ensure the correctness of the transformation process. To the best of our knowledge, very few transformations applying on the analysis problem and which are discussed in the literature are proved to be correct, e.g. see [76] for a discussion on the subject. This is a huge problem as soon as an analysis result is the by-product of a transformation process which is itself not trustworthy. Verifying the correctness of models transformations is actually a problem in its own right which is the object of ongoing and dedicated researches, e.g. see works by Amrani [16].

In conclusion, in a transformation-based analysis approach, one must define a large number of *ad hoc* transformations with weak guarantees on their correctness.

Improving transformations through a pivot model. The previous strategy can be improved by using a pivot model. As depicted in Figure III.13, a pivot model is used to carry out several analyses. Pivot models can be connected with designs-specific models and/or analysis-specific models through model transformations.

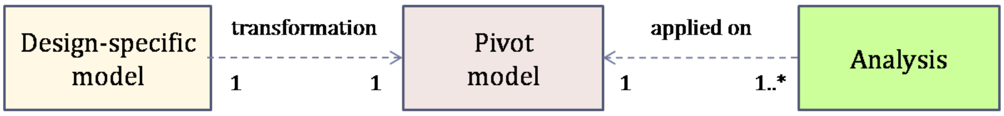


Figure III.13: Analysis of a design model via a pivot model. *The pivot model is used to apply several analyses. Model transformations can be used to map the pivot with design-specific models.*

For instance, MoSaRT and Cheddar have been used as intermediate frameworks between AADL models and temporal analysis tools [69, 149]; Fiacre is an intermediate formal language that is mapped to design languages (e.g. AADL, SDL, UML and SysML) in input and model checkers (e.g. Tina and CADP toolboxes) in output [150]; ADAPT is an intermediate framework between AADL models and dependability analysis tools based on Generalized Stochastic Petri Nets (GSPNs) [85].

Using a pivot model brings the benefit of reducing the number of transformations that are necessary to connect design models with analyses. Yet, this approach still requires implementing an important number of *ad hoc* transformations with little guarantees on their correctness.

III.4.2 Model query vs. model transformation

Figure III.14 depicts the elements that can be involved in the analysis of a design model. We can observe that the query and transformation issues are actually "orthogonal".

In the modeling space (bottom part of the figure), a model called $Model_A$ represents the system. $Model_A$ is defined by its metamodel $Metamodel_A$. $Model_A$ can possibly be transformed into another model named $Model_B$ (to switch from a design-specific model to an analysis-specific model for example). In this case the transformation is defined in terms of the source and target metamodels, i.e. $Metamodel_A$ and $Metamodel_B$ respectively. Notice that model transformations are defined as *syntactic* transformation, transforming a model's syntax into another.

In the analysis space (top part of the figure), the analysis algorithm processes precise data structures such as the periodic tasks model or graph-based tasks models. The analysis can directly query the models by taking into account the mapping between the analysis data structure and the model concepts. Notice that the link between models and analyses are *data*.

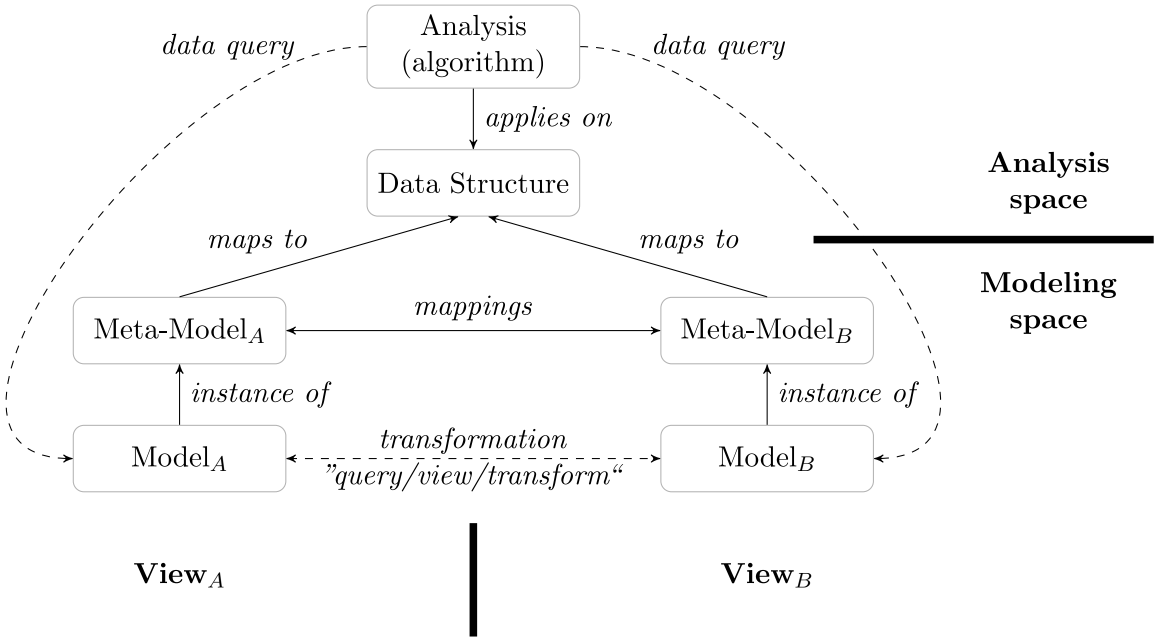


Figure III.14: Model queries vs. model transformations. *Analyses query data over models according to the mapping between data structures and metamodels. Multiple views involve model transformations defined in terms of source and target metamodels.*

From those observations, we can clearly distinguish between a model transformation and a model query. A model transformation is a *syntactic* operation, allowing to transform a model's syntax into another. A transformation occurs in the modeling space as it only affects models. A model transformation requires a full definition of metamodels and transformations (e.g. following the "query/view/transform" standard). In contrast, an analysis operates on *data*. Thus, a model query enables to extract some relevant data from a model in order to process them in a program (i.e. data structures + algorithm). A model query does not imply to reason in terms of transformations between a source metamodel and a target metamodel (i.e. language syntaxes) but data extraction by taking into account the mapping between analysis data structures and model concepts. Thus, model queries connect the analysis space with the modeling space.

Positioning with related works. The transformation-based analysis approaches presented in Section III.4.1 mix the elements displayed in Figure III.14 in a confusing manner.

In Figure III.15, we firstly distinguish between the tool spaces: the design tool on the left, and the analysis tool on the right. We secondly note that elements of the analysis space, i.e. the data structures and analysis algorithms, are not explicitly handled. Therefore, the data structures are only visible through the analysis-specific model, and targeting the analysis-specific model through model transformation is the only way to connect with the analysis. The transformation used to connect the tool spaces is:

- unidirectional: a model used for design in a first tool is translated into a model used for analysis in a third-party tool,
- exogenous: the models are defined in different technical spaces.

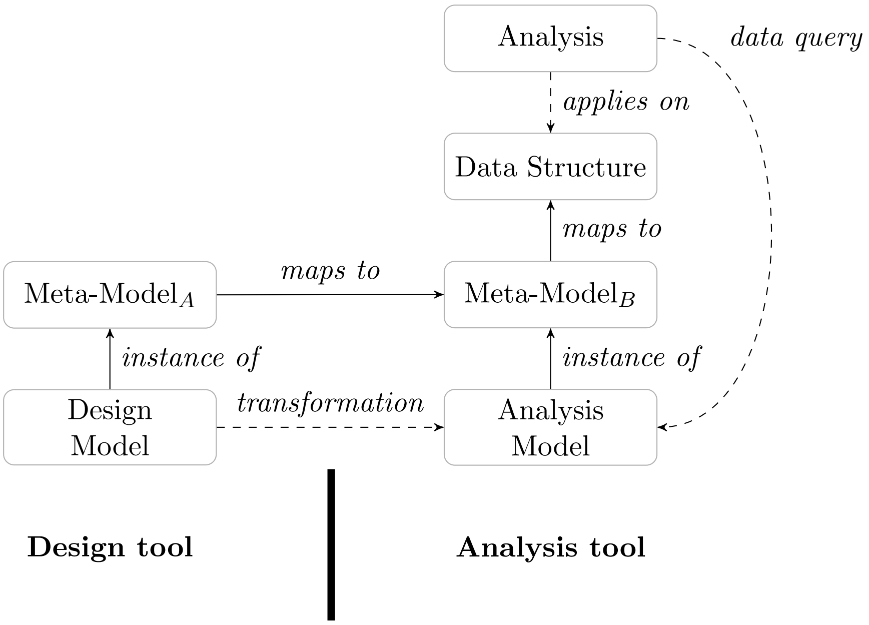


Figure III.15: Positioning with related model transformation approaches. *The approaches discussed in Section III.4.1 are a specific case of the view presented on Figure III.14: a model used for design in a first tool is translated into a model used for analysis in a second tool. The data structure is implicitly represented by the target metamodel, and the analysis applied on its own model.*

The main bias in this approach is to reason in terms of tool spaces, based on models, rather than business spaces, based on operations (i.e. modeling and analysis). Analyses are hard-wired in a transformation framework that translates a model used for design in a first tool into a model used for analysis in another tool, in order to comply with the execution needs of a specific analysis engine. Therefore, the analysis occurs at the end of a two-steps process: "horizontally" the transformation process, and "vertically" by letting the analysis tool using its own mechanisms to query the analysis-specific model. We already commented on the limitations of this approach in Section III.4.1.

III.5 Synthesis and conclusion

In this chapter, we presented query mechanisms called accessors to analyze the non-functional properties of a system from architectural models.

We firstly presented the rationale behind model queries. In particular, we identified the elements which are involved in the analysis of an architectural model – models and metamodels are the design components on the one hand, algorithms and data structures are the analysis components on the other hand – and the relations between them. We underlined the crucial role of the data structures at the core of the analysis definition, and reviewed several data structures that can be used for the analysis of real-time properties. We also emphasized the mapping that exists between analysis data structures and metamodels, making it possible to link an analysis to an architectural model.

This perspective led us to completely revisit the way analyses are applied on architectural models in Model-Driven Engineering. We showed that the application of an analysis on a model does not always require to translate a model used for design into a model used for analysis, and thus implement a complex and untrustworthy transformation chain. In fact, we showed that accessors enable just as well to analyze a model. These mechanisms enable to extract some relevant data from a model, and then analyze them. We implemented accessors through a dedicated Application Programming Interfaces in Python. As an example, we used this API to analyze real-time properties from AADL models.

In conclusion, accessors completely shift the way analyses are applied on domain-specific models. It is no longer necessary to take a “detour” via an analysis-specific models or a pivot model as soon as an implementation of accessors towards model internals is provided. The distinction between “design-specific” and “analysis-specific” models does not hold anymore. An analysis-specific model is not a model to be implemented in order to comply with a specific analysis engine, but simply represent the system from a particular point of view. Furthermore, we are able to analyze any model, as soon as an implementation of accessors towards these models is provided. Finally, implementation of accessors through a dedicated API facilitates the collaboration between designers and analysts while enhancing the reliability of the application.

In the next chapter IV, we will use those accessors to fully implement analyses, including their preconditions and postconditions.

Chapter IV

Semantics of an analysis

Abstract

Accessors introduced in the previous chapter enable to query and analyze non-functional properties of a system from architectural models. This chapter focuses on the analysis itself, especially its semantics. *As an introductory example (Section IV.1), we explain the difficulty to apply real-time scheduling analysis in a model-based engineering approach. We present our solutions in the following sections. In Section IV.2, we propose a general formalism to define the semantics of an analysis, and instantiate it to a simple real-time scheduling analysis. Section IV.3 evaluates several implementations with the help of accessors. This chapter terminates with a discussion about related works (Section IV.4) and a conclusion (Section IV.5).*

IV.1 Introductory example: model-based real-time scheduling analysis

In this introductory example, we consider real-time scheduling theory in general and feasibility tests in particular. We quickly remind the real-time task model used in such tests and the real-time scheduling problem. We then discuss feasibility tests and the difficulty to apply them in a model-based approach.

Task model. Let us consider a system that has to carry out a set of tasks. Flight control, flight guidance or fuel control are some examples of tasks in an airplane. In Figure IV.1, a task $\tau_i \in \mathcal{T}$ $(card(\mathcal{T}) = n,\ i, n \in \mathbb{N})$ is a software module, that is a set of instructions to execute. A task can have several characteristics, e.g. in the context of seminal works by Liu and Layland [127] the tasks are periodic. A periodic task τ_i consists of an infinite sequence of jobs $\tau_{i,j}$ $(j \in \mathbb{N})$. A task can admit an offset O_i that is the amount of time for the first release of the task. This implies that the j^{th} job of a periodic task is released at time $r_{i,j} = O_i + (j-1) \cdot T_i$ where T_i is the task period. Each job consumes an amount of processor time C_i called computation time. Finally, a task has a relative deadline D_i, or expressed on the j^{th} job of a periodic task: $d_{i,j} = r_{i,j} + D_i$.

Real-time scheduling. Real-time scheduling is the problem of building up an execution order such that timing constraints are met, usually deadline constraints. In the case of on-line scheduling, a scheduling algorithm decides the scheduling of a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on a set of processor $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ and, possibly, a set of shared resources $\mathcal{R} = \{R_1, R_2, \ldots, R_s\}$. Rate Monotonic (RM) is an example of scheduling algorithm which is mainly characterized by preemption (i.e. it is able to suspend a task execution, to execute one or several other tasks, and then resume the execution of the first task), deterministic deadlines ($D_i = T_i$) and fixed priorities according to the rule "the smaller the period, the higher the priority".

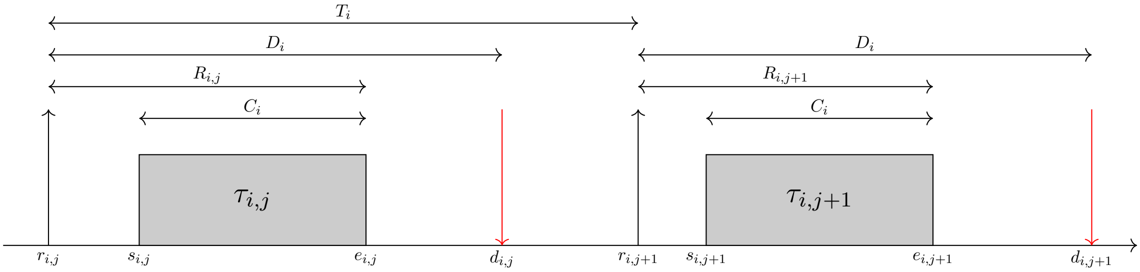


Figure IV.1: Usual representation of a real-time task with a Gantt diagram. *For a task τ_i: T_i the period, C_i the computation time and D_i the relative deadline. $\tau_{i,j}$ denotes the j^{th} job of a task i: $r_{i,j}$ is the release time, $s_{i,j}$ the start time, $e_{i,j}$ the completion time, $d_{i,j}$ the absolute deadline. A system is schedulable if $\forall \tau_i \in \mathcal{T}$, $\forall R_{i,j}$ the response time respects $R_{i,j} \leq d_{i,j}$.*

Feasibility tests. Feasibility tests are analytical methods to state if there is a schedule that will meet all deadlines for a given task set and scheduling algorithm, i.e. check whether a task set is *schedulable* according to a given scheduling algorithm.

For instance, Liu and Layland [127] proposed a test which is based on the analysis of the *processor utilization factor* U – the fraction of processor time used by the tasks. They have shown that a set of n periodic tasks is schedulable with the RM algorithm if:

$$U \leq n(2^{\frac{1}{n}} - 1); \text{ with } U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{LL-test}$$

The LL-test is a proved *sufficient condition* for the scheduling of a set of tasks with the Rate Monotonic algorithm and under the assumptions that all the tasks have no offsets, the deadlines are equal to the periods and the tasks are independent (that is, have no shared resources or precedence constraints).

Later developments have improved or proposed new feasibility tests, relaxed the assumptions, or considered new task models. For instance, Sha et al. [148] deal with real-time tasks with shared resources; the access to the resources is managed with a concurrency control protocol. They have shown that a set of n periodic tasks using the Priority Ceiling Protocol is schedulable with Rate Monotonic if the following test is verified:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{\frac{1}{n}} - 1) \tag{SRL-test}$$

In Equation (SRL-test), B_i denotes the worst-case blocking time for a task τ_i, that is the time that this task can be blocked by all the lower level tasks that can access a shared resource.

Another approach is to not use a feasibility test but calculate the worst-case response time R_i of each task. The set of task is schedulable by a given algorithm if and only if the worst-case response time of each task is less or equal to its deadline, e.g. see [151] for the response time analysis of a set of tasks scheduled under the Rate Monotonic algorithm.

How to use feasibility tests? Since the origins of the real-time scheduling theory in the 1970s, the research community has provided a multiplicity of feasibility tests, targeting many task models and providing numerous feedbacks on these models.

In the first place, the application of a feasibility test depends on the model which is provided for the analysis, e.g. there are at least 9 task models that can be used to analyze real-time workloads for preemptive uniprocessors [152, 131]. Those task models offer different trade-offs between expressiveness (modeling precision) and computation cost (analysis complexity). For instance, the aforementioned Liu and Layland's task model [127] is simple and easily computable (a few parameters, its complexity is linear in the number of tasks) but restricts the tasks to a representation that does not always fit the reality. At the opposite, the timed automata formalism [153] provides an accurate representation at the price of a much higher algorithmic complexity.

For a given model, numerous feasibility tests can be chosen. To give an idea, 200+ articles are cited by Sha et al. [21] as regards the advances in real-time modeling and associated analyses! In another survey, Davis et al. [126] examine the feasibility tests which are provided for multiprocessor architectures; and list about 120 different works. Last but not least, each analysis may report on schedulability in a different way: computed metrics (e.g. processor utilization factor, worst-case response times), scope of the result (e.g. exact test, sufficient condition only, necessary condition), etc.

Thus, applying the *right* real-time scheduling analysis on the *right* model is a tedious and error-prone task. The problem for the designer is first to define the conditions under which an analysis can be applied (e.g. assumptions on the system model) and then to state whether the input model complies with these conditions or not. In addition, the analysis result (processor utilization factor, worst-case response times, ...) must be completely interpreted in order to report on the schedulability status. In the next section, we propose solutions to address this problem:

- by fully defining an analysis with pre and postconditions in Section IV.2,
- by exploring implementation means in Section IV.3.

IV.2 Semantics of an analysis

An elementary model-based analysis process consists of the following computation chain, as pictured in Figure IV.2:

❶ the analysis inputs data from a model,

❷ the analysis program processes the data,

❸ the analysis outputs data about the model (i.e. analysis results).

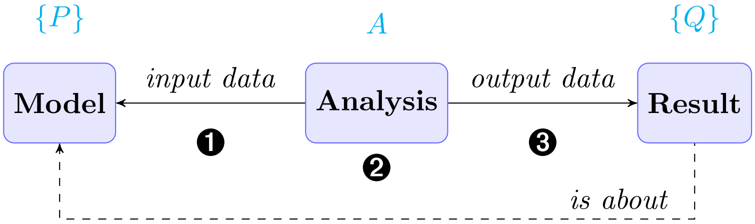


Figure IV.2: Elementary model-based analysis process. *An analysis can be made equivalent to a Hoare triple* $\{P\}\ A\ \{Q\}$*. Preconditions* P *express the properties to hold true in an input model to successfully execute an analysis* A*. Postconditions* Q *are the properties guaranteed on the model after the analysis execution, i.e. the analysis results.*

We can formally define the semantics of an analysis with a triple analogue to a *Hoare triple*.

Definition 12 (Analysis (semantics))**.** *An analysis is a triple* $\{P\}\ A\ \{Q\}$*:*

- P *is a logical assertion expressed on input data called the* precondition *of* A*,*
- A *is an analysis program to compute output data from input data,*
- Q *is a logical assertion expressed on output data called the* postcondition *of* A*.*

Preconditions P express the properties that the model must satisfy prior to execute an analysis. Postconditions Q express the properties that the analysis guarantees in return. Thus, if P is true on a model, executing A can lead to a model where Q is true. In practice, assertions can be expressed with first-order logic formulas and checked through a dedicated verification engine.

Example: semantics of a feasibility test. Let us consider a simple input data model that can be used for real-time scheduling analysis, consisting of the tuple ($\mathcal{T}$, $\mathcal{G}$, $\mathcal{P}$, S):

- $\mathcal{T}$ is the set of task, with each $\tau_i \in \mathcal{T}$ is a tuple (T_i, C_i, D_i, O_i) (respectively: the period, the computation time, the deadline and the offset),
- $\mathcal{G}$ is the graph (V,E) giving the dependencies between tasks,
  - V are vertices, each vertex is a task of the model $V \subseteq \mathcal{T}$,
  - $E \in V \times V$ are edges and represent dependencies between tasks,
- $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ is the set of processors,
- S is the scheduling algorithm, $S \in \{FP, RM, DM\}$.

Liu and Layland defined up to 10 assumptions on the task model to analyze with their feasibility test [127]:

- **mono-processor** (p_1): there is just one processor,
- **periodic tasks** (p_2): all tasks are periodic,
- **no jitter** (p_3): all tasks are released at the beginning of periods,
- **implicit deadlines** (p_4): all tasks have a deadline equal to their period,
- **independent tasks** (p_5): all tasks are independent, that is, have no shared resources or precedence constraints,
- **fixed computation times** (p_6): all tasks have a fixed computation time (or at least a fixed upper bound on their computation times) which is less than or equal to their period,
- **no self-suspension** (p_7): no task may voluntarily suspend itself,
- **preemption** (p_8): all tasks are fully preemptive,
- **no overheads** (p_9): all overheads are assumed to be null,
- **fixed priority** (p_{10}): all tasks have a fixed priority.

According to the input model defined previously and the assumptions given above, we can define the preconditions with predicates in First-Order Logic:

$$P_{LL-test} = \{p_1 \wedge \ldots \wedge p_{10}\}$$

with:

- $p_1 : \{P \mid card(\mathcal{P}) = 1\}$
- $p_2 : \{\forall \tau_i \in \mathcal{T} \mid T_i \neq \emptyset\}$
- $p_3 : \{\forall \tau_i \in \mathcal{T} \mid O_i = 0\}$
- $p_4 : \{\forall \tau_i \in \mathcal{T} \mid T_i = D_i\}$
- $p_5 : \{\mathcal{G} \mid card(V) = 0\}$
- $p_4 : \{\forall \tau_i \in \mathcal{T} \mid C_i \leq= T_i\}$
- $p_{10} : \{P \mid S = RM \vee S = FP\}$
- p_7, p_8 and p_9 are axioms, alternatively the data model could be extended with any suitable data structure onto which those predicates could be expressed (for example a graph explaining the task behaviors).

Provided the respect of the preconditions, the analysis by Liu and Layland computes the processor utilization factor U (see LL-test). Hence, the postcondition that determines the schedulability of the task set is given by $Q_{LL-test} = \{q_1\}$ with:

- $q_1 : \{U \mid U \leq card(\mathcal{T})(2^{\frac{1}{card(\mathcal{T})}} - 1)\}$

Section IV.3 presents a practical implementation of this formalism.

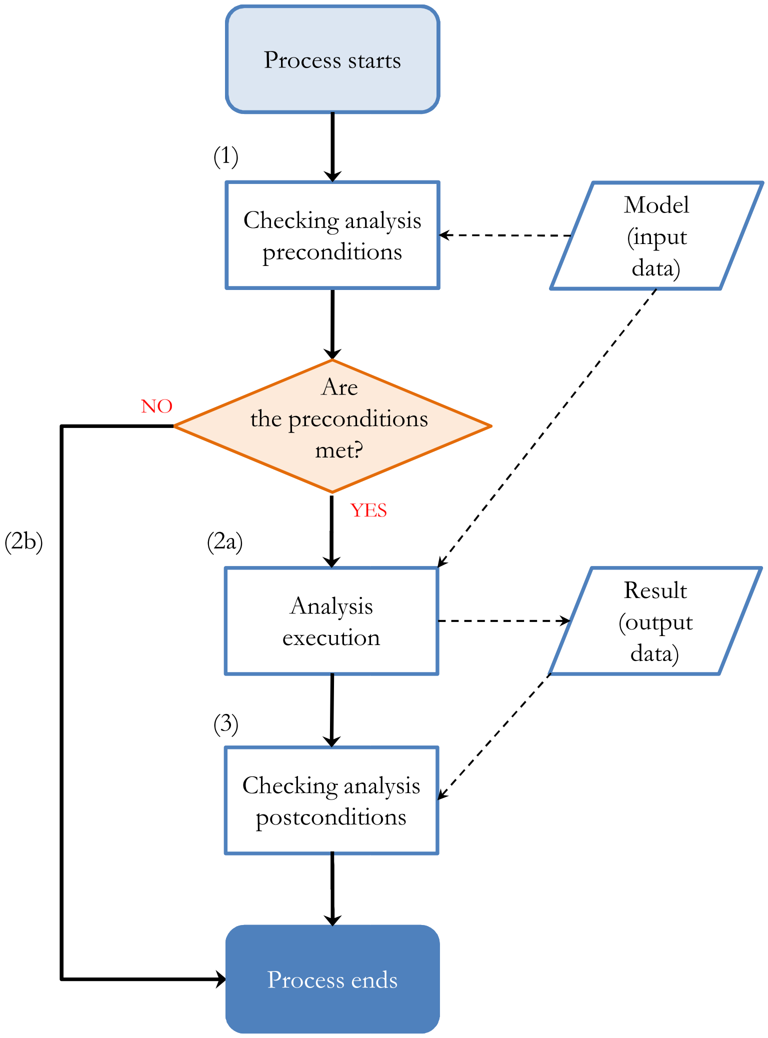


Figure IV.3: Process Flowchart describing the analysis execution. *The analysis execution depends on the verification of analysis preconditions. The analysis result is checked at the end of the analysis execution.*

IV.3 Implementation of the analysis execution

In the previous section, we discussed a general formalism to define the semantics of an analysis. In this section, we explain how this formalism can be used to completely and correctly analyze architectural models. We quickly explain our approach before reviewing several implementations.

IV.3.1 Proposed approach

In the previous section, we showed that an analysis can be made equivalent to a Hoare triple. In particular, the preconditions are the properties to be checked true in an input model to successfully execute an analysis. The postconditions are the properties guaranteed on the model after analysis execution. At run time, we hence evaluate the preconditions prior to execute the analysis, and check the postconditions at the end of the analysis execution.

Figure IV.3 explains the analysis process in greater detail with a Process Flow Diagram. At the very beginning, we verify the analysis preconditions on the model (1). If the model fulfills the preconditions then we can the carry out the analysis (2a). Otherwise, the process terminates (2b). Lastly, we check the analysis postconditions (3). The process ends whether the postconditions are confirmed or not.

In the following sections, we evaluate several ways to implement this approach:

- using general-purpose constraint languages, e.g. REAL [154] with AADL models (Section IV.3.2),
- using the generic query mechanisms introduced in the previous Chapter III together with the Python programming language (Section IV.3.3),
- using transformation or metamodeling approaches (Section IV.3.5).

IV.3.2 A first implementation with constraint languages

We firstly implement the approach in Figure IV.3 by using the REAL constraint language which can be used with AADL models.

IV.3.2.A REAL at a glance

In former works, Gilles et al. [154] proposed REAL (Requirements Enforcement and Analysis Language) to express and verify constraints on AADL models. It has been designed as an AADL annex language and comes with its own interpreter.

REAL considers `theorems` as basic execution units. A theorem expresses one or more constraints to check on an AADL model based on model queries and analysis capabilities.

For our concern, REAL provides key features:

- it makes it possible to manipulate the elements of an AADL instance model as many sets (`thread_set`, `bus_set`, `memory_set`, etc.) with getters for their properties (`get_property_value`),

- it enables mathematical computing with classical operators (+, −, ×, etc.) or high-level functions (`cardinal`, `min`, `max`, etc.),
- it provides a syntax for predicate calculus with quantifiers (∀, ∃), logical operators (¬, ∧, ∨, etc.) and predicate functions (`is_subcomponent_of`, `is_bound_to`, etc.).

IV.3.2.B Application to the Liu and Layland's feasibility test

We can implement the analysis and its preconditions through dedicated REAL theorems.

Preconditions. In Listing IV.1, the `periodic_task` theorem implements the precondition p_2="all tasks are periodic" in Section IV.2. The expression of this precondition in a theorem is straightforward: we check that the `Period` property is provided (`property_exists` predicate function) for each element in the task set (the `thread_set` in the AADL instance model).

The theorems needed to express the **mono-processor** (p_1), **no jitter** (p_3), **implicit deadlines** (p_4), **fixed computation times** (p_6) and **fixed priority** (p_{10}) preconditions are of similar complexity.

```
-- This theorem checks that the release period of each task exists
theorem periodic_tasks
   foreach t in thread_set do
   check (property_exists (t, "Period"));
end periodic_tasks;
```

Listing IV.1: An example of REAL theorem. *A REAL theorem expresses constraints on a AADL model. The simple theorem here is used to check that the `threads` described in the model are periodic.*

Listing IV.2 depicts the theorem for the precondition p_5="all tasks are independent". It translates the assertion for AADL models with two sub-theorems: `no_tasks_precedences` and `no_shared_data`.

The first sub-theorem assumes that a precedence (`task_precedence`) involves a connection between AADL threads (`Is_Connected_To (t2, t1)` with `t1` and `t2` are elements in `thread_set`) and checks that the number of precedences is null (`cardinal (task_precedence) = 0`).

In the second sub-theorem, we assume that a shared data situation occurs when at least two AADL threads access a same AADL data (`Is_Accessing_To (t,d)` with `d in Data_Set` and `t in Threads_Set`). We thus check that at most one thread accesses each data (`Cardinal (accessor_threads) <= 1`).

```
-- independent_tasks : this theorem checks that tasks are mutually independent, ie
-- (1) tasks do not share (access) a same resource and
-- (2) tasks have no precedence relationships

theorem independent_tasks
   foreach e in local_set do
      requires(no_tasks_precedences and no_shared_data);
   check (1=1);
end independent_tasks;

-- subtheorem
theorem no_tasks_precedences
   foreach t1 in thread_set do
   task_precedence := { t2 in thread_set | Is_Connected_To (t2, t1)};
   check ((cardinal (task_precedence) = 0));
end no_tasks_precedences;

-- subtheorem
theorem no_shared_data
   foreach d in Data_Set do
   accessor_threads := {t in Thread_Set | Is_Accessing_To (t, d)};
   check (Cardinal (accessor_threads) <= 1);
end no_shared_data;
```

Listing IV.2: Independent tasks theorem. *The theorem on top checks that the* `threads` *in the AADL model are independent: (1) a task cannot precede another, i.e. in AADL a* `thread` *cannot be connected to another one (second theorem); (2) the* `threads` *cannot share* `data` *with each other (third theorem).*

Analysis and postconditions. Listing IV.3 depicts the full implementation of the Liu and Layland's feasibility test with REAL theorems. The topmost theorem `liu_layland_feasibility_test` implements the feasibility test.

In this theorem, we first evaluate the (pre)conditions under which the analysis is applicable (`require` keyword at line 6). The preconditions are listed in the `liu_layland_assumptions` sub-theorem (lines 19 to 23) and fully defined in other sub-theorems (e.g. we discussed the `periodic_tasks` and `independent_tasks` theorems in previous paragraphs, see Listings IV.1 and IV.2). If the preconditions are met, then the test can be executed (the `requires` command at line 6 aborts the main theorem if any predicate is false).

The analysis then executes (`compute` keyword at line 10). We calculate the processor utilization factor (`var U`, line 10) via the `processor_utilization_factor` sub-theorem (lines 30 to 34). This sub-theorem needs the set of threads, previously retrieved from the AADL model at lines 9.

Lastly, we evaluate the postcondition (`check` keyword at line 12). We check that the processor utilization factor is under the acceptable limit. If the test succeeds, then the task set represented in the AADL model is schedulable.

IV.3.2.C Lessons learned in using REAL

We firstly observe that a constraint language is defined by a precise (meta)model. Consequently, a constraint language can only be used with (meta)models that belong to the same technical space. For example, REAL is defined by the AADL metamodel and can only be used with AADL models, OCL can only be used with MOF-compliant models such as UML, etc.

From our practical experience, we note that a constraint language such as REAL do not always meet our needs in terms of expressiveness, e.g. because of restricted operators, limited control flow, etc. In particular, a major shortcoming is that model queries must be defined in terms of design-oriented data, obliging to reason about analysis data through design-oriented concepts. For example in this subsection, the Liu and Layland's feasibility test is defined through REAL theorems. Consequently, the analysis must be tailored to both the REAL syntax and AADL design-oriented concepts. Preconditions such as **non self-suspension**, **preemption** and **no overheads** cannot be expressed because AADL models do not enable to model such cases. Behavioral modeling would be more adapted to represent these real-time systems; provided that the constraint language enables to query this kind of models.

In conclusion, constraint languages such as REAL or OCL enable to query and analyze data structures. However, as "design-specific" query languages, these languages suffer strong limitations in terms of queried models and expressiveness. In the following, we solve this problem by combining generic accessors with the Python programming language.

```
1  -- liu_layland_feasibility_test : this main theorem implements a
     feasibility test
2
3  theorem liu_layland_feasibility_test
4    foreach e in Processor_Set do
5      -- verification of the analysis preconditions
6      requires ( liu_layland_assumptions );
7      -- analysis of the "model of tasks"
8        Proc_Set(e) := {x in Process_Set |  Is_Bound_To (x, e)};
9        Threads := {x in Thread_Set | Is_Subcomponent_Of (x,
             Proc_Set)};
10     var U := compute processor_utilization_factor (Threads);
11     -- Liu and layland's test
12     check (U <= (Cardinal (Threads) * (2 ** (1 / Cardinal (Threads
          ))) -1));
13 end liu_layland_feasibility_test;
14
15 -- subtheorem: verification of the test assumptions
16
17 theorem liu_layland_assumptions
18   foreach t in thread_set do
19     requires (mono_processor and periodic_tasks
20        and  no_offsets and  implicit_deadlines
21        and  independent_tasks and fixed_computation_times
22        and fixed_priority);
23     check (1=1);
24 end liu_layland_assumptions;
25
26 -- subtheorem: computation of the processor utilization factor
27
28 theorem processor_utilization_factor
29   foreach e in Local_Set do
30     var Period := get_property_value (e, "period");
31     var WCET := last (get_property_value (e, "
          compute_execution_time"));
32     var U := WCET/Period;
33     return (MSum (U));
34 end processor_utilization_factor ;
```

Listing IV.3: A complete feasibility test implemented in REAL. *The analysis starts in the theorem on top. At line 6, the preconditions are verified by calling the second theorem. If all the assumptions associated to the test are true, then the processor utilization factor is calculated by calling the third theorem at line 10. The postconditions are finally checked at line 12.*

IV.3.3 Implementation through accessors in Python

We acknowledged that constraint languages are dedicated to Domain-Specific Modeling Languages (e.g. REAL is used with AADL models, OCL with UML models, etc.) neither enable to address analysis data structures nor to express the analysis logic easily. This second implementation combines accessors with the Python programming language to fully implement an analysis.

IV.3.3.A Motivations for Python

In Chapter III, we introduced accessors so as to retrieve data from a model and analyze them. Figure IV.4 reminds the approach: an analysis implemented in Python is carried out via access to data on a model written for example with AADL. We use the Python programming language to both extract the data from AADL models and analyze them. In this subsection, we extend this approach to fully implement an analysis, as previously depicted in Section IV.3.1.

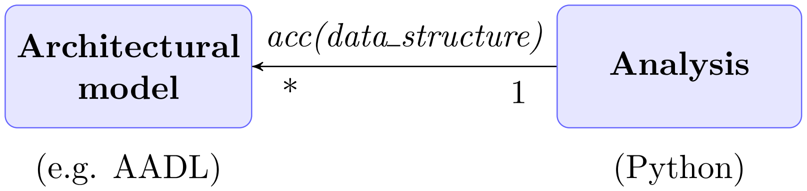


Figure IV.4: Analysis of an architectural model using accessors. *We carry out the analysis via access to data on an architectural model represented with AADL. We firstly retrieve the data in the model before analyzing them in a Python program.*

For our concern, Python provides key features:

- usability: Python is high-level, general-purpose programming language that support multiple programming paradigms, including object-oriented programming,
- rich and simple syntax: Python provides numerous data types (e.g. Boolean, signed integers or float to represent numbers, strings for sequence of characters, sets or lists for sequence types, etc.) and operators on them (assignment, arithmetic, logical, relational, etc.), usual control flow and decision mechanisms (loops, branches, and function calls), and many customized tools via built-in functions or in external libraries,
- extendibility: Python enables to structure the program with modules and packages and, in doing so, to create reusable libraries, e.g. to help stakeholders to reuse existing model accessors and analyses or develop new libraries,
- portability: Python is a cross-platform software that can run on a wide variety of systems through code interpretation.

IV.3.3.B Application to the Liu and Layland's feasibility test

We implement both the analysis and its preconditions through a related function, the necessary data structures being passed as function arguments. Listing IV.4 shows a prototype of an analysis function with the `def` keyword in Python.

```

    def analysis(self, required_data_structure):

     """ both the analysis and its preconditions are implemented in Python through an analysis function with data structures passed as arguments """
```

Listing IV.4: Definition of a precondition through a Python function.

Preconditions. Listing IV.5 depicts three precondition checks in a Python program: `mono_processor`, `fixed_computation_times` and `independent_tasks`. Each function is carried out on its own data structure passed as a function parameter, that is `processors_list`, `tasks_list` and `dependency_graph` respectively.

```
""" Examples of functions to check preconditions in Python
    Arguments: data structures
"""

    # precondition 1
    def __mono_processor(self, processors_list):
        if len(processors_list) != 1 :
            return False
        return True

    # precondition 6
    def __fixed_computation_times(self, tasks_list):
        for task in tasks_list:
            if task.worst_case_execution_time != None and task.worst_case_execution_time > task.period:
                return False
        return True

    # precondition 5
    def __independent_tasks(self, dependency_graph):
        for task in dependency_graph:
            dependent_tasks=dependency_graph[task]
            if len(dependent_tasks) > 0:
                return False
        return True
```

Listing IV.5: Three precondition checks implemented with Python. *Preconditions are verified within functions (i.e. `mono_processor`, `fixed_computation_times` and `independent_tasks`) with help of data structures passed as parameters (`processors_list`, `tasks_list` or `dependency_graph` respectively).*

The `mono_processor` function implements the precondition p_1="there is just one processor" in Section IV.2 by simply checking that there is only one element in the `processors_list`.

The `fixed_computation_times` function checks the precondition p_6. We verify that all tasks in `tasks_list` have a computation time (`task.worst_case_execution_time`) which is less than or equal to their period (`task.period`).

The verification of the precondition p_5="all tasks are independent" is carried out directly on the `dependency_graph` data structure. The graph is built out of lists and dictionaries in Python: each key in the dictionary is a vertex of the graph (a task) and the corresponding value is a list containing the vertices that are connected via an edge to this vertex (that is, dependent tasks). For each key in the dictionary (i.e. `task in dependency_graph`), we check that the corresponding value (i.e. `dependent_tasks`) is empty.

```
1  """ Liu and Layland feasibility test in Python """
2
3  class liu_layland_fesibility_test(Analysis):
4
5     def analysis(self, model):
6
7     #input data structures (model access)
8     tasks_list=model.get("LIST_OF_TASKS")
9     processors_list=model.get("LIST_OF_PROCESSORS")
10    dependency_graph=model.get("TASKS_DEPENDENCIES")
11
12    try:
13       #check analysis preconditions
14       assert (self.__mono_processor(processors_list)), "p1"
15       assert (self.__periodic_tasks(tasks_list)), "p2"
16       assert (self.__no_offsets(tasks_list)), "p3"
17       assert (self.__implicit_deadlines(tasks_list)), "p4"
18       assert (self.__independent_tasks(dependency_graph)), "p5"
19       assert (self.__fixed_computation_times(tasks_list)), "p6"
20
21       #compute the analysis and check postcondition is true
22       assert(self.__ll_test(tasks_list),"q1"
23
24    except AssertionError as e:
25       print ’analysis aborted ’, e.args
```

Listing IV.6: A complete feasibility test implemented in Python. *The test is implemented by the* `analysis` *method in the* `liu_layland_feasibility_test` *class. The preconditions are checked at the beginning of the function (*`assert` *statements). If no exception is raised (*`try-except` *statement), we execute the feasibility test via the* `ll_test` *function.*

Analysis and postconditions. Listing IV.6 shows a complete implementation of the Liu and Layland's feasibility test with Python. We implement the feasibility test through a `liu_layland_fesibility_test` class which has an `analysis` method. First of all, the analysis retrieves the input data structures from the AADL model: `processors_list`, `tasks_list` and `dependency_graph` (lines 8 to 10). Then the preconditions are checked with useful functions and data structures (lines 14 to 19). We use `assert` statements to evaluate the preconditions. An assertion raises an exception, caught and handled with the `try-except` statement, if a precondition is evaluated to false. If an exception occurs in the try clause, the analysis terminates. Contrariwise if all the preconditions are true, then the feasibility test is to be executed via the `ll_test` function (line 22). Violating the postcondition (i.e. assert statement at line 22) raises a last program exception synonym of analysis failure.

IV.3.4 Constraint Language vs. accessors+Python

We saw that constraint languages enable to query model instances and analyze them. In this chapter, we used the REAL constraint language to analyze AADL models. However, we noted two main drawbacks when using constraint languages:

- *queried models*: constraint languages are limited in terms of addressable models as they are included in a specific technical space, e.g. REAL with AADL, OCL with UML,
- *expressiveness* of constraint languages is not adapted to our use, e.g. operations concern design-oriented data (obliging to reason about analysis data through design-oriented concepts), languages may have limited operations or control flow (according to our practical experience with REAL), or be at times unnecessary verbose and hard to read (both OCL and REAL languages).

To overcome these issues, we presented an improved implementation that combined accessors (introduced in Chapter III) with the Python programming language. We clearly separate data definition (data structures), from data extraction (accessors), from data analysis (Python program). In this way, we enhance the implementation of the analysis and its preconditions:

- *queried models*: accessors enable to analyze any model as soon as an implementation of accessors towards these models is provided; in addition, both the data structures and the language used to evaluate these data structures are independent of the models,
- *expressiveness*: this approach makes it possible to directly analyze analysis-specific data structures rather than interpreting them through a third-party metamodel; in addition, Python is general-purpose programming language which enables to easily express analysis operations with a simple and rich syntax, and additional libraries.

IV.3.5 Other possible implementations

The implementations presented in the previous sections combined different kind of accessors with a dedicated language to analyze architectural models. Accessors enable to extract data from a model, whereas a constraint language (e.g. REAL) or a general-purpose programming language (e.g. Python) makes it possible to analyze such data. We sketch some other possible implementations that we found less optimal.

Implementing preconditions as part of the transformation process. This first alternative implementation would apply in a situation where a transformation is necessary to translate a model used for design into a model used for analysis (notice that we dismissed this approach in the previous Chapter III, more arguments are provided Section III.4). In such a case, precondition checks could be implemented as specific transformation rules, e.g. expressed with ATL [15]. Yet, we note several limitations with this approach. First, this approach is only applicable within a fully defined modeling framework with models, metamodels and transformation languages; and is restricted to this specific technical space. The second main disadvantage is that analysis preconditions must be checked according to model syntaxes (obliging to think analysis data structures in terms of design-oriented concepts) and with help of the transformation language (which can have limited capabilities in terms of data exploration, operations, and so on).

Well-formedness rules – constraining model instantiation. Another approach is to implement preconditions via well-formedness rules (WFR) as part of the metamodel definition. Through WFRs, models are tuned to conform to a specific analysis (or transformation) engine. In other words, model instances must satisfy the WFR rules to apply an analysis. A constraint language such as OCL can be used with UML-based metamodels. First, we note that this approach is very restrictive as it constrains apply on the construction of the models (i.e. definition of the metamodels). Secondly, we claim that this approach is paradoxical: models are tailored to fit analysis aims, and not to implement system-specific requirements. Lastly, we observe that analysis preconditions must be adjusted to DSML syntaxes (that is, to both the design language and the constraint language).

The main advantage of our approach is to separate precondition checks from off-topic transformation and metamodeling issues. First, we clearly separate concerns, i.e. data definition, from data extraction, from data analysis. Secondly, we can use all the powerful features provided by a general-purpose programming language, instead of fitting preconditions into an unsuitable constraint or transformation language.

IV.4 Discussion: related works

This section discusses related works that aim at providing context-aware analyses, and then compare to our works. We distinguish between implementation means and analysis frameworks.

Implementation through constraint languages. OCL (Object Constraint Language) [155, 156] is a constraint language working with UML models. In practice OCL can be used for expressing many sorts of (meta)model queries, manipulations and requirements. OCL is adopted as a standard by the OMG, the latest specification of OCL is version 2.4 [157].

REAL (Requirements Enforcement and Analysis Language) is a language proposed by Gilles [158], [154] aiming at expressing constraints on AADL models. REAL has initially been designed to support system optimization in a model-based process [159] but can be used more generally to enforce some semantics or consistency checks on AADL models. REAL is available as an AADL annex language and comes with its own interpreter integrated in the OCARINA tool [136, 88].

We firstly note that constraint languages can be used to express many kinds of model queries. Thus, they do not provide specific guidelines to implement analysis at large. Constraint languages are query language towards design-specific models (e.g. REAL/AADL, OCL/UML). Thus, constraints languages cannot inter-operate and have a limited expressiveness (e.g. design-oriented data structures, operations, control flow, etc.). See our experimentation with REAL in Section IV.3.2.

Analysis frameworks. We can mention works from Ouhammou [70] and Gaudel [160] that preceded our works.

Ouhammou et al. [161] proposed an intermediate framework between real-time design languages and real-time analysis tools called MoSaRT (Modeling-oriented Scheduling analysis of Real-Time systems). MoSaRT consists of a Domain Specific Modeling Language providing the core concepts of real-time systems, and an Analysis Repository to analyze the models with help of the real-time scheduling theory. The main novelty is the automatic selection of real-time scheduling analyses. For this purpose, the authors firstly formalize the applicability of real-time scheduling analyses with *real-time contexts* (i.e. a set of assumptions related to the tasks). Real-time contexts are then automatically checked on the models to choose a suitable analysis. This feature is implemented through a set of OCL invariants representing analysis assumptions.

Gaudel [160] build on architectural design patterns to select real-time feasibility tests [162, 163] in the Cheddar tool. The authors define an architectural design pattern as a set of applicability constraints applying on architectural models. They implement their own algorithms to select the feasibility tests in the Cheddar tool [164]. This algorithm aims at detecting the design patterns which are present in a model and analyze their composition if multiple patterns are represented.

We note that the two approaches are devoted to specific DSMLs and tools (Cheddar and MoSaRT). It is hence necessary to either re-implement the approach to reuse it in another tool or to define bridges between tools, e.g. a transformation chain exists from AADL to Cheddar or MoSaRT tools [137, 70], or more recently Gaudel et al. [149] redefined architectural design patterns for AADL models though AADL *subsets*. Let us note that these palliatives have several disadvantages: the semantics gap between design languages, adaptation of technical solutions (for instance, constraint language vs. *ad hoc* implementation of selection algorithms), weak guarantees on the transformation correctness, etc.

Our approach builds on and generalizes the related works while proposing different implementation means. We introduced a more general formalization of the semantics of an analysis based on the Hoare notation. We note that this notation applies for real-time scheduling analysis just as well as for any sort of analysis (dependability, security, etc.). A full analysis, including preconditions and postconditions, can be implemented in several ways, e.g. with constraint languages (REAL in Section IV.3.2) or through accessors combined with a general-purpose programming language (Python in Section IV.3.3). Our approach is naturally portable and interoperable through the generic accessors introduced earlier in Chapter III. We finally note that we are able to implement an improved decision process based on contracts to define analysis interfaces and SAT methods to evaluate them (this contribution is presented in Chapter V).

IV.5 Synthesis and conclusion

We showed in an introductory example the difficulty to correctly apply real-time scheduling analyses in a MDE process. The problem for the designer is firstly to define the conditions under which an analysis can be applied, and secondly to state whether the input model fulfills these expectations or not. In addition, the analysis result should be completely interpreted in order to report on the schedulability status.

In this chapter, we formalized the analysis execution. We showed that an analysis can be made equivalent to a Hoare triple $\{P\}\ A\ \{Q\}$. The preconditions P in this triple are the properties to hold true in an input model to successfully execute an analysis A. The postconditions Q are the properties guaranteed on the model after analysis execution. With preconditions and postconditions, an analysis is complete and sound. Hence, a full analysis requires first validating the preconditions, and lastly checking the postconditions. We experimented two implementations of this approach: constraint languages (REAL on AADL models) first, and accessors introduced earlier in Chapter III combined with a general-purpose programming language (Python) next. We noticed that the second implementation is more efficient: easier implementation, better portability and interoperability.

We extend the work presented in this chapter to provide greater decision and orchestration support. The approach presented in Chapter V uses analysis contracts to define analysis interfaces and SAT methods to evaluate them.

Chapter V

Contract-driven analysis

Abstract

In the previous chapter we presented an approach based on pre and postconditions to deal with the context associated to an analysis. We showed that model query implemented either by a classic constraint language (e.g. REAL or OCL) or a general-purpose programming language (e.g. Python) combined with a set of model accessors could be used to check the preconditions. Yet, these artifacts, in the current state, offer a limited decision support when analyses have to be considered in a design workflow: e.g. which analysis can be applied on a given model? How to handle the analysis results? Is it possible to combine analysis results? Are there interference between analyses? Etc. To answer those questions, one must be able to better characterize analyses with their interfaces and properties first and be able to exhaustively interpret the analysis features afterwards.

In Section V.1, we explain that analyses as an integral part of Model-Based Systems Engineering (MBSE) approaches must be handled in a systematic manner. In Section V.2, we introduce contracts as a means to formally define the design components – models, analyses, goals. We then explain in Section V.3 how contracts can be used to systematize the analysis activities in the design process. In particular, we present the proof-of-concept of a contract-driven analysis approach supported by Alloy. This chapter ends with a discussion in Section V.4 and a conclusion in Section V.5.

V.1 Motivating context: analysis in a design process supported by an architectural language

Architectural languages provide a support for the Model-Based Engineering of real-time embedded systems [18]. An advanced design process supported for instance by AADL involves conjoint modeling and analysis activities, as shown by Figure V.1:

1. the AADL model is the centerpiece of the process. The AADL model represents the top-level architecture of the system. It depicts the static software architecture, the computer platform architecture with behavioral descriptions in a single model,

2. analyses are carried out on the AADL model to provide *feedbacks* about the system design. Analyses can be used for validation purposes (e.g. to assess the

processor workload or analyze the schedulability of the task set) or to compute new data that could be added to the model (for instance, we combined AADL models with an analysis chain in order to define some network parameters for a Flight Management System in [165, 166]),

3. the system is progressively defined and validated via the successive modeling and analysis steps. Platform-Specific Models (runnables, configuration files, etc.) can be fully or partially generated from higher level models (for instance, see works by Lasnier [167])

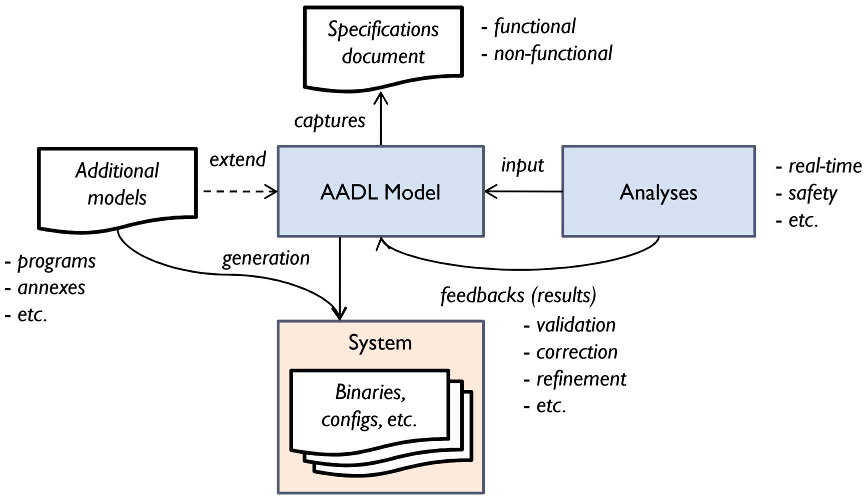


Figure V.1: Architecture-centric Model-Based Systems Engineering process supported by AADL. *AADL models capture the functional and non-functional architecture of an embedded system. We conduct analysis from AADL analytical representations, e.g. to assess real-time or safety properties. The system is progressively defined and validated via the successive modeling and analysis steps. Finally, we generate Platform-Specific Models (PSM) such as runnables.*

Yet, we note that, apart high-level principles and abstract guidelines, MBSE tools such as OSATE (Open Source AADL Tool Environment) [134] provide little support to carry out the modeling and analysis steps.

How to make the analysis systematic? Let us discuss a simple design flow represented with a directed graph in Figure V.2. The vertices depict the modeling and analysis activities while the directed edges represent the transitions between activities:

- M: the designer starts by modeling the system with AADL,
- $preA_1$, A_1: the designer can apply a feasibility test (A_1 vertex) to assess a real-time property: `are the tasks schedulable?` Before the designer must check the analysis preconditions ($PreA_1$ vertex) as discussed in Chapter IV,
- G_1, M': if the feasibility test succeeds the model is validated (G vertex: *the deadlines are met*), if not the designer must propose a correction (vertex M').

The process continues to assess other properties based on the validated AADL model or its correction.

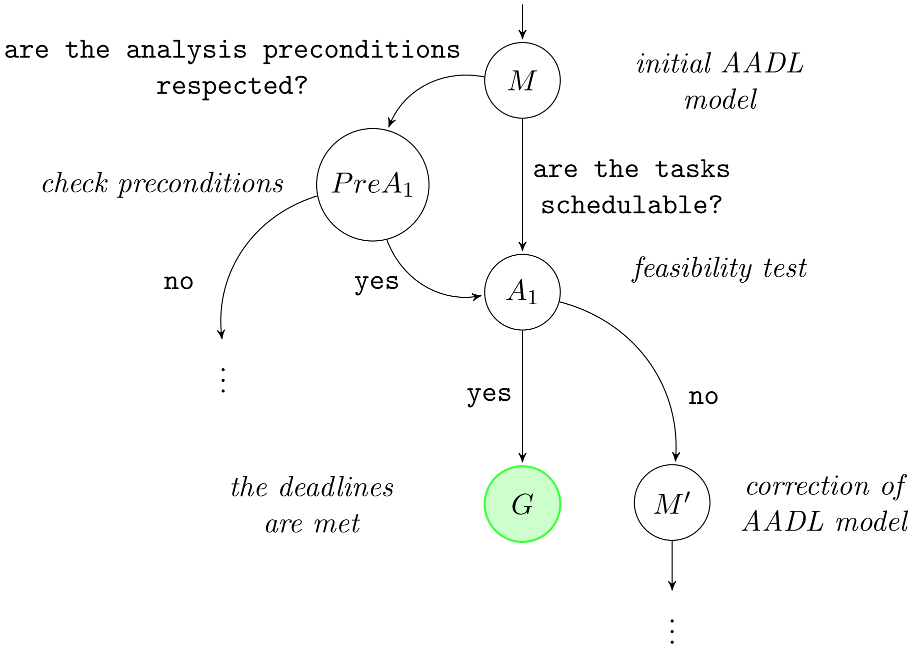


Figure V.2: An example of design workflow. *The design flow involves modeling and analysis activities to achieve goals. Vertically: we assess temporal constraints (goal) on an initial AADL model with feasibility tests (analysis). If the analyses do not succeed we must correct the model. The process continues to achieve other goals.*

We notice that the design flow on the figure V.2 systematically involves the following elements:

1. one or several *models* that must be analyzed: M and M',
2. *goals* which are the properties that must be assessed on those models: G,
3. *analyses* that must be applied on the models to achieve goals: $PreA_1$, A_1. Analyses can be combined to provide intermediate data ($PreA_1$) or end data (A_1).

The problem for the designer is hence to handle a workspace which is made up of *models* representing the system and *analyses* that should be applied to meet specific *goals*. As stated by Vaziri and Jackson [168], classic constraint languages such as OCL or REAL used for preconditions checks in the previous Chapter IV do not provide the adequate decision support to tackle this problem. Indeed, one must be able to fully characterize the design components – models, analyses, goals – with their interfaces and properties. Analysis features should then be exhaustively interpreted in order to answer specific questions such as: which analysis can be applied on a given model? For a given goal? Are there analysis results to possibly combine? Are there interference to forbid between analyses? Etc.

We present our solutions for this purpose in the next sections:

- we first present analysis contracts in the Section V.2,
- we then explain how contracts can be used to systematize the analysis activities in a design workflow Section V.3.

V.2 Contracts

In this section, we introduce *contracts* as a means to formally define models, analyses and goals.

We provide preliminary definitions for models, analyses and goals. We then introduce contracts and their properties. All the concepts discussed in this section are illustrated with help of a simple example coming from the real-time scheduling theory.

V.2.1 Preliminary definitions: models, analyses and goals

Model. A model is "a sound abstraction of an original that depends on the science matter" [16], i.e. here an embedded system. We propose the following definition for the approach presented in this chapter.

Definition 13 (Model)**.** *A model is a couple $M = (S, P)$:*

- *S is a set of data,*
- *P is a set of properties. A property is an association of data structures $P : S \rightarrow S$.*

Data are defined as per basic data types such as mathematics-oriented types (e.g. Boolean, integers, floats, etc.) or domain-specific types (e.g. scheduling algorithms in real-time theory), or according to more sophisticated data structures (e.g. using sets, lists, graphs, etc.). Data structures are closely related to the facets of the system being considered (e.g. a set of tasks in a real-time system).

Properties can specify invariants such as tasks periods, processor scheduling policies, but also a model status like being schedulable, safe, etc.

Analysis. Informally, we can define an analysis as a *"a careful study of something to* learn *about its parts, what they do, and how they are related to each other ; an explanation of the nature and meaning of something*[1]*".* We propose the following definition.

Definition 14 (Analysis (function))**.** *An analysis is a function that operates over a model $A : M \rightarrow M$.*

Goal. According to the previous definitions, we can combine models and analyses to produce other models. We finally define goals as particular models.

Definition 15 (Goal)**.** *Let $\mathcal{M}$ be a set of models and $\mathcal{A}$ be a set of analyses. A goal is a particular model required over a set of models and analyses $G : \mathcal{M} \times \mathcal{A} \rightarrow M$.*

[1] general definition by `http://www.merriam-webster.com/`

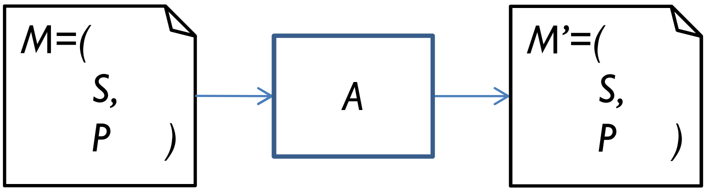


Figure V.3: Analysis seen as a function. *An analysis A inputs a model M and outputs another model: $M' = A(M)$.*

Example. Let us discuss the models, analyses and goals of the design workflow represented in Figure V.4.

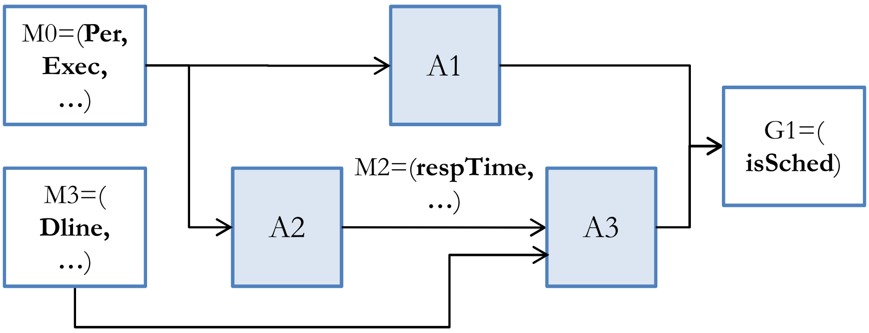


Figure V.4: Models, analyses and goals in the design workflow of a real-time system. *From the periodic task model M_0, either* feasibility test *(A_1) or* response time analysis *(A_2) enables to conclude about the schedulability of the tasks (G_1). An additional analysis A_3 inputs $M2$ together with an extra model $M3$ in order to compare the responses times against the deadlines.*

We can represent a real-time system through the periodic task model proposed by Liu and Layland [127] (referred to as M_0 in the following). This model represents a set of tasks with periods and execution times plus a processor with a scheduling policy, as specified by Table V.1.

The latter tasks model allows for several sorts of schedulability analysis. For instance, we can use a *feasibility test* based on the computation of the processor utilization factor [127] or a *response time analysis* [129] (respectively referred to as A_1 and A_2 in the following).

Table V.2 depicts the results of the two analyses. The first model($M_1 = A_1(M_0)$) is made up of a property (**isSched**) which associates a Boolean value to a set of tasks. $true$ means that the set of tasks is schedulable and $false$ means it is not. The second model ($M_2 = A_2(M_0)$) associates a worst-case response time to each task (**respTime** data). The response time is the time taken to complete a task in the worst-case scenario.

Any of M_1 and M_2 models can be a goal. In that case, it is referred to as G_1 or G_2. We consider G_1 in Figure V.4:

- A_1 (*feasibility test*) can be directly used to conclude about the schedulability of the tasks: if the Boolean-valued property **isSched** is true, the system is schedulable,
- A_2 (*response time analysis*) requires further interpretation of the resulting model M_2. An additional analysis A_3 inputs $M2$ together with an extra model $M3$ in order to compare the task responses times against the task deadlines: if the response-times **respTime** are lower than the deadlines **Dline**, the system is schedulable (**isSched** is true).

V.2.2 Contracts

A contract formally defines the interfaces of a model, an analysis or a goal in terms of data and properties.

Definition 16 (Contract)**.** *A contract, related to an element (i.e. a model, an analysis or a goal), is a tuple: K=(I,O,A,G):*

- *I are inputs: the data required by the element,*
- *O are outputs: the data provided by the element,*
- *A are assumptions: the properties required by the element,*
- *G are guarantees: the properties provided by the element.*

Notice that the 'data' directly refer to the accessors presented in Chapter III, whereas the 'properties' relate to the preconditions and postconditions introduced in Chapter IV. Hence, a contract is semantically equivalent to a Hoare triple as set out in Chapter IV.

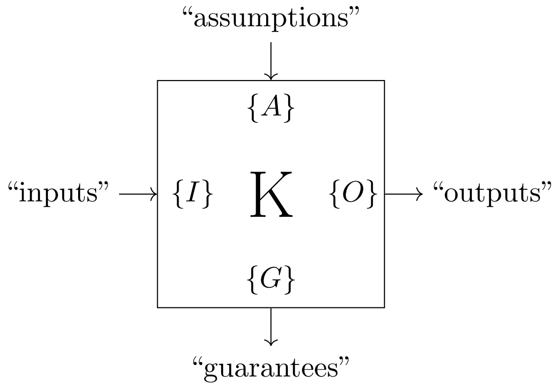


Figure V.5: Representation of a contract. *A contract formally defines the interfaces of a model, an analysis or a goal in terms of required and provided data and properties. It specifies the data through inputs and outputs, and properties via assumptions and guarantees.*

data	*designation*	*data type or structure*
TasksSet	tasks set	**TasksSet** $=\{$**Task**$\}$
Task	task	**Task** $=($**Per**, **Exec**$)$
Proc	processor	**Proc** $=($**Sched**$)$
Per	period	**Per** $\in \mathbb{N}$
Exec	(worst-case) execution time	**Exec** $\in \mathbb{N}$
Sched	scheduling policy	**Sched** $\in \{FP, RM, DM\}$

Table V.1: Data represented by the periodic tasks model M_0.

model	*data or property*	*designation*	*data type or structure or association*
M_1	**isSched**	schedulability of the tasks set	$result \in \{true, false\} \rightarrow$ **TasksSet**
M_2	**TasksSet**	tasks set	**TasksSet** $=\{$**Task**$\}$
	Task	task	**Task** $=($**respTime**$)$
	respTime	response time	**respTime** $\in \mathbb{R}$

Table V.2: Two models provided by schedulability analyses. *A* feasibility test *outputs the model* M_1. *A* response-time analysis *outputs the model* M_2.

property	*designation*	*association*
perTasks	all tasks are periodic	$result \in \{true, false\} \rightarrow$ **TasksSet**
fixedExec	all tasks have a fixed computation time	$result \in \{true, false\} \rightarrow$ **TasksSet**
fixedSched	the processor implements a fixed-priority scheduling policy	$result \in \{true, false\} \rightarrow$ **Proc**
...	other assumptions (see Section IV.2)	

Table V.3: Properties required to apply the Liu and Layland's *feasibility test*.

Notation convention. A number of notation conventions are used throughout this chapter:

- $K.I$, $K.O$, $K.A$ and $K.G$ denotes the various interfaces of a specific contract, that is its inputs, outputs, assumptions and guarantees respectively,
- we use the notation $K(x)$ to refer to the contract of an element x. We use an uppercase letter M, A or G instead of x to refer to a model, an analysis or a goal respectively,
- we use several indexes to point out an element of a set of contracts, models, analyses or goals, e.g. $K_{i\in\mathbb{N}}$, $M_{j\in\mathbb{N}}$, $A_{k\in\mathbb{N}}$, $G_{l\in\mathbb{N}}$.

Example. Let us define contracts for our simple example.

The *feasibility test* A_1 requires several data from the input model M_1 (see previous Table V.1). In addition, the test proposed by Liu and Layland relies on a set of assumptions as specified in Table V.3 (see Section IV.2 for the complete list of the assumptions).

contract	I	O	A	G
$K_1(M_0)$	$\emptyset$	**Per**, **Exec**, ...	$\emptyset$	$\emptyset$
$K_2(A_0)$	**Per**, **Exec**, ...	$\emptyset$	$\emptyset$	**perTasks**, **fixedExec**, ...
$K_3(A_1)$	**Per**, **Exec**, ...	**U**	**perTasks**, **fixedExec**, ...	**isSched**
$K_4(A_2)$	**Per**, **Exec**, ...	**respTime**	**perTasks**, **fixedExec**, ...	$\emptyset$
$K_5(M_3)$	$\emptyset$	**Dline**	$\emptyset$	$\emptyset$
$K_6(A_3)$	**respTime**, **Dline**	$\emptyset$	**perTasks**, **fixedExec**, ...	**isSched**
$K_7(G_1)$	$\emptyset$	$\emptyset$	**isSched**	$\emptyset$

Table V.4: Contracts for the various models, analyses and goals from Section V.2.1. *We must use an additional analysis A_0 to check Liu and Layland's assumptions.*

Under the Liu and Layland's assumptions, this analysis provides the processor utilization factor (**U** data) and a guarantee about the schedulability of the system (**isSched** property).

We hence define the contract for A_1 as follows: $K_3(A_1) = (I_3, O_3, A_3, G_3)$ with

$$I_3 = \{\mathbf{Per}, \mathbf{Exec}, \mathbf{Sched}, \ldots\},$$

$$O_3 = \{\mathbf{U}\},$$

$A_3 =$ "Liu and Layland's assumptions" $= \{\mathbf{perTasks}, \mathbf{fixedExec}, \mathbf{fixedSched}, \ldots\}$

and $G_3 = \{\mathbf{isSched}\}$.

Following the same method we are able to define the contracts for the various models, analyses and goals represented in Figure V.4. Table V.4 summarizes those contracts. Notice that we use an additional analysis (A_0 in the following) to check Liu and Layland's assumptions.

V.2.3 Contracts complementarity

We note that the interfaces (inputs and outputs, assumptions and guarantees) of two distinct contracts can be complementary. In that case, there is a precedence order between the underlying elements (models, analyses or goals).

Vertical precedence. A *vertical* precedence denotes a precedence between two elements with respect to properties computation (from assumptions to guarantees).

Property 1 (Vertical precedence (informal))**.** *There is a vertical precedence of an element X over a distinct element Y if and only if the guarantees of X and the assumptions of Y are complementary.*

Property 2 (Vertical precedence (formal))**.** *Let:*

- *$\mathcal{E}$ be a set of elements, with $(X, Y) \in \mathcal{E}$ distinct elements ($X \neq Y$),*
- *$K(X)$ and $K(Y)$ be the contracts of X and Y respectively.*

X vertically precedes Y, that is $next\_vertical(X, Y) = true$, iff $K(Y).A \cap K(X).G \neq \emptyset$.

Horizontal precedence. An *horizontal* precedence denotes a precedence between elements with respect to data computation (from outputs to inputs).

Property 3 (Horizontal precedence (informal))**.** *There is an horizontal precedence of an element X over a distinct element Y if and only if the outputs X and the input of Y are complementary,* and *there are elements K and N to satisfy the assumptions of X and Y respectively.*

Property 4 (Horizontal precedence (formal))**.** *Let:*

- *$\mathcal{E}$ be a set of elements, with $(K, N, X, Y) \in \mathcal{E}$ distinct elements ($K \neq N \neq X \neq Y$),*
- *$K(X)$ and $K(Y)$ be the contracts of X and Y respectively,*
- *$next\_vertical(K, X)$ and $next\_vertical(N, Y)$ be vertical precedences over elements of $\mathcal{E}$.*

X horizontally precedes Y, that is $next\_horizontal(X, Y) = true$, iff $K(Y).I \cap K(X).O \neq \emptyset$ and ($K(X).A = \emptyset$ or $next\_vertical(K, X) = true$) and ($K(Y).A = \emptyset$ or $next\_vertical(N, Y) = true$).

Example. A graphical representation of the precedences involving the contracts of Table V.4 is given in Figure V.9.

According to Properties 1 and 2, there are 5 cases of vertical precedence. For instance, between an analysis and another: $K_3.A \cap K_2.G = \{\mathbf{perTasks}, \mathbf{fixedExec}, \ldots\} \neq \emptyset \iff next\_vertical(A_0, A_1) = true$.

According to Properties 3 and 4, there are 5 cases of horizontal precedence. For instance, between a model and an analysis: $K_3.I \cap K_1.O = \{\mathbf{Per}, \mathbf{Exec}, \ldots\} \neq \emptyset \wedge K_1.A = \emptyset \wedge \; next\_vertical(A_0, A_1) = true \iff next\_horizontal(M_0, A_1) = true$.

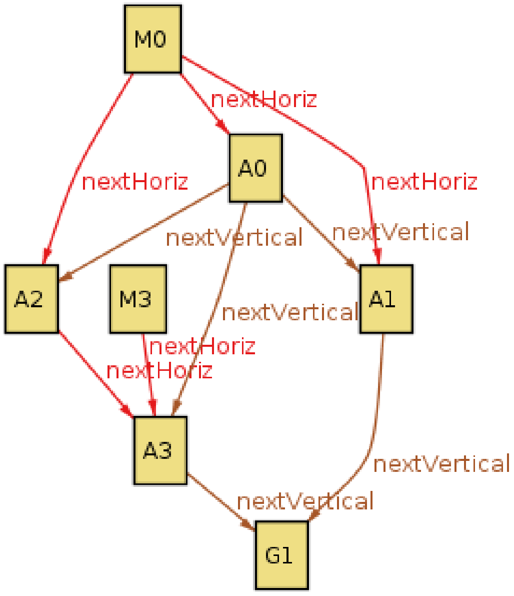


Figure V.6: Example of data flow between models, analyses and goals. *The complementarity between the contracts in Table V.4 bring out precedences between the models, analyses and goals. Horizontal precedences refer to data (computed from outputs to inputs) while vertical ones concern properties (computed from guarantees to assumptions).*

V.3 Contract-driven analysis

In this section, we explain how contracts can be used to systematize the analysis activities in a design workflow. We discuss the general approach first. We then present a proof-of-concept with Alloy.

V.3.1 Proposed approach

We propose the approach represented with a Process Flow Diagram in Figure V.7. Our approach relies on the detection of contracts complementarity to set up analysis paths in order to reach goals for any input model. The approach consists of 3 main steps.

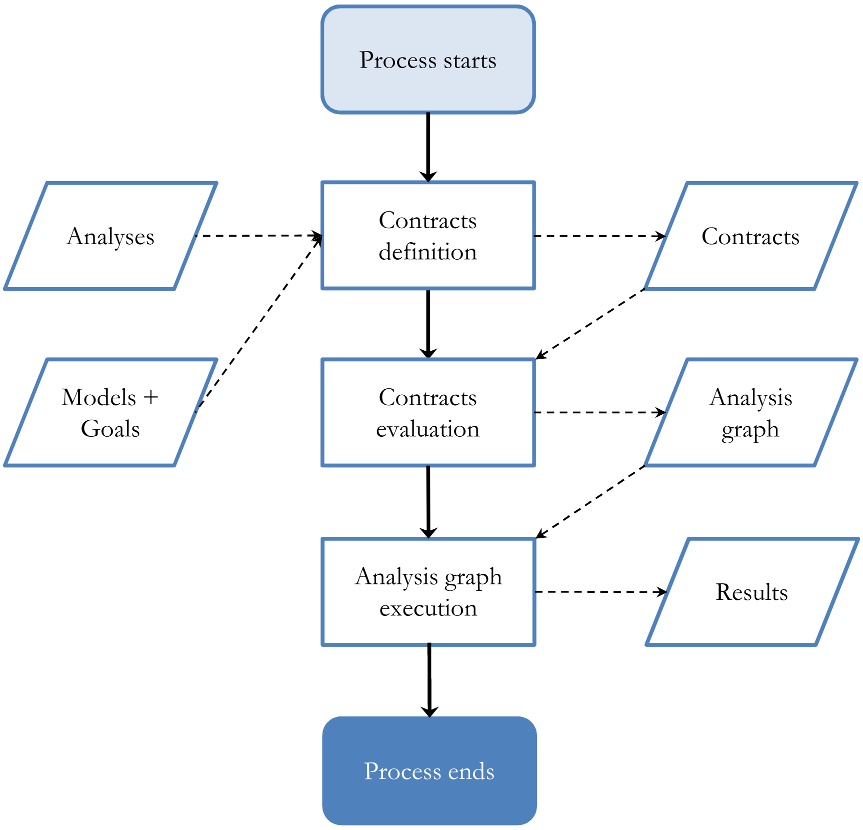


Figure V.7: Process Flowchart for contract-driven analysis. *The analysis flow is executed (step 3) according to the definition (step 1) and evaluation (step 2) of analysis contracts.*

(1) Definition of the contracts. We define contracts for the input configuration. A configuration consists of:

- a set of models,
- a set of analyses,
- a set of goals.

Contracts specify the interfaces of an element with FOL formulas from both the data (inputs/outputs) and properties (assumptions/guarantees) points of view.

(2) Evaluation of the contracts. Subsequently, we evaluate the contracts. During this step, we use (a) the contracts from step 1 together with (b) the rules describing under which conditions two contracts are complementary (Properties 1 to 4 in Subsection V.2.3). Then, we proceed as follows:

(i) given (a) and (b), we search the complementarity between the contracts,

(ii) if a complementarity between two contracts exists, we set the precedence between the underlying elements.

This is a problem about the satisfiability of contract formulas. A satisfiable interpretation of the contracts defines an analysis graph compliant with a model and a goal.

The implementation with Alloy discussed in the next subsection is optimal in the sense that it allows us to identify all the analysis paths to fulfill a goal according to an input model.

(3) Execution of the analyses. Finally, we proceed to the execution of the analysis graph. The analyses are executed with their tools according to the analysis graph resulting of step 2 to produce sound result(s) on the model.

Implementation of steps 1 and 2 is discussed in next Section V.3.2. Implementation of step 3 is discussed in Chapter VI.

V.3.2 Proof-of-concept with Alloy

As a proof-of-concept of the general approach represented in Figure V.7, we implement *contracts definition* (step 1) and their *evaluation* (step 2) with help of Alloy. Notice that the execution of the *analysis graph* (step 3) is not part of the Alloy problem. We discuss the execution part in the context of a more advanced prototype in Chapter VI (in particular, Section VI.2.4 deals with the visit of the analysis graph).

In this section, we firstly give a quick overview of Alloy. We then describe the toolchain used for proof-of-concept, mixing modeling and analysis tools together with the Alloy tool. We finally discuss an experimentation of the contract-based approach on several case studies.

V.3.2.A Alloy at a glance

Motivations. Our objective is to define contracts as precisely as possible to then provide a correct, exhaustive and time-efficient interpretation of these contracts.

We chose not to use a classic constraint language such as OCL or REAL for several reasons. As state by Vaziri and Jackson [168]:

1. constraint languages are not stand-alone languages: they need an accompanying model, e.g. OCL needs an UML, REAL focus on AADL models. In our case, contracts must be expressed independently of any design-oriented model,
2. constraint languages are not conceptual languages: they use low-level operations and complicated type systems, expressions are hard to read, etc. Consequently, they are hardly amenable to automatic and extensive analysis.

Instead, we choose Alloy [169], a language for expressing complex structural constraints completed with a tool for analyzing them. For our concern, it provides key advantages:

- Alloy is a formal language with abstract and analytical notations based on first-order logic that we use to specify contracts,
- Alloy provides a constraint solver to analyze and Alloy specification; we use the Alloy analyzer to evaluate contracts.

Alloy specification. Alloy is based on a specification that contains *signatures*. Signatures may have *fields* to define relationships with other signatures. In addition, *facts* express constraints on the signatures and fields.

We define contracts with Alloy in two parts:

- a basic *signature* specifies the structure of a contract: fields are not only used to represent the contract interfaces (`inputs`, `outputs`, `assumptions` and `guarantees`) but also dependencies with other contracts (`nextHoriz` and `nextVertical`). Listing V.1 depicts the contract structure in Alloy syntax,
- signature *facts* specify the specific constraints about the contract instances. Listing V.2 represents a contract instance of a feasibility test called `DC_FPP_RTA` with the specification of `inputs`, `outputs`, `assumptions` and `guarantees`.

The Alloy specification is completed in Listing V.3 with `VerticalPrecedence` and `HorizontalPrecedence` *facts*. They define the logical conditions under which the `nextHoriz` and `nextVertical` relationships hold between two contracts.

Alloy analysis. The Alloy analyzer provides full and automatic analysis of an Alloy specification. The Alloy analyzer is a 'model finder': it searches a model that satisfies the logical formula generated from the Alloy specification. If there is a solution that makes the formula true, Alloy will find it. Alloy provides several SAT solvers for this purpose.

```
/*Basic signatures manipulated in Alloy specification*/

/*Definition of Data and Properties signatures*/
abstract sig Data {}
abstract sig Property {}

/*Definition of the structure of a contract*/
abstract sig Contract{
   //interfaces
   input: set Data,                       //required-provided data
   output:set Data,
   assumption: set Property,              //required-provided properties
   guarantee: set Property,
   // relationships with other contracts
   nextHoriz:set Contract,                // output->input
   nextVertical:set Contract              // guarantee->assumption
}
```

Listing V.1: Basic signatures of the Alloy specification. *Signatures in Alloy describe the entities to reason about. Here, the contract signature specifies the structure of a contract: fields are not only used to represent the contract interfaces (`input`, `output`, `assumption` and `guarantee`) but also dependencies with other contracts (`nextHoriz` and `nextVertical`).*

```
/* Data structures in a AADL model */
abstract sig Component extends Data {
   subcomponents: set Component,
   type: lone ID,
   properties: set ID
}

/* An analysis contract on such structure*/
one sig DC_FPP_RTA extends Contract{
}{
   //specification of input data structure
   input={S:Component |
               S.type=system and (
                  some sub:S.subcomponents | sub.type =processor and (scheduling_protocol+preemptive_scheduler) in sub.properties) and (
                  some sub:S.subcomponents | sub.type=process and
                     thread in sub.subcomponents.type and
                     ( let th=sub.subcomponents & thread.~type |
                           (dispatch_protocol +period + compute_execution_time +priority +deadline) in th.properties and
                           (not (offset) in th.properties)
                     )
                  )
            }
   //specification of output data structure
   //assumptions and guarantees
   [...]
}
```

Listing V.2: Specification of an analysis contract. *`Input/output` fields are defined with respect to the `Component` data structure used for AADL modeling. Here, the analysis expects a precise hierarchy of components which consists of a system with processors and threads; with properties attached to the components, e.g. `periods` are required, `offset` are not required.*

```
/* Predicate specifying contracts inter-dependencies */

//between inputs/outputs
fact HorizontalPrecedence{
  all c_current:Contract |
   c_current.nextHoriz={c_next:Contract|
      (c_current.output & c_next.input != none) and
      (all a :c_current.assumption| a in Contract.guarantee) and
      (all a :c_next.assumption| a in Contract.guarantee)
}

//between assumptions/guarantees
fact VerticalPrecedence{
    all c_current:Contract |
      c_current.nextVertical={c_next:Contract|
         (c_current.guarantee & c_next.assumption != none)
   }
```

Listing V.3: Additional constraints on signatures and fields expressed with facts. *Here, the inter-dependencies between inputs/outputs and assumptions/guarantees fields of contracts are defined by `HorizontalPrecedence` and `VerticalPrecedence` facts respectively.*

Given contracts in an Alloy specification, the analyzer finds dependencies between models, analyses and goals. The solution visualized from Alloy in Figure V.9 represents dependencies under the form of a graph. Here, the graph exhibits the analysis paths that should be executed to conclude about the schedulability of a satellite system modeled with AADL. We experiment Alloy more exhaustively in Section V.3.2.V.3.2.C.

V.3.2.B Toolchain

We propose a toolchain to model and analyze real-time systems (see Figure V.8):

- Modeling: the system architecture is represented with AADL [170, 134],
- Analysis:
  - MAST [9] and Cheddar [8] tools provide several analyses to assess real-time workloads,
  - we use RTaW-Pegase [121] and RTaW-Sim [171] tools to calculate networks traversal times. RTaW-Pegase focuses on network calculus to compute communication delays in Rate-Constrained networks (e.g. AFDX networks). RTaW-Sim provides a set of analyses for the performance evaluation of CAN networks,
  - we can define user-specific analyses (e.g. precondition checks) with help of REAL or Python (see section IV.3),
- Orchestration: we use Alloy to both define the contracts and evaluate them.

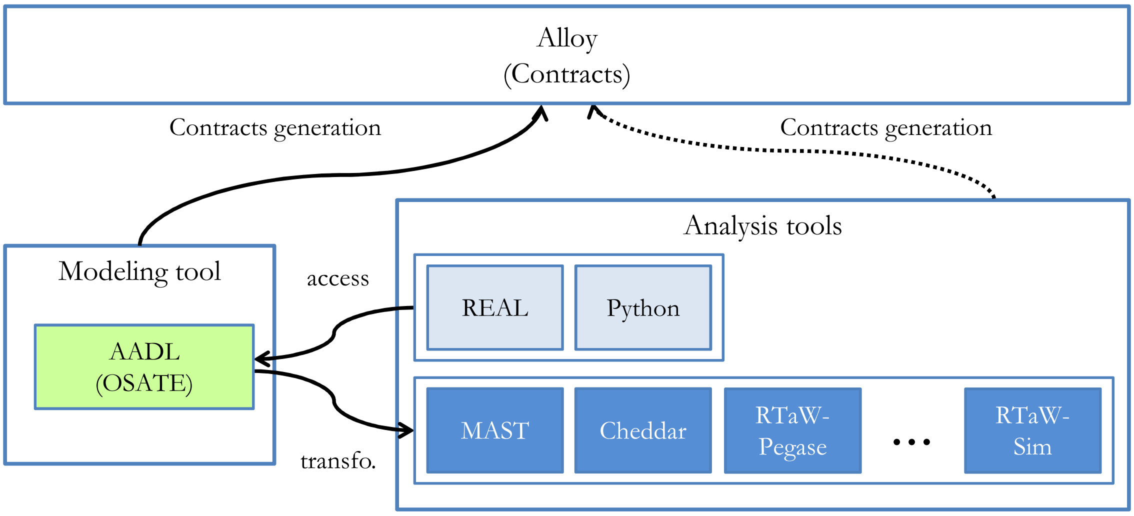


Figure V.8: Proposed toolchain for the proof-of-concept. *The toolchain mixes modeling and analysis tools together with Alloy. An AADL model represents the system. Several analysis tools enable to assess real-time workloads at tasks and networks levels. We use Alloy to both define the contracts and evaluate them. Solid arrows represent currently implemented bridges between tools.*

Backends. REAL is natively supported by the OSATE/OCARINA plugin. Python-based analyses rely on the query mechanisms introduced in Section IV.3. Transformations from AADL models to tool-specific models and contracts are partly supported by the OCARINA tool [88]. Currently implemented bridges are represented with solid arrows in Figure V.8.

V.3.2.C Experimentation and lessons learned

We evaluated the strengths and shortcomings of an implementation with Alloy. We experimented the orchestration of real-time scheduling analysis for various AADL models.

Models. We consider 5 models[2]:

- M_1 : a multitasked real-time system implementing the *ravenscar profile* [173]. Several tasks access a shared resource in an asynchronous way according to a priority inheritance protocol,
- M_2 : a simple *distributed real-time system*. The system is made up of 2 calculators to execute tasks. We consider a Fixed-Priority Preemptive (FPP) policy to schedule the tasks. An *Avionics Full Duplex-Switched Ethernet* (AFDX) network supports inter-calculators communications,
- M_3 : the *mars pathfinder system* [125]. The system consists of a stationary lander and a micro-rover. Each sub-system schedules the tasks following the

[2] the models are part of the AADLib project accessible online [172]

Rate-Monotonic Scheduling (RMS) algorithm. CAN buses support communications,

- M_4 : a simplified *satellite system*. We represent the software part together with the platform part of an on-board satellite system. Functional chains involves task scheduling according to a Fixed-Priority Preemptive (FPP) algorithm and 1553B-based communications,
- M_5 : a *Flight Management System* (FMS) [174, 175]. We consider a subpart of the FMS that consists of five functions to execute in 4 calculators according to the ARINC653 standard. CAN buses and AFDX virtual links support inter-calculators communications,
- we finally consider an "all-in-one" model $M_6 = M_1 \cup M_2 \cup M_3 \cup M_4 \cup M_5$.

The AADL models represent systems of different complexity. Table V.5 depicts some metrics related to the complexity of the AADL models: number of lines of code, number of components, number of properties, and average number of properties by component. We propose an additional metrics $\mathcal{O}_{AADL}$ mixing the number of components and the number of properties present in the AADL model:

$$\mathcal{O}_{AADL}(M_n) = \frac{NOC(M_n) \times NOP(M_n)}{NOC(M_5) \times NOP(M_5)} \tag{V.1}$$

In Table V.5, apart from M_6, the model of the FMS is the most complex according to $\mathcal{O}_{AADL}$: $\mathcal{O}_{AADL}(M_5) = 1$. The AADL model using the ravenscar profile is the least complex: $\mathcal{O}_{AADL}(M_5) \approx 16 \times \mathcal{O}_{AADL}(M_1)$. The "all-in-one" model M_6 is obviously more complex than the model of the FMS as $\mathcal{O}_{AADL}(M_6) = 9 \times \mathcal{O}_{AADL}(M_5)$.

AADL model	*LOC*	*NOC*	*NOP*	$\frac{NOP}{NOC}$	$\mathcal{O}_{AADL}$
M_1	148	7	39	5,57	0,06
M_2	337	20	57	2,85	0,25
M_3	395	24	51	2,125	0,27
M_4	464	27	85	3,148	0,5
M_5	753	47	97	2,064	1
M_6	2097	125	329	2,632	9,02

Table V.5: Several metrics representing the complexity of the AADL models. *We consider the number of number of lines of code (LOC), the number of components (NOC), the number of properties (NOP) and the average number of properties defined per component (NOP/NOC). We propose an additional metrics $\mathcal{O}_{AADL}$ mixing the number of components and the number of properties present in the AADL model. The models are ordered by ascending complexity following $\mathcal{O}_{AADL}$.*

Analyses. The toolchain (see Figure V.8) provides 14 analyses in total. MAST, Cheddar, RTaW-Pegase and RTaW-Sim tools implement 7 analyses to assess tasks schedulability and networks traversal times. In addition, 4 analyses use REAL or Python to check analysis preconditions. We finally consider 3 analyses also based on REAL or Python to compare response times and traversal times against their deadlines.

Goals. We focus on a single goal which is to conclude about the schedulability of the system, that is schedulability at task and network levels.

Analysis graph. The Alloy analyzer found a solution satisfying the Alloy specification for each AADL model.

Figure V.9 represents the analysis graph visualized from Alloy for the satellite case study:

- the Alloy analyzer finds the analyses which are directly applicable on the AADL model (6 analyses connected to the `aadl_model` vertex),
- it also finds all the dependencies between analyses (15 dependencies represented by edges between analyses),
- it finally identifies the analyses that that are needed to reach the goal (4 analyses connected to the `is_schedulable` vertex).

We can then use the graph found by Alloy to execute the analyses: here, there are 4 complete paths to execute (from `aadl_model` down to `is_schedulable`).

Contracts processing times. Let us now focus on the time taken by the Alloy analyzer to find the analysis graph. We call *contracts processing time* (CPT) the time taken by Alloy to analyze the contracts together with the precedence constraints in the Alloy specification, and find the solutions that satisfy the specification. The CPT encompasses two dimensions: (1) the *generation time* (GT) of the formula to be solved and (2) the *resolution time* (RT) of the formula. This is simply summarized by:

$$CPT = GT + RT \tag{V.2}$$

Figure V.10 outlines the Contracts Processing Times (CPT) experienced for the various case studies.

Firstly notice that the generation times (GT) increase exponentially with the complexity of the AADL models ($\mathcal{O}_{AADL}$). The best case ($GT = 639ms$) corresponds to the *ravenscar profile* model (M_1). The worst case is experienced with the model of the *flight management system* (M_1) with $GT = 121159ms$ ($\approx 2min$). In that case where we handle all the models at once (M_6), the generation time is multiplied by 20 ($GT \approx 40min$) compared to the case involving the FMS only. A better strategy is to break such a wide resolution space in smaller affordable pieces, interpret them separately and then aggregate the results. For instance, we are able to reduce the processing time of M_6 from 40 minutes to less than 3 minutes by simply handling the input models independently and subsequently.

We secondly observe that, for all the case studies, almost entire part of the contracts processing times (CPT) is devoted to the generation of the formula to be solved (GT). The resolution time itself (RT) never exceeds 1 second ($RT = 856ms$ being the worst-case experienced).

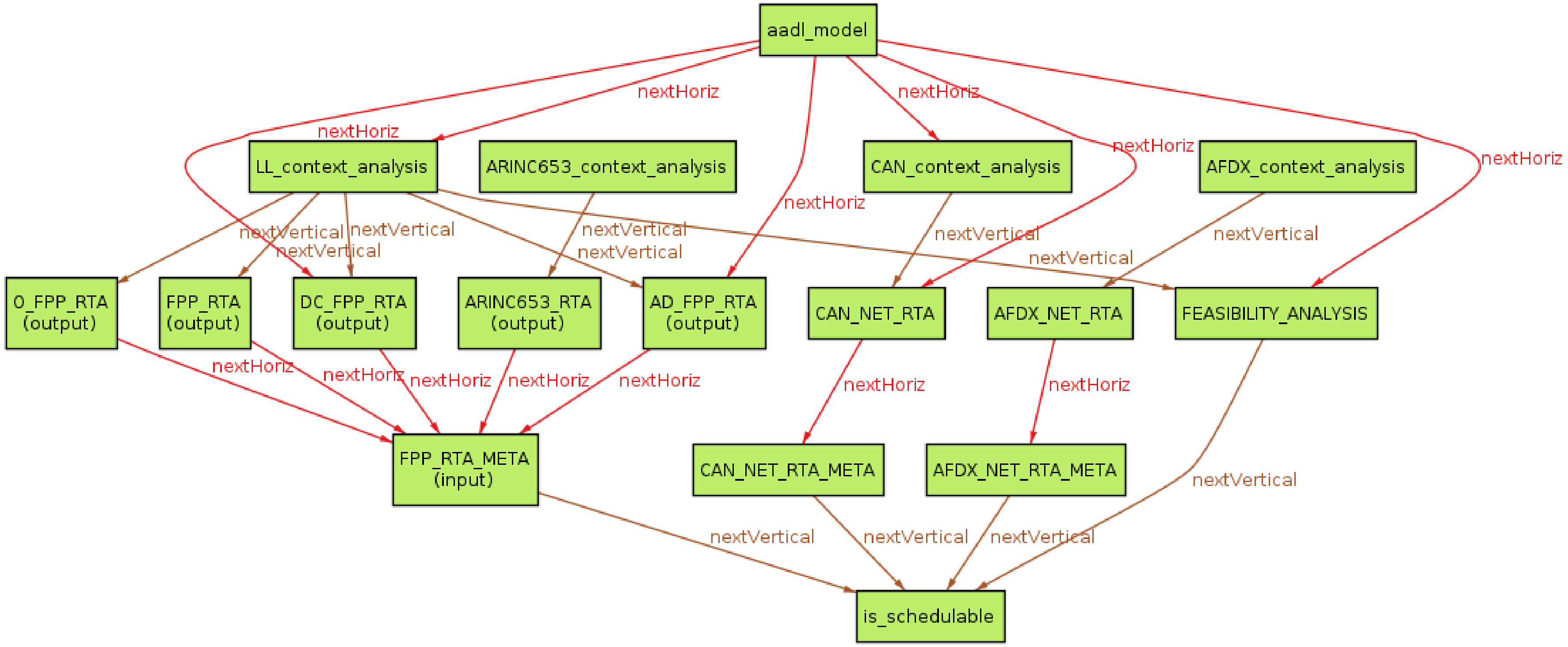


Figure V.9: Visualization of a solution found by the Alloy analyzer for contracts specified in Alloy (satellite system case study). *Here, the graph represents data dependencies and precedences between the models, analyses and goals. We then execute the analyses from this graph to conclude about the schedulability of a satellite system modeled with AADL.*

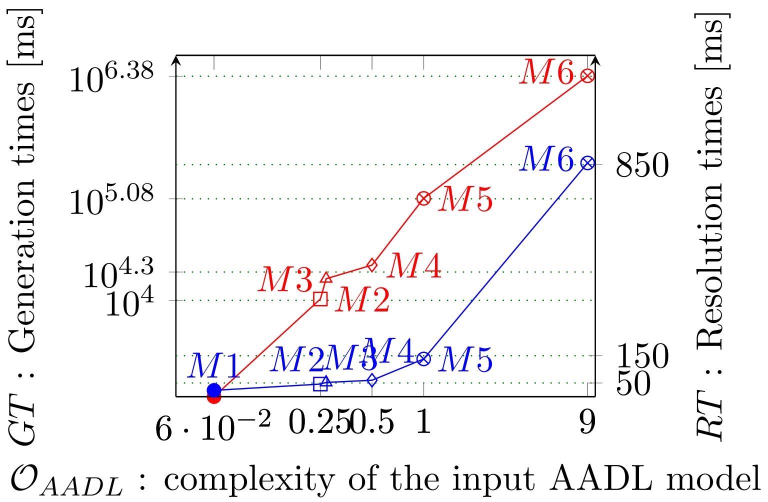


Figure V.10: Contracts processing time $CPT = GT + RT$ dependence of the input model complexity $\mathcal{O}_{AADL}$. *The generation times (GT) of the Boolean formula increases exponentially with the complexity of the AADL models ($\mathcal{O}_{AADL}$). The resolution times (RT) evolves to a lesser extent as it never exceeds 1 second ($RT = 856ms$ is the worst-case experienced).*

Lessons learned. We showed that our approach is applicable on sets of models, analyses and goals of realistic complexity in an affordable time.

Despite of the important resolution spaces to handle, the Alloy analyzer is able to find solutions in a reasonable time (the worst processing time is about 2 minutes).

We secondly experienced the scalability of our approach: we applied our approach on a configuration including all the models together (which represents 5 models, 125 components and 329 properties). Notwithstanding that this strategy is poorly efficient (the processing time increases exponentially to 40 minutes), we are able to find all the solutions. A better strategy to avoid huge processing times is to break wide resolution spaces into smaller pieces, compute them subsequently and finally aggregate the results. That way we reduce the processing time from 40 minutes to less than 3 minutes.

A major benefit of an implementation of the contract-based approach with Alloy is that if any solution exists for the specification, the analyzer will *always* find it. Furthermore, the Alloy analyzer is able to find *all* the solutions. As disadvantages, the use of Alloy requires a minimal expertise to define the contracts and, possibly, adjust manually the resolution scope of the SAT solver.

V.4 Discussion

The notion of *contract* is the keystone of the approach presented in this chapter: it formally captures analysis features and enables to reason about them. Contracts can then be used in various settings to systematize the analysis activities in a design workflow. We discuss related works on contracts and sketch possible future improvements around this notion.

V.4.1 Related works

Background on contracts. Contracts have been formerly introduced and used in various contexts.

Assume-guarantee and contracts reasoning have their roots in Floyd-Hoare logic [176, 177]. Contracts explicitly handle pairs of properties called *assumptions* and *guarantees*: assumptions represent the properties expected by a given system on its environment while guarantees summarize the properties provided by the system under these assumptions. Intuitively, a contract says: (1) under which context the system operates and (2) what its obligations are. A contract can refer to any kind of system but usually allude to software or systems components in *contract-based design* [178].

A well-known application of contracts is *design-by-contract*, an approach to design software popularized by Meyer [179]. More recently, contracts have been investigated for the design of Cyber-Physical Systems [180, 181]. A more exhaustive description of general contracts together with a meta-theory is discussed in [182].

Related works. To the best of our knowledge, few works focus on contracts to deal with analysis problems.

We can cite works from Ruchkin et al. [183, 184, 185] that focus on a problem close to ours. Ruchkin et al. [183] deal with integration of Cyber-Physical Systems analysis in the context of the OSATE/AADL tool environment. They acknowledge that properties of AADL models can be computed by tools coming from different scientific domains (e.g. schedulability, power consumption, safety or security). They hence use the contract formalism to capture the semantics of analysis domains and avoid the execution of conflicting tools (invalidation of properties computed by a tool with one another). This is made possible with a language to specify contracts (being part of AADL) and a verification algorithm (based on SMT solving) to find inter-dependencies between contracts. They detail implementation of this approach through the ACTIVE tool in [184].

Although we share a root formalism, we develop and investigate it in quite distinct contexts. Firstly, the works of Ruchkin et al. take place in the development of the OSATE tool. Contracts are thereby intrinsically bound to AADL in their development. For instance: Ruchkin et al. [183] define the contracts in terms of the AADL type system (AADL property sets, AADL components such as `threads` and `processors`) through a sub-language annex; the ACTIVE tool presented in [184] is developed within the OSATE/AADL infrastructure; analyses as part of OSATE relies on more traditional *ad hoc* model transformations. For our part, we define an holistic approach based on (i) analysis data structures and accessors to query them on any type of model (be it designed with AADL, CPAL, SysML or another language) in Chapter III, (ii) the definition of the semantics of an analysis in Chapter IV, (iii) analysis contracts as an extension of (ii) to make the analysis systematic in a design workflow.

Let us note secondly that Ruchkin et al. focus on the interaction between analyses coming from heterogeneous domains (e.g. schedulability, power consumption, safety or security). This problem is here again strongly linked with the AADL/OSATE tool platform that integrates analysis plugins from multiple domains. They thus use contracts to prevent the incorrect order of analyses where the result of one analysis is invalidated by the result of another analysis executed afterwards. In our view, contracts are neither relevant to analysis domains only (but also to intra-domain analysis) nor to be considered from a "destructive" point of view (but should be rather handled in a "constructive" way). In this thesis for instance, we use contracts to handle various kind of analyses coming from the real-time scheduling domain only (we consider for instance execution times analysis, schedulability analysis, response times analysis, or traversal times analysis). We have shown in addition that data dependencies between analyses could be used to (1) build wider analyses and (2) compute expected results (goals). We believe that contracts can be extended to cover more advanced use cases (see the next discussion about possible improvements).

V.4.2 Improvements

Advanced use cases. A first improvement is to enrich contracts with metrics: e.g. complexity, rapidity of an analysis execution, precision of a result. This will enable to deal with more advanced use cases through additional reasoning capabilities. For instance:

- to handle the analysis dynamics more precisely: coarse-grained but fast analyses such as *feasibility tests* can be used during the early design stages, e.g. for prototyping; in-depth and costly analyses such as *model-checking* are more relevant at the last stages of the design process (before the implementation phase), when early results should be consolidated,
- to enable more advanced design space exploration and/or optimization [186, 187]. In this case, numerous design strategies could be proposed on the base of heuristics mixing model states, analysis properties and multiple goals expressed in terms of non-functional requirements.

Let us note that the evaluation of the metrics adds little algorithmic complexity and can be quite easily integrated to our approach, e.g. by looking for the shortest analysis paths on a weighted analysis graph. Yet, investigation of design strategies and heuristics is a problem on its own that will require fully dedicated researches (see works by Gilles [158] and Cadoret [187] for instance).

Contract language. The proof-of-concept presented in this chapter is based on Alloy. We motivated our use of Alloy through several key arguments, mainly: Alloy is standalone high-level language with powerful analysis features.

We already reported some limitations from experimentation of Alloy. In particular, it is necessary to modify manually the Alloy specification in some contexts, e.g. to define manually the contracts for analyses and goals, or to adjust the resolution scope. Moreover, we note that the grammar of Alloy does not enable a neophyte to deal with contracts in a straightforward way.

A perspective is hence to define a domain-specific language that captures well the concept of contract and allows for automatic processing. Additional investigations will enable to move forward this topic and find the most efficient implementation of the contract-based approach. For our concern, we present our prototype including Alloy in Chapter VI.

V.5 Synthesis and conclusion

Analysis, as a set of model assessment activities, takes an active part in the construction of a system, be it to validate a specific property or compute new data that could be added to the model.

In this chapter we presented an approach to systematize the analysis activities in a design workflow. We define the interfaces of a model, an analysis or goal through generic contracts, semantically equivalent to a Hoare triple as set out in Chapter IV. We then use SAT methods to reason about the data structures and properties defined in contracts. In particular, we are able to find: (1) the analyses that are applicable on a model; (2) the analyses that meet a given goal; (3) the data dependencies that bring out analysis combinations. In the proof-of-concept, we used Alloy to support both the contracts definition and their evaluation. We can use the analysis graph thereby obtained to execute the analyses in a systematic manner. A typical case study is to combine heterogeneous real-time analyses to assess the schedulability of a system including tasks and networks.

Defined through contracts in close relation with system models and engineering goals, analyses are no longer considered apart from the design process but become first-class citizens in the design workflow. We present a more advanced prototype involving contracts in the next Chapter VI.

Part 2

Application

Chapter VI

Tool prototype

Abstract

In this chapter, we present a tool prototype that implements the various concepts introduced in the first part of this thesis. The tool implements various functions so as to automatically handle the analysis process when designing an embedded system. We firstly present the modular architecture of the tool (Section VI.1). In particular, we introduce the basic functions of the tool and provide an object-oriented design of the software. We implemented the first version of the prototype with a set of scripts written in Python that we run on top of modeling tools – OSATE, CPAL-Editor – and, possibly, external analysis tools – TkRTS, MAST, Cheddar, etc. Section VI.2 deals with the key elements of implementation. We present the workflow supported by the tool in Section VI.3. Section VI.4 finally concludes this chapter.
The tool prototype presented in this chapter will allow us to experiment a type of design process that systematically combine architectural models and analyses. We further explore the case studies in the next Chapter VII.

VI.1 Tool architecture

We firstly describe the general architecture of the tool and the basic functions. We then present the object-oriented architecture that we implemented in Python.

VI.1.1 General architecture and basic functions

The tool is made up of 4 modules-functions (or layers) as represented in Figure VI.1. Tool modules shown in colors interface with external resources shown in light gray. We presented the foundations of each layer in the first part of the thesis. The next paragraphs present the modules in a few words. We sketch the object-oriented architecture of the software in Section VI.1.2 and introduce some key elements of implementation in Section VI.2.

Models enable to fully or partly represent an embedded system. We use Domain-Specific Languages such as AADL or CPAL (see Section II.2.4) for this purpose.

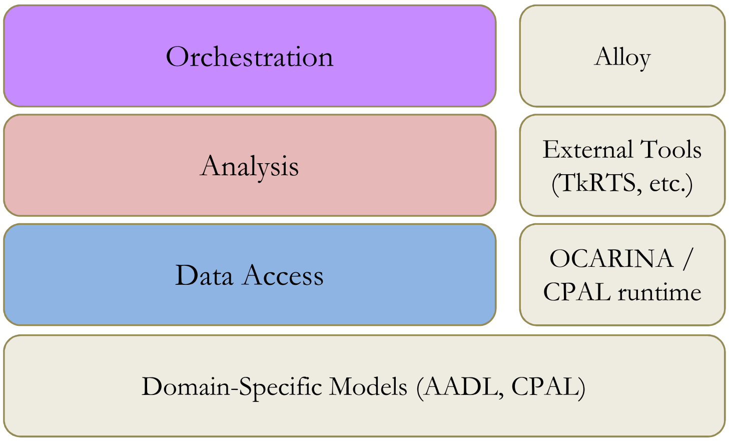

Orchestration
Alloy
Analysis
External Tools (TkRTS, etc.)
Data Access
OCARINA / CPAL runtime
Domain-Specific Models (AADL, CPAL)

Figure VI.1: Modular and layered tool architecture. *The tool involves the solutions presented in the previous chapters within separate modules in colors. The modules from modeling to orchestration are organized in layers. We interface these modules with external tools in light gray.*

Analysis. This module makes it possible to analyze some properties of the system from (one of) its model(s). This module provides domain-specific analysis such as real-time scheduling analyses (feasibility of a task set, computation of communication delays in embedded networks for example), dependability, etc. This module provides in addition some specific analyses to verify the preconditions of above-mentioned analyses (see Chapter IV). It is possible to outsource the analysis to third-party tools (e.g. bridges exist towards REAL, TkRTS, Cheddar, MAST, etc. through OCARINA).

Accessors. The interaction between models and analyses is managed by means of accessors (see Chapter III).

From an analysis perspective, accessors consist of programming interfaces to be used in an analysis program, that is getters and setters to the data model. The data model hold data about the system from one of its representations (e.g. in AADL or CPAL). It relies on standard data structures. For example, real-time tasks, processors, shared resources and scheduling algorithms are some data structures required to analyze real-time workloads.

Accessors to model internals must then be implemented. These are implemented in three parts: (1) providing access to the data model at the topmost level; (2) providing access to the domain-specific models in possibly distinct technical spaces (mapping for example the data model to AADL and CPAL models); (3) possibly, combining the accessors to build wider accessors. We use functionalities provided by dedicated tools to interface with the domain-specific models. We use for example OCARINA to parse AADL models, or the `cpal2x` tool to extract data from CPAL source files.

Orchestration. The orchestration module directs the analysis process according to the input model(s), the repository of analysis, and the analysis goal(s).

The orchestration relies on the concept of contract to firstly represent and then evaluate the interfaces of an analysis (see Chapter V). We use SAT resolution methods to find the interdependencies between analyses. Alloy is used to that end. The orchestration module finally visits the graph, executing the analysis path that both fulfill input models and goals.

VI.1.2 Object-oriented design

We developed the tool with the Python language. We implement the basic functions in Figure VI.1 with classes in Python. Figure VI.2 shows the architecture of the tool as a class diagram. The diagram represents the basic functions-classes as well as the relations between them. From top to down:

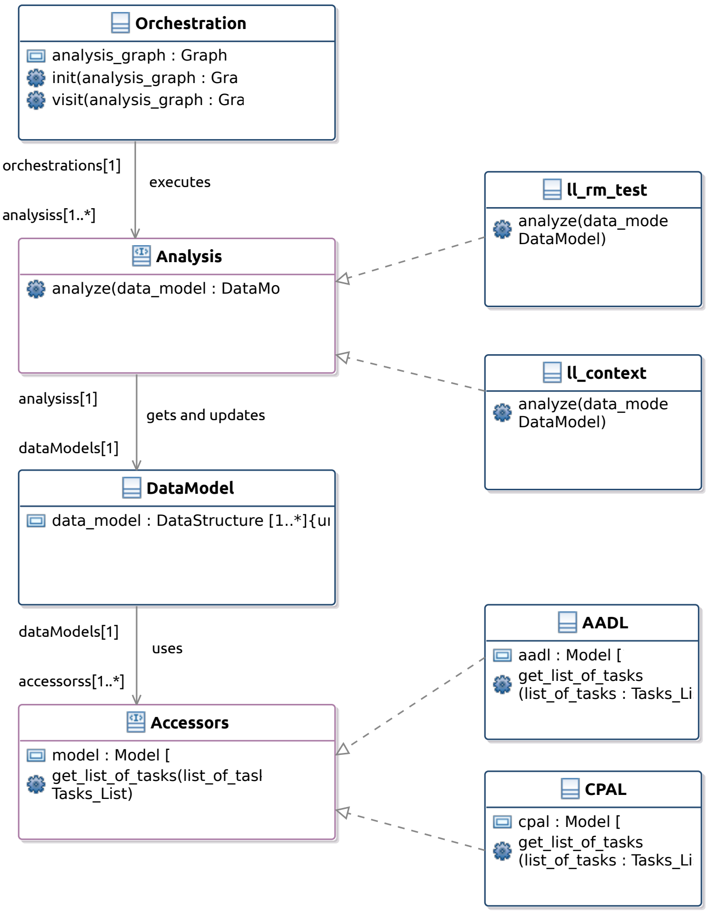


Figure VI.2: Object-oriented tool architecture. *We implement the various modules with classes in Python.*

The `orchestration` class relies on an analysis graph. It provides two methods to use the graph: (1) a method to initialize the graph; (2) a method to visit the graph (analyses are executed when the graph is visited).

An `analysis` is an interface. It makes possible the analysis of a data model via a specific method `analyze(data_model Data_Model)`. The `ll_rm_test` is a specific implementation of this interface that analyzes a task set with the help of the test of Liu and Layland, and determines whether the task set is feasible or not. The `ll_context` is another analysis that checks if the test assumptions defined by Liu and Layland hold or not.

The data model (i.e. `DataModel`) is made up of a set of data structures (i.e. `DataStructure`) to organize the data about the system. The aforementioned `ll_rm_test` uses data structures such as real-time tasks, processors, scheduling algorithms, etc. On the one hand, the data model provides methods to `get` and `update` the data structures, i.e. the high-level accessors. On the other hand, the data model uses low-level accessors to the domain-specific models.

The `accessor` interfaces define the methods to implement in order to retrieve data from a domain-specific model, e.g. `get_list_of_tasks`. An implementation of this interface is specific to a technical space. For example, the class `AADL_accessor` implements the method `get_list_of_tasks` for the AADL technical space with the help of the Python/OCARINA API. The class `CPAL_accessor` implements the same method working on top of CPAL models by using the `cpal2x` tool.

Interaction between the modules. The sequence diagram in Figure VI.3 represents a typical execution of the tool:

- the orchestration module directs the analysis process. The `init()` method compute the analysis graph first. The orchestration module then visit the analysis graph with the `visit()` method, and execute the analyses with the `analysis()` method. As defined by the analysis graph, we execute the `ll_context` analysis, that checks a set of preconditions, before the `ll_rm_test`,

- the analyses `ll_context` and `ll_rm_test` compute result from input data. These analyses firstly use the `get` method to retrieve input data from the data model. The analyses finally update the data model (i.e. `update` method) with the computed result,

- the data model use accessors to domain-specific models when necessary, for example the `get_list_of_tasks` method retrieves data about real-time tasks from a AADL or CPAL model.

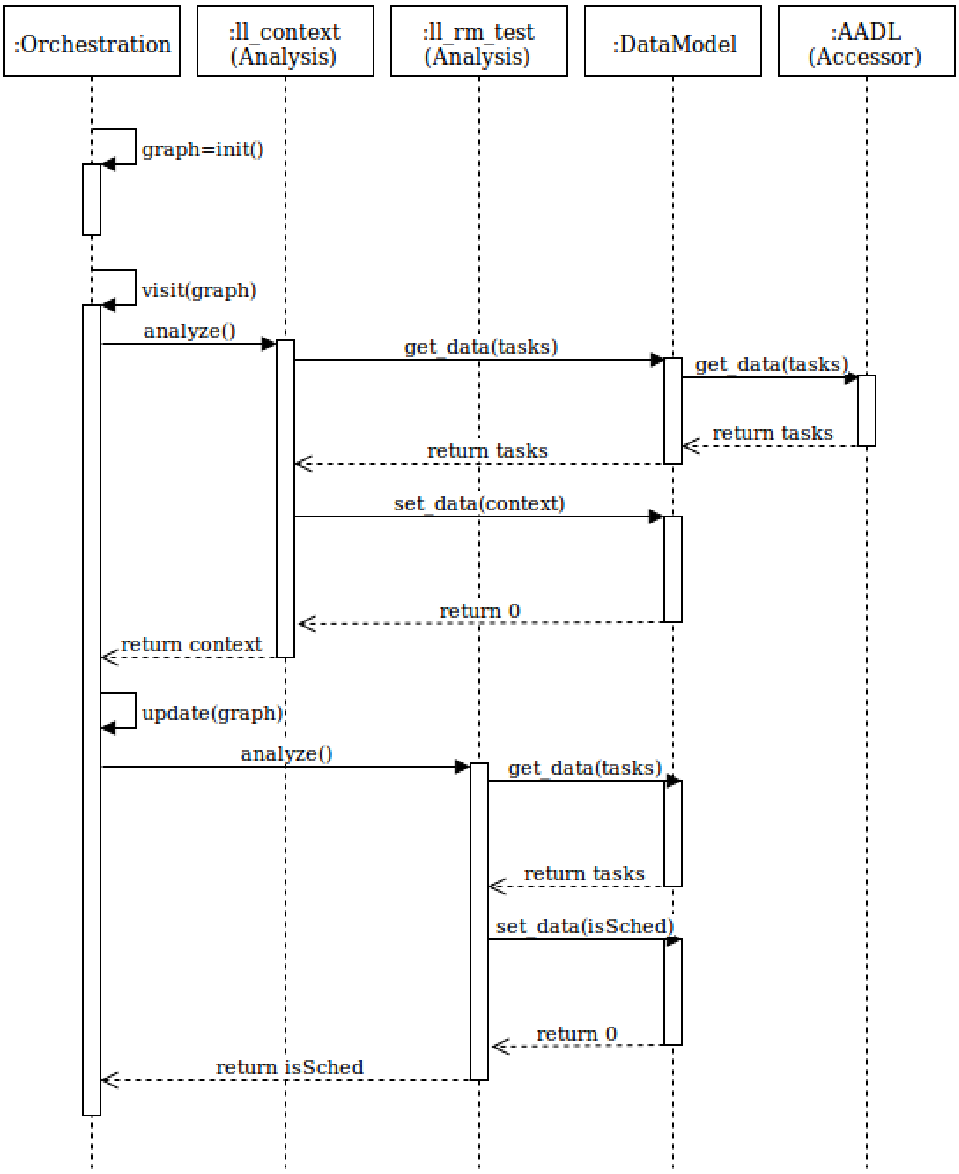


Figure VI.3: Sequence diagram representing a typical tool execution. *The diagram represents the timeline of each object (i.e. module), the various functions executed within each timeline and the interactions between the objects.*

VI.2 Key elements of implementation

In this section, we present some key elements of implementation. We implement the software architecture described in the previous section. We used the Python programming language to develop a first prototype of the tool.

VI.2.1 Data model and data structure

The data model consists of a collection of data structure instances and methods to access them. Listing VI.1 illustrates the principles of implementation of the data model and its use in a Python program.

```

""" A simple script to create and use a data model
"""

# declaration of a ’Task’ data structure via a class
class Task:

   def __init__(self, name, period, best_case_execution_time,
       worst_case_execution_time, deadline, offset):
   """ This function initializes the class
       Arguments: task properties
   """
   self.name=name
   self.period=period
   self.best_case_execution_time=best_case_execution_time
   self.worst_case_execution_time=worst_case_execution_time
   self.deadline=deadline
   self.offset=offset

# declaration of several objects using that class

# some tasks
T1=Task("A task", 10, 2, 3, 10, 0)
T2=Task("Another task", 5, 1, 3, 5, 0)
T3=Task("A third task", 20, 1, 1, 20, 0)

# a list of tasks with previous objects
list_of_tasks=[T1, T2, T3]

# a a graph of dependencies between tasks
dependency_graph=dependency_graph={T1: [T2],T2: [T1],T3: []}

# declaration of the data model and update with previous objects
data_model={}
data_model.update({"LIST_OF_TASKS": list_of_tasks})
data_model.update({"TASKS_DEPENDENCIES": dependency_graph})
```

Listing VI.1: Implementation and use of a simplified data model in a Python program.

In this simplified example, we firstly declare a data structure that represents a task with the help of a class. We can then instantiate several tasks with their respective properties, i.e. `T1`, `T2` and `T3`. We can also use the task data structures to build more complex data structures: a list of tasks and a graph of task dependencies.

Last, we can `update` the data model with these various objects. These objects can be used later by any analysis through the reverse method `get`.

Our prototype implements a little more sophisticated data model than the one sketched in Listing VI.1. In particular, a comprehensive data model must bind the high-level `get` and `update` methods to low-level accessors so as to retrieve data represented in domain-specific models. Figure VI.4 (copied from Chapter III) describes the extended procedure to get a data structure from the data model. This procedure executes an alternative thread in the event that the required data structure is not yet present in the data model: the sub-procedure `Get Data Structure from Design Model` builds a data structure from its counterpart in a domain-specific model. We discuss a more detailed implementation of this data sub-procedure in the next subsection.

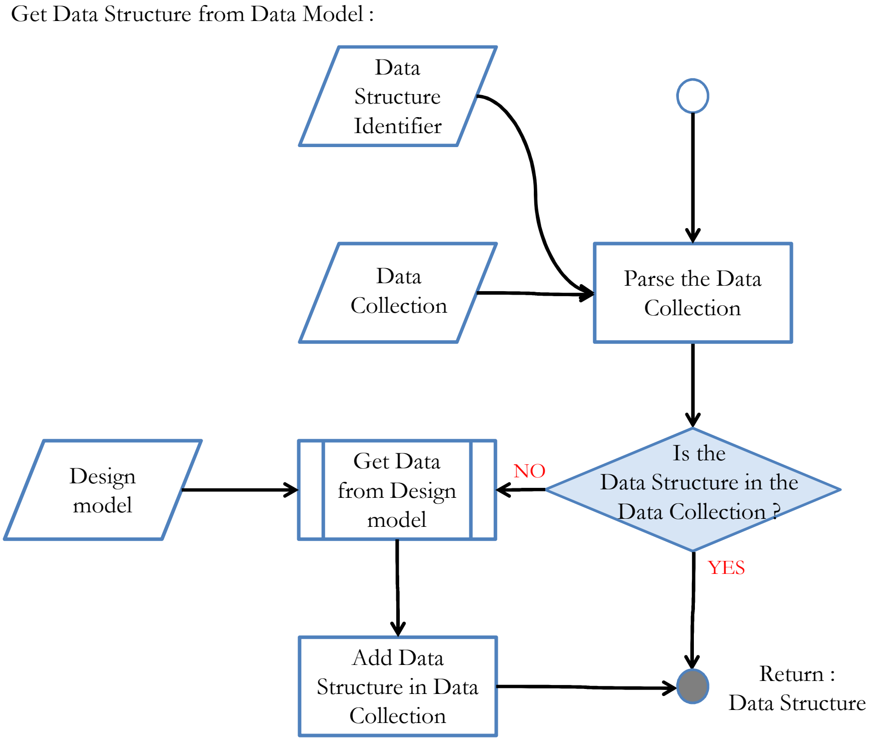

Get Data Structure from Data Model :
Data Structure Identifier
Data Collection
Parse the Data Collection
Design model
Get Data from Design model
NO
Is the Data Structure in the Data Collection ?
YES
Add Data Structure in Data Collection
Return : Data Structure

Figure VI.4: Process Flowchart depicting the procedure to get a data structure from the data model. *If necessary, the data structure is accessed in the design model via the sub-process `Get Data Structure from Design Model`.*

VI.2.2 Accessors

The low-level accessors are the methods to retrieve data about a system from its model. These methods implement low-level routines to query the domain-specific models such as reading of the AADL Instance Tree (AIT) or extraction of data from CPAL source files. Notice that data are accessed once in the domain-specific model and then stored as data structures in the data model, thus minimizing costly and useless operations on the domain-specific models (see Figure VI.4 and previous subsection).

AADL accessors. Listing VI.2 shows for example the Python code of the `ListOfTasks` accessor towards an AADL model. In this method, we explore the AADL Instance Tree (AIT) so as to retrieve the task set in the AADL model. We use the Python API provided by the OCARINA tool:

- the method `lmp.getInstances('thread')` at line 12 returns a list of tasks from an AADL model, i.e. it returns all instances of AADL `thread` components from an AADL model,
- the methods `lmp.getInstanceName` and `ocarina.getPropertyValueByName` respectively return task names and various properties of tasks, i.e. `'period'`, `'compute_execution_time'`, `'dispatch_offset'`, etc. in the AADL syntax.

CPAL accessors rely on the `cpal2x` tool which is part of the CPAL development environment [101]. Among other features, this tool extracts given data from CPAL source files and formats them in an easy-to-read output data file, e.g. in a `JSON` or `rt-format` textual data format. Figure VI.5 represents the `cpal2x` toolchain underlying CPAL accessors.

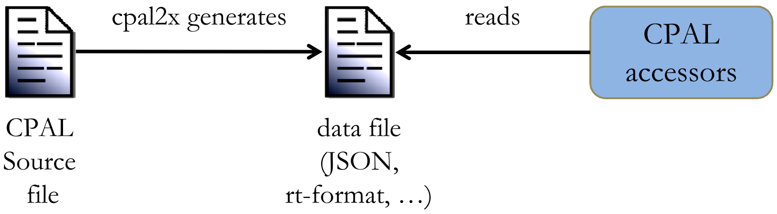


Figure VI.5: Implementation of CPAL accessors by means of the `cpal2x` tool.

Generation of tool-specific data models. Accessors can also be used to generate a data file in a tool-specific format when the analysis is to be externalized. Figure VI.6 depicts the toolchain that generates of a tool-specific data file from accessors to analyze these data with an external tool.

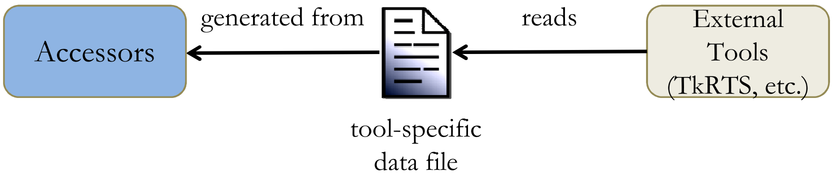


Figure VI.6: Using accessors to generate a tool-specific data file.

Each generation program uses its own adequate means to generate a tool-specific data file, according to the expected target format. Target data files range from lightweight text files (e.g. TkRTS [188]) to comprehensive data models defined by target metamodels (e.g. Cheddar [8], MAST [9], RTaW-Pegase [121], etc.).

```python

""" A function to return a list of tasks from an AADL model
"""

def ListOfTasks(self):

   #local variables
   _task_name=_period=_best_case_execution_time=_worst_case_execution_time=_deadline=_offset=_priority=_respTime=None
   _list_of_tasks=[]

   #we must specify the properties to get in the aadl instance model
   properties=['period', 'priority', 'deadline', 'compute_execution_time', 'dispatch_offset']
   property_value=None

   #we then explore- the AADL Instance Tree
   #get tasks from the AADL Instance Tree
   aadlInstances=lmp.getInstances('thread')[0]

   #get task properties
   for task in aadlInstances :
      #task name
      _task_name=lmp.getInstanceName(task)
      print ' ' * self._indentation, _task_name
      #various properties
      for prop in properties :
         #if the property exists
         if ocarina.getPropertyValueByName(task,prop)[0][1] !=  'KO':
            property_value=ocarina.getPropertyValueByName(task,prop)[0][1]
            print ' ' * self._indentation, prop+'='+property_value
            #we process values and store them
            for case in switch(prop):
                if case('period'):
               _period=util.getValueFromAADLTime(property_value, 'ms')
               break
[...]
         else:
            print  ' ' * self._indentation, prop+' not found in the model!'
      #we create a Task object and add it to the list
      _list_of_tasks.append(Task(_task_name,_period,_best_case_execution_time,_worst_case_execution_time,_deadline,_offset,_priority,_respTime))
   #we finally return the list of tasks
   return _list_of_tasks
```

Listing VI.2: Implementation of a specific AADL accessor using the OCARINA-Python API (`ListOfTasks` accessor).

VI.2.3 Analysis

An analysis carries out a set of operations and calculations from the data model. When completed, the analysis updates the data model with the calculated results. Our prototype enables two types of implementations: through an internal program in Python or by referencing an external tool.

Analysis with a Python program. Listing VI.3 shows a schedulability analysis written in the Python programming language. We implement the analysis from Sha et al. [148] via the `analysis()` method of the specific class `srl_pcp_test_th16`. The data model is an argument of this method. Any analysis must implement the following procedure:

1. *retrieve the data to analyze from the data model.* Here, the analysis requests a list of tasks at line 13,
2. *analyze the data.* Schedulability analysis is performed with a call to the built-in function `__srl_pcp_test_theorem16` at line 16. This function firstly calculates an upper admissible bound of the processor utilization factor (line 39). It then compares the actual utilization rate against the threshold (the effective processor utilization is calculated in the `for` loop at line 42, comparison to the upper limit occurs at line 47). The test result is stored in the `isSched` variable from the function return,
3. *update the data model with the analysis result.* The analysis updates the data model with the schedulability property (line 21) through a specific data structure `task_meta` that contains the `isSched` result (set at line 20).

Analysis through an external tool. The analysis can be outsourced to a third-party tool. Listing VI.4 shows how to reference an external tool. In this example, we use the commands provided by the MAST tool to launch the remote analysis (line 18). We must beforehand generate a tool-specific data model from accessors (line 15, see also *Generation of tool-specific data models* in Section VI.2.2).

VI.2.4 Orchestration

The orchestration module is implemented in two parts. First, we initialize the analysis graph. Then, we visit the analysis graph to execute the analyses.

Workspace Alloy – generation of Alloy files from AADL models. Initialization of the analysis graph relies on contracts manipulated in Alloy. In particular, we write the contracts with the Alloy language and evaluate them with the SAT solvers provided by the Alloy tool.

Figure VI.7 gives an overview of the Alloy workspace:

- `main` is the file to execute with Alloy. It defines the resolution scope and references all the files to analyze with the SAT solvers,

```python
""" Example of analysis class
"""

class srl_pcp_test_th16(Analysis): # define an analysis

   def analysis(self, model):
   """ This function implements the basic analysis process
       Arguments:
       model (DataModel): the data model
   """

   # read data from the data model
   tasks_list=model.get("LIST_OF_TASKS")[0]

   # execute the main test
   isSched=self.__srl_pcp_test_theorem16(tasks_list)

   # write data in the data model
   task_meta=model.get("TASKS_META")
   setattr(task_meta, 'isSched', isSched)
   model.update("TASKS_META", task_meta)


   # return the result to the orchestration module
   return isSched

   def __srl_pcp_test_theorem16(self, tasks_list):
   """ This function implements the business analysis

       Arguments:
       tasks_list ([Task]): a list of tasks
   """

   # local variables
   utilization_factor=0.0
   res=None
   blockingTime_factor=[]

   # compute the test bound
   test_bound=float(len(tasks_list))*(2.0**(1.0/float(len(
       tasks_list)))-1.0)

   # compute the utilization factor
   for task in tasks_list:
      utilization_factor+=task.worst_case_execution_time/task.
          period
      blockingTime_factor.append(task.blockingTime/task.period)

   # compare the utilization factor against the test bound
   if utilization_factor+max(blockingTime_factor)<=test_bound:
      # test is successful
      res=True
   else:
      # test is successful
      res=False

   return res
```

Listing VI.3: An example of schedulability analysis written in Python.

```
""" A class to externalize an analysis
"""

class classic_rm_MAST(Analysis):

    def analysis(self, model):
    """
    This function outsources the ’classic_rm’ analysis to the MAST tool

    Arguments:
    model (DataModel): the data model
    """

    # generate the MAST model from the data model
    self.generate_mast_input(model)

    # run the mast analysis with that generated model
    os.system("mast_analysis classic_rm mast-model.txt")
```

Listing VI.4: An analysis can be outsourced to a third-party tool.

- `model`, `analysis` and `goals` describes the contracts associated to models, analyses and goals respectively,
- `lib` (i.e. library) defines the set of data structures and properties that can be declared in contracts,
- `meta` defines the concepts manipulated in the Alloy specification, that is to say the concept of contract and associated constraints (in particular the precedence constraints).

We implemented new functionalities to OCARINA in order to generate part of the Alloy workspace within an AADLib workspace. We generate the blue-colored files in Figure VI.7 from AADL models. The other files (in red- and white-color) are generated as static files. The red-colored files can be edited manually to add new contracts (we generate samples only).

The generated Alloy specification is ready to be evaluated by the Alloy analyzer. We finally inject the graph found by Alloy in the orchestration Python program. This is done either manually in the first version of the prototype, or through an intermediate graph-formatted file in improved versions.

Visit of the analysis graph. The analysis graph found by Alloy provides the analysis paths to execute. We implemented methods methods to visit the analysis graph and execute the analyses.

Listing VI.5 shows how to define and use an analysis graph in a Python program. We must previously define the accessors (line 3), the data model (line 4) and the various analyses (lines 7 to 10). The orchestration module visits the graph, defined from line 16 to line 23 and represented in Figure VI.8, with the method `exec_analysis(...)`. This method considers a starting node that is the `AADL_model`, and an arrival node that is the `isSched` goal.

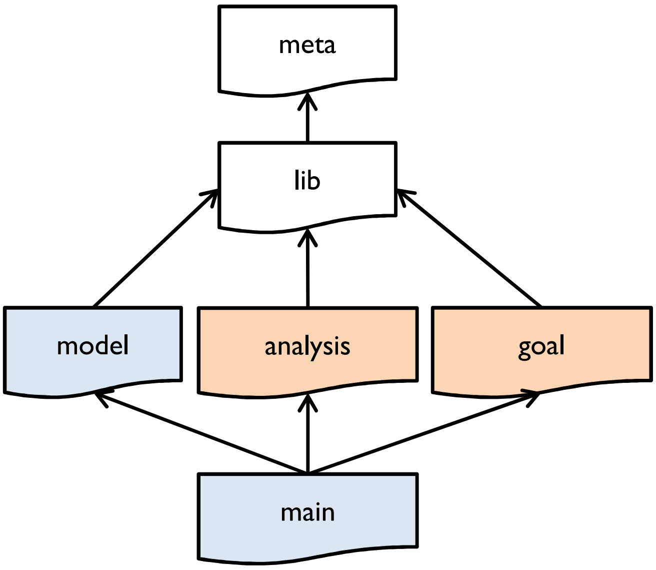


Figure VI.7: Components of the Alloy workspace.

```python

""" A script that creates an analysis graph with its components and visit it
"""

# declaration of the data model with accessors
aadl_accessors=AADL_accessors()         # aadl accessors
data_model=DataModel(aadl_accessors);     # data model using the accessors

# declaration of analyses
ll_context=ll_context()           # preconditions
srl_pcp_context=srl_pcp_context()
ll_rm_test=ll_rm_test()           # analyses
srl_pcp_test=srl_pcp_test()

# declaration of the orchestration module

# the analysis graph is hardcoded according to an Alloy solution
# an example of graph for the mars pathfinder case study
analysis_graph = {
     "AADL_model" : [ll_context, srl_pcp_context, ll_rm_test, srl_pcp_test],
     ll_context : [ll_rm_test],
     srl_pcp_context : [srl_pcp_test],
     ll_rm_test : ["isSched"],
     srl_pcp_test : ["isSched"],
     "isSched" : [],
}

o = Orchestration(analysis_graph)

# visit the analysis graph from "AADL_model" to "isSched" goal
# execute the analyses with help of the data model
o.exec_analysis("AADL_model", "isSched", data_model)
```

Listing VI.5: Creation and visit of an analysis graph in a Python program

We could visit the graph in many ways, e.g. performing topological ordering prior to execute the analyses, finding the shortest paths between nodes, or using common algorithms to traverse a graph such as Depth-First Search (DFS) or Breadth-First Search (BFS), etc. The chosen strategy must fulfill two constraints:

1. the graph must be visited such that the data and properties used by an analysis are computed the graph must be visited such that the data and properties used by an analysis are computed beforehand,
2. the analyses for which the preconditions are no met must not be executed; more widely, the analysis paths that include analyses for which the preconditions are no met must be aborted.

For the time being, the prototype implements a Breadth First Search (BFS) algorithm. In addition, the preconditions are verified in priority, i.e. before the subsequent analyses. When a precondition is not met, the subsequent analysis paths are removed from the execution stack. That way, the orchestration module fulfills the above-mentioned constraints. According to this policy, the graph represented in Figure VI.8 is visited in the following order: `aadl_model -> ll_context -> srl_pcp_context -> ll_rm_test -> srl_pcp_test -> isSched`. Notice that the `aadl_model` and `isSched` elements that denote the starting and ending nodes are not be executed. This execution stack enables to compute the data and properties in a correct order. In addition, if a precondition (represented with red arrows in Figure VI.8) is not satisfied the subsequent elements are removed from the execution stack. For instance, if the property computed by the `ll_context` analysis is *false* then the subsequent `ll_rm_test` analysis will not be executed. Let us finally note that discarding a path does not prevent from reaching the goal `isSched` if an alternative – correct – path exists. That is, by executing here the `srl_pcp_context` and `srl_pcp_test` analyses for which the results must be *true*.

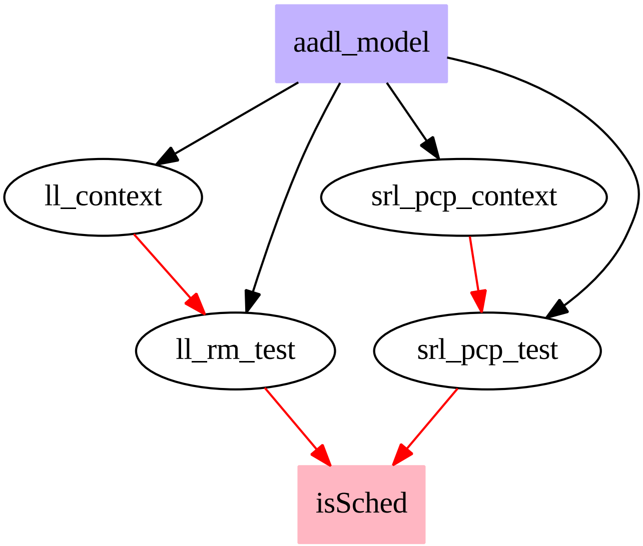


Figure VI.8: Example of analysis graph to be visited by the orchestration module.

More sophisticated algorithms to visit analysis graphs can be proposed in future version of the prototype in order to address more advanced use cases, e.g. to detect redundant paths, to find optimal paths according to customized metrics, to (re-)execute a subpart of a graph, and so on.

VI.3 Working with the tool

Figure VI.9 illustrates the activities that are supported by the tool. The tool implements the following workflow:

1. **Creation of the analysis repository**: the first task is to create the models, analyses and goals that form together the analysis repository. We can make a model with the help of a language such as AADL or CPAL. We can fully program an analysis in Python, or reference an external tool, and add it to the analysis repository. Last, we can specify the analysis objectives. Presently, the models can be created via their respective editors, i.e. OSATE and the CPAL-Editor. The analyses must be coded separately and then included manually in the tool program. The goals must be defined in Alloy.

2. **Analysis of the repository**, in two steps:

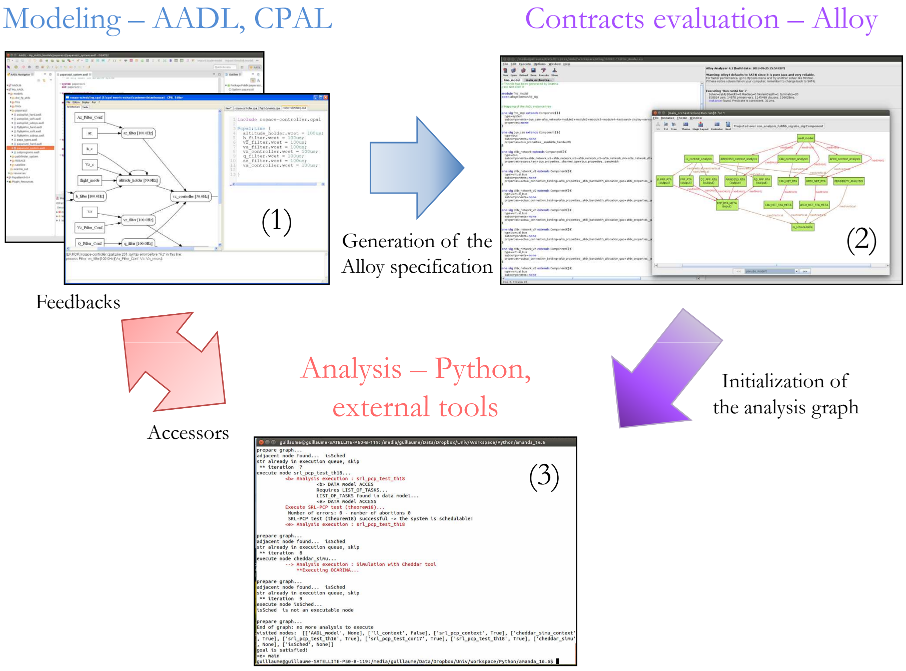
   (a) **Evaluation of contracts**: we semi-automatically generate the Alloy specification that we then evaluate through SAT resolution methods. The analysis graph found by the Alloy solvers is injected in the Python program,

   (b) **Execution of the analyses**: the tool automatically executes the analyses from the analysis graph. The execution takes into account the input model(s) and the analysis goal(s). The analyses use accessors to query data from models.

3. **Feedbacks**: the tool finally provides feedbacks about the models. These feedback are trustworthy (contextualized) and fulfill the analysis objectives, e.g. answering questions about the schedulability of the system, computing precise dependability attributes, etc. The tool is able to adapt the analysis process to the input models, the available analyses and the analysis goals.

Listing VI.6 shows a typical execution of the tool. The trace involves the various modules presented earlier in this chapter: orchestration, analysis, data model and accessors. We firstly initialize the different modules, by choosing for instance the input model which is an AADL model of the mars pathfinder robot in this example (see Section VII.2 for a complete description of this case study). We also initialize the orchestration module with the analysis graph, the data model with the AADL accessors as well as the various analyses referenced by the tool. The tool then visits the nodes-analyses of the graph according to the Breadth-First Search algorithm. At each visited node, we execute the associated analysis and update the execution stack in accordance with the analysis result.

The graph which is the same of Figure VI.8 includes some nodes-analyses in order to verify the schedulability of the mars pathfinder system modeled with AADL. The visit starts with the `AADL_model` node at `iteration 1`. At `iteration 2`, the `ll_context` analysis is unsuccessful, meaning that the preconditions of the


Modeling – AADL, CPAL
Contracts evaluation – Alloy
(1)
Generation of the Alloy specification
(2)
Feedbacks
Analysis – Python, external tools
Initialization of the analysis graph
Accessors
(3)

```
guillaume@guillaume-SATELLITE-P50-B-119: /media/guillaume/Data/Dropbox/Univ/Workspace/Python/amanda_16.6
prepare graph...
adjacent node found...  isSched
str already in execution queue, skip
 ** iteration  7
execute node srl_pcp_test_th18...
          <b> Analysis execution : srl_pcp_test_th18
                     <b> DATA model ACCES
                     Requires LIST_OF_TASKS...
                     LIST_OF_TASKS found in data model...
                     <e> DATA model ACCESS
          Execute SRL-PCP test (theorem18)...
           Number of errors: 0 - number of abortions 0
           SRL-PCP test (theorem18) successful -> the system is schedulable!
          <e> Analysis execution : srl_pcp_test_th18

prepare graph...
adjacent node found...  isSched
str already in execution queue, skip
 ** iteration  8
execute node cheddar_simu...
          --> Analysis execution : Simulation with Cheddar tool
              **Executing OCARINA...

prepare graph...
adjacent node found...  isSched
str already in execution queue, skip
 ** iteration  9
execute node isSched...
isSched  is not an executable node

prepare graph...
End of graph: no more analysis to execute
visited nodes:  [['AADL_model', None], ['ll_context', False], ['srl_pcp_context', True], ['cheddar_simu_context'
, True], ['srl_pcp_test_th16', True], ['srl_pcp_test_cor17', True], ['srl_pcp_test_th18', True], ['cheddar_simu'
, None], ['isSched', None]]
goal is satisfied!
<e> main
guillaume@guillaume-SATELLITE-P50-B-119:/media/guillaume/Data/Dropbox/Univ/Workspace/Python/amanda_16.6$
```

Figure VI.9: Workflow supported by the tool.

`ll_rm_test` analysis are not met. The `ll_rm_test` is therefore discarded. Instead, at `iteration 4`, the graph executes the `srl_pcp_test` by firstly checking its preconditions via the `srl_pcp_context` analysis at `iteration 3`. The system is schedulable according to the `srl_pcp_test`. The visit of the graph ends at `iteration 5`: the goal `isSched` is met. The tool finally summarizes the nodes-analyses visited and their results.

VI.4 Synthesis and conclusion

In this chapter, we presented a tool prototype that implements the concepts introduced in the first part of this thesis. The prototype implements several functions in order to integrate models and analyses in a same framework. By that means, the tool takes on the analysis process when designing an embedded system. In particular, the tool is able to adapt the analysis process to the input models, the available analyses and the analysis goals.

Our prototype implements several modules-functions, each one implementing a part of the concepts presented in the first part of this thesis. We implemented the first version of the prototype through a set of scripts written in Python and various model processors (e.g. parsers, model generators, SAT solvers, etc.). We run the scripts on top of modeling tools – OSATE, CPAL-Editor – and, possibly, external analysis tools – TkRTS, MAST, Cheddar, etc.

This tool prototype will allow us to apply a design workflow that systematically combines architectural models and analyses. We present case studies in Chapter VII.

```
$ python main.py
main thread... initializes components...
      Available models are: (1) fms (2) paparazzi (3) pathfinder (4) satellite
      Please choose a model (number): 3
      **Files directory: aadl_model/pathfinder
[...]
main thread... execute analyses from contracts...
visiting graph according to bfs algorithm...
 ** iteration  1
execute node AADL_model...
AADL_model  is not an executable node

prepare graph...
adjacent node found...
ll_context added in queue for execution
srl_pcp_context added in queue for execution
ll_rm_test added in queue for execution
srl_pcp_test_th16 added in queue for execution
 ** iteration  2
execute node ll_context...
   Check preconditions for LL-test...
   precondition failed  ('tasks are dependent',)

prepare graph...
update graph... delete subsequent paths...
 ** iteration  3
execute node srl_pcp_context...
   Check preconditions for SRL-PCP-test...
   OK

prepare graph...
adjacent node found...  srl_pcp_test_th16
srl_pcp_test_th16 already in execution queue, skip
 ** iteration  4
execute node srl_pcp_test_th16...
   SRL-PCP-test is satisfied , U=0.725420 <= 0.728627 -> the tasks set is
       schedulable!

prepare graph...
adjacent node found...  isSched
str added in queue for execution
 ** iteration  5
execute node isSched...
isSched  is not an executable node

prepare graph...
End of graph: no more analysis to execute
visited nodes:  [['AADL_model', None], ['ll_context', False], ['
    srl_pcp_context', True], ['srl_pcp_test_th16', True], ['isSched', None]]
```

Listing VI.6: Record of a typical tool execution displayed in the terminal.

Chapter VII

Case studies

Abstract

In the first part of this thesis, we reviewed several concepts in order to analyze non-functional properties in a Model-Driven Engineering approach. We implemented these concepts through a tool prototype introduced in Chapter VI. In this last contribution chapter, we show several case studies that systematically combine architectural models and analyses so as to design embedded systems.
We present three case studies in this chapter. Section VII.1 deals with the continuous timing validation of the Paparazzi *drone throughout the design process. In the second case study (Section VII.1), we use our approach to resolve the original design error that caused a serious failure of the* Mars Pathfinder *system. The last case study in Section VII.1 concerns the design space exploration of an avionic system that comprises a Flight Management System (FMS) and a Flight Control System (FCS). To implement these case studies, we use the tool prototype presented in the previous chapter together with architecture description languages (i.e. AADL and/or CPAL) and many real-time scheduling analyses.*

VII.1 Continuous validation of the Paparazzi UAV design

This section deals with the Paparazzi case study [189, 190]. We firstly present the Paparazzi UAV project. We then introduce the analysis problem that occurs at design time. We finally apply our approach to resolve this problem.

VII.1.1 System overview

Paparazzi UAV and Papabench. Paparazzi UAV (Unmanned Aerial Vehicle) is an open-source drone launched at the ENAC school in 2003 [189, 191]. The Paparazzi project encompasses hardware and software such as the source code – airborne and ground station – and various design documents. As a free and open-source project, Paparazzi encourages reuse, extension and improvement of these elements, in particular to port the UAV on various platforms. Paparazzi developers include researchers, companies or hobbyists.

In our case, we consider the Paparazzi UAV in order to experiment the approach presented in this thesis. We updated, corrected and extended the AADL models originally developed by Nemer et al. [190], Nemer [192]. The source models are part of *Papabench* [190, 193], a benchmark for WCET evaluation at IRIT (used in [194, 192] or more recently in [195]).

Architecture. The Paparazzi system basically consists of an airborne system and a ground control station. The subsystems communicate with each other via a radio link. We only consider the embedded system for our experimentation.

The embedded system includes hardware, e.g. a control card with power supply and processors (dual micro-controllers), sensors (infrared sensors, GPS, Gyroscope), actuators (servos, motor controllers) and other payloads (camera and video transmitter). The airborne system also comprises a R/C receiver and a radio modem to communicate with the ground station.

The software uses a dual processor architecture: the first processor **MCU1** commands the aircraft (Fly-By-Wire system) while the second **MCU0** manages navigation, sensors, payloads communications and other processing (Autopilot system). The two micro-controllers communicate via a Serial Peripheral Interface (SPI) bus.

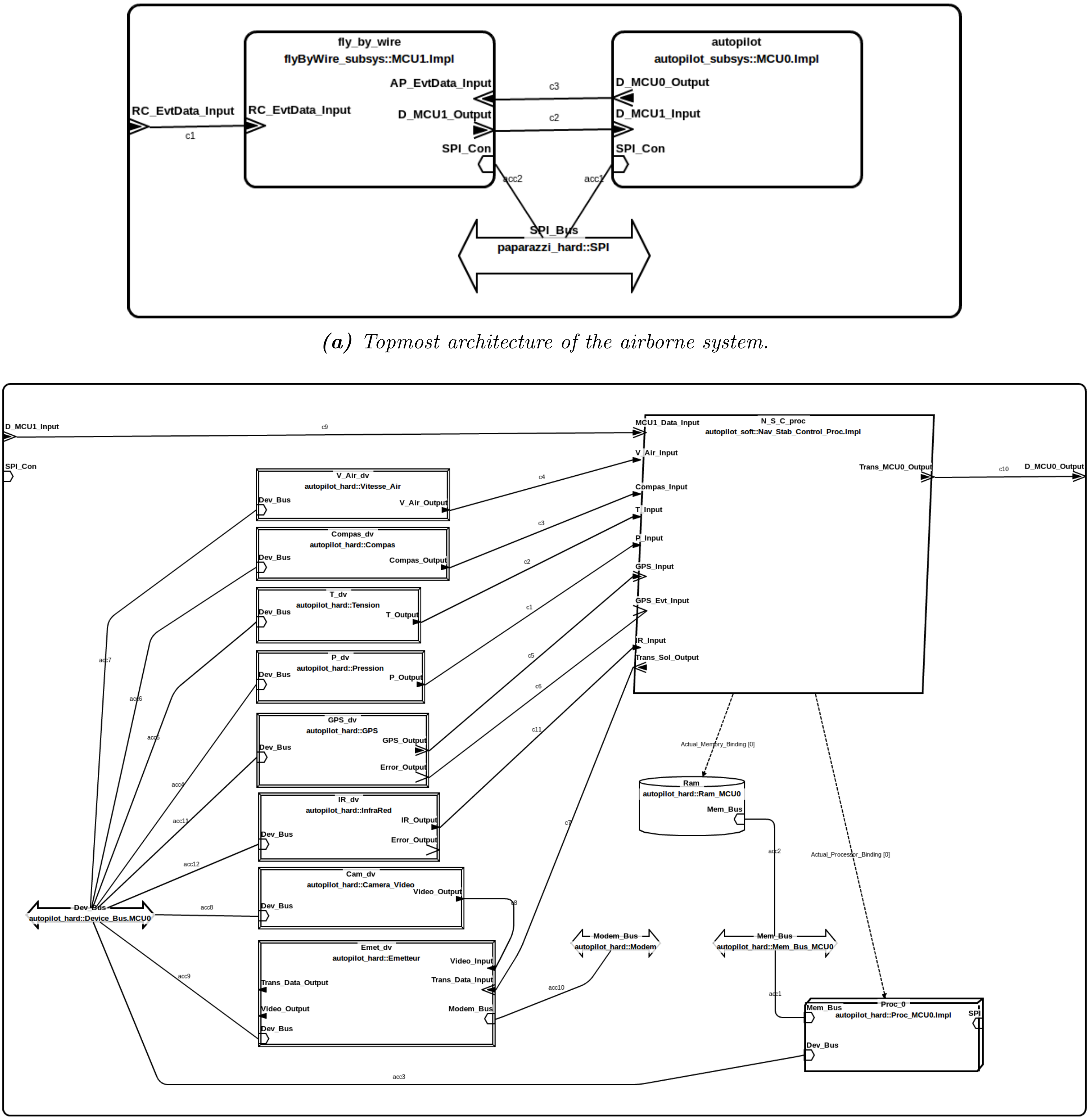
More information about the functions or hardware and software components is available in the Paparazzi documentation, for instance [189, 191].

VII.1.2 Problem: validation throughout the design process

The design of an embedded system such as the Paparazzi UAV is progressive. The designer starts for example with a definition of the task set from the functional description of the system. He or she then defines the way these tasks are to be activated (e.g. strictly periodically, sporadically, according to a mixture of periodic, sporadic and aperiodic activation, etc.) according to task parameters (e.g. periods or minimum inter-release times, worst-case execution times, deadlines, etc.). In addition, the designer has to set the scheduling policy that will meet the timing constraints. A full and correct design must also take account of task dependencies. To this end, the designer defines some appropriate policy for inter-task data exchanges, possibly implements synchronization mechanisms, enforces task dependencies, etc.

Throughout the design process, the designer must be able to evaluate the hypotheses and choices made. An analysis enables to validate, or conversely, invalidate some choices. The goal is to define an architecture that will meet the functional and non-functional requirements (in our case, the real-time constraints). It is thus necessary to adjust the analysis process to the models provided at each stage in the design process. The analysis to apply greatly differ at the early and late stages in the design process, whether a model is simplified, coarse-grained, far from reality at the beginning of the design, or, on the contrary, more exhaustive, complex, and close to the final system in the last design stages.

To illustrate this case study, we defined, through AADL models, several task sets at various stages of the design process of the Paparazzi system:



(a) *Topmost architecture of the airborne system.*

(b) *Architecture of the autopilot subsystem.*

Figure VII.1: Architecture of the Paparazzi system in AADL

- **step 1**: we assume periodic, non-preemptive tasks and aim to evaluate either a Fixed Task Priority (e.g. Rate Monotonic) or Fixed Job Priority scheduling algorithm (e.g. Earliest Deadline First),
- **step 2**: we rather consider preemptive tasks, still periodic and scheduled according to a Fixed Priority algorithm,
- **step 3**: we model the system more accurately and consider a mixture of periodic and aperiodic tasks, with preemptive and Fixed Priority scheduling,
- **step** n: the design can continue to further describe task dependencies, task precedences, inter-task caches, etc.

Our problem is thus to adapt, at each step in the design process, the scheduling analysis to the input AADL model in order to check that the system fulfills the timing constraints (i.e. satisfies all the deadlines). Table VII.1 summarizes the various task parameters. We study the Autopilot system only (the process would be identical for the Fly-By-Wire).

Task	**Description**	**Parameters**	
		T	**C**
I_4	interrupt-spi-1	50 ms^1	{251 μs, 447 μs}
I_5	interrupt-spi-2	50 ms^1	{151 μs, 228 μs,}
I_6	interrupt-modem	100 ms^1	{303 μs,520 μs}
I_7	interrupt-gps	250 ms^1	{283 μs,493 μs}
T_6	radio-control	25 ms	{15,6 ms, 21,1 ms}
T_7	stabilization	50 ms	{5681 μs, 6654 μs}
T_8	link-fbw-send	50 ms	{233 μs, 471 μs,}
T_9	receive-gps-data	250 ms	{5987 μs, 6659 μs}
T_{10}	navigation	250 ms	{44,42 ms, 54,35 ms}
T_{11}	altitude-control	250 ms	{1478 μs, 1660 μs}
T_{12}	climb-control	250 ms	{5429 μs, 6241 μs}
T_{13}	reporting	100 ms	{5 ms, 12,22 ms}

[1] applies for **step 1** and **step 2** only.

Table VII.1: Task parameters of the Paparazzi UAV (taken from [192] and [195]).

VII.1.3 Application of our approach

We apply our approach in order to analyze the schedulability of the Paparazzi system throughout the design process. We model the software architecture with the help of the AADL language, at each stage in the design process described in the previous section (i.e. **step 1**, **step 2** and **step3**).

Analysis repository. We consider the following analyses:

- schedulability tests:

- *srl_rm_test* which is a schedulability test contributed by Sha et al. [148],
- *lss_sporadic_test*, another schedulability test by Lehoczky [196] and studied later by Bernat and Burns [197],
- *rts_periodic_npfp* that is a schedulability test based on worst-case response times [130].

- analyses to check the preconditions of above-mentioned schedulability tests: *srl_rm_context*, *lss_sporadic_context* and *periodic_npfp_context*

Table VII.2 sums up the preconditions of the various analyses.

Analysis / Precondition	srl_rm_test	lss_sporadic_test	rts_periodic_npfp
mono-processor	✓	✓	✓
periodic tasks	✓	✓	✓
aperiodic tasks	✗	①	✗
offsets	$O_i \geq 0$		
jitters	✗	✗	✗
implicit deadlines	✓	✓	✓
fixed computation times	✓	✓	✓
dependent tasks	✗	✗	✗
self-suspension	✗	✗	✗
preemption	✓	✓	✗
overheads	✗	✗	✗
scheduling algorithm	RM	RM	$NP - FP$

① aperiodic tasks must be scheduled via a Sporadic Server (SS).

Table VII.2: Analysis preconditions for the Paparazzi case study. ✓*: the predicate must be true.* ✗*: the predicate must be false.* ○*: special conditions. Otherwise, the expected condition is stated explicitly.*

First of all, we set the precedences between these analyses. Figure VII.2 portrays the analysis graph. The rectangular-shaped elements represent the starting (the `aadl_model`) and ending nodes (the `isSched` goal). Elliptic forms represent the analyses. Black arrows display data dependencies while red arrows show property dependencies. This graph, which is to be executed at each stage in the design process, will enable us to evaluate the AADL model in a systematic and dynamic way.

Step 1. At the first stage in the design process, we assume strictly periodic tasks. The model describes a set of n tasks $\Pi = \{\tau_1, \ldots, \tau_n\}$ with $\tau_i = (C_i, T_i, D_i)$, C_i is the worst-case execution time, T_i is the period and D_i is the deadline such that $D_i = T_i$. We consider a non-preemptive scheduling algorithm, either with Fixed Task Priorities (FTP, e.g. Rate Monotonic) or Fixed Job Priorities (FJP, e.g. Earliest Deadline First).

We visit the graph displayed in Figure VII.2. Figure VII.3 recaps the analysis process during the first design stage. First and foremost, we execute the preconditions analyses: ❶ *lss_sporadic_context*, ❷ *srl_rm_context* and ❸ *periodic_npfp_context*.

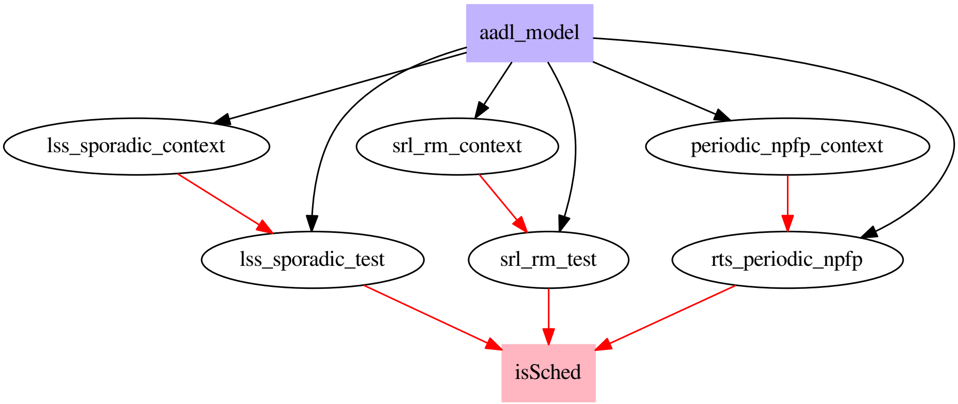


Figure VII.2: Analysis graph for the Paparazzi UAV case study.

The properties computed by the *lss_sporadic_context* and *srl_rm_context* are *false*: the tasks are non-preemptive. Thus, the preconditions of the *lss_sporadic_test* and *srl_rm_test* are not fulfilled, meaning that these analyses cannot be executed. The properties calculated by the *periodic_npfp_context* analysis are *true*. Therefore, we can execute the ❹ *rts_periodic_npfp* analysis.

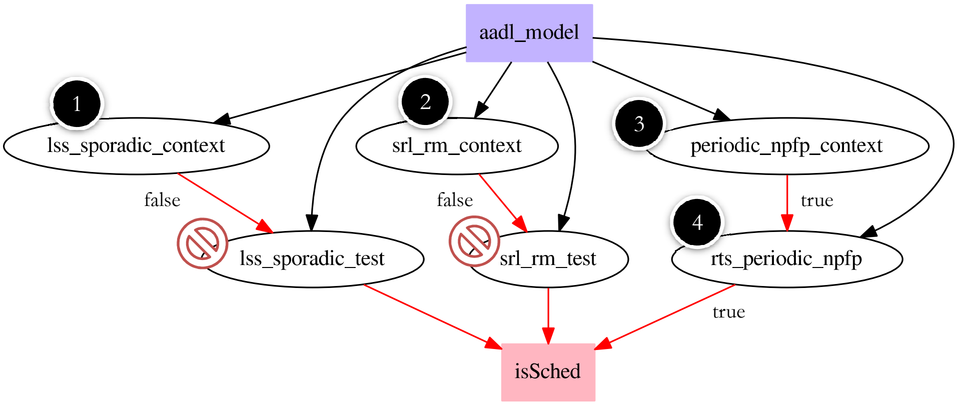


Figure VII.3: Analysis process during the first design stage of the Paparazzi UAV.

We carry out the *rts_periodic_npfp* analysis via the `TkRTS` tool [188, 198]. We evaluated both the FTP and FJP scheduling cases through NP-FP (priorities defined according to RM) and NP-EDF algorithms respectively. Table VII.3 and Table VII.4 summarize the results.

The results are successful in the case of a FTP scheduling given an optimal priority assignment calculated by the tool. For each task, the worst-case response time *bound* in Table VII.3 is lower than the deadline D. On the contrary, in the case of a FJP scheduling, the produced schedule does not meet all the deadlines. The *laxity* in Table VII.4, that is the remaining time to deadline at the task completion, can be negative for two tasks (i.e. `interrupt_spi_th1` and `interrupt_spi_th2`), meaning that several deadlines can be missed. As a consequence, the designer would select the NP-FP scheduling algorithm instead of the NP-EDF.

Algorithm	Task	C (µs)	T (µs)	D (µs)	*bound* (µs)
np-fp	`ctrl_by_rc_th`	21100	100000	100000	21100
	`interrupt_gps_th`	493	250000	250000	493
	`interrupt_modem_th`	520	100000	100000	520
	`interrupt_spi_th2`	228	50000	50000	228
	`interrupt_spi_th`	447	50000	50000	447
	`send_grd_station_th`	12220	100000	100000	12220
	`send_mcu1_th`	471	250000	250000	471
	`stab_th`	6654	100000	100000	6654
	`climb_ctrl_th`	6241	250000	250000	6241
	`nav_th`	53350	250000	250000	53350
	`alt_ctrl_th`	1660	250000	250000	1660
	`data_acq_filt_th`	6659	250000	250000	6659

Table VII.3: Result of the *rts_periodic_npfp* analysis computed via the `TkRTS` tool.

Algorithm	Task	C (µs)	T (µs)	D (µs)	*bound* (µs)	*laxity* (µs)
np-edf	`ctrl_by_rc_th`	21100	100000	100000	95193	4807
	`interrupt_gps_th`	493	250000	250000	152562	97438
	`interrupt_modem_th`	520	100000	100000	95193	4807
	`interrupt_spi_th2`	228	50000	50000	54024	**-4024**
	`interrupt_spi_th1`	447	50000	50000	54024	**-4024**
	`send_grd_station_th`	12220	100000	100000	95193	4807
	`send_mcu1_th`	471	250000	250000	152562	97438
	`stab_th`	6654	100000	100000	95193	4807
	`climb_ctrl_th`	6241	250000	250000	152562	97438
	`nav_th`	53350	250000	250000	152562	97438
	`alt_ctrl_th`	1660	250000	250000	152562	97438
	`data_acq_filt_th`	6659	250000	250000	152562	97438

Table VII.4: Result of the *rts_periodic_npedf* analysis computed via the `TkRTS` tool

Step 2. During this second analysis stage, we aim at evaluating the following scheduling configuration: periodic tasks to be scheduled according to a Fixed Tasks Priority, preemptive algorithm.

Similarly to step 1, we execute the precondition analyses in the first place: *lss_sporadic_context*, *srl_rm_context* and *periodic_npfp_context*. That time, only the *srl_rm_test* can be carried out as: (i) the result of the *srl_rm_context* analysis is *true*; (ii) the properties calculated by the *lss_sporadic_context* and *periodic_npfp_context* are *false* (the tasks are not to be periodic for the first analysis and must not be preemptive for the second). Therefore, the *lss_sporadic_test* and the *rts_periodic_npfp* analysis cannot be used.

We carry out the *srl_rm_test* with the help of our tool. According to the analysis result shown in Listing VII.1, the task set does not pass the test. Indeed, the amount of processor time used by the task set is above the limit not to be exceeded so as to be sure that the task set is schedulable. This test in exact (i.e. provides a sufficient and necessary condition) and, thus, we conclude that the task set is in fact unschedulable.

```
$ python main.py
[...]
Execute SRL-RM-test (theorem 15)...
[...]
Number of errors: 0 - number of abortions 3
SRL-RM-test aborted: the system is not schedulable!
```

Listing VII.1: Result of the *srl_rm_test* computed via our tool.

Step 3. At the third design step, we model the system more accurately. We no longer assume that all the tasks are periodic. Rather, we characterize the Paparazzi system with a mixture of periodic and aperiodic tasks. Thus, the model describes a set of n periodic tasks $\Pi_p = \{\tau_1, \ldots, \tau_n\}$ and an additional tasks τ_s to serve the k aperiodic tasks $\Pi_{ap} = \{\tau_1, \ldots, \tau_k\}$. Aperiodic tasks are scheduled through a Sporadic Server (SS) characterized by a maximum capacity C_s^{SS} and a replenishment period T_s^{SS} [199]. We define these parameters as follows:

- the server capacity such that $C_s^{SS} = \sum_{\{\tau_j \in \Pi_{ap}\}} C_j$,
- $T_s^{SS} = \min\ T_{i,\tau_i \in \Pi_p}$ in order to execute the server task with the highest priority.

We still consider a FTP priority algorithm (i.e. Rate Monotonic), which is able to preempt tasks.

In this new context, the preconditions of the *srl_rm_test* and *rts_periodic_npfp* are no longer satisfied: the tasks are not periodic. Hence, we cannot execute these analyses. Alternatively, we can use the *lss_sporadic_test* as the properties computed by the *lss_sporadic_context* analysis are *true*.

The test by Lehoczky [196] computes the amount of processor time that is used by the set of tasks. In this case, the processor utilization factor encompasses two dimensions: the fraction of processor time consumed by the periodic tasks U_p and the fraction of processor time used by the sporadic server U_s^{SS}. Lehoczky [196] defined a limit not to be exceeded:

$$U_p \leq \ln \frac{2}{U_s^{SS} + 1} \tag{LSS-test}$$

According to the result of the *lss_sporadic_test*, displayed in Listing VII.2, this threshold is respected, meaning that the system is schedulable under Rate Monotonic.

```
$ python main.py
[...]
Execute lss_sporadic_test...
lss_sporadic_test is satisfied, U is 0.673264 <= 0.676408064556 -> the tasks
    set is schedulable!
```

Listing VII.2: Result of the *lss_sporadic_test* computed via our tool.

Figure VII.4 recaps the analysis paths applied at each design step, displayed as Step 1, Step 2 and Step 3. The analysis paths shown with plain-blue arrows comprise the analyses used to verify the schedulability of the task set at each stage in the design process (**step 1 to 3** described in previous paragraphs). Sub-paths shown with dashed-red arrows include analyses in order to verify the preconditions of the diverse schedulability tests.

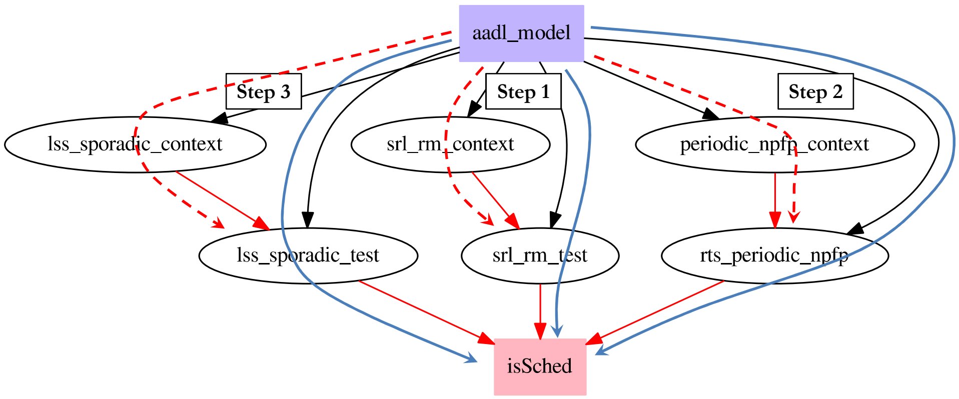


Figure VII.4: Analysis paths executed at each design stage of the Paparazzi UAV.

Step *n*. The model can be enriched to represent the Paparazzi system even more finely: data dependencies and/or precedences between tasks, synchronization mechanisms, inter-task caches, etc.

The approach that we applied during the early stages can be applied at any stage in the design process, including the late stages. Our approach is applicable to any type of model (nature, complexity) and to a large panel of analyses.

VII.1.4 Conclusion

The design of an embedded system such as the Paparazzi drone is progressive. During this process, the designer conceives the system through a multitude of models, e.g. from a simple, coarse-grained model at an early design stage to a more complex and accurate one during late design steps. The designer must be able to evaluate a model at any stage in the design process. An analysis enables to validate, or on the contrary, discard some design choices, assumptions made about the system, etc. It is hence necessary to automatically tune the analysis process according to the models provided at each stage in the design process.

We illustrated this case study with various AADL models to represent the Paparazzi UAV at different design stages. These models delineates several task sets, e.g. strictly periodic tasks versus a mixture of periodic and aperiodic activation, preemptive against non-preemptive scheduling, Fixed Task Priority or Fixed Job Priority scheduling algorithms, etc.

First of all, our approach identifies the interdependences between analyses. For this specific case study, this information enables us to find any analysis A_0 that can be used to check the set of preconditions $\{P_1\}$ of any analysis A_1. Afterwards, our tool executes the analyses according to the (completeness of the) input model, the interdependences between analyses and the analysis goal. We have been able to adjust the analysis process to verify the schedulability of the task sets defined through the AADL models at different stages in the design process (i.e. $\boxed{\text{Step 1}}$, $\boxed{\text{Step 2}}$ and $\boxed{\text{Step 3}}$ in Figure VII.4)

Let us finally note that the approach applied in this case study could be used similarly at more advanced design stages: to model and analyze task dependencies and/or task precedences, to propose and evaluate policies for inter-task data exchanges and/or synchronization mechanisms between tasks, to represent and assess inter-tasks caches, etc. In addition, this approach can be applied just as well with more complex analyses, more important analysis repositories, and models of diverse kinds (e.g. see the case study including CPAL in Section VII.3), typically as part of a complete design environment.

VII.2 Correct design of the Mars pathfinder system

This section deals with the Mars Pathfinder case study [200, 125]. First of all, we provide an overview of the Mars Pathfinder system. Next, we present the software error that occurred during the Mars Pathfinder mission and caused a major failure of the system. Last, we show that our approach would have detected and fixed this error trough a combination of architectural models and systematic analysis of these models at an early design stage.

VII.2.1 System overview

Mars Pathfinder mission. The Mars Pathfinder mission was a discovery mission that took place in the late 1990s in the frame of the MESUR (Mars Environmental SURvey) program conducted by the NASA.

Mars Pathfinder is a robotic spacecraft that landed on Mars and released an exploratory robot. The Mars Pathfinder system consists of a stationary lander and a microrover named Sojourner.

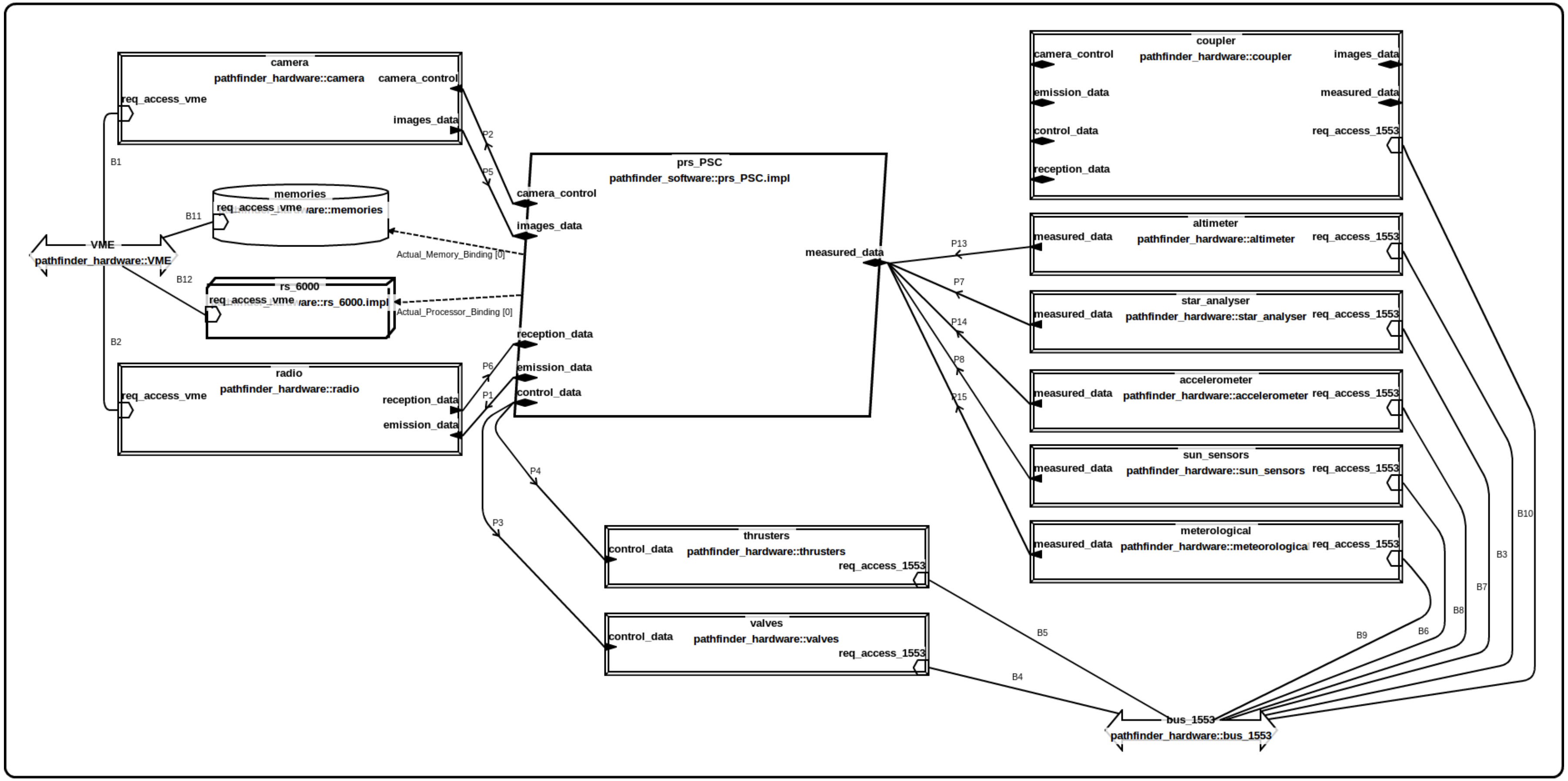


Figure VII.5: Hardware architecture of the Mars Pathfinder system in AADL.

Sojourner is a six-wheeled vehicle controlled from Earth. This control is done by means of high frequency radio waves, between the lander and Earth and between the lander and the rover. Both the lander and the rover are equipped with instruments to investigate the surface of Mars: cameras, spectrometers, atmospheric structure instrument and meteorology. Among those instruments, we can mention an altimeter and an accelerometer embedded on the station on Mars and a sun sensor and a star analyzer on the rover. During the mission the spacecraft collected gigabytes of data about the Martian environment (images, measurements about the atmosphere, etc.).

Hardware and software architecture. Figure VII.5 represents the simplified hardware architecture of the Mars Pathfinder system. The subsystems (lander and rover) include processing and memory resources together with control and measurement devices (radio, altimeter, accelerometer, thrusters, etc.). The components communicate with each other through VME or 1553 buses. Two couplers connect the subsystems (high frequency communication link).

The software architecture is based on real-time operating system (VxWorks) and includes over 25 tasks. Figure VII.5 depicts the simplified software architecture of the Mars Pathfinder system. The tasks in the *exploration* mode are:

- *bus_scheduling* task that controls the transactions on the 1553 bus,
- *data_distribution* task to collect the data from the 1553 bus and write them in the shared *data* buffer,
- *control_task* to control the rover,
- *radio_task* to communicate between the lander and Earth,
- *measure_task* to control the lander camera.
- *measure_task* and *meteo_task* for the various measurements (altimeter, accelerometer, meteorological, etc.).

All the tasks are to be executed by the RTOS according to their periods. In addition, four tasks access a `Data` resource in a concurrent way. Table VII.5 summarizes the tasks with their properties.

VII.2.2 Problem: original design error

During the Mars Pathfinder mission, the spacecraft experienced several resets, each one resulting in losses of data. After some investigations, the failure proved to come from a typical priority inversion phenomenon.

Figure VII.7 shows the execution sequence leading to the system failure with a temporal diagram. In that scenario, the *meteo_task* has an execution time equal to 75 ms (3 with reduced parameters). The RTOS schedules the tasks according to the priority given in Table VII.5. Yet, the temporal diagram shows that the *data_distribution* task misses its deadlines during its third job. This fault causes a reset of the system.

The failure comes from a priority inversion problem. The *meteo_task* accesses the resource at time 9 and blocks during it whole execution. The *data_distribution*

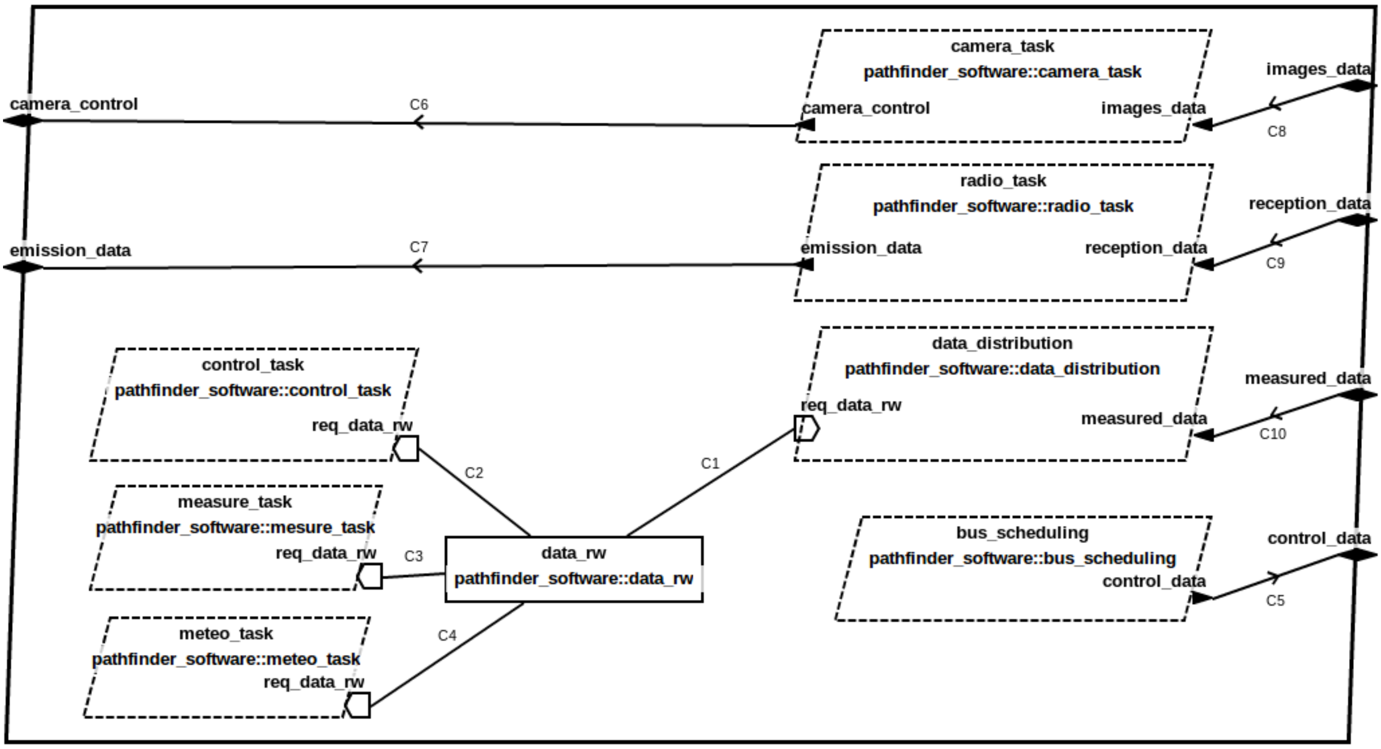


Figure VII.6: Software architecture of the Mars Pathfinder system in AADL.

Task	Priority	Parameters (ms)		Reduced parameter		Critical section
		T	C	T	C	
bus_scheduling	1	125	25	5	1	-
data_distribution	2	125	25	5	1	1
control_task	3	250	25	10	1	1
radio_task	4	250	25	10	1	-
camera_task	5	250	25	10	1	-
measure_task	6	5000	50	200	2	2
meteo_task	7	5000	{50,75}	200	{2,3}	{2,3}

Table VII.5: Task parameters of the Mars Pathfinder system (taken from [125]).

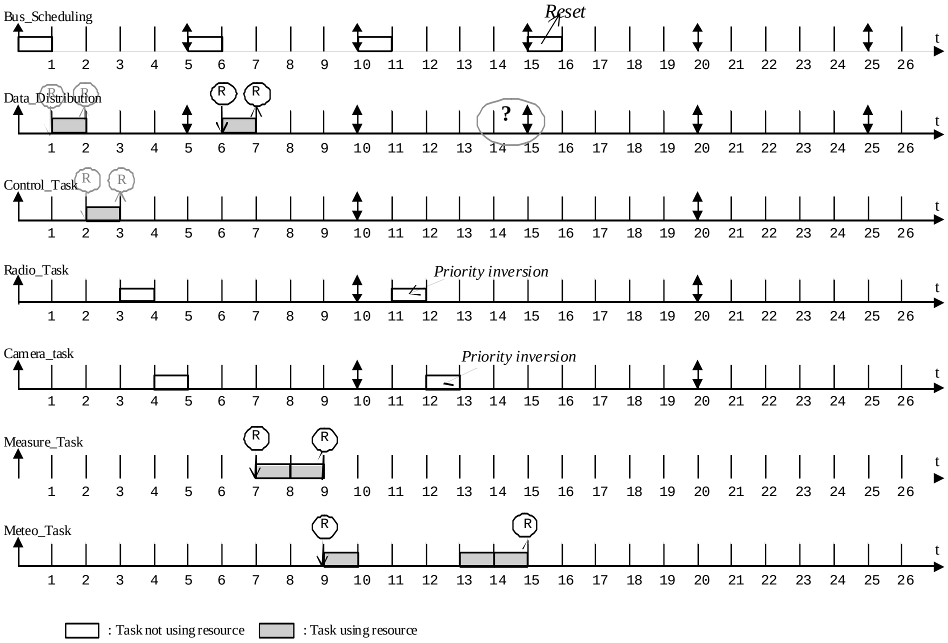


Figure VII.7: Faulty schedule of the Mars Pathfinder task set (taken from [125])

task, which has an higher priority, awakes at time 10. Nevertheless, it cannot execute has the *data* resource is blocked. During this blocking time, the *radio_task* and *camera_task* can execute as $p_{radio\_task} < p_{camera\_task} < p_{meteo\_task}$. There is a priority inversion phenomenon as tasks with intermediate priorities (*radio_task* and *camera_task*) execute before the task which has the higher priority (*data_distribution*) because it share a resource with a task of lower priority (*meteo_task*). The priority inversion brings about an abnormal blocking time of the *data_distribution* task which finally leads to a violation of deadline.

The system failure experienced during the Mars Pathfinder comes from a design error due to a lack of analysis during early-stages design activities. We show in the next section how this system can be designed correctly by combining architectural models with systematic analyses.

VII.2.3 Application of our approach

We apply our approach to design the software architecture of the Mars Pathfinder system. We model the system with the help of AADL on the one hand, and systematically analyze these models on the other hand.

Analysis repository. We consider the following analyses:

- schedulability tests: *ll_rm_test* [127] and *srl_pcp_test* [148],
- a schedule simulator: *cheddar_simu* [8],
- several analyses to check preconditions: *ll_context*, *srl_pcp_context* and *cheddar_simu_context*.

Table VII.6 summarizes the preconditions of the various analyses. Figure VII.8 depicts the precedences between analyses, computed from contracts.

Analysis **Precondition**	**ll_rm_test**	**cheddar_simu**	**srl_pcp_test**
mono-processor	✓	✓	✓
periodic tasks	✓	*N.R*	✓
offsets	*N.R.*	*N.R.*	*N.R.*
jitters	✗	*N.R.*	✗
implicit deadlines	✓	*N.R.*	✓
fixed computation times	✓	✓	✓
dependent tasks	✗	*N.R.*	✓
self-suspension	✗	✗	✗
preemption	✓	*N.R.*	✓
overheads	✗	✗	✗
scheduling algorithm	*RM*	*N.R.*	*RM*
concurrency control protocol	*N.A.*	*N.R*	*PCP*

Table VII.6: Analysis preconditions for the Mars Pathfinder case study. ✓*: the predicate must be true.* ✗*: the predicate must be false. Otherwise, the expected condition is stated explicitly.* N.A.*=not applicable,* N.R.*=no restriction.*

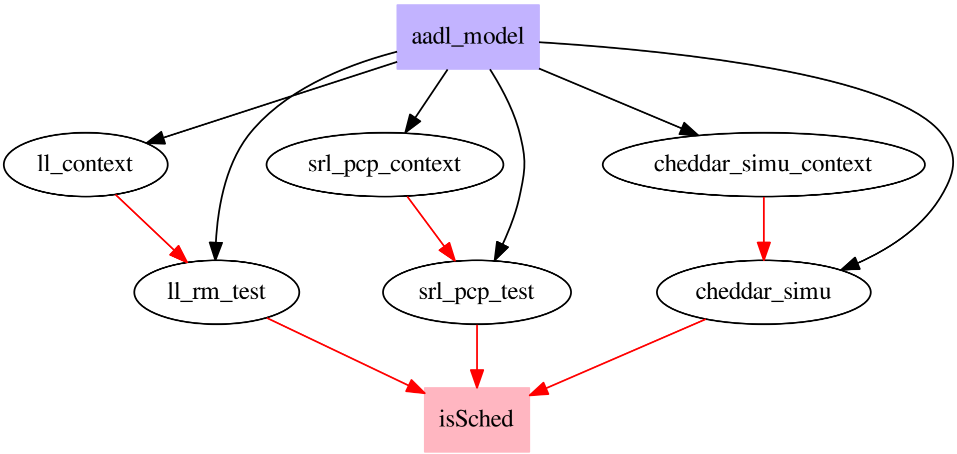


Figure VII.8: Analysis graph for the Mars Pathfinder case study.

Analysis of the original model. We firstly consider a faulty AADL model that would lead to the execution error and final system failure that we explained in the previous Section VII.2.2.

The tool visits the graph from Figure VII.8 as presented in Figure VII.9. It firstly checks the various preconditions with the following analyses: ❶ *ll_context*, ❷ *srl_pcp_context* and ❸ *cheddar_simu_context*. The results of the *ll_context* and *srl_pcp_context* analyses are *false*: on the one hand the tasks are not independent, on the other hand no protocol is defined to access the shared resources. Thus, we cannot execute the *ll_rm_test* and *srl_pcp_test*. On the contrary, the result of the *cheddar_simu_context* analysis is *true*; therefore we apply the ❹ *cheddar_simu*.

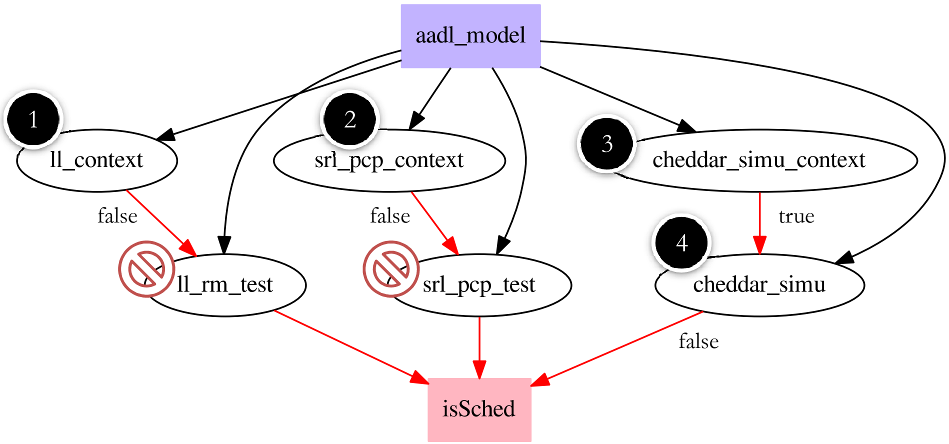


Figure VII.9: Analysis process performed from the original AADL model of the Mars Pathfinder system.

The simulation is carried out with the Cheddar tool. We observe from the result schedule displayed in Figure VII.10 that the third job of the *data_distribution* task does not complete before its deadline at time 15 (i.e. $375ms$). This violation of deadline comes from a priority inversion phenomenon as explained in Section VII.2.2.

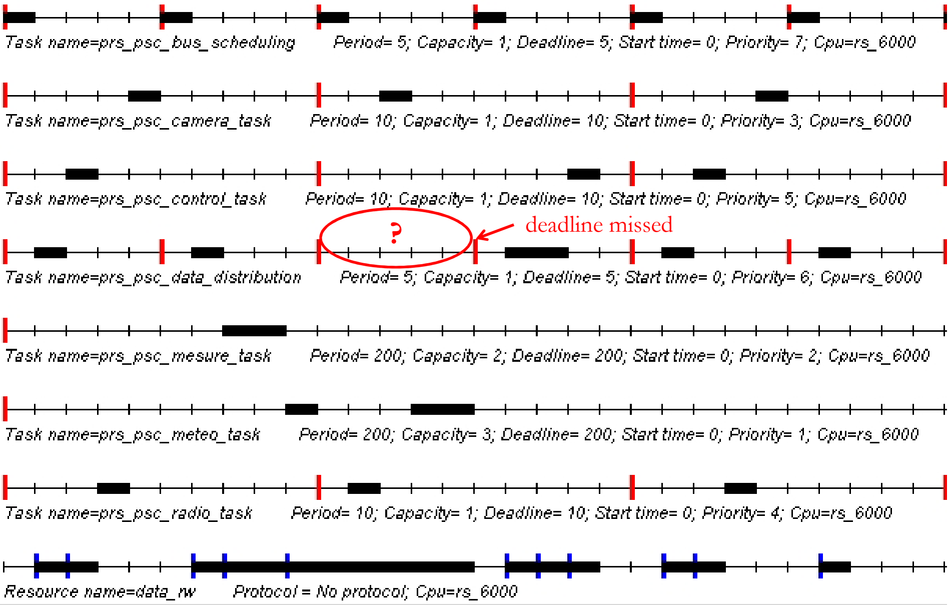


Figure VII.10: Simulation of an invalid schedule of the Mars Pathfinder task set computed with Cheddar (*cheddar_simu*).

Correction. We propose to implement a dedicated protocol called Priority Ceiling Protocol (PCP) in order to handle concurrent access to the shared resource. This protocol enables to avoid priority inversions and also prevent from blocking the system due to mutual exclusions (i.e. deadlocks).

Listing VII.3 shows a `sys_mars_pathfinder.correct` extension of the initial `sys_mars_pathfinder.impl` AADL model. This corrective specifies the aforementioned `Priority_Ceiling` protocol as a specific property of the `prs_PSC.data_rw` resource.

```
system implementation sys_mars_pathfinder.correct
   extends sys_mars_pathfinder.impl
properties
  Concurrency_Control_Protocol => Priority_Ceiling applies to
      prs_PSC.data_rw;
end sys_mars_pathfinder.correct;
```

Listing VII.3: Extension and correction of the original AADL model of the Mars Pathfinder system.

Validation. We finally analyze the corrected AADL model. Figure VII.11 summarizes the analysis process at the second design iteration. We check the analysis preconditions first, through the ❶ *ll_context*, ❷ *srl_pcp_context* and ❸ *cheddar_simu_context* analyses. The result of the *ll_context* analysis is negative because the tasks are dependent: we must not use the *ll_rm_test*. According to the result of the *srl_pcp_context* which is *true*, we can execute now execute the ❹ (a) *srl_pcp_test*. Indeed, the corrected model specifies a protocol to access the shared resource (the Priority Ceiling Protocol). Alternatively, the ❹ (b) *cheddar_simu* is still applicable as the result of the *cheddar_simu_context* analysis remains *true*.

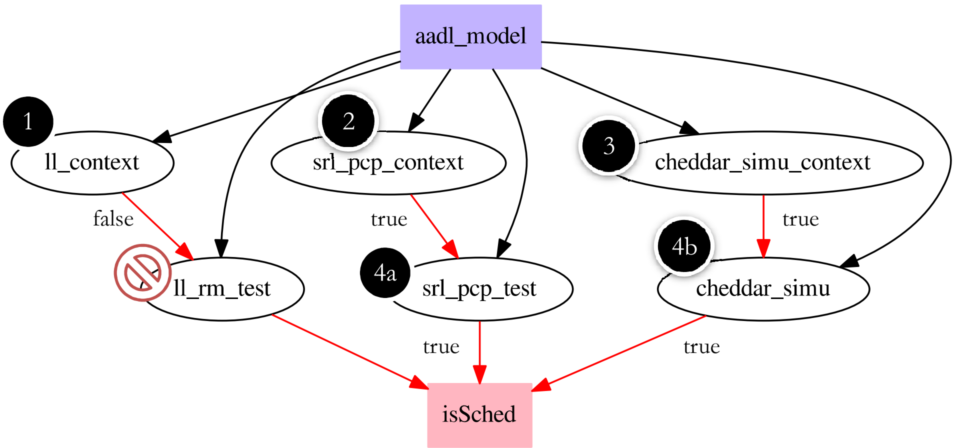


Figure VII.11: Analysis process performed from the corrected AADL model of the Mars Pathfinder system.

The *srl_pcp_test* checks that the amount of processor time needed to execute the tasks is acceptable (in other words, the actual processor utilization factor must be under a specific threshold to make sure that the task set is schedulable under a given algorithm). Unlike the test by Liu and Layland [127], the processor utilization factor that is computed with the test by Sha et al. [148] takes into account the time that each task may be blocked when attempting to access a shared resource. The result of the *srl_pcp_test*, computed from our tool, is displayed in Listing VII.4. As the calculated utilization factor is under the acceptable limit, the system is schedulable (sufficient condition), meaning that all the tasks will meet their deadlines at run time. Cheddar also simulates a valid schedule. However, a valid simulation is only a necessary condition, in contrast to the test by Sha et al. [148] that provides a sufficient condition.

```
$ python main.py
[...]
Execute SRL-PCP-test (theorem16)...
SRL-PCP-test is satisfied , U=0.725420 <= 0.728627 -> the tasks set is
    schedulable!
```

Listing VII.4: Result of the *srl_pcp_test* computed via our tool.

VII.2.4 Conclusion

This case study showed that analyses are of paramount importance to design an embedded system. Indeed, this is a design error in the software architecture that caused a significant failure of the system used during the Mars Pathfinder mission. Although it could have been fixed at design time, this error was very difficult to detect at that time given the lack of an holistic modeling/analysis approach and associated tools. Thus, the early design error came undetected before system operation and caused the system to shut down.

We showed that our approach was suitable to resolve this problem. First, our tool detects the interdependences between the analyses. This information is important to build a correct sequence of analyses. Next, the tool executes the analyses according to the input model (an AADL model representing the Pathfinder system in this example) and expected results (here, the goal was to verify the schedulability of the system).

We saw that the analysis process changes with the input model and analysis results (in particular the preconditions). We were able to firstly select an appropriate analysis for each AADL model, and then analyze the models to correct or validate them. That way, we were able to detect the original design error of the Mars Pathfinder system, propose a corrective and finally validate the corrective.

VII.3 Design space exploration of an avionic system

In this section, we deal with the design space exploration of an avionic system. First, we give an overview of the system with a functional description and a brief presentation of the target platform called Integrated Modular Avionics (IMA). Next, we combine two architecture description languages, AADL and CPAL, to model the various aspects of the avionic system. Last, we apply our approach to automatically analyze timing properties from the architectural models. We show that the systematic analysis of the architectural models enables to explore the design space of the embedded system.

VII.3.1 System overview

Firstly, we present the avionic system that we study in this section. Secondly, we give an introduction to the Integrated Modular Avionics (IMA) platform that hosts the avionic embedded system.

VII.3.1.A Avionic system

The avionic system comprises a Flight Management System (FMS) [175, 174] and a Flight Control System (FCS) [110, 201].

Flight Management System. The primary task of a Flight Management System (FMS) is in-flight management of the flight plan. The Flight Management System uses values measured from various sensors to compute the flight plan in flight and guide the aircraft. The crew interacts with the FMS by means of a Multi-Function Control and Display Unit (MCDU).

Figure VII.12 portrays the functional architecture of the Flight Management System. This system is made up of five main functions. The *Keyboard and cursor control Unit (KU)* handles requests from the crew while the *Multi Functional Display (MFD)* displays data from the flight plan such as *waypoints* or the *Estimated Time of Arrival*. The *Flight Manager (FM)* computes the flight plan by querying static data (waypoints, airways, etc.) from the *Navigation Data Base (NDB)* and dynamic data (altitude, speeds, position, etc.) from the *Air Data Inertial Reference Unit (ADIRU)*.

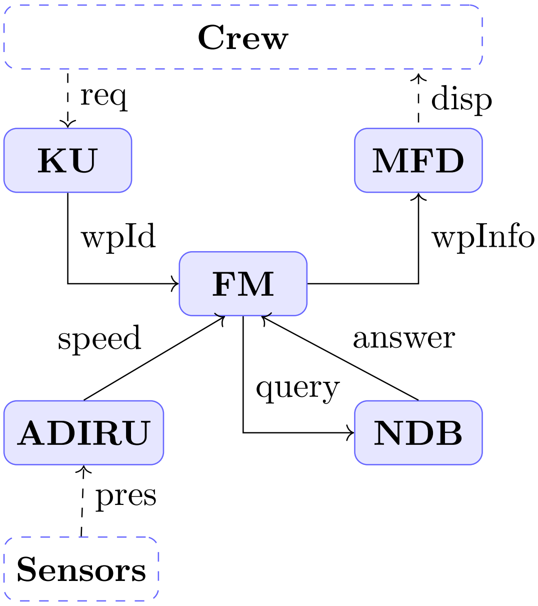


Figure VII.12: Functional architecture of the flight management system. *The functional architecture depicts the set of functions and the data flow among the functions.*

Flight Control System. The Flight Management System also interfaces with several other avionic systems in order to accomplish these functions. Figure VII.13 shows the connection between the Flight Management System and the Flight Control System (FCS). The aim of this system is to control the altitude, the speed and the trajectory of the aircraft from the flight plan [110]. In this section, we use the functional architecture coming from the ROSACE (Research Open-Source Avionics and Control Engineering) case study [201].

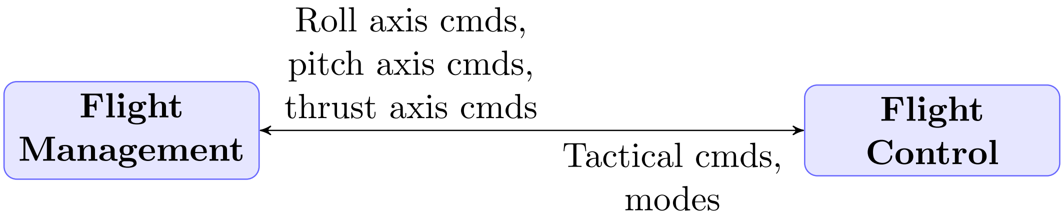


Figure VII.13: Interface between the Flight Management System and the Flight Control System.

VII.3.1.B Integrated Modular Avionics platform

The *functions* are to be stored and executed an *Integrated Modular Avionics (IMA)* platform. The IMA defines the use of the hardware and software resources through two standards:

- the ARINC 653 [202] for computational resources,
- the ARINC 664 (part 7) [203] for communication resources.

One particular objective of the IMA is to ensure timing predictability. In the following, we review some important concepts of its core standards. This description emphasizes on the parameters that are to be analyzed later on in this section.

Calculators – ARINC 653. The ARINC 653 is a standard to share processing and memory resources between several functions in a hardware *module*, or calculator. According to the ARINC 653, each function is to be hosted in a specific *partition* with a strict access to processing and memory resources:

- *temporal* partitioning ensures that partitions are executed during specific time slots defined at system start-up,
- *spatial* partitioning guarantees that each partition has a reserved memory space defined at system start-up.

Hence, an ARINC 653 schedule is both static and cyclic. Partitions are scheduled according to several parameters:

- at module level: a *major time frame* is defined for each module (MAF_m); possibly, a minor cycle can also be defined (MIF_m).

- at partition level: an *offset* (O_{mp}) that is the delay between the MAF_m origin and the start of the partition execution; and a *duration* (D_{mp}) that is the time allocated to each partition to access the processor.

Each partition is planned one or several times during the major cycle. This major cycle is then repeated indefinitely. In a partition, a function is realized through one or several processes. These processes are scheduled at the partition level according to a specific scheduling algorithm (e.g. FIFO or NP-FP).

Networks – ARINC 664. The ARINC 664 standard defines a predictable communication network called *Avionics Full Duplex-Switched Ethernet* (AFDX). It uses full-duplex links to convey the packets and switches to route a packet from a source to one or several sink(s). AFDX implements the core concept of *Virtual Link* (VL) to share the network bandwidth between the data flow. A VL is a unidirectional logical connection from one sender to one or several receiver(s) (i.e. unicast or multicast VLs). In particular, each VL has:

- a limited bandwidth (ρ_v) according to two parameters: the *Bandwidth Allocation Gap* (bag_v) that is the minimum time interval between two frames sending; and the *maximal allowed packet size* ($smax_v$); $\rho_v = \frac{smax_v}{bag_v}$,
- a predefined and static *route* ($route_v$) crossing one or several switch(es).

VII.3.2 Co-modeling with AADL and CPAL

We model the various aspects of the avionic system with two Architecture Description Languages: AADL and CPAL.

Operational architecture in AADL. We represent the highest-level operational architecture of the avionic system with AADL. Initially, only the Flight Management System (FMS) is represented. The model uses AADLv2 core specifications and the ARINC653 Annex [68]. Figure VII.14 shows the graphical view of the model. The model represents four ARINC653 calculators to host the avionic functions connected through an AFDX network[1].

The model follows the initial specifications and AADL design patterns for ARINC653 systems: a module is a distinct `system` (containing a global `memory` and a `processor`) that hosts partitions (each is a `process`) bound to separate `memory` segments and `virtual processors` (representing spatial and temporal partitioning). `thread` components contained in partitions realize the avionic functions. Thanks to annex guidelines, we can model precisely the ARINC653 components and associated parameters (modules Major Frames, partition duration, partition scheduling policies, etc.).

AADL does not provide specific guidelines for modeling AFDX networks. The AADL concept of `virtual bus` defines a connection supported in a `bus`. We use this concept to define AFDX virtual links. Switches are represented by `device` components

[1] The full AADLv2 textual model is part of the AADLib project, see `http://www.openaadl.org` for more details.

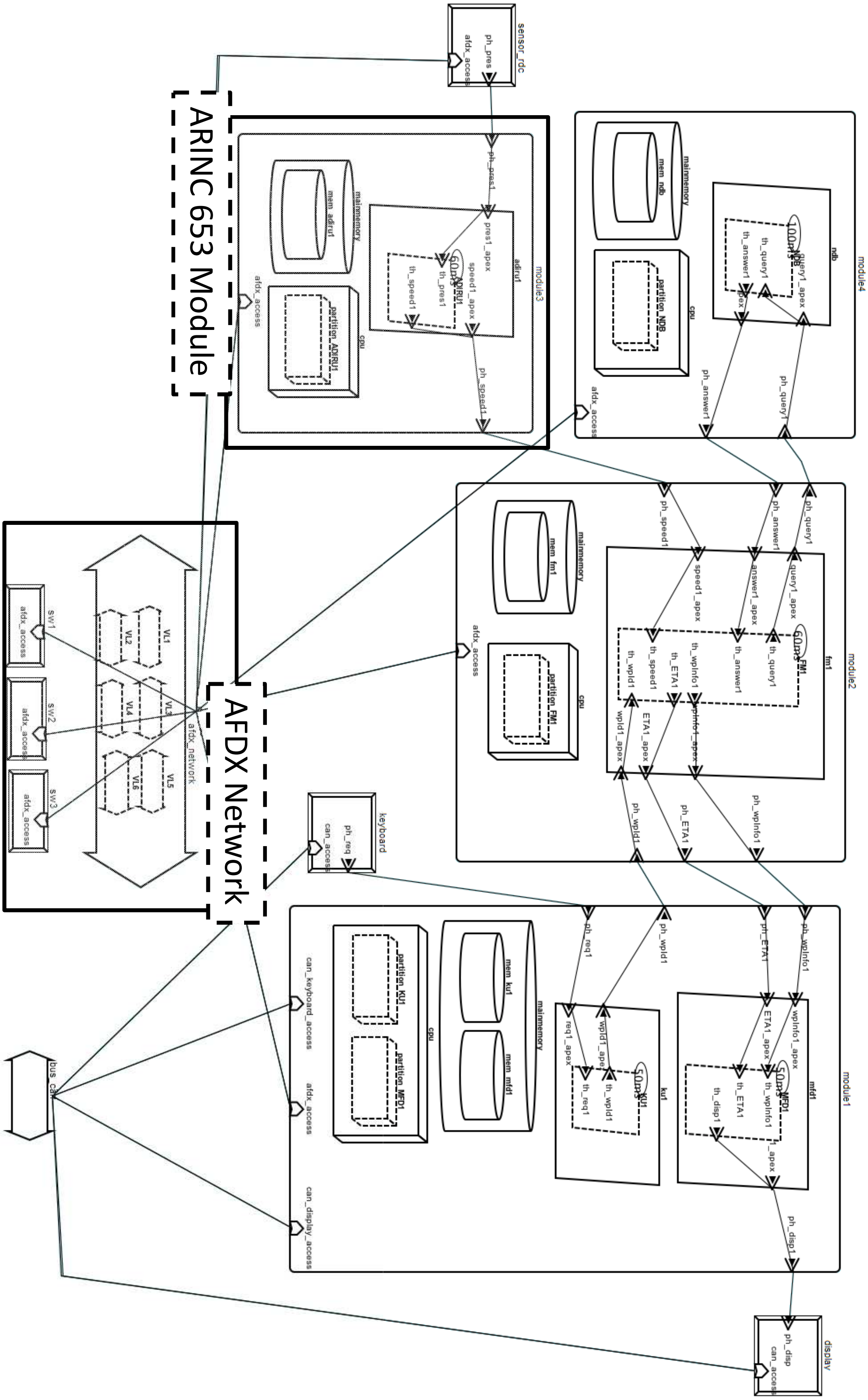


Figure VII.14: Overview of the operational architecture of the Flight Management System in AADLv2. *AADL components represent the ARINC653 calculators and the AFDX network.*

bound to the virtual links. A dedicated property set has been defined to model parameters attached to virtual links, end systems and switches.

FCS processes in CPAL. A functional description of the calculators completes the highest-level operational architecture. For example, we model the functions (i.e. processes) of the Flight Control System (FCS) with the CPAL language (the CPAL models of the FCS come from [204]).

Figure VII.15 shows the functional architecture of the FCS in CPAL graphical syntax. The functional architecture represents the processes, their activation scheme and the data flow between them. For instance, the process `az_filter` executes at a rate of 100Hz (i.e. $T_{az\_filter} = 10ms$). It computes an output variable `az_meas` used by another process named `vz_controller` from input variables `Az_Filter_Conf` and `az` .

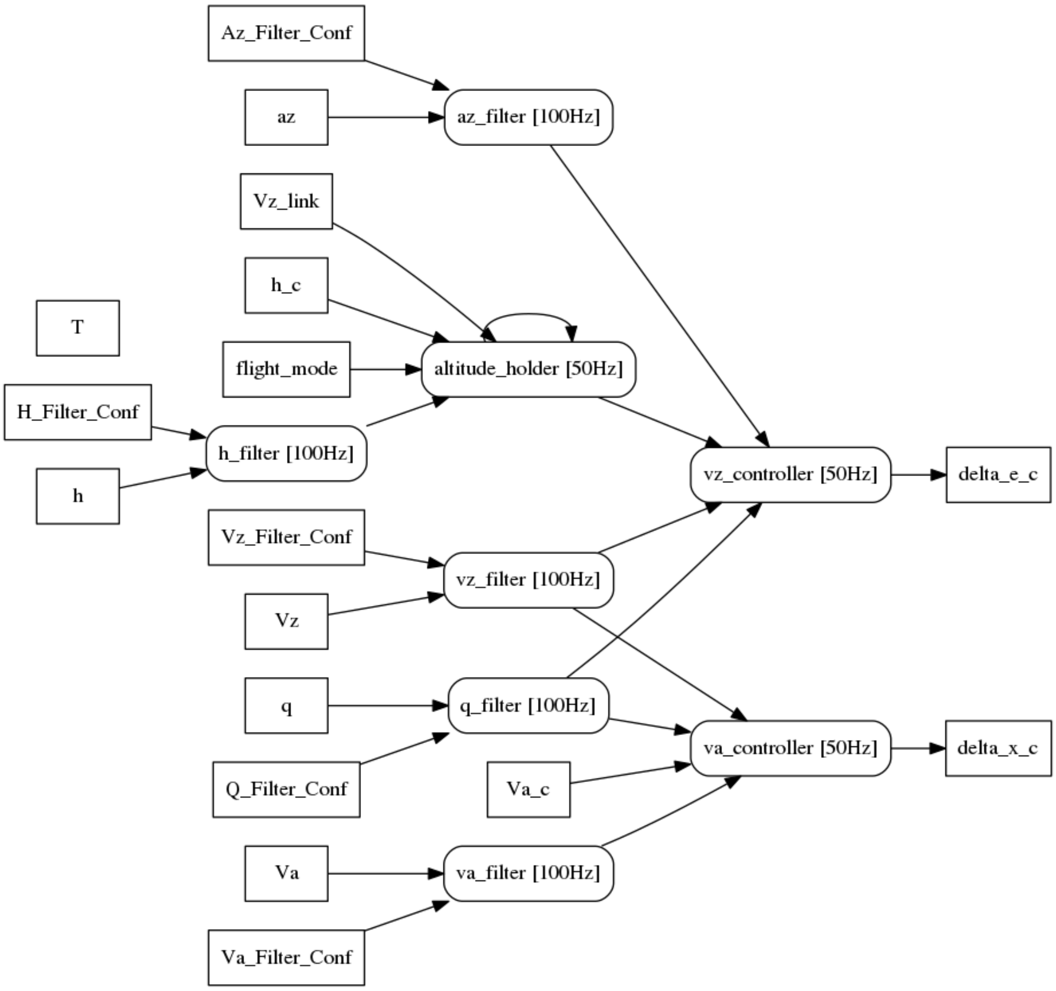


Figure VII.15: Functional architecture of the flight controller in CPAL.

In addition, the CPAL model describes the logic of each process with a Finite-State Machine (FSM). For example, the states of the FSM in Figure VII.16 implements two distinct running modes of the `altitude_holder` process: `Manual` and `Auto`. The operations in each state are specified in a textual syntax close to the syntax of the C language, e.g. `Altitude_Holder` process in Listing VII.5.

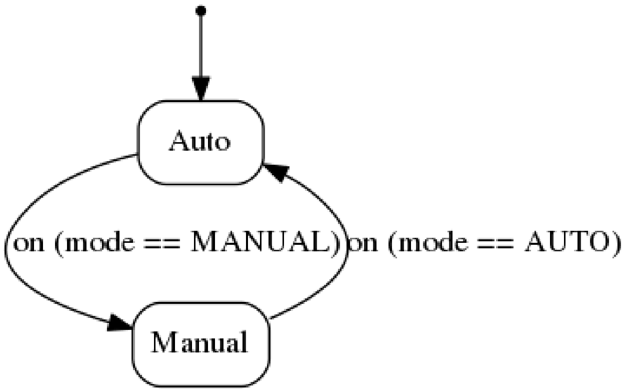


Figure VII.16: Logic of the `altitude_holder` process defined as a Finite-State Machine.

VII.3.3 Problem: exploration of the design space

An architectural model captures different facets of a system. For instance, we used AADL together with CPAL to represent three aspects of the FMS as shown in Figure VII.17: the functions, the IMA platform that implements the functions and the non-functional properties to comply with. We observe that the modeling views depicted in Figure VII.17 are interdependent:

Allocation. The functional architecture must be allocated to the hardware architecture. The operational architecture maps the functions and variables to the IMA platform. For example:

- we fix the cyclic frame of the modules (MAF_m and MIF_m) according to the periods of the functions (T_f): MAFs and MIFs are the *lcm* (least common multiple) of the periods and the shorter period respectively,
- we define the duration of a module partition depending on the related function execution time: $D_{mp} \geq C_f$,
- we set the parameters of the virtual links (bag_v and $smax_v$) from the number of messages to be sent by the linked function (n_f), and the maximum size of the messages that can be sent by this function (m_f).

Compliance with non-functional constraints. In addition, the operational architecture has to fulfill non-functional constraints. For instance:

- response time is the time needed to realize an activity,
- traversal times are communication delays between functions,

```

processdef Altitude_Holder(
  in Flight_Mode: mode,
  in float64: Vz_input,
  in float64: Vz_link,
  in float64: h_f,
  in float64: h_input,
  out float64: y
)
{
  static var float64: integrator = 532.2730285;

  state Auto {
    var float64: error = h_f - h_input;

    if (error < -50.0) {
      y = Vz_link;
    } else if (error > 50.0) {

      y = -Vz_link;
    } else {
      /* Output */
      y = Kp_h * error + Ki_h * integrator;
      /* state */
      integrator = integrator + (float64.as(self.period) / float64
          .as(1s)) * error;
    }
  }
  on (mode == MANUAL) to Manual;

[...]

}
```

Listing VII.5: Textual description of the `altitude_holder` process.

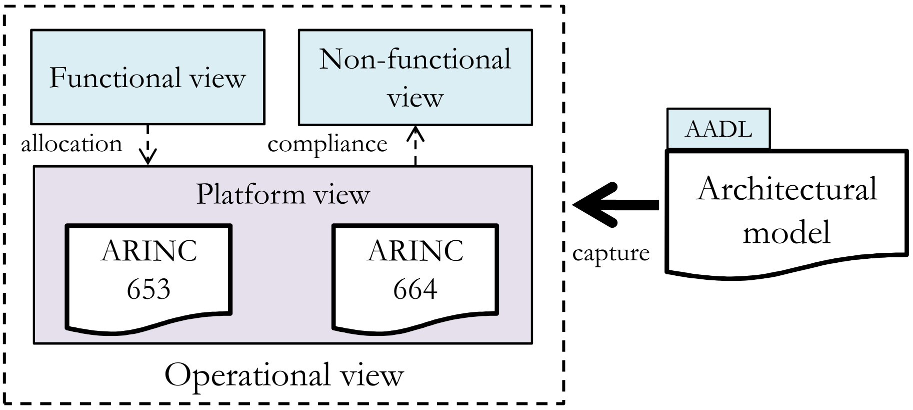


Figure VII.17: Different Views captured in an architectural model *An AADL model represents the functions, the IMA platform that implements the functions and the non-functional properties to comply with.*

- end-to-end latencies encompass response times and traversal times.

One must take these constraints into account when defining the architecture:

- the parameters of calculators (scheduling policies, execution times, etc.) impact the response times,
- the configuration of the AFDX network (VLs parameters, topology and routing strategies) influences the traversal times,
- the interaction between the platform components (calculators, networks) causes latencies along functional chains.

Towards exploration of the design space. The problem is hence to explore potentially large design spaces that integrate multiple interrelated views, e.g. functional aspects, platform concerns, non-functional constraints. We show in the following that the automatic analysis of architectural models enables to explore and evaluate many different design proposals. In particular, we explain how to dimension some important platform parameters from a functional description of the system, and fulfill the timing constraints.

VII.3.4 Application of our approach

We apply our approach to explore design proposals and evaluate them. We apply a systematic analysis approach based on the AADL and CPAL models presented in Section VII.3.2. In particular, we dimension several parameters of the avionic system in order to meet the real-time constraints expressed at tasks and networks levels.

VII.3.4.A Analysis repository

We set up the analysis graph in Figure VII.18 from contracts. The graph depicts the analysis process that will enable us to check that the avionic system represented with AADL and CPAL models (respectively aadl_model and cpal_model nodes in the graph) respects the timing constraints (isSched node in the graph). See Section VII.3.2 for a presentation of the AADL and CPAL models.

The analysis graph comprises two analysis flows that run separately at the beginning of the process and then converge towards the same goal:

(1) the left-hand analysis flow, starting from the aadl_model, includes several analyses in order to iteratively define parameters of the AFDX network and finally validate them;

(2) the right-hand analysis flow, starting from the cpal_model, enables to check the schedulability of the tasks described in CPAL models, which are part of the ARINC653 processes to be represented in AADL;

(3) the distinct flows meet at the arinc653_dimensioning analysis. First, we define the ARINC653 parameters in the AADL model from tasks parameters defined in the CPAL model. Then, we validate the ARINC653 parameters.

We explain the various analysis flows in greater depth, providing experimental results, in the following sections.

VII.3.4.B From the analysis of CPAL processes to the dimensioning of ARINC 653 modules

This first experimentation aim at fully validating the timing behavior of the software, that is to verify that all the processes will meet their deadlines at run time. For this purpose, we need to dimension a new ARINC 653 module for the Flight Control System.

❶ WCET analysis and ❷ (b) simulation . A CPAL model can be simulated so as to evaluate the timing behavior of the software. The CPAL simulator uses the following data:

- the scheduling algorithm which can be FIFO, NP-FP or NP-EDF in a CPAL model,
- the task activation model that basically consists of few tasks parameters, e.g. periods and offsets,
- timing annotations that may be execution times, jitters, priorities or deadlines.

The processes execute in zero time when the code is not annotated. Timing annotations defined within a @cpal:time block specifies the timing behavior that must have a CPAL program at run time. In the first place (step A), we measure the WCET experienced by the processes on several target platforms with the help of the CPAL-interpreter option --stats. Next (step B), we inject the measured WCET as timing annotations in the CPAL model in order to make the simulation more accurate.

Table VII.7 and Table VII.8 display the WCET measured on two execution platforms:

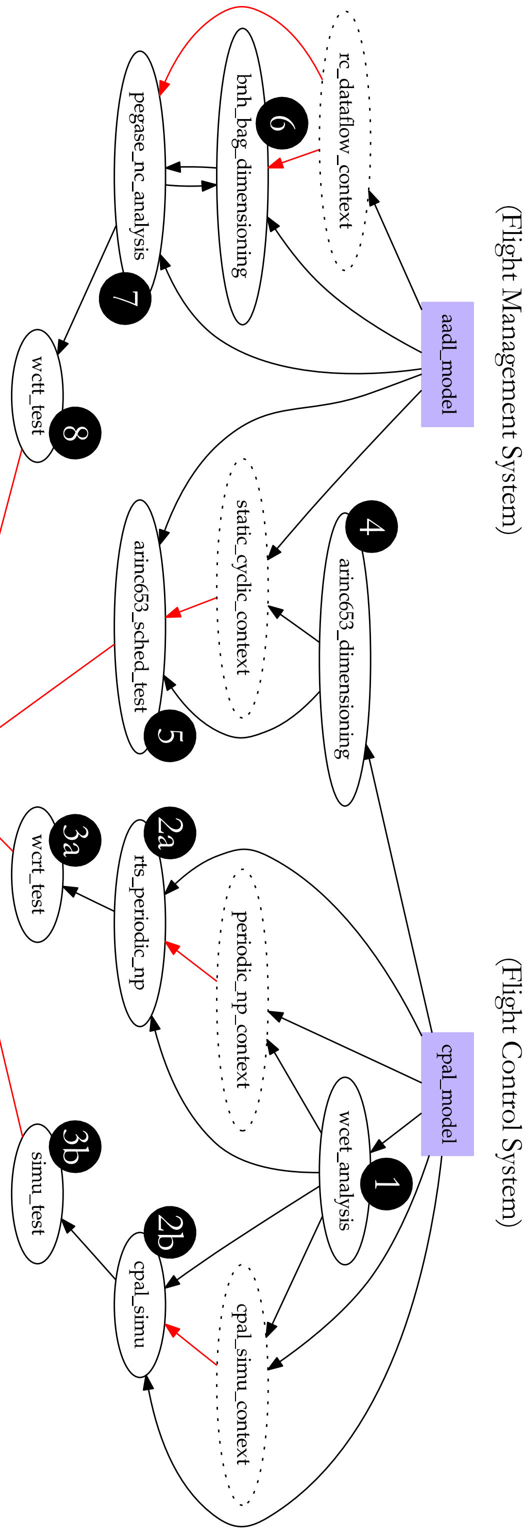


Figure VII.18: Analysis graph for the avionic case study.

- *Embedded Linux 64-bit*: a laptop with a processor Intel Core i7-4710HQ @2,50GHz (4 cores), 7895 MiB of RAM, and running under Ubuntu 14.10 operating system,
- *Raspberry Pi*: a single-board embedded computer Raspberry Pi 2 - Model B V1.1 with a processor ARM Cortex-A7 (Broadcom BCM2836) @900MHz (4 cores), 1 GiB of RAM, and running under Raspbian operating system.

Process	**WCET (µs)**		
	Vertical Speed	**Airspeed**	**Climb**
va_filter	298.961	71.177	39.989
vz_filter	218.330	70.387	103.836
q_filter	131.875	29.189	70.725
az_filter	55.561	71.162	43.751
h_filter	298.590	69.999	110.573
altitude_holder	43.108	70.526	74.800
vz_controller	207.780	270.470	123.423
va_controller	170.519	1326.751	32.260

Table VII.7: WCET measured on an Embedded Linux platform (*wcet_analysis*).

Process	**WCET (µs)**		
	Vertical Speed	**Airspeed**	**Climb**
va_filter	498.210	241.769	259.894
vz_filter	188.797	252.915	192.916
q_filter	440.518	218.801	209.739
az_filter	3402.323	371.920	190.832
h_filter	543.221	303.957	238.227
altitude_holder	162.448	164.531	262.551
vz_controller	194.634	263.957	216.561
va_controller	208.125	232.967	241.405

Table VII.8: WCET measured on a Raspberry Pi platform (*wcet_analysis*).

Figure VII.19 displays the timing simulation of the CPAL model of the flight controller in the *Vertical Speed* scenario. The bars represent process activation according to the periods and offsets (which are null). The processes are scheduled according to a FIFO (First-In First-Out) policy, i.e. the processes are executed in the exact order of their activation. The width of the bars represents process execution times.

We observe from the simulation result in Figure VII.19 that the schedule fulfills the timing constraints: (1) the process activation respects the periods; (2) only one process is scheduled on the processor at every time; (3) all the processes complete before their deadlines, i.e. before the activation of the next job.

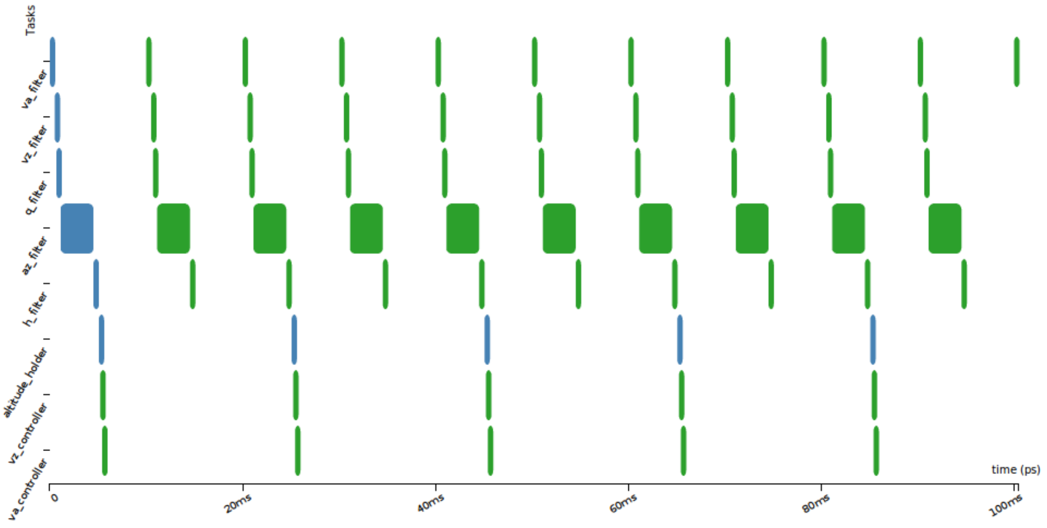


Figure VII.19: Timing simulation of the flight controller (*cpal_simu*) under FIFO scheduling in the $VerticalSpeed$ scenario.

❷ (a) Scheduling analysis. Static scheduling analyses (i.e. schedulability tests) are in general safer than a simulation. Indeed, the simulation of a valid schedule is usually a necessary condition while schedulability tests provide sufficient and, possibly, necessary conditions. We evaluate the task response times from the CPAL model with the help of the `TkRTS` tool. Table VII.9 shows the worst-case response times (i.e. *bound*) under NP-FP scheduling in the $Airspeed$ scenario. Table VII.10 displays the worst-case response times under NP-EDF scheduling in the $Climb$ scenario.

Algorithm	Task	C (ns)	T (ns)	D (ns)	*bound* (ns)	*laxity* (ns)
np-fp	`altitude_holder`	164531	20000000	20000000	2049989	17950011
	`va_controller`	232967	20000000	20000000	2049989	17950011
	`vz_controller`	263957	20000000	20000000	1885458	18114542
	`va_filter`	241769	10000000	10000000	1652491	8347509
	`h_filter`	303957	10000000	10000000	1410722	8589278
	`az_filter`	371092	10000000	10000000	1146765	8853235
	`q_filter`	218801	10000000	10000000	842808	9157192
	`vz_filter`	252915	10000000	10000000	624007	9375993

Table VII.9: Worst-case response times computed by the *rts_periodic_np* analysis under NP-FP scheduling in the $Airspeed$ scenario.

The results are conclusive in the two scenarios. Every calculated worst-case response times *bound* is less than its related deadline D. Thus, every laxity, which is the remaining time to deadline, is positive. Therefore, the task set is schedulable in the $Airspeed$ scenario, resp. $Climb$ scenario, according to the NP-FP scheduling algorithm, resp. NP-EDF scheduling algorithm.

❹ dimensioning ARINC653 partitions and ❺ validation. From a validated schedule of the FCS processes, we can now safely set up an ARINC 653 module M_5 to host these processes.

Algorithm	Task	C (ns)	T (ns)	D (ns)	*bound* (ns)	*laxity* (ns)
np-edf	`va_controller`	241405	20000000	20000000	1812125	18187875
	`vz_controller`	216561	20000000	20000000	1812125	18187875
	`altitude_holder`	262551	20000000	20000000	1812125	18187875
	`h_filter`	238227	10000000	10000000	1354158	8645842
	`az_filter`	190832	10000000	10000000	1354158	8645842
	`q_filter`	209739	10000000	10000000	1354158	8645842
	`vz_filter`	192916	10000000	10000000	1354158	8645842
	`va_filter`	259894	10000000	10000000	1354158	8645842

Table VII.10: Worst-case response times computed by the *rts_periodic_np* analysis under NP-EDF scheduling in the *Climb* scenario.

The simplest approach is actually to define a unique partition for all the processes. We can simply dimension this partition from the parameters of the processes:

- the MAF_5 is equal to the least common multiple of the process periods,
- a different MIF_5 is not necessary as there is only one partition, hence $MIF_5 = MAF_5$,
- the duration to execute the single partition is $D_{51} = MAF_5$

In this particular case, the scheduling analysis is quite trivial as there is only one partition and the MAF is set to the hyperperiod of the processes. Figure VII.21 depicts a schedule of the FCS partitions and processes. The MAFs depict the repetition of the major cycle. A unique partition is scheduled during this major cycle, as represented with red rectangles. We note that the duration of the partition is equal to the MAF. Finally, the CPAL processes are scheduled within the partition according to a FIFO algorithm.

If we choose a different partitioning of the processes (i.e. by assigning processes to different partitions), we must use a specialized scheduling analysis. In fact, the global schedule encompasses two hierarchical levels, i.e. the partition level schedule and the process level schedule(s). For example, a compositional analysis methodology could be applied to determine whether the processes are schedulable [205].

VII.3.4.C Iterative definition of the Bandwidth Allocation Gap (BAG) from the AADL model

Let us consider an incomplete AADL model. Listing VII.7 partly depicts the system architecture: the modules that implement the functions and part of the AFDX network (the connections (variables) between the functions and the network devices). One problem at that level is to allocate the dataflow to network resources (e.g. Virtual Links) and define the routing strategy. In Listing VII.6, the problem is to define the properties of a Virtual Link (e.g. the Bandwidth Allocation Gap) that meet the latency constraints expressed on the dataflow.

At this stage, dimensioning the BAG can be a difficult task. According to [203], any BAG must be defined such that BAG= $2^k ms$ with $k \in \{1, 2, \ldots, 7\}$. If we

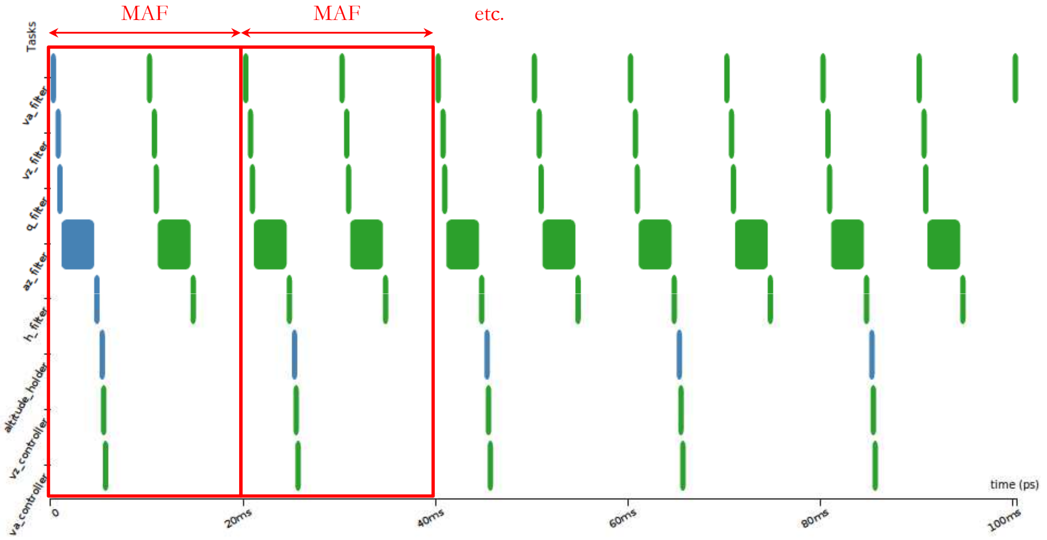


Figure VII.20

Figure VII.21: “Pen & paper” simulation of an ARINC 653 schedule (FCS hardware module, $VerticalSpeed$ scenario).

assume one VL per dataflow then the design space comprises 8^{α} solutions, with α the number of dataflow.

```
-- This subpart of the AADL model defines the Virtual Links

virtual bus VL
   properties
   -- generic parameters from the standard
   AFDX_Properties::AFDX_Frame_Size => AFDX_Properties::
       AFDX_Std_Frame_Size;
   AFDX_Properties::AFDX_Tx_Jitter => AFDX_Properties::
       AFDX_Std_Tx_Jitter;
end VL;

-- definition of a Virtual Link
virtual bus implementation VL.vl1
   properties
   -- we must define the properties to meet the latency
       constraints
   AFDX_properties::AFDX_Bandwidth_Allocation_Gap => 32 ms;
end VL.vl1;
```

Listing VII.6: Specification of a Virtual Link in AADL.

We visit the analysis graph in Figure VII.18. We use two analyses to define the BAG:

1. `bnh_bag_dimensioning` to define the suitable BAG for each VL in the network,
2. `pegase_nc_analysis` that relies on Network Calculus to compute upper bounds on communication delays (worst-case traversal times) in AFDX networks.

```
-- This AADL model represents a basic architecture of the Flight
    Management System

-- root system
system fms end fms;

-- system implementation = FMS architecture
system implementation fms.impl
   subcomponents
   -- ARINC653 modules
   module1 : system subsystem::m1_system.impl;
   module2 : system subsystem::m2_system.impl;
[...] -- other modules and devices

   -- AFDX components
   afdx_network : bus fms_hardware::physical_afdx_link.impl;
   sw1 : device subsystem::afdx_switch;
   sw2 : device subsystem::afdx_switch;
   sw3 : device subsystem::afdx_switch;

   -- we define the data flow with connections
   connections
   nt_wpId : port module1.ph_wpId1 -> module2.ph_wpId1;
[...] --other connections between modules

   flows
   wpId_fl : end to end flow module1.wpId_src ->
                   nt_wpId -> module2.wpId_sink ;
[...] -- other data flow: wpInfo, query, answer, etc.

   -- and we finally define the temporal constraints
   properties
   Latency => 0ms .. 15 ms applies to wpId_fl;
[...] -- other latency constraints

   -- one problem is to allocate the dataflow to Virtual Links
   -- for instance:
   Actual_connection_binding => (reference (afdx_network.VL1))
                   applies to nt_wpId;

   -- we must also define the routing strategy
end fms.impl;
```

Listing VII.7: Incomplete specification of the Flight Management System in AADL.

❻ **BAG dimensioning from latency evaluation.** We proposed in [166] an analysis to evaluate the latency experienced by any message in the AFDX network, including the delay in the end systems. The latency suffered by a message in the network is the sum of the delays experienced in each crossed element: from the source end system, through the successive switches, up to the sink end system(s). In few words, the formula of the Worst-Cased Latency Time ($WL_{n,v}$) suffered by the last frame of the message n in the VL v is:

$$WL_{n,v} = bag_v \times (p_{n,v} - 1 + \sum_{k=1}^{n-1} p_{k,v}) + \Big(lag + 2 \times \frac{smax_v}{BW} \times (1 + r_v) + jmax\Big) + D_{sw\_v} \tag{VII.1}$$

$$\text{with} \begin{cases} D_{sw\_v} & = \sum_{k=1}^{r_v} WSCL_{n,k} \\ lag & = 2 \times WETeL + r \times WSTeL \\ sub_v - 1 & = 1 \text{ (sub-vl are not considered)} \end{cases}$$

From that formula, we can calculate the BAG of each VL to meet the latency constraints expressed on the message LC_n (i.e. $WL_{n,v} \leq LC_n$):

$$bag_v \leq \frac{LC_n - Dsw - \big(lag + 2 \times \frac{smax_v}{BW} \times (1 + r_v) + jmax\big)}{p_{n,v} - 1 + \sum_{k=1}^{n-1} p_{k,v}} \tag{VII.2}$$

Thus, the model must provide several data to calculate the BAG:

- information about the messages: the maximal number of messages (nbr_f) that a function can send through a virtual link; the maximal size of each message (m_n); the latency constraint expressed on each message LC_n,
- AFDX-specific parameters defined in the standard: the bandwidth (BW), technological delays (lag) and a maximal transmission jitter in a end system ($jmax$).

We can do the following assumptions if the other data are not set in the model:

- one virtual link is allocated to each dataflow (i.e. set of messages sent by a function) with the same source/receiver(s) couple,
- the $smax_v$ is set to:
  - $smax_v = m_v + 67$ bytes if $m_v \leq 1471$ bytes,
  - its maximum value $smax_v = 1538$ bytes else,
- all the messages can be fragmented, that means that $p_{n,v} \geq 1$ with $p_{n,v} = \lceil \frac{m_n}{smax_v - 67} \rceil$,
- if the routing strategy is missing, we assume that there is one crossed switch per VL:
  $r_v = Card(route_v) = 1$,
- if unknown, the delay in the switches $D_{sw\_v} = 0$.

❼ **Network Calculus.** Network Calculus is a mathematical theory designed to compute worst case performances of networks [119]. The Network Calculus (NC) theory can be used to compute upper bounds on communication delays in AFDX networks. For example, the NC has been used to certify the AFDX network of the Airbus' A380 [206].

NC handles incoming *flows* expressed by an *arrival curve* $\alpha(t)$ and *server* elements offering a minimal service specified through a *service curve* $\beta(t)$. Given $\alpha(t)$ and $\beta(t)$, at time t, it is possible to estimate the *backlog* – the amount of bits held in the network element – and the *virtual delay* – the delay suffered by a bit to cross the element. The worst delay experienced by a flow in a server is given by the greatest horizontal deviation between the curves: $d = h(\alpha, \beta)$. Furthermore, in accordance with the input flows and the offered service expressions, the resulting output flow $\alpha^*(t)$ is given by $\alpha^*(t) = \alpha(t + d)$. Afterwards, it is possible to cascade the servers, *i.e.* to bind the output of a server to the input of another, in order to propagate the data flow along its route and to compute the end-to-end delay.

We can use the NC technique to calculate the delay in the switches D_{sw_v}. For this purpose, the model must detail the data needed to set the arrival curves belonging to each virtual link v and the service offered by the end systems e and switches s:

- $\alpha_v(t)$ depends on the bag_v and the $smax_v$,
- $\beta_e(t)$ and $\beta_s(t)$ depend on: $smax_v$, BW, lag and $jmax$.

We define $smax_v$, BW, lag and $jmax$ as for the BAG dimensioning. In addition to the VL parameters, the NC considers:

- the network topology made up of end systems, switches and links,
- the static routing table.

These data can either be part of the input model or we can assume them. In particular, we can combine data from the model with assumptions. This brings two advantages: (1) we can evaluate and possibly refine model parameters according to a virtual but realistic network configuration; (2) we can evaluate several routing strategies.

We carry out the NC analysis with the help of the RTaW-Pegase tool [121]. We also use NETAIRBENCH [207], an AFDX benchmark generator provided with the RTaW-Pegase tool. NETAIRBENCH makes it possible to generate realistic avionic data flow according to user-defined parameters. Figure VII.22 shows one instance generated by NETAIRBENCH where the FMS and the FCS are included in a network architecture of realistic size.

Iterative process. We execute the iterative process represented in Figure VII.23. We refine the model at each iteration ($m1$, $m2$ and $m3$) according to (1) analysis results (successive BAG_v and D_{sw_v}) and (2) modeling assumptions ($as1, as2$).

At $\boxed{\text{Step 1}}$, the `bnh_bag_dimensioning` analysis (BAG in Figure VII.23) inputs the incomplete model ($m1$, also represented in Listing VII.7) together with some assumptions ($as1$) discussed in the previous paragraphs (in particular, D_{sw_v} is unknown

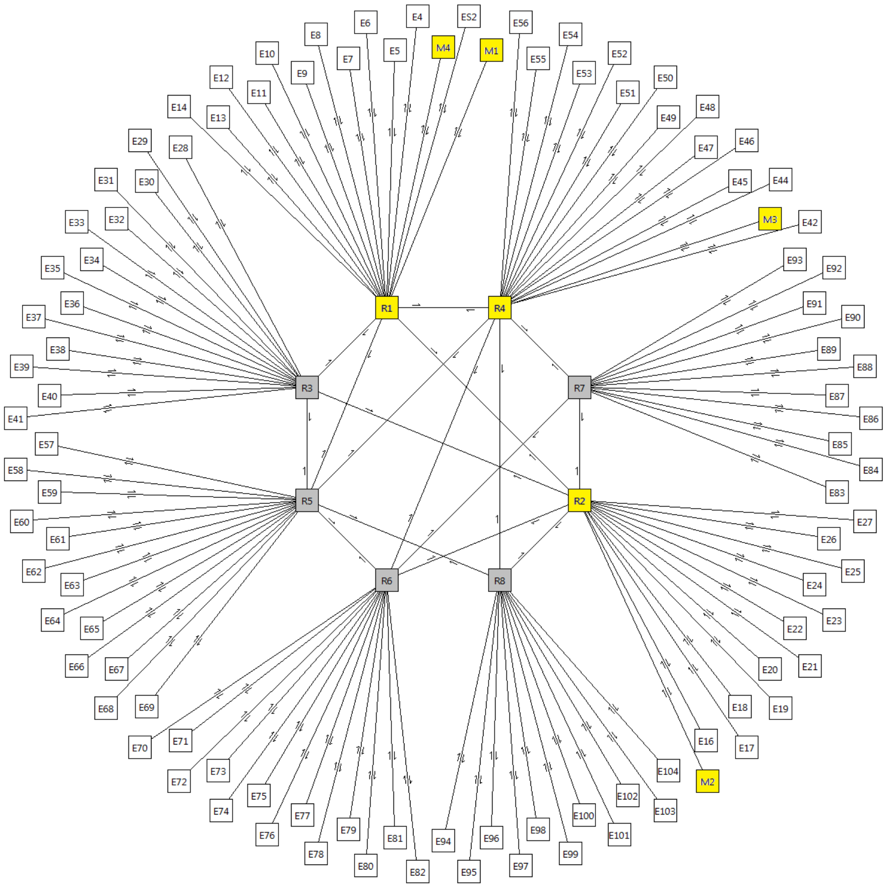


Figure VII.22: Realistic network architecture and background traffic generated by the NETAIRBENCH tool. *The topology and background traffic is generated from user-defined parameters. A typical AFDX topology can contain 100+ end systems and 8 switches. The AFDX switches form the central backbone. The FMS and FCS represented with yellow-colored components are included in the overall system generated by NETAIRBENCH.*

and thus assumed to be null). We define five VLs for the FMS, following the assumption ‘one virtual link per dataflow’. Table VII.11 summarizes the analysis results, that is the maximal BAGs that meet the latency constraints. Notice that this first coarse-grained analysis discards: $bag\ solutions\ for\ m1 - bag\ solutions\ for\ m2 = 8^5 - 1440 = 31328$ incorrect BAG solutions.

We execute the `pegase_nc_analysis` (NC in Figure VII.23) at $\boxed{\text{Step 2}}$. The NC analysis evaluates the upper delay suffered by each frame in a Virtual Link ($D^{m_2}_{sw\_vl_i}$) from the first evaluation of the BAG. We defined each BAG in $m2$ with the greatest value available in the range computed by the `bnh_bag_dimensioning`. In addition, we assume the topology computed by NETAIRBENCH depicted in Figure VII.22 with an average utilization of switch ports of 25% ($as2$). We also suppose a static *shortest path* routing. Table VII.12 details the analysis results ($D^{m_2}_{sw\_vl_i}$).

At $\boxed{\text{Step 3}}$, the `bnh_bag_dimensioning` analysis refines the BAG to meet the latency constraints ($LC^{m_1}_{vl_i}$) according to the delays computed by the `pegase_nc_analysis`. We narrow the set of correct BAGs (BAG^{m_3}) for all the VL excepted for VL_1 and

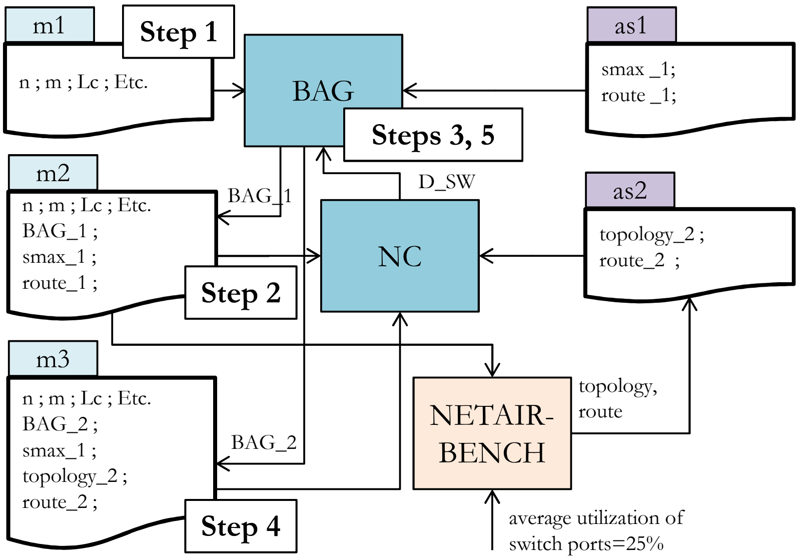


Figure VII.23: Bandwidth Allocation Gap dimensioning process *Dimensioning the BAG requires an iterative process of modeling and analysis, i.e. Steps 1 to 5. Model data used at the various steps are represented on the left in blue-headed shapes. Analyses are represented on the center in green and orange rectangles. Assumptions, which are represented on the right in purple-headed shapes, may be required at certain steps to move forward. Dashed arrows, resp. solid arrows, depict analysis inputs, resp. analysis results.*

VL_2: $bag\ solutions\ for\ m2 - bag\ solutions\ for\ m3 = 1440 - 720 = 720$ additional solutions do not meet the latency constraints.

At $\boxed{\text{Step 4}}$, we must calculate the delays suffered by the frames in the Virtual Links ($D^{m3}_{sw\_vl_i}$) with the `pegase_nc_analysis` according to the new definition of the BAGs ($BAG^{m3}_{vl_i}$); and then refine the BAGs sets ($BAG^{m4}_{vl_i}$) with the `bnh_bag_dimensioning` analysis if necessary ($\boxed{\text{Step 5}}$). This iteration from m_3 shows that: (1) the delays in a VLs do not evolve ($D^{m3}_{sw\_vl_i} = D^{m2}_{sw\_vl_i}$); thus, (2) it is not necessary to adjust the BAGs ($BAG^{m4}_{vl_i} = BAG^{m3}_{vl_i}$). We reach a fixed-point: the model m_3 cannot be refined anymore with respect to the Bandwidth Allocation Gap if the analysis data (input data and assumptions) remain identical.

Virtual Link	n^{m_1}	$s^{as_1}_{max}$ (bytes)	LC^{m_1} (ms)	$BAG^{m_2}_{max}$ (ms)	BAG^{m_2} (ms)
VL_1	2	142	15	14,27456	$\{1, 2, 4, 8\}$
VL_2	3	692	15	7,04928	$\{1, 2, 4\}$
VL_3	2	192	10	9,25856	$\{1, 2, 4, 8\}$
VL_4	2	567	35	34,13856	$\{1, 2, 4, 8, 16, 32\}$
VL_5	2	567	20	19,13856	$\{1, 2, 4, 8, 16\}$

Table VII.11: Results of the *bnh_bag_dimensioning* analysis at $\boxed{\text{Step 1}}$ *The analysis computes the set of suitable BAG from the input model m_1 and assumptions as_1. AFDX parameters (BW, lag, $jitter_{max}$) not appearing in the table are set according to the standard, i.e. $r^{as_1}_{vl_i} = 1$ and $D^{m_1}_{sw\_vl_i} = 0$*

Virtual Link	$s_{max}^{as_1}$ (bytes)	BAG^{m_2} (ms)	$D_{sw}^{m_2}$ (ms)	LC^{m_1}	r^{as_2}	BAG^{m_3} (ms)
VL_1	142	8	2,774	15	2	$\{1,2,4,8\}$
VL_2	692	4	2,922	15	2	$\{1,2,4\}$
VL_3	192	8	3,118	10	2	$\{1,2,4\}$
VL_4	567	32	2,774	35	2	$\{1,2,4,8,16\}$
VL_5	567	16	4,189	20	3	$\{1,2,4,8\}$

Table VII.12: Results of the *pegase_nc_analysis* at $\boxed{\text{Step 2}}$ and *bnh_bag_dimensioning* analysis at $\boxed{\text{Step 3}}$ *First, the NC analysis computes the upper bound on communication delays in each VL ($D_{sw\_vl_i}^{m_2}$) from the largest BAG calculated at the previous step, and maximal frame sizes. Then, the BAG analysis computes the set of BAG that meet the latency constraints LC. Apart from the number of crossed switches ($r_{vl_i}^{as_2}$), the other inputs remain identical.*

VII.3.5 Conclusion

This third case study dealt with the design of complex system: an avionic system composed of a Flight Management System (FMS) and a Flight Control System (FCS). Our design approach includes:

1. a description of the system architecture at different levels of abstraction: overall and operational architecture of the system in AADL, functional architecture of the applications in CPAL,
2. a repository of multiple analyses: WCET, scheduling, communication delays, various simulators, etc.
3. a tool that automatically executes analyses according to input models and analysis goals.

We have been able to explore the design space of the FMS/FCS from the systematic analysis of the architectural models. In particular, we defined several parameters of the ARINC653 calculators and the AFDX network in order to fulfill the real-time constraints.

VII.4 Synthesis and conclusion

In this chapter, we experimented the concepts contributed in this thesis (see chapters III, IV and V), and implemented through a tool prototype (see Chapter VI). We presented three case studies: the timing validation of the Paparazzi drone, the design of the Mars Pathfinder system, and the design of an avionic system an avionic system composed of a Flight Management System (FMS) and a Flight Control System (FCS).

These case studies highlight several use cases of our approach. Table VII.13 summarizes the use cases encountered in this chapter.

Interoperability: our approach separates models from accessors and from analyses. Therefore, analyses are independent of models; or, in other words, analyses can work

Case studies Use cases	Paparazzi	Pathfinder	FMS
Interoperability	✗	✗	✓
Interdependencies	✓	✓	✓
Context-aware analysis	✓	✓	✓
Iterative process	✗	✓	✓

Table VII.13: Use cases of our approach.

with any architectural model for which an implementation of accessors to model internals is provided. For example, we analyzed the avionic system equally from AADL or CPAL models.

Interdependencies between analyses: for each case study, we evaluate analysis contracts to initialize the analysis graph. This graph represents the data flow between analyses and, thereby, the precedences between analyses. The analysis graph is necessary to execute the analyses in a correct order (preserving results) or to build wider analyses (building results). We experimented the two cases in the case studies, for example when we check the preconditions before applying an analysis (preserving results); or, when data computed by an analysis are used by another (building results).

Context-aware analysis: our tool is able to adapt the analysis process depending on an input model, available analyses and some analysis goals. In the Paparazzi UAV case study, we were able to analyze the AADL models at different design stages with suitable schedulability analyses in order to verify timing constraints throughout the design process. In the Mars Pathfinder case study, we automatically chose schedulability analyses depending on a AADL model. Thereby, we were able to select an appropriate analysis to detect the design error that caused an important failure of the system during the Mars Pathfinder mission.

Iterative process: more generally, we can apply an automatic or semi-automatic design process that takes into account three parameters: a set of system models, a repository of multiple analyses and goals in terms of non-function requirements. Analysis becomes an integral part of the design process as depicted in Figure VII.24:

- analyses determine whether the system models meet some non-functional requirements (system *validation*),
- analyses enable to fulfill the non-functional requirements from a model (system *dimensioning*).

We applied the iterative process represented in Figure VII.24 to design a subpart of the avionic system and, to a lesser extent, design the software architecture of the Mars Pathfinder system. In the Mars Pathfinder case study, we applied different scheduling analyses to AADL models in order to verify that the system satisfies the timing constraints. We were able to detect the original design error with the right analysis, correct the model and finally validate the corrected model. In the more complex avionic case study, we combined different analyses so as to dimension and then validate the architecture of the avionic system based on AADL and CPAL models. We defined important parameters of the ARINC653 calculators and the

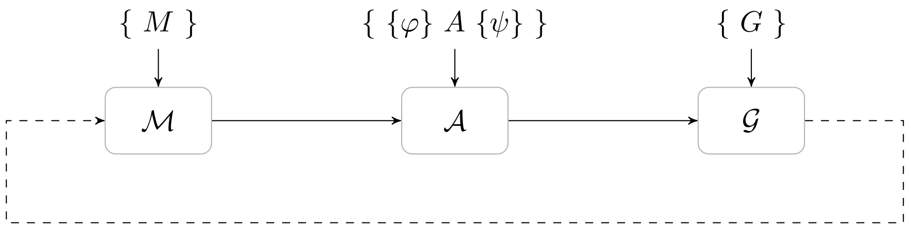


Figure VII.24: Iterative design process.

AFDX network, including Virtual Link parameters for example, and checked them to comply with the real-time constraints.

Chapter VIII

Conclusion

Abstract

This thesis dealt with the coupling between models and analyses so as to increase the efficiency and quality of critical embedded systems development, especially within Model-Driven Engineering. This penultimate chapter summarizes our contributions regarding this problem and provides the main results of this thesis. The last Chapter IX details some possible perspectives and future works.

VIII.1 Summary of the thesis

Non-functional requirements. The development of embedded systems is a complex and critical task, especially because of the non-functional requirements. In fact, embedded systems have to fulfill a set of non-functional properties dictated by their environment, expressed for example in terms of timing, dependability, security, or other performance criteria. In safety-critical applications for instance (e.g. an airplane), missing a non-functional requirement can have severe consequences, e.g. loss of life, personal injury, equipment damage, environmental disaster, etc.

A better integration of the analysis of non-functional properties in Model-Driven Engineering will increase the efficiency and quality of critical embedded systems development. This thesis aims at providing a general and coherent view on this problem by investigating two fundamental questions:

- How to apply an analysis on a model? (technical issue)
- How to manage the analysis process? (methodological issue)

In Part 1, we advanced several important concepts regarding the integration issue.

1) **revisiting model transformation for analysis.** First of all, we revisited the way model transformations are done to accommodate specific analysis engines (Chapter III). Arguing that an analysis is less based on a particular model syntax than specific data, we promoted query mechanisms called accessor to analyze the non-functional properties of a system at design time. These accessors enable to

extract data from a model and then analyze them. Expected benefit is that an analysis can be integrated to any kind of model as soon as an implementation of accessors to model internals is provided. Another advantage is that an analysis can be easily implemented by using a general-purpose programming language (e.g. Python) instead of relying on specific analysis engines.

2) **semantics of an analysis.** Next, we formalized the analysis execution (Chapter IV). We showed that an analysis is basically a program with preconditions and postconditions. The preconditions are the properties to hold true on an input model to successfully execute the analysis, whereas the postconditions are the properties guaranteed on the model after the analysis execution. With preconditions and postconditions, an analysis is complete and sound. We showed that a full analysis, including preconditions and postconditions, can be implemented through above-mentioned accessors.

3) **contract-driven analysis.** We abstracted away from the execution aspect through the notion of contract (Chapter V). A contract formally defines the interfaces of an analysis in terms of processed data and properties. Inputs/Outputs (I/O) describe input and output data. Assumptions/Guarantees (A/G) describe input and output properties. Notice that the 'data' directly refer to the accessors, whereas the 'properties' relate to the preconditions and postconditions. SAT methods can then be used to automatically reason about these interfaces, and provide greater automation support: which analysis can be applied on a given model? Which are the analyses that meet a given goal? Are there analyses to be combined? Are there interference between analyses? Etc. In practice, contracts can be defined with the help of a specification language such as Alloy, and evaluated through associated SAT solvers.

Then in Part 2, we implemented these concepts and experimented them through various case studies.

4) **prototyping and application.** We implemented a proof-of-concept tool to demonstrate and evaluate these concepts (Chapter VI). This tool implements several functions, each one implementing a part of the concepts introduced earlier. In particular, our tool provides accessors towards AADL and CPAL models, various real-time scheduling analyses programmed in Python, and an orchestration module based on Alloy. We finally illustrated the capabilities of our approach to deal with concrete systems coming from the aerospace: a drone, an exploratory robot and a flight management system (Chapter VII).

VIII.2 Main results

We experimented our contributions for the timing analysis of architectural models. On the one hand, we demonstrated that accessors enable to apply real-time scheduling analyses onto different kinds of architectural models, e.g. written with the industry standard AADL (Architecture and Analysis Design Language) or the new time-triggered language CPAL (Cyber-Physical Action Language). In fact, the benefit of using accessors is dual:

1. analyses can be applied on various types of models,
2. as analyses can originate from many models, one can combine these models to build wider analyses.

In addition to accessors and analyses, contracts make it possible to automate complex analysis procedures and, to some extent, to mechanize the design process itself. From a modeling and analysis repository, we are able to define and execute the analysis process that fulfills precise goals, e.g. is the system schedulable? To answer this final question, the analysis process may need to consider tasks and networks defined in the models, compute some missing data in the model, build a sound analysis order, etc.

The Flight Management System (FMS) provided a good illustration of the capabilities of our approach. We designed a subpart of the FMS from a combination of two architectural descriptions languages (i.e. AADL and CPAL) and various timing analyses. The models provided different abstractions from which we were able to carry out the analysis process, whereas the analysis process enabled to dimension and then validate the system from these complementary views.

This thesis provided some arguments and contributions supporting the idea that analysis should become first-class citizen in the design of critical embedded systems. Defining the coupling between models and analyses was a first step in this direction. This thesis advanced important concepts to make analysis visible and usable by engineers in the design workflow. Future work may improve or extend the concepts presented in this thesis, relax some initial work hypotheses, equip the approach with tools or additional language constructs, or explore the notion of design space and design space exploration through analysis contracts. Chapter IX presents these perspectives in more detail.

Chapter IX

Perspectives

Abstract

In this chapter, we sketch possible directions to continue the work initiated in this thesis. Some of the future works are direct improvements that may be carried out in the short term; others are part of more substantial research works to be pursued on their own. We detail five possible lines of research that follows the development of this thesis: immediate improvements and extensions of the concepts presented in this thesis (Section IX.1), definition of (a) language(s) that improve(s) the efficiency of these concepts (Section IX.2), development of a more advanced analysis and orchestration tool (Section IX.3), researches around the notion of design space and design space exploration (Section IX.4), and several relaxations of the initial work hypotheses (Section IX.5).

IX.1 Improvement and extension of the concepts

Part 1 presented several concepts so as to analyze the multiple non-functional properties of embedded systems in a MDE approach. A natural perspective will be to enhance and/or extend these concepts. The next subsections outline some potential improvements.

IX.1.1 Factorization of accessors

Accessors must be implemented in a one-to-one fashion, pairing an accessor implementation with a specific model (Chapter III). Therefore, there exist as many accessors to implement as there are technical spaces (i.e. metamodeling pyramids) to address.

A possible improvement will be to “factorize” the implementation of accessors. A particular way to proceed would be to implement something like an interchange data format between several modeling environments. This approach would bring several benefits:

- *reducing the number of accessor implementation*: the number of accessor implementations would be reduced to the number of interchange data formats,

- *further separation of concerns and reliability*: being implemented in two parts (i.e. generic accessors towards the interchange data format at the highest level, generation of the interchange data format at the lowest level) accessors are more reliable.

The definition of a data interchange format can be a consensus *a minima* between several domain experts (e.g. the `rt-format` to exchange data between tools of the real-time research community); or defined through of a more systematic approach (e.g. works about ontology).

IX.1.2 Additional contract evaluations and strategies

Another improvement would be to enrich contracts (presented in Chapter V) with quality metrics (e.g. rapidity of an analysis execution, precision of a result). This will allow to handle the analysis dynamics more precisely: coarse-grained but fast analyses can be used during the early design stages, e.g. for prototyping; in-depth and costly analyses are more relevant at the last stages in the design process (before the implementation phase), when early results should be consolidated. We note that the evaluation of the quality metrics adds little algorithmic complexity as it can be performed on a weighted analysis graph, e.g. by looking for the shortest analysis paths.

IX.2 Analysis and orchestration language(s)

The notion of language is prominent in this thesis. In fact, we mentioned multiple languages throughout this thesis: architecture description languages to represent a system architecture (AADL, CPAL), metalanguages to define metamodels, languages to program analyses (Python), language to express model constraints (REAL, OCL) or specify contracts (Alloy).

Future works may investigate the set of languages that capture well the concepts presented in this thesis. Defining one or several domain-specific languages would improve the:

- *effectiveness* of the concepts through optimal implementation means,
- *usability* of these concepts by engineers through customized representations.

Several languages may be defined:

Analysis and query language to express both model queries and analysis operations.

Constraint languages (e.g. OCL, REAL) can be used to express queries on domain-specific languages. Yet, we note two important shortcomings with constraint languages. First, constraints languages are defined by specific metamodels (e.g. UML or AADL metamodels) and, consequently, can only express queries about domain-specific concepts (rather than focusing on *analysis data*). Secondly, these languages have a syntax that does not enable to describe easily the analysis logic (data structures, control flows, operations, etc. are limited). We showed in this thesis that a high-level programming language such as Python is perfectly able to describe all the aspects of the analysis logic. However, a general-purpose programming language comes with a sometimes too rich syntax, and extra, useless, features.

Thus, an ideal query and analysis language would provide an intermediate level of abstraction between constraint languages (model query with analysis data structures) and programming languages (analysis logic with control flows, basic operators, mathematical operators, etc.), without all the unnecessary features and superfluous syntactic elements.

Constraint and contract language in order to describe analysis contracts, i.e. the analysis interfaces. In fact, contracts have two purposes: (1) check whether an analysis can be applied on a model, (2) check whether the analyses can be combined. Thus, the choice of the language is strongly related to its final use. In the first case, a constraint language (e.g. OCL, REAL) or a classic programming language (e.g. Python), may be sufficient. The second case is a constraint satisfaction problem. We used Alloy that provides both abstractions to represent contracts and SAT methods to automatically reason about analysis interfaces.

New, existing or original, languages may be experimented in order to find the most efficient way to capture contracts and evaluate them (for example, see works by Ruchkin et al. [184] where the authors define contracts through an AADL annex language).

Goal language to specify goals. In this thesis, we specify analysis goals through specific contracts written in Alloy. Goals may be expressed through a dedicated formalism (e.g. see the *goal-structuring notation* [208, 209]). A dedicated notation would enable to exhaustively specify the goals in terms of expected data and/or properties, and reason about them (hierarchization of goals, definition of assumptions, presentation of solutions, etc.).

IX.3 Analysis and orchestration tool

Part 2 firstly presented a proof-of-concept tool that implements the various concepts introduced in this thesis (Chapter VI). Through diverse case studies, we showed that this tool is capable to automate the analysis process at design time but also to enhance the design process by systematically combining models and analyses (Chapter VII). However, at this stage, this tool is not mature enough to be used by engineers. An interesting direction will be to further implement the concepts presented in this thesis in a working prototype, either as a standalone tool or as a tool add-on (e.g. as an Eclipse plug-in). This working prototype would be used to both carry on experimentation and act as a demonstrator showing the capabilities of our approach. We discuss possible lines of research and/or development hereinafter.

Models and accessors. Models and accessors form together the first part of the prototype. We already implemented accessors towards AADL and CPAL models.

Accessors towards other architectural models will be implemented in the short term, e.g. towards SysML, MARTE, Cheddar ADL, MoSaRT, etc.

From accessors, it will be interesting to explore the modeling and analysis synergies offered by these various kinds of models. First of all, system-wide models could be used to represent the essential aspects of a system: overall system representation using SysML, operational architecture in AADL, functional architecture and real-time execution from CPAL, etc. System-wide models may be completed with more specialized models, representing particular system views: real-time (e.g. Cheddar ADL, MoSaRT, MAST, etc.), behavioral, dependability, security, etc. All these models may overlap (provide the same data), complement one another (provide complementary data), or be totally distinct (provide different data).

Analysis repository. The analysis repository is the second fundamental component of the tool. The analysis repository should be implemented in two parts. First, every analysis must be programmed. For this purpose, we may use a constraint language, a general-purpose programming language as done in this thesis, or use a dedicated language (see next Section IX.2). Secondly, it is necessary to add every analysis to the repository. More advanced plug-in mechanisms may be provided to this end.

From a broad repository of models, accessors and analyses, we will be able to explore more combinations of these elements, and, we hope so, implement more powerful analysis and design processes.

Feedbacks. Providing information about the analysis process, i.e. feedbacks, would be a great functionality for the user. We may envision three main types of feedbacks, besides raw analysis results:

- analysis solutions: indicate the analyses that are applicable on a (subpart of a) model, signify the analyses that fulfill goals, show possible (or to avoid) analysis combinations, show all analysis paths, or only optimal analysis paths according to quality metrics (e.g. complexity, rapidity, precision), ...
- advanced analysis results: explain analysis results, notify the corrections to make on a model if applicable, provide automatic integration of results in models, ...
- debugging: point out missing data to apply an analysis, handle assumptions, provide a full trace of the analysis process, indicate which part of the analysis process is to be re-executed when a model is modified, ...

From effective feedbacks, we may greatly improve the way engineers interact with models and analyses when designing an embedded system, thereby increasing the impact of models and analyses on the design activity.

IX.4 Integration of models and analyses in the design process

This thesis emphasized on the coupling between models and analyses so as to design embedded systems. Applying our solutions on concrete case studies coming from the aerospace (Chapter VII), we showed that our approach enables not only to automate the analysis process at design time but also to automate some part of the design process through analysis. In other words, analysis, as a set of model assessment activities, is an integral part of the design process.

Hence, a natural research direction would be to explore more deeply such a design that encompasses models and analyses, i.e. the notion of *design space* quickly saw through the FMS case study (Section VII.3). The main idea would be to define the overall system (i.e. design space) and process (i.e. design space exploration) that includes the notions of models and analyses. For example, these works will define (1) the elements that make up the design space (models, analyses, goals, etc.); and (2) the techniques that enable to explore the design space (algorithms, constraints solving, heuristics, optimization techniques, etc.).

This substantial research work will build on our contributions to move forward the formal definition of the design process and its automation. Bridges may exist with more specialized works: requirements engineering, systems synthesis, systems optimization, etc.

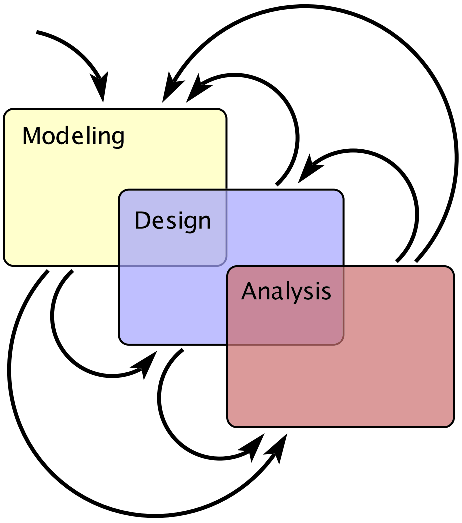


Figure IX.1: Modeling and analysis is the design process (taken from Lee and Seshia [17]). *In future works, we may define more precisely how models and analyses drive the design process.*

IX.5 Relaxing the work hypotheses

In this thesis, we sought to define generic concepts that can be applied as widely as possible, i.e. concepts that are not specific to particular kind of models or a specific analysis domain. We successfully applied these concepts to architectural models and real-time scheduling analyses.

Widening abstraction levels and semantics domains. A major challenge is now to apply these concepts to other types of models and analyses. We believe that our approach provides enough stability and genericness for this purpose. Yet, some improvements and/or extensions may be necessary in order to address new abstraction levels, or semantic domains. For example, in another analysis context such as *model-checking* , the analysis must be realized at a different level of abstraction, i.e. on a behavioral model (Petri nets, behavioral annex or AADL, etc.) rather than an architecture representation. In consequence, extensions and improvements include:

(1) New accessors to address various kinds of models: architectural, behavioral, etc. In the short term, accessors towards models at the same level of abstraction as AADL and CPAL will be implemented with a minimum of effort (e.g. UML-based languages SysML and MARTE, or Analysis-specific Languages such as MoSaRT, Cheddar ADL and MAST, synchronous dataflow languages). Accessors towards other types of abstractions will require more investigations to precisely define the data structures and mappings with metamodels,

(2) Enriched contracts to express and evaluate all types of analysis interfaces, i.e. all types of data and properties that can be computed by analyses. We can proceed as follows:

(a) list the *interface types* for different analysis domains (real-time, behavioral, dependability, security, etc.)

(b) define the suitable means to *express* these types of interfaces (for example the type of logic to use: First-Order Logic, Linear Temporal Logic, etc.); and maybe propose new methods to evaluate them (SAT resolution methods, SMT methods, etc.).

Application to the development of complex systems? Other work hypotheses were about development phases (the design phase) and the type of system (embedded systems). We believe that our approach is more general than just these activities and systems. Future works may experiment our approach to support other development phases such as requirements engineering, implementation, and even operation phases; and target all complex systems which have non-functional requirements.

Appendix A

Summary of publications

This appendix provides a list of the publications issued from this thesis. We presented the motivations behind this thesis in a position paper [HB14]. The Conference paper [BHN15] introduced contract-driven analysis and constituted the core of Chapter V. Proposed Journal article [BHN17] will provide an overview of the systematic analysis problem, thus covering the most important aspects of Chapter IV and Chapter V, and some parts of Chapter VI. Part of the Flight Management System case study in Chapter VII has been published as a Technical Report [BHN13a], and presented in a short version in a Workshop [BHN13b].

References

[BHN13a] G. Brau, J. Hugues, and N. Navet. Refinement of AADL models using early-stage analysis methods. In Gabor Karsai David Broman, editor, *Proceedings of the 4th Analytic Virtual Integration of Cyber-Physical System (AVICPS) Workshop*, pages 29–32, Vancouver, Canada, December 3 2013. Linköping University Electronic Press.

[BHN13b] G. Brau, J. Hugues, and N. Navet. Refinement of AADL models using early-stage analysis methods – An avionics example. Technical Report TR-LASSY-13-06, Laboratory for Advanced Software Systems, 2013.

[BHN15] G. Brau, J. Hugues, and Nicolas Navet. A contract-based approach for goal-driven analysis. In *18th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2015.

[BHN17] G. Brau, J. Hugues, and N. Navet. Towards the systematic analysis of non-functional properties in model-based systems engineering. *Science of Computer Programming*, 2017.

[HB14] J. Hugues and G. Brau. Analysis as a first-class citizen: An application to architecture description languages. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, pages 214–221, June 2014.

List of Figures

List of Tables

List of Listings

Bibliography

[1] P. H. Feiler, J. Hansson, D. De Niz, and L. Wrage, “System architecture virtual integration: An industrial case study,” DTIC Document, Tech. Rep., 2009.

[2] R. France and B. Rumpe, “Model-driven development of complex software: A research roadmap,” in *2007 Future of Software Engineering.* IEEE Computer Society, 2007, pp. 37–54.

[3] SAE International, *Architecture Analysis and Design Language (AADL) AS-5506A*, Std., 2009.

[4] P. Cuenot, P. Frey, R. Johansson, H. Lonn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. Tavakoli Kolagari, M. Torngren, and M. Weber, “The EAST-ADL Architecture Description Language for Automotive Embedded Software,” in *Model-Based Engineering of Embedded Real-Time Systems.* Springer, 2011, pp. 297–307.

[5] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst, “Automotive open system architecture - an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures,” *Convergence International Congress & Exposition On Transportation Electronics*, pp. 325–332, 2004.

[6] B. Selic and S. Gerard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*, ser. The MK/OMG Press. Morgan Kaufmann, 2013.

[7] T. Weilkiens, *Systems Engineering with SysML/UML: Modeling, Analysis, Design*, ser. The MK/OMG Press. Morgan Kaufmann, 2008.

[8] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, “Cheddar: a flexible real time scheduling framework,” in *ACM SIGAda Ada Letters*, vol. 24, no. 4. ACM, 2004, pp. 1–8, software available at http://beru.univ-brest.fr/~singhoff/cheddar/.

[9] M. González Harbour, J. G. García, J. P. Gutiérrez, and J. D. Moyano, “Mast: Modeling and analysis suite for real time applications,” in *13th Euromicro Conference on Real-Time Systems (ECRTS).* IEEE, 2001, pp. 125–134, software available at http://mast.unican.es/.

[10] K. G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a nutshell,” *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "Cadp 2010: A toolbox for the construction and analysis of distributed processes," in *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2011, pp. 372–387.

[12] B. Xu and M. Lu, "A survey on verification and analysis of non-functional properties of aadl model based on model transformation," in *5th International Conference on Education, Management, Information and Medicine (EMIM).* Atlantis Press, 2015.

[13] Object Management Group (OMG), *Meta Object Facility (MOF) Core Specification Version 2.5*, Std., June 2015.

[14] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework.* Addison-Wesley, 2008.

[15] F. Jouault and I. Kurtev, "Transforming models with atl," in *Workshop on Model Transformations in Practice (MTiP)*, 2005.

[16] M. Amrani, "Towards the Formal Verification of Model Transformations: An Application to Kermeta," Ph.D. dissertation, University of Luxembourg, november 2013.

[17] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach.* Lee & Seshia, 2011.

[18] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley, 2012.

[19] N. Navet, L. Fejoz, L. Havet, and S. Altmeyer, "Lean Model-Driven Development through Model-Interpretation: the CPAL design flow," in *Embedded Real-Time Software and Systems (ERTS)*, 2016.

[20] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, no. 10, pp. 10–19, 1988.

[21] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, no. 2-3, pp. 101–155, 2004.

[22] T. A. Henzinger and J. Sifakis, "The embedded systems design challenge," in *International Symposium on Formal Methods.* Springer, 2006, pp. 1–15.

[23] T. Noergaard, *Embedded systems architecture: a comprehensive guide for engineers and programmers.* Newnes, 2012.

[24] P. Koopman, "Design constraints on embedded real time control systems," 1990.

[25] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX.* Addison-Wesley Educational Publishers Inc, 2009.

[26] P. Marwedel, *Embedded system design: Embedded systems foundations of cyber-physical systems.* Springer Science & Business Media, 2010.

[27] P. Koopman, "Embedded system design issues (the rest of the story)," in *Computer Design: VLSI in Computers and Processors, 1996. ICCD'96. Proceedings., 1996 IEEE International Conference on.* IEEE, 1996, pp. 310–317.

[28] D. Harel and A. Pnueli, "On the development of reactive systems," in *Logics and models of concurrent systems.* Springer, 1985, pp. 477–498.

[29] H. Kopetz, *Real-time systems: design principles for distributed embedded applications.* Springer Science & Business Media, 2011.

[30] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, vol. 6, no. 3, pp. 116–128, 1991.

[31] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications.* Springer Science & Business Media, 2011, vol. 24.

[32] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo, *Soft Real-Time Systems.* Springer, 2005.

[33] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International.* IEEE, 2007, pp. 239–243.

[34] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[35] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on.* IEEE, 2002, pp. 547–550.

[36] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[37] I. Sommerville, *Software Engineering*, seventh edition ed., Addison-Wesley, Ed., 2004.

[38] W. W. Royce, "Managing the development of large software systems," in *proceedings of IEEE WESCON*, vol. 26, no. 8. Los Angeles, 1970, pp. 328–338.

[39] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[40] J. Bézivin, "On the unification power of models," *Software & Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

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

[42] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 01, pp. 31–57, 1992.

[43] P. B. Kruchten, "The 4+ 1 view model of architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.

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

[45] D. Ghosh, *DSLS IN ACTION.* Wiley India Pvt. Limited, 2011. [Online]. Available: https://books.google.lu/books?id=9U9TpwAACAAJ

[46] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional, 2003.

[47] G. Booch, I. Jacobson, and J. Rumbaugh, "The unified modeling language reference manual," 1999.

[48] D. Harel and B. Rumpe, "Meaningful modeling: what's the semantics of" semantics"?" *Computer*, vol. 37, no. 10, pp. 64–72, 2004.

[49] B. Combemale, "Approche de métamodélisation pour la simulation et la vérification de modèle–application à l'ingénierie des procédés," Ph.D. dissertation, Institut National Polytechnique de Toulouse-INPT, 2008.

[50] X. Blanc and O. Salvatori, *MDA en action: Ingénierie logicielle guidée par les modèles.* Editions Eyrolles, 2011.

[51] A. Kleppe, *Software language engineering: creating domain-specific languages using metamodels.* Pearson Education, 2008.

[52] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving executability into object-oriented meta-languages," in *International Conference on Model Driven Engineering Languages and Systems.* Springer, 2005, pp. 264–278.

[53] I. Kurtev, J. Bézivin, and M. Akşit, "Technological spaces: An initial appraisal," 2002.

[54] J. Bézivin, I. Kurtev *et al.*, "Model-based technology integration with the technical space concept," in *Metainformatics Symposium*, vol. 20, 2005, pp. 44–49.

[55] J.-M. Favre, J. Estublier, and M. Blay-Fornarino, "L'ingénierie dirigée par les modèles: au delà du mda, traité ic2, série informatique et systèmes d'information," 2006.

[56] M. Amrani, J. Dingel, L. Lambers, L. Lúcio, R. Salay, G. Selim, E. Syriani, and M. Wimmer, "Towards a model transformation intent catalog," in *Proceedings of the First Workshop on the Analysis of Model Transformations.* ACM, 2012, pp. 3–8.

[57] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, and M. Wimmer, "Model transformation intents and their properties," *Software & systems modeling*, pp. 1–38, 2014.

[58] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006.

[59] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, vol. 45, no. 3. USA, 2003, pp. 1–17.

[60] A. Kleppe, "Mcc: A model transformation environment," in *European Conference on Model Driven Architecture-Foundations and Applications.* Springer, 2006, pp. 173–187.

[61] Object Management Group (OMG), *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.3*, Std., June 2016.

[62] I. Kurtev, "State of the art of qvt: A model transformation language standard," in *International Symposium on Applications of Graph Transformations with Industrial Relevance.* Springer, 2007, pp. 377–393.

[63] P. C. Clements, "A survey of architecture description languages," in *Proceedings of the 8th international workshop on software specification and design.* IEEE Computer Society, 1996, p. 16.

[64] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[65] P. Binns, M. Englehart, M. Jackson, and S. Vestal, "Domain-specific software architectures for guidance, navigation and control," *International Journal of Software Engineering and Knowledge Engineering*, vol. 6, no. 02, pp. 201–227, 1996.

[66] S. Vestal, "Metah support for real-time multi-processor avionics," in *Parallel and Distributed Real-Time Systems, 1997. Proceedings of the Joint Workshop on.* IEEE, 1997, pp. 11–21.

[67] J. Hugues, "Architecture in the Service of Real-Time Middleware," Institut Sup'erieur de l'A'eronautique et de l'Espace (ISAE), HDR, 2017.

[68] SAE/AS2-C, *Data Modeling, Behavioral and ARINC653 Annex document for the Architecture Analysis & Design Language v2.0 (AS5506A)*, October 2009.

[69] O. Yassine, G. Emmanuel, and H. Jérôme, "Mapping AADL models to a repository of multiple schedulability analysis techniques," in *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing*, IEEE, Ed., jun 2013, p. 8.

[70] Y. Ouhammou, "Model-based framework for using advanced scheduling theory in real-time systems design," Ph.D. dissertation, Ecole Nationale Supérieure de Mécanique et d'Aérotechique de Poitiers, december 2013.

[71] O. Sokolsky, I. Lee, and D. Clarke, "Schedulability Analysis of AADL Models," in *20th International Parallel & Distributed Processing Symposium (IPDPS).* IEEE, 2006.

[72] O. Sokolsky and A. Chernoguzov, "Analysis of AADL Models Using Real-Time Calculus with Applications to Wireless Architectures," University of Pennsylvania Department of Computer and Information Science, Tech. Rep. No. MS-CIS-08-25., July 2008.

[73] M.-Y. Nam, K. Kang, R. Pellizzoni, K.-J. Park, J.-E. Kim, and L. Sha, "Modeling Towards Incremental Early Analyzability of Networked Avionics Systems Using Virtual Integration," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 4, pp. 81:1–81:23, 2013.

[74] X. Renault, F. Kordon, and J. Hugues, "Adapting models to model checkers, a case study: Analysing aadl using time or colored petri nets," in *20th International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2009, pp. 26–33.

[75] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, and F. Vernadat, "Formal verification of AADL specifications in the Topcased environment," in *14th International Conference on Reliable Software Technologies Ada-Europe*. Springer, 2009, pp. 207–221.

[76] J.-P. Bodeveix, M. Filali, M. Garnacho, R. Spadotti, and Z. Yang, "Towards a verified transformation from aadl to the formal component-based language fiacre," *Science of Computer Programming*, vol. 106, pp. 30–53, 2015.

[77] P. C. Ölveczky, A. Boronat, and J. Meseguer, "Formal semantics and analysis of behavioral aadl models in real-time maude," in *Formal Techniques for Distributed Systems*. Springer, 2010, pp. 47–62.

[78] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat, "Automated verification of aadl-specifications using uppaal," in *14th International Symposium on High-Assurance Systems Engineering (HASE)*. IEEE, 2012, pp. 130–138.

[79] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*. Ieee, 2006, pp. 3–12.

[80] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating aadl into bip-application to the verification of real-time systems," in *Models in Software Engineering*. Springer, 2009, pp. 5–19.

[81] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "Cadp 2011: a toolbox for the construction and analysis of distributed processes," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 2, pp. 89–107, 2013.

[82] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel, "From aadl model to lnt specification," in *20th International Conference on Reliable Software Technologies Ada-Europe*. Springer, 2015, pp. 146–161.

[83] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems," in *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, ser. SAFECOMP '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 173–186. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04468-7_15

[84] A.-E. Rugina, "Dependability modeling and evaluation–from aadl to stochastic petri nets," Ph.D. dissertation, Institut National Polytechnique de Toulouse, 2007.

[85] A.-E. Rugina, K. Kanoun, and M. Kaâniche, "The ADAPT tool: From AADL architectural models to stochastic petri nets through model transformation," in *Dependable Computing Conference, 2008. EDCC 2008. Seventh European.* IEEE, 2008, pp. 85–90.

[86] J. Hansson, B. Lewis, J. Hugues, L. Wrage, P. Feiler, and J. Morley, "Model-Based Verification of Security and Non-Functional Behavior using AADL," *IEEE Security & Privacy*, vol. PP, no. 99, pp. 1–1, 2009.

[87] J. Delange, L. Pautet, and F. Kordon, "Design, verification and implementation of mils systems," in *Proceedings of the 21th International Symposium on Rapid System Prototyping*, 2010, pp. 1–8.

[88] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications," in *14th International Conference on Reliable Software Technologies Ada-Europe.* Springer, 2009, software available at http://www.openaadl.org/ocarina.html.

[89] Y. Ma, H. Yu, T. Gautier, J.-P. Talpin, L. Besnard, and P. Le Guernic, "System synthesis from aadl using polychrony," in *Electronic System Level Synthesis Conference (ESLsyn), 2011.* IEEE, 2011, pp. 1–6.

[90] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens, "Virtual execution of AADL models via a translation into synchronous programs," in *7th International Conference on Embedded Software (EMSOFT).* ACM, 2007, pp. 134–143.

[91] R. Varona-Gomez and E. Villar, "AADL Simulation and Performance Analysis in SystemC," in *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 323–328. [Online]. Available: http://dx.doi.org/10.1109/ICECCS.2009.11

[92] A. Johnsen and K. Lundqvist, "Developing Dependable Software-Intensive Systems: AADL vs. EAST-ADL," in *Ada-Europe 2011*, A. Romanovsky and T. Vardanega, Eds. Springer-Verlag, June 2011, pp. 103–117. [Online]. Available: http://www.es.mdh.se/publications/1753-

[93] M. Faugere, T. Bourbeau, R. de Simone, and S. Gerard, "MARTE: Also an UML Profile for Modeling AADL Applications," *Engineering of Complex Computer Systems, IEEE International Conference on*, vol. 0, pp. 359–364, 2007.

[94] R. Behjati, T. Yue, S. Nejati, L. Briand, and B. Selic, "Extending sysML with AADL Concepts for Comprehensive System Architecture Modeling," in *Proceedings of the 7th European Conference on Modelling Foundations and Applications*, ser. ECMFA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 236–252.

[95] C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard, "Optimum: A MARTE-based Methodology for Schedulability Analysis at Early Design Stages," *SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, Jan. 2011.

[96] S. Bernardi, J. Merseguer, and D. Petriu, "A dependability profile within MARTE," *Software & Systems Modeling*, vol. 10, no. 3, pp. 313–336, 2011.

[97] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and assessment in uml-based software development," *The Scientific World Journal*, vol. 2012, 2012.

[98] D.-J. Chen, R. Johansson, H. Lönn, H. Blom, M. Walker, Y. Papadopoulos, S. Torchiaro, F. Tagliabo, and A. Sandberg, "Integrated safety and architecture modeling for automotive embedded systems," *Elektrotechnik und Informationstechnik*, vol. 128, no. 6, pp. 196–202, 2011.

[99] D. Chen, L. Feng, T.-N. Qureshi, H. Lonn, and F. Hagl, "An architectural approach to the analysis, verification and validation of software intensive embedded systems," *Computing*, vol. 95, no. 8, pp. 649–688, 2013.

[100] J. Hugues and G. Brau, "Analysis as a First-Class Citizen: An Application to Architecture Description Languages," in *IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, June 2014, pp. 214–221.

[101] L. Fejoz and N. Navet, *The CPAL Programming Language: An introduction*, v1.05 ed., RealTime-at-Work and University of Luxembourg, October 2016.

[102] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[103] N. Halbwachs, *Synchronous programming of reactive systems.* Springer Science & Business Media, 2013, vol. 215.

[104] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Embedded software.* Springer, 2001, pp. 166–184.

[105] F. Maraninchi and Y. Rémond, "Mode-automata: a new domain-specific construct for the development of safe critical systems," *Science of computer programming*, vol. 46, no. 3, pp. 219–254, 2003.

[106] S. Altmeyer and N. Navet, "The case for fifo scheduling," technical report from the University of Luxembourg, to appear, Tech. Rep., 2015.

[107] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," in *Proceedings of the IEEE*, vol. 79, no. 9. IEEE, 1991, pp. 1305–1320.

[108] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the signal language and its semantics," *Science of Computer Programming*, vol. 16, no. 2, pp. 103–149, 1991.

[109] G. Berry and G. Gonthier, “The esterel synchronous programming language: Design, semantics, implementation,” *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[110] J. Forget, “A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints,” Ph.D. dissertation, Université de Toulouse, 2009.

[111] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, “A real-time architecture design language for multi-rate embedded control systems,” in *25th Symposium on Applied Computing (SAC)*. ACM, 2010, pp. 527–534.

[112] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, “Polychrony for system design,” *Journal of Circuits, Systems, and Computers*, vol. 12, no. 03, pp. 261–303, 2003.

[113] G. Berry, “Scade: Synchronous design and validation of embedded control software,” in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[114] Y. Ma, H. Yu, T. Gautier, P. L. Guernic, J. Talpin, L. Besnard, and M. Heitz, “Toward polychronous analysis and validation for timed software architectures in AADL,” in *Design, Automation and Test in Europe (DATE)*, 2013, pp. 1173–1178.

[115] Z. Yang, K. Hu, J.-P. Bodeveix, L. Pi, D. Ma, and J.-P. Talpin, “Two formal semantics of a subset of the aadl,” in *16th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2011, pp. 344–349.

[116] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[117] C. A. Petri, “Kommunikation mit automaten,” Ph.D. dissertation, 1962.

[118] R. Alur and D. Dill, “Automata for modeling real-time systems,” in *International Colloquium on Automata, Languages, and Programming*. Springer, 1990, pp. 322–335.

[119] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Springer Science & Business Media, 2001, vol. 2050.

[120] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat, *Synchronization and linearity: an algebra for discrete event systems*. John Wiley & Sons Ltd, 1992.

[121] M. Boyer, J. Migge, and M. Fumey, “PEGASE - A Robust and Efficient Tool for Worst-Case Network Traversal Time Evaluation on AFDX,” in *SAE AeroTech Congress & Exhibition*, Toulouse, France, October 18-21 2011, http://www.realtimeatwork.com/software/rtaw-pegase/.

[122] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[123] M. Utting and B. Legeard, *Practical model-based testing: a tools approach.* Morgan Kaufmann, 2010.

[124] N. Audsley and A. Burns, "Real-time system scheduling," Tech. Rep., 1990.

[125] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, *Scheduling in real-time systems*, 2002.

[126] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, p. 35, 2011.

[127] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[128] E. Bini and G. Buttazzo, "A hyperbolic bound for the rate monotonic algorithm," in *Real-Time Systems, 13th Euromicro Conference on, 2001.* IEEE, 2001, pp. 59–66.

[129] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[130] L. George, N. Rivierre, and M. Spuri, "Preemptive and non-preemptive real-time uniprocessor scheduling," Inria, Tech. Rep., 1996.

[131] M. Stigge and W. Yi, "Graph-based models for real-time workload: a survey," *Real-Time Systems*, vol. 51, no. 5, pp. 602–636, 2015.

[132] S. Friedenthal, R. Griego, and M. Sampson, "Incose model based systems engineering (mbse) initiative," in *INCOSE 2007 Symposium*, 2007.

[133] J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies," INCOSE MBSE Initiative, Tech. Rep., 2007.

[134] Software Engineering Institute, "OSATE2 : An open-source tool platform for AADLv2," https://wiki.sei.cmu.edu/aadl/index.php/Osate_2, june 2016.

[135] J.-M. Favre, "Towards a basic theory to model model driven engineering," in *3rd Workshop in Software Model Engineering, WiSME.* Citeseer, 2004, pp. 262–271.

[136] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the Ocarina AADL tool suite," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 4, pp. 42:1–42:25, 2008.

[137] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, simulate, and implement ARINC653 systems using the AADL," in *SIGAda annual international conference on Ada and related technologies*, ser. SIGAda '09, New York, NY, USA, 2009, pp. 31–44. [Online]. Available: http://doi.acm.org/10.1145/1647420.1647435

[138] MathWorks, "Simulink - Simulation and Model-Based Design," https://www.mathworks.com/products/simulink.html (accessed January, 2017).

[139] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity-the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[140] A. Khoroshilov, I. Koverninskiy, A. Petrenko, and A. Ugnenko, "Integrating aadl-based tool chain into existing industrial processes," in *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, 2011.

[141] M.-Y. Nam, R. Pellizzoni, L. Sha, and R. M. Bradford, "ASIIST: Application Specific I/O Integration Support Tool for Real-Time Bus Architecture Designs," in *14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. Potsdam, Germany: IEEE Computer Society, 2-4 June 2009, pp. 11–22.

[142] N. Wirth, *Algorithms+ data structures= programs*. Prentice Hall PTR, 1978.

[143] C. J. Date and H. Darwen, *A guide to the SQL Standard: a user's guide to the standard relational language SQL*. Addison-Wesley Longman, 1993, vol. 55822.

[144] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu, "Xquery 1.0: An xml query language," 2002.

[145] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," 1983.

[146] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," *Software Engineering, IEEE Transactions on*, vol. 23, no. 10, pp. 635–645, 1997.

[147] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, no. 1, pp. 5–22, 1999.

[148] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers (TC)*, vol. 39, no. 9, pp. 1175–1185, 1990.

[149] V. Gaudel, A. Plantec, F. Singhoff, J. Hugues, P. Dissaux, and J. Legrand, "Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets," in *International Symposium on Rapid System Prototyping (RSP)*. IEEE, 2013.

[150] B. Berthomieu, J.-P. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang, and F. Vernadat, "Fiacre: an Intermediate Language for Model Verification in the Topcased Environment," in *Embedded Real-Time Software (ERTS) Congress 2008*, Toulouse, France, 2008.

[151] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[152] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *17th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2011, pp. 71–80.

[153] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, “Task automata: Schedulability, decidability and undecidability,” *Information and Computation*, vol. 205, no. 8, pp. 1149–1172, 2007.

[154] O. Gilles and J. Hugues, “Expressing and enforcing user-defined constraints of AADL models,” in *5th UML and AADL Workshop (UML and AADL 2010)*, 2010.

[155] J. B. Warmer and A. G. Kleppe, *The object constraint language: getting your models ready for MDA.* Addison-Wesley, 2003.

[156] J. Cabot and M. Gogolla, “Object constraint language (ocl): a definitive guide,” in *Formal methods for model-driven engineering.* Springer, 2012, pp. 58–90.

[157] Object Management Group, *Object Constraint Language, Version 2.4*, Std., February 2014.

[158] O. Gilles, “Vers une prise en compte fine de la plate-forme cible dans la construction des systemes temps réel embarqués critiques par ingénierie des modeles,” Ph.D. dissertation, Télécom ParisTech, 2010.

[159] O. Gilles and J. Hugues, “A mde-based optimisation process for real-time systems,” in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing.* IEEE, 2010, pp. 50–57.

[160] V. Gaudel, “Des patrons de conception pour assurer l’analyse d’architectures: un exemple avec l’analyse d’ordonnancement,” Ph.D. dissertation, Université de Bretagne Occidentale, november 2014.

[161] Y. Ouhammou, E. Grolleau, P. Richard, and M. Richard, “Reducing the Gap Between Design and Scheduling,” in *20th International Conference on Real-Time and Network Systems (RTNS).* ACM, 2012, pp. 21–30.

[162] P. Dissaux and F. Singhoff, “Stood and cheddar: Aadl as a pivot language for analysing performances of real time architectures,” in *4th European Congress on Embedded Real Time Software and Systems (ERTS)*, 2008, p. 21.

[163] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand, “Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns,” in *Leveraging Applications of Formal Methods, Verification, and Validation.* Springer, 2010, pp. 4–17.

[164] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand, “An Ada Design Pattern Recognition Tool for AADL Performance Analysis,” in *Annual International Conference on Special Interest Group on the Ada Programming Language (SIGAda).* ACM, 2011, pp. 61–68.

[165] G. Brau, J. Hugues, and N. Navet, “Refinement of AADL models using early-stage analysis methods,” in *Proceedings of the 4th Analytic Virtual Integration of Cyber-Physical System (AVICPS) Workshop*, G. K. David Broman, Ed. Vancouver, Canada: Linköping University Electronic Press, December 3 2013, pp. 29–32.

[166] ——, “Refinement of AADL models using early-stage analysis methods – An avionics example,” Laboratory for Advanced Software Systems, Tech. Rep. TR-LASSY-13-06, 2013.

[167] G. Lasnier, “Une approche intégrée pour la validation et la génération de systèmes critiques par raffinement incrémental de modèles architecturaux,” Ph.D. dissertation, Télécom ParisTech, 2012.

[168] M. Vaziri and D. Jackson, “Some shortcomings of ocl, the object constraint language of uml.” in *TOOLS (34)*, 2000, pp. 555–562.

[169] D. Jackson, *Software Abstractions: logic, language, and analysis.* MIT press, 2012.

[170] SAE/AS2-C, “Architecture Analysis & Design Language V2 (AS5506A),” Jan. 2009.

[171] RealTime-at-Work, “RTaW-Sim: Controller Area Network simulation and configuration,” http://www.realtimeatwork.com/software/rtaw-sim/ (accessed January, 2017).

[172] “Open AADL/AADLib – Library of reusable AADL Models,” http://www.openaadl.org/aadlib.html, (accessed January, 2017).

[173] A. Burns, B. Dobbing, and G. Romanski, “The ravenscar tasking profile for high integrity real-time programs,” in *International Conference on Reliable Software Technologies.* Springer, 1998, pp. 263–275.

[174] M. Lauer, “Une méthode globale pour la vérification d’exigences temps réel - Application à l’Avionique Modulaire Intégrée,” Ph.D. dissertation, Institut National Polytechnique de Toulouse, 2012.

[175] C. Spitzer, U. Ferrell, and T. Ferrell, *Digital avionics handbook.* CRC Press, 2014.

[176] R. W. Floyd, “Assigning meanings to programs,” *Mathematical aspects of computer science*, vol. 19, no. 19-32, p. 1, 1967.

[177] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[178] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, “Multiple viewpoint contract-based specification and design,” in *Formal Methods for Components and Objects.* Springer, 2007, pp. 200–225.

[179] B. Meyer, “Applying “Design by Contract”,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[180] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, “Taming dr. frankenstein: Contract-based design for cyber-physical systems*,” *European journal of control*, vol. 18, no. 3, pp. 217–238, 2012.

[181] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren, “Cyber-physical system design contracts,” in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems.* ACM, 2013, pp. 109–118.

[182] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for systems design: Theory," Inria Rennes Bretagne Atlantique, Tech. Rep., 2015.

[183] I. Ruchkin, D. De Niz, S. Chaki, and D. Garlan, "Contract-based integration of cyber-physical analyses," in *14th International Conference on Embedded Software (EMSOFT).* ACM, 2014, p. 23.

[184] ——, "Active: A tool for integrating analysis contracts," in *5th Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS).* LiU Electronic Press, 2014.

[185] I. Ruchkin, A. Rao, D. De Niz, S. Chaki, and D. Garlan, "Eliminating inter-domain vulnerabilities in cyber-physical systems: An analysis contracts approach," in *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy.* ACM, 2015, pp. 11–22.

[186] M. Walker, M.-O. Reiser, S. Tucci-Piergiovanni, Y. Papadopoulos, H. Lönn, C. Mraidha, D. Parker, D. Chen, and D. Servat, "Automatic optimisation of system architectures using EAST-ADL," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2467 – 2487, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121213000885

[187] F. Cadoret, "Génération stratégique de code pour la maîtrise des performances de systèmes temps-réel embarqués," Ph.D. dissertation, Télécom ParisTech, 2014.

[188] J. Migge, "Scheduling of recurrent tasks on one processor : a trajectory based model," Ph.D. dissertation, Universit'e de Nice, 1999.

[189] P. Brisset, A. Drouin, M. Gorraz, P.-S. Huard, and J. Tyler, "The paparazzi solution," in *2nd US-European Competition and Workshop on Micro Air Vehicles (MAV)*, 2006.

[190] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. De Michiel, "Papabench: a free real-time benchmark," in *6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[191] "Paparazzi - the free autopilot," http://wiki.paparazziuav.org/wiki/Main_Page, (accessed January, 2017).

[192] F. Nemer, "Optimisation de l'estimation du wcet par analyse inter-tâche du cache d'intructions," Ph.D. dissertation, Université de Toulouse, Université Toulouse III-Paul Sabatier, 2008.

[193] Institut de Recherche en Informatique de Toulouse (TRACES team), "Papabench," https://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97, (accessed January, 2017).

[194] F. Nemer, H. Cassé, P. Sainrat, and J. P. Bahsoun, "Inter-task wcet computation for a-way instruction caches," in *2008 International Symposium on Industrial Embedded Systems.* IEEE, 2008, pp. 169–176.

[195] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, “On the effectiveness of cache partitioning in hard real-time systems,” *Real-Time Systems*, pp. 1–46, 2016.

[196] J. P. Lehoczky, “Enhanced aperiodic responsiveness in hard real-time environments,” in *Proceedings of the IEEE Symposium on Real-Time Systems*, 1987, pp. 261–270.

[197] G. Bernat and A. Burns, “New results on fixed priority aperiodic servers,” in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE.* IEEE, 1999, pp. 68–78.

[198] J. Migge, *TkRts: A tool for computing response time bounds (v 0.5.5)*, December 2000.

[199] B. Sprunt, L. Sha, and J. Lehoczky, “Aperiodic task scheduling for hard-real-time systems,” *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[200] Jet Propulsion Laboratory, NASA (website), “Mission to mars, mars pathfinder / sojourner rover,” http://www.jpl.nasa.gov/missions/details.php?id=5913 (accessed September, 2016).

[201] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, “The rosace case study: from simulink specification to multi/many-core execution,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, 2014, pp. 309–318.

[202] *ARINC Report 653P0 Avionics Application Software Standard Interface, Part 0, Overview of ARINC 653.* Aeronautical Radio Incorporated.

[203] *ARINC Report 664P7-1 Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network.* Aeronautical Radio Incorporated.

[204] L. Fejoz, “ROSACE Case Study: A CPAL implementation (version 1.0),” RealTime-at-Work, Tech. Rep., September 2016.

[205] J. P. Gonçalves Crespo Craveiro, “Real-Time Scheduling in Multicore Time- and Space-Partitioned Architectures,” Ph.D. dissertation, Universidade de Lisboa, 2013.

[206] F. Frances, C. Fraboul, and J. Grieu, “Using network calculus to optimize the AFDX network,” in *European Congress on Embedded Real-Time Software*, Toulouse France, 2006.

[207] M. Boyer, N. Navet, and M. Fumey, “Experimental assessment of timing verification techniques for AFDX,” in *European Congress in Embedded Real Time Software and Systems (ERTS)*, Toulouse, France, February 1-3 2012.

[208] T. Kelly and R. Weaver, “The goal structuring notation–a safety argument notation,” in *Proceedings of the dependable systems and networks 2004 workshop on assurance cases.* Citeseer, 2004.

[209] G. W. Group, *GSN COMMUNITY STANDARD VERSION 1*, http://www.goalstructuringnotation.info/, Std., 2011.