République Tunisienne
Ministère de l'Enseignement
Supérieur, de la Recherche
Scientifique

Université de Sfax
École Nationale d'Ingénieurs de Sfax

Ecole Doctorale
Sciences et Technologies

Thèse de *DOCTORAT*

Ingénierie des Systèmes Informatiques

THESE

Présentée à

L'École Nationale d'Ingénieurs de Sfax

En vue de l'obtention du

DOCTORAT

Dans la discipline Informatique
Ingénierie des Systèmes Informatiques

Par

Hana Mkaouar

Ingénieur en informatique

A Formal Approach for Real-time Systems Engineering

Soutenu le 09 02 2019, devant le jury composé de :

M.	**Ahmed Hadj Kacem** (Professeur à la FSEG)	**Président**
M.	**Mohamed Abid** (Professeur à l'ENIS)	**Rapporteur**
M.	**Mohamed Kaâniche** (Directeur de recherche)	**Rapporteur**
M.	**Mohamed Ben Aouicha** (MCF à la FSS)	**Examinateur**
M.	**Mohamed Jmaiel** (Professeur à l'ENIS)	**Directeur de Thèse**
M.	**Bechir Zalila** (Maitre-assistant à l'ENIS)	**Invité**
M.	**Jérôme Hugues** (Professeur à ISAE-SUPAERO)	**Invité**

A Formal Approach for Real-time Systems Engineering

Hana MKAOUAR

Acknowledgments

It is with great pleasure that I reserve this page as a sign of deep gratitude to all those who have kindly provided the necessary support for the smooth running of this thesis.

I present my thanks to *Prof. Ahmed Hadj Kacem*, for the honor he had accorded me for agreeing to be the committee chair of my thesis. I also thank *Prof. Mohamed Ben Aouicha* for the valuable service to examine my thesis and to be a member of the committee. My distinguished thanks go also to *Prof. Mohamed Abid* and *Prof. Mohamed Kaâniche* for taking their time to review my dissertation and for their relevant comments.

I would like to express my deep gratitude to my supervisor *Prof. Mohamed Jmaiel* for his outstanding commitment to this thesis. I am also grateful for the support he gave me. His professionalism, friendliness and pedagogical and scientific qualities have been invaluable.

I am also indebted to my co-supervisors *Dr. Bechir Zalila* and *Prof. Jérôme Hugues* for supervising my thesis work, for their human qualities, for their patience, and especially for the time they have spent for me. They have always been the source of inspiration and motivation to me. Their knowledge, insight, scientific rigor and advices improved immensely the quality of the scientific and technical contributions of this work, but also its presentation. They have helped me gain invaluable skills as a researcher. May they find in this work the fruit of their effort and the expression of my deep gratitude.

I am thankful to all of my friends and colleagues at the ReDCAD Laboratory (Sfax-Tunisia). Not only were our technical discussions very interesting, but it was wonderful working with them.

Last but not least I would like to express a very special gratitude to my parents Wahid and Hajer, my whole family (Salma, Mohamed, Ahmed, Faouzi, Mohamed Sadak and Omar) and all my friends for their support, patience, love and for making this journey as pleasant as it can be.

Contents

1 Introduction **1**

1.1 Context et motivation 1

1.2 Problem statement 2

1.3 Objectives 5

1.4 Contributions 5

1.4.1 Formal pattern for a real-time task model 6

1.4.2 Application in an MDE approach 6

1.4.3 Tool-chain, analysis results and scalability 8

1.5 Thesis organization 8

2 Concepts and state of the art **11**

2.1 Introduction 12

2.2 Concepts 12

2.2.1 Software engineering 12

2.2.1.1 Model-driven engineering 13

2.2.1.2 Architectural description languages 15

2.2.2 Model transformation 16

2.2.2.1 Model transformation concepts 17

2.2.2.2 Model transformation classifications 18

2.2.3 Safety-critical systems 18

2.2.3.1 Certifications and criticality levels 19

2.2.3.2 Ravenscar profile 19

2.2.4 Real-time systems 20

2.2.4.1 Real-time task model 21

2.2.4.2 Real-time scheduling theory and analysis 23

2.2.5 Formal methods 25

2.2.5.1 Formal specification 25

2.2.5.2 Formal verification 26

2.3 Modeling of real-time systems . . . 28
2.3.1 MARTE . . . 28
2.3.2 AADL . . . 30
2.3.3 Discussion . . . 32
2.3.3.1 AADL modeling tools . . . 33
2.4 Formal specification of real-time systems . . . 34
2.4.1 Automata . . . 35
2.4.2 Petri nets . . . 35
2.4.3 Process algebra . . . 36
2.4.4 Discussion . . . 37
2.5 AADL related approaches . . . 39
2.5.1 Classification according to the source model . . . 40
2.5.2 Classification according to the target formalism . . . 40
2.5.3 Discussion . . . 44
2.6 Conclusion . . . 45

3 Formal pattern **47**
3.1 Introduction . . . 48
3.2 LNT language . . . 48
3.2.1 Syntax . . . 48
3.2.2 Some definitions . . . 49
3.2.2.1 Module . . . 49
3.2.2.2 Types and channels . . . 49
3.2.2.3 Function . . . 50
3.2.2.4 Process . . . 51
3.2.2.5 Gates and parameters . . . 51
3.2.2.6 Statement . . . 52
3.2.3 LNT specification . . . 53
3.3 A Ravenscar compliant task model . . . 54
3.4 Formal mapping . . . 56
3.4.1 Task definition . . . 56
3.4.2 Scheduler definition . . . 59

3.4.2.1 Time allocation . . . 62
3.4.2.2 Task state updating . . . 63
3.4.2.3 Task activation . . . 63
3.4.2.4 Sporadic task checking . . . 63
3.4.3 Communication mapping . . . 64
3.4.4 Composition and synchronization . . . 65
3.4.5 Discussion . . . 67
3.5 Conclusion . . . 68

4 AADL model transformation 69
4.1 Introduction . . . 70
4.2 AADL in a nutshell . . . 70
4.2.1 Core language . . . 70
4.2.1.1 Components . . . 71
4.2.1.2 Connections . . . 73
4.2.1.3 Properties . . . 73
4.2.1.4 AADL system modeling . . . 73
4.2.2 Behavior annex . . . 74
4.2.3 AADL subset . . . 75
4.3 Model transformation . . . 76
4.3.1 Scheduling mapping . . . 78
4.3.1.1 Thread mapping . . . 78
4.3.1.2 Processor mapping . . . 80
4.3.2 Communication mapping . . . 81
4.3.2.1 Port mapping . . . 81
4.3.2.2 Port connection mapping . . . 83
4.3.3 Hierarchical mapping . . . 85
4.3.3.1 System mapping . . . 86
4.3.3.2 Other rules . . . 86
4.3.4 Discussion . . . 88
4.4 Transformation of a larger AADL subset . . . 88
4.5 Behavioral mapping . . . 90

4.5.1 Data mapping . . . 90
4.5.2 Port and port connection new mapping . . . 91
4.5.3 Behavior specification mapping . . . 93
4.5.3.1 Variables and states . . . 93
4.5.3.2 Transitions . . . 94
4.5.3.3 Actions . . . 97
4.6 Conclusion . . . 98

5 Implementations and validation 99
5.1 Introduction . . . 100
5.2 Ocarina architecture . . . 100
5.3 Ocarina extensions . . . 102
5.3.1 Behavior annex parsing . . . 102
5.3.2 LNT code generation . . . 103
5.3.2.1 Model transformation . . . 104
5.3.3 SVL script generation . . . 105
5.3.3.1 SVL language . . . 105
5.3.3.2 SVL script for AADL model . . . 105
5.4 Tool-chain . . . 106
5.5 Case studies . . . 108
5.5.1 AADL modeling . . . 108
5.5.1.1 Flight control system . . . 108
5.5.1.2 Line follower robot . . . 109
5.5.1.3 Pacemaker . . . 111
5.5.2 LNT code generation . . . 115
5.5.3 Formal verification . . . 117
5.5.3.1 Compilation: state space generation . . . 118
5.5.3.2 Verification: model-checking . . . 120
5.5.4 Analysis results . . . 123
5.5.5 Manual verification . . . 124
5.6 Scalability . . . 126
5.6.1 Test suite . . . 126
5.6.2 Results and interpretations . . . 126
5.7 Conclusion . . . 129

6 Conclusion and perspectives **131**

6.1 Conclusions 131

6.1.1 Reminder of the contributions and results 131

6.2 Perspectives 133

Bibliography **135**

List of Figures

1.1 Integration of formal methods in the design phase 3
1.2 Integration of formal methods in an AADL model-based development process 7

2.1 Main concepts of model transformation 17
2.2 Main parameters of a real-time task (Gantt diagram) 22
2.3 Basic idea of model-checking [93] 27
2.4 Architecture of the MARTE Profile [7] 30
2.5 Main AADL graphical components 31

3.1 Example of LNT specification: graphical representation 53
3.2 Task state automaton 55
3.3 `SCHEDULER` algorithm: ready task 62
3.4 `SCHEDULER` algorithm: task-loops 62
3.5 *Producer-Consumer*: LNT graphical `MAIN` 67

4.1 Summary of AADL elements [61] 71
4.2 Example of AADL system model: graphical representation 74
4.3 Overview of the AADL2LNT transformation 77
4.4 AADL `thread` transformation rule 78
4.5 AADL *thread scheduling and execution states* automaton [10] . . . 80
4.6 AADL `processor` transformation rule 80
4.7 AADL port connection transformation rule 83
4.8 AADL `device` transformation rule 87
4.9 `SCHEDULER` algorithm: EDF scheduling 90

5.1 Ocarina compiler architecture 101
5.2 Ocarina-CADP tool-chain 107
5.3 Flight control system 109
5.4 *FCS* AADL model 109

5.5 *Robot* AADL model 110
5.6 A pacemaker implantation (taken from [47]) 112
5.7 *Pacemaker* AADL model 113
5.8 Generated LTS corresponding to Listing 5.11 120
5.9 Analysis results corresponding to Listing 5.10 125

List of Tables

2.1 Classification according to the source model 40
2.2 Classification according to the target formalism 41

4.1 LNT channels for the behavioral mapping 92
4.2 AADL port transformation rule 92
4.3 AADL port connections transformation rule 92
4.4 Behavior annex transitions transformation rule 96
4.5 Behavior annex actions transformation rule 97

5.1 Case studies 108
5.2 Case studies transformation and verification metrics 116
5.3 State spaces results of family (i) 128
5.4 State spaces results of family (ii) 128
5.5 State spaces results of family (iii) 128

List of Listings

3.1 LNT module definition . . . 49
3.2 LNT type definition . . . 50
3.3 LNT channel definition . . . 50
3.4 LNT function definition . . . 50
3.5 LNT data statements . . . 50
3.6 LNT process definition . . . 51
3.7 LNT behavior statements . . . 51
3.8 LNT formal gates and parameters . . . 52
3.9 LNT communication definition . . . 53
3.10 `TASK` LNT skeleton . . . 58
3.11 `SCHEDULER` LNT skeleton . . . 60
3.12 `CONNECTOR` LNT skeleton . . . 65
3.13 LNT types and channels for `TASK-SCHEDULER` synchronization . 66
3.14 *Producer-Consumer*: LNT `MAIN` . . . 66
4.1 Example of AADL system model . . . 74
4.2 Example of AADL `thread` component . . . 76
4.3 LNT `THREAD` for AADL independent `thread` . . . 79
4.4 LNT type and channel for AADL `data` component . . . 81
4.5 LNT `THREAD` for AADL `thread` with port connections . . . 82
4.6 LNT `CONNECTOR` for AADL data port connection . . . 84
4.7 LNT `Event_CONNECTOR` I . . . 85
4.8 LNT `Event_CONNECTOR` II . . . 85
4.9 LNT `DEVICE` for AADL `device` component (example) . . . 87
4.10 LNT `THREAD` for AADL `thread` with Behavior annex (example) 91
4.11 `Behavior_Specification` . . . 93
4.12 LNT `Type_STATES` type for Behavior annex states . . . 94
5.1 `Behavior_annex` grammar rule [9] . . . 102
5.2 SVL script skeleton for AADL models . . . 106
5.3 *Robot*: `Control Behavior_Specification` . . . 111
5.4 *Pacemaker*: `Pacemaker system implementation` . . . 112
5.5 *Pacemaker*: `VVIMode Behavior_Specification` . . . 114
5.6 *Pacemaker*: `Dual_Or_Timer_VRP Behavior_Specification` 114
5.7 *Robot*: extract of `THREAD_CONTROL` process . . . 116
5.8 *Pacemaker*: extract of `THREAD_VVIMode` process . . . 117
5.9 *Pacemaker*: extract of `THREAD_DUAL_OR_TIMER_VRP` process . . 117
5.10 *Pacemaker*: mini SVL script . . . 118
5.11 *Robot*: smart generation with the SVL language . . . 119
5.12 SVL schedulability property . . . 122

5.13 SVL preemption property 122
5.14 SVL connection property 122
5.15 SVL data loss property 122
5.16 SVL FIFO property 123
5.17 SVL transition property 123
5.18 *Pacemaker*: SVL normal rhythm property 125
5.19 *FCS*: SVL order reachability property 125

List of Algorithms

1 SCHEDULER algorithm: *Operational part* 61

1 Introduction

1.1 Context et motivation

Software engineering in safety-critical domains, like transport and health, is a quite delicate field in computer science. In such a context, designers often cope with distributed, real-time and embedded systems with several constraints, requiring an exhaustive verification as early as possible in the development process. Particularly, real-time systems are an important research topic in software engineering. They manifest an exponential growing in size, complexity and criticality, since they are used in sensitive large scale systems such as public transport systems or nuclear power stations. For example, a flight control system requires up to several hundred functions for the aircraft safe flight, communication with airport stations and additional services for the passenger comfort.

Recently, several approaches concerning the software specification, design, implementation and verification/validation have been proposed for the development of safety-critical real-time systems. The MDE (Model-Driven Engineering) methodology is a development trend based mainly on modeling language, model transformation, production of documentation and code generation. The MDE approaches aim to abstract system representations (models) and allow a coherent evaluation of the system from the specification until the final application. The basic idea of MDE is to describe the system through different models to cover its different aspects (architecture, behavior, performance, deployment, etc.), which lays the emphasis on models rather than programs. In this context, system programs are often generated from models. For example, the architectural models, which are designed using ADLs (Architectural Description Language), allow to describe systems in sufficient detail (software, hardware and configuration) so that system programs can be generated from them.

Taking such an important role in the development process, models have to be analyzed and verified carefully to allow the early detection of design flaws that can lead to serious errors in the final application: it is well known that a majority of errors is often introduced at early stages, while they are discovered late in the development process [62]. Especially for real-time systems considering both concurrency and real-time requirements, it is necessary to validate temporal and scheduling choices from the design phase. During the past decades, important research approaches have been devoted for system modeling and analysis, which have given rise to sophisticated modeling languages (may also be referred to as design languages) with rich ecosystems (tools, extensions, analysis, etc.), such as the UML standard [8] (with its profiles) and AADL standard [10] (an ADL for real-time systems modeling). These modern languages provide the ability to analyze models by supporting non-functional properties (e.g. timing constraints) or to analyze specific notations (e.g. data flows).

In the context of system verification and validation, formal methods have become one of the advocated techniques in safety-critical software engineering [166]. They are mathematically-based techniques designed to aid in the specification and verification of software and hardware systems. The integration of formal methods in MDE approaches seems rewarding, especially that their use has become recommended in system certification (e.g. RTCA/DO-333 [6] , formal methods supplement to RTCA/DO-178C [5] and RTCA/DO-278A certifications in the avionics domain). For this end, a common approach is to translate design models (such as UML diagrams or AADL models) into formal specifications to be verified by formal analysis tools [63].

The work performed in the present thesis fits into the context of the formal verification of real-time systems in safety-critical domains. In the following, we detail different problems around the integration of the formal methods in an MDE approach and then we give an overview of our objectives and contributions.

1.2 Problem statement

Different key issues are of concern in this work. Generally, they are related to a common challenge, which is the integration of formal methods within an MDE approach for the verification of real-time systems.

The use of formal methods requires a formal expertise: the system should be specified with specific languages (formalisms) such as Petri nets, automata and process algebras, based on formal semantics described using mathematical approaches, to be explored by dedicated analysis tools. Indeed, the lack of formal semantics of design languages makes them inappropriate for formal verification (formal techniques can not be directly applied on design models). For this reason,

the verification in MDE approaches is often based on a transformation process. Thus, the adoption of formal methods requires the definition and the implementation of an interface between modeling and verification tools. A model transformation and formal verification steps are integrated, as shown in Figure 1.1, in order to transform the design model into a formal specification and then allow its formal verification. This seems a practical solution to join modeling and formal verification tools, however, it raises a set of fundamental problems, detailed in the rest of this section.

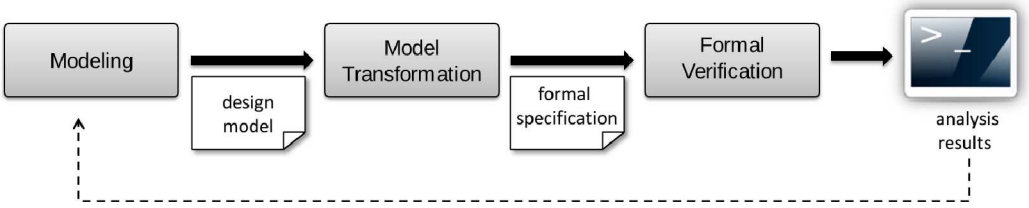


FIGURE 1.1: Integration of formal methods in the design phase

Complexity of real-time system design

A first common issue concerns the complexity of the development of real-time systems. Aside from the functional requirements, real-time systems are distinguished by additional non-functional information compared to traditional systems. They should respect a set of timing constraints, defined as deadlines that have to be met. The violation of these constraints is considered as a system failure even if the functional behavior is correct. In addition, real-time systems are often designed according to multi-tasking architectures where system functions are implemented with several tasks. These tasks may share computation resources and critical hardware/software resources such as processor, memory, variables, etc. They are then in concurrency for the execution, communication and resource access, so they must be scheduled and synchronized to ensure their timing constraints. During the design phase, all these real-time features should be fixed and validated, which requires a timing and concurrency based-design in modeling, as well as in formal specification. Therefore, a first problem in this thesis is the following: *How to model real-time systems so as to enable their verification?*

Model transformation issues

The model transformation is a challenging research topic, particularly in front of the complex structure of models designed by modern languages. The design models are increasingly condensed with information about different modern aspects (time, resources, networks, etc.). The model transformation can be simply defined as the translation of a source model, designed for one purpose, to a target

model better suited for other purposes. It is based on a transformation description that defines the relation between the source and target models. A transformation description may be expressed using transformation languages such as ATL [92], or it can be a direct model manipulation. During the transformation, some information are considered, while some others are ignored, and some constructs from the source model must undergo significant simplification or refinement before being translated into new constructs in the target model. Defining a model transformation is then a complex exercise with several steps, requiring skills (a clear understanding of the syntax and semantics) about both the source and target languages. Therefore, a basic problem in this thesis is the following: *How to hide the model transformation complexity?*

Another challenge in the model transformation is the semantics preservation, which means that the transformation does not jeopardize the semantics of the models under consideration. In the case of formal verification ends, the model transformation is defined between design models (written in design languages) and formal specifications (written in formal languages). The formal languages are based on a well-defined formal semantics, in contrary, the semantics of design languages, such as UML or AADL, is often given in natural language (i.e. standard and manual documents). Due to this semantic gap between design and formal models, the semantics preservation of this kind of model transformation can not be proved, which represents a challenging issue considered yet as an unsolved problem [16]. Thus, an important problem in this thesis is the following: *How to decrease the semantic gap between design and formal models?*

Formal verification challenges

The application of formal methods in the verification of safety-critical systems has produced promising results in the industrial context [23, 127]. Yet, they still have a reputation to be hard to use and to require an expertise both in advanced mathematics and also in the internal operation of dedicated tools. Providing the formal specification is an important step, but further steps are required to accomplish the verification. Initially, the formal techniques are applied on the state space of the system, built from the formal specification according to the semantics of the considered language. The properties, representing the system requirements to be verified, should be also specified as graphs or temporal logic properties using dedicated formalisms. Thereafter, the verification can be performed by means of analysis tools. In that respect, a crucial problem is addressed in this thesis: *How to guide/assist designers in the use of formal techniques?*

Analysis results usefulness

The formal verification phase ends by the generation of analysis results. Being produced by analysis tools, these results may be hard to understand and to inter-

pret in terms of the design model. In addition, the verification process may fail, because of the well-known state space explosion problem. This problem occurs when the system state space (mostly of large specifications) becomes too large to be analyzed, which represents a major obstacle to the application of formal methods, especially in the industrial context. This raises two key questions: *How to interpret the analysis results?* and *How to avoid the state space explosion problem?*

1.3 Objectives

The main objective of this thesis is to assist real-time systems designers during the design phase. Precisely, we aim to provide a systematic and rigorous development process by the integration of formal methods in an MDE approach. The proposed solution is based on a model transformation operation allowing the generation of a formal specification, in order to enable the verification with an existing analysis tool. To achieve this ultimate objective, the following points have to be taken into account:

- Providing an adaptable solution, easily integrated within an MDE approach;
- Providing a solution to decrease the semantic gap between design and formal models;
- Ensuring a complete automatic and transparent solution for both the model transformation and formal verification;
- Generating a comprehensible analysis results that are easily interpreted by non-formal expert designers;
- Providing a rapid, traceable, scalable solution.

1.4 Contributions

To tackle the problems mentioned above and achieve our objectives, we propose our solution in the context of the formal verification of real-time systems, that consist of the main lines summarized in the following.

1.4.1 Formal pattern for a real-time task model

We propose a formal pattern for real-time systems. To tackle the semantics presentation problem, discussed in section 1.2, we use a standard task model as pivot representation between design and formal models: the design model is abstracted as a set of tasks with their temporal parameters, which simplifies and reduces semantic ambiguity in transformation. Instead of considering the whole model to define the transformation, the real-time constructs such as tasks and connections are extracted and translated into equivalent formal patterns.

The considered task model is based on a conventional tasking model inspired from Liu and Layland [110] model. In addition, rigorous semantics and strong requirements are applied following the Ravenscar profile [43] for safety-critical systems. We consider periodic and sporadic tasks, which are asynchronously connected and concurrently executed by a preemptive fixed-priority scheduler.

The proposed pattern is described and justified in chapter 3. It is designed to be modular, generic and comprehensible and so it can be easily extended and used in MDE approaches. This pattern is specified with the LNT (LOTOS New Technology) [48] language, which is a process algebra based on two standards LOTOS [1] and E-LOTOS [3]. This choice is encouraged by the expressiveness and richness of LNT (discussed in section 2.4 of chapter 2). It provides sufficiently expressive operators for data and behavior with a user-friendly notations to simplify writing and extension. LNT is supported by the CADP (Construction and Analysis of Distributed Processes) [69] toolbox, which is a well experimented analysis tool, used in many industrial applications (e.g. Airbus [66]).

1.4.2 Application in an MDE approach

In this second contribution, we propose an automatic solution for the integration of formal methods within an MDE approach. For real-time systems design, we opt for architectural models, where the overall structure of the system may be specified in one model composed of the principal components, their relationships and their configurations. We choose the AADL (Architecture Analysis and Design Language) [10] modeling language (for reasons discussed in section 2.3 of chapter 2). AADL is an industrial SAE [1] standard for real-time embedded systems modeling in safety-critical domains such as avionics, automotive electronics and robotics. The AADL core language provides a rich syntax and semantics, sufficient enough to describe concepts required for real-time systems design as discussed in section 1.2. In addition, the AADL semantics can be extended via user-defined properties and annexes (separate sub-languages). For instance, the

[1] SAE: Society of Automotive Engineers

standard Behavior annex [9] extends AADL models with behavioral specifications.

As shown in Figure 1.2, we deal with an AADL model-based development process, where the design phase is completed with the following verification related activities:

- AADL2LNT model transformation (chapter 4): based on the proposed LNT pattern, we define the AADL2LNT model transformation to translate an AADL model into an LNT specification. The transformation is described by a set of correspondence rules between AADL and LNT. Firstly, an architectural mapping is ensured through three levels: scheduling, communication and hierarchical mapping. Then, the transformation is extended by a behavioral mapping level, to support the AADL Behavior annex. To tackle problems discussed in section 1.2, all the AADL2LNT transformation steps are automated to generate an LNT specification ready for the verification with the CADP toolbox.

- Automatic formal verification (chapter 5): to address problems discussed in section 1.2, we propose an automatic formal verification phase allowing to simplify and encourage the practice of formal methods in software engineering. In addition to the LNT specification, a second input is provided for the CADP toolbox. As shown in Figure 1.2, a script file written in the SVL (Script Verification Language) [67] language is also generated to orchestrate the verification phase. It contains mainly a set of generic properties to be verified by model-checking. These properties allow the detection of serious problems at the design phase, such as the deadlock detection, schedulability test and the detection of connection failures (FIFO overflow, loss of data, broken links).

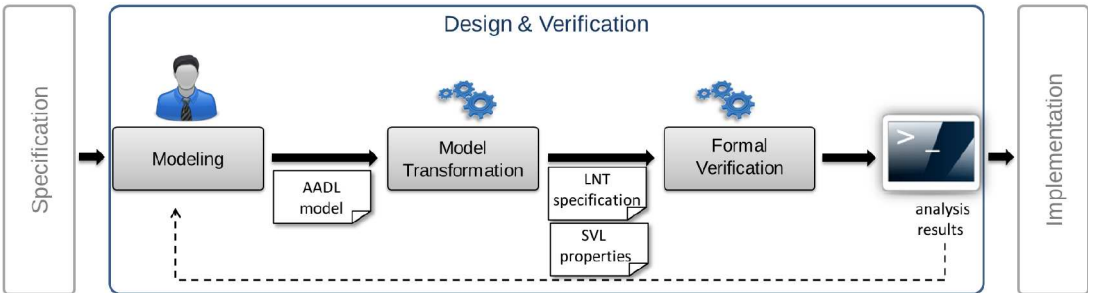


FIGURE 1.2: Integration of formal methods in an AADL model-based development process

1.4.3 Tool-chain, analysis results and scalability

The AADL2LNT transformation is implemented as an extension of Ocarina [72], an existing tool suite for AADL modeling and code generation. This extension [2] allows to form a tool-chain based on Ocarina for architectural modeling and CADP for formal verification. We illustrate the capabilities of the Ocarina-CADP tool-chain with various case studies from the AADLib library [3] (chapter 5).

For the analysis results (problems discussed in section 1.2), which are generated by the CADP model-checkers, we provide a user-friendly (simple) output form that is easily interpreted with non-formal expert designers. This is based on a traceable transformation of the LNT specification and SVL properties, in order to preserve information from the AADL model in the analysis results (commented to explain each property, decorated with AADL identifiers, etc.).

As part of our experiment, a scalability study is carried out to address the state space explosion problem discussed in section 1.2. The proposed solution is exhaustively tested using a test suite composed of 100 AADL models, to evaluate its time and memory performances (section 5.6 of chapter 5).

1.5 Thesis organization

This section gives a brief summary of the contents of the chapters of this thesis as follows:

- Chapter 2 is devoted to the research foundations and the state of the art. Firstly, it presents key concepts related to software engineering of real-time systems about the development process, MDE methodology, modeling language and model transformation. It also highlights the common timing constraints and scheduling policies of real-time systems. In addition, it gives an overview of formal methods for the specification and verification of safety-critical systems. Secondly, this chapter includes detailed discussions about the languages and tools for modeling and formal specification considered in this thesis. Finally, we end this chapter with a study of a set of related approaches about the formal verification of AADL models.

- Chapter 3 represents the first contribution in this thesis, it develops the proposed LNT pattern for a real-time task model. It begins with an introduction of the LNT language. Then, the supported task model is described. Finally, different parts of the LNT pattern are defined and discussed.

[2] The AADL2LNT extension is deployed in the official Ocarina GitHub repository to be available for the academic and industrial users: https://github.com/OpenAADL/ocarina

[3] AADLib is a library of reusable AADLv2 models under the OpenAADL project (https://github.com/OpenAADL/AADLib)

- Chapter 4 represents the AADL model transformation. The AADL language is firstly introduced. Then, the AADL2LNT transformation is described through its transformations rules. Finally, we present the behavioral extension that adds the mapping of the AADL Behavior annex.
- Chapter 5 gives an overview of our implementations and experiments. Firstly, this chapter explains how the model transformation and formal verification phase are automated through the provided tool-chain. Secondly, three case studies are developed to illustrate different contributions from the modeling phase until the analysis results interpretation. Finally, a scalability study is carried out to show the effectiveness of our solution in the formal verification of realistic large scale systems.
- Chapter 6 concludes the thesis and outlines some directions for future work.

2

Concepts and state of the art

Contents

2.1 Introduction 12

2.2 Concepts 12

2.2.1 Software engineering 12

2.2.2 Model transformation 16

2.2.3 Safety-critical systems 18

2.2.4 Real-time systems 20

2.2.5 Formal methods 25

2.3 Modeling of real-time systems 28

2.3.1 MARTE 28

2.3.2 AADL 30

2.3.3 Discussion 32

2.4 Formal specification of real-time systems 34

2.4.1 Automata 35

2.4.2 Petri nets 35

2.4.3 Process algebra 36

2.4.4 Discussion 37

2.5 AADL related approaches 39

2.5.1 Classification according to the source model 40

2.5.2 Classification according to the target formalism 40

2.5.3 Discussion 44

2.6 Conclusion 45

2.1 Introduction

The main intent of this chapter is to present this thesis domain of interest. Firstly, we present the general concepts that are necessary to comprehend the tackled issues and proposed contributions. Then, a set of modeling languages and related approaches are studied to justify our choices and compare our work with existing solutions.

This chapter is organized as follows: section 2.2 highlights fundamental notions about respectively software engineering, model transformation, safety-critical systems, real-time systems and formal methods; section 2.3 and 2.4 expose and discuss different existing solutions for real-time modeling and specification; finally, section 2.5 surveys a set of related work about AADL model transformation approaches.

2.2 Concepts

The work of this thesis presents a software engineering solution based on the intersection of different domains, mainly, the safety-critical domain, real-time systems, architectural languages and formal methods. In this section, these notions are presented according to reference papers and surveys.

2.2.1 Software engineering

The software engineering (or software development) is the discipline that concerns all aspects of software production. It is based on a development process (or development life-cycle) defining the approach that is taken as software is engineered [141]. Otherwise, the development life-cycle can be defined as an assisted treatment process that decomposes the development of a product into a set of steps (phases) structured according to a certain philosophical approach [35]. There exist different software processes, but all must include a set of fundamental activities [160] as follows:

- *Software specification* (requirements engineering) is the phase of understanding/defining services of the system and identifying the constraints on its operation and development. This phase aims to produce an agreed requirements document, which contains a set of requirements represented at two levels: end-users high-level statement of the requirements; and system developers detailed system specification.

- *Software design* is the phase of describing the structure of the software to be implemented, that may include data models and structures, system components and the interfaces between these components, algorithms, protocols, etc. In addition, it may include other activities depending on the type of system. For example, real-time systems require timing design and database systems require a database design phase. This phase involves the production of several design documents (may be diagrams) of the system at different levels of abstraction.

- *Software implementation* is the phase of converting a system design into an executable system. To facilitate the programming task, there are many software development tools that provide the generation of a skeleton program from a design (code generation) to be completed by programmers, especially in the case of large and complex systems. This phase is normally achieved with testing and debugging activities to detect, localize and correct program defects.

- *Software validation* phase aims at verifying that a system conforms to its specification (verification) and that it is compliant with the expectations of the system end-users (validation). This phase is ensured by: the design analyses, where different produced documents are checked at each stage of the development process from the user requirements definition to the program development; and the program testing, where the system is executed using simulated tests.

These basic development activities are differently organized and completed with other steps in different conventional development processes, such as linear sequential mode, waterfall mode, prototyping mode, V-cycle mode, incremental mode, spiral mode, etc. Theoretically speaking, different phases are successively handled: the development process moves to the next phase by using the deliverables of the previous phase as inputs. In practice, the development process is rarely linear, especially with large systems where changes are inevitable (end-user new requirements, new technologies, etc.). The development process may then involve several refinements over preceding and succeeding phases. For example, in the waterfall mode [138], the development phases are organized in sequence, whereas in the incremental mode [105], they are interleaved.

2.2.1.1 Model-driven engineering

The concept of MDE (Model Driven Engineering) has emerged as a generalization of the MDA (Model Driven Architecture) which is a software engineering

approach for the development of software systems proposed by the OMG [1] organization in 2001. Today, MDE and MDA are often seen as the same thing, to designate a software development trend where models are the principal outputs: system models are created (at different levels of abstraction) and transformed throughout the development life-cycle phases into the concrete final implementation [63, 95]. In this context, important research work has been devoted for the system modeling, concerning the definition of modeling languages, model transformation techniques, code generation and the creation of tools for model processing.

Model

The "what is a model ?" question was well discussed in the literature, yet there is no unified definition of the model. Definitions such as *Everything is a model* [32] or *A model is a set of statements about some system under study* [152] are among the famous watchwords used in MDE to generally define a model. In the context of this thesis, we consider a model as an abstract view (a simplified representation) of a system built for specific purposes [33].

Modeling language

A model is written in a well-defined modeling language which is a language with well-defined form (syntax) and meaning (semantics), suitable for automated interpretation by a computer [95].

Defining the structure of a modeling language consists of the design of its abstract syntax (the basic structure of the language), its concrete syntax (the graphical and/or textual representations for end-users in order to manipulate models) and its static and dynamic semantics (the meaning of different elements of the model).

A formal grammar (such as BNF (Backus-Naur Form) [122]) may be used to describe the syntax of the language. The static semantics defines structural properties of models, that can be determined without considering either inputs or execution (e.g. identifiers, statements and expressions). The dynamic semantics is concerned with the execution and behavior of the specified models [17]. These semantic rules may be defined informally by natural language or formally by describing syntactical elements in terms of a formal approach. In this case, the language is commonly known as a formal language (or formal specification language). The formal semantics may be defined using two main approaches:

- *Translational semantics* is based on the mapping of the language constructs using another language with an already defined formal semantics. The

[1] OMG: Object Management Group

target language may be a mathematical formalism (denotations), known as the denotational semantics;

- *Operational semantics* specifies explicitly the execution of the language (rather than by translation). There are two operational semantics categories: SOS (Structural Operational Semantics) [140] to describe the computation steps of the execution of the language; and the natural semantics to describe how the overall results of the execution are produced.

Classification of modeling languages

In the literature, languages are commonly classified as domain-specific or general-purpose languages. So a modeling language may be a GPML (General-Purpose Modeling Language) or a DSML (Domain-Specific Modeling Language).

The GPML is a modeling language, providing generic concepts to model systems in any domain. The UML (Unified Modeling Language) language, proposed by the OMG organization since 1997 to support the object-oriented programming, is a typical example of GPML with a very broad scope that covers a diverse set of application domains. UML [8] is a standard language with graphical notations representing different views of a system using a set of 13 different diagram types (mainly structure, behavior and interaction diagrams).

Contrary to GPMLs based on universal concepts, the DSMLs are specialized languages which capture the concepts of a specific domain. There is a wide range of domain-specific language used in common domains, such as the HTML (Hypertext Markup Language) for developing web pages, VHDL for hardware descriptions and SQL (Structured Query Language) for relational database queries, that has evolved into the PL/SQL general-purpose procedural language. The UML language includes a profile mechanism (to extend the language to a particular domain) that can be used to develop domain-specific modeling languages [153]. A set of UML profiles has been proposed which may be considered as DSMLs such as MARTE (Modeling and Analysis of Real-Time Embedded systems) for modeling real-time embedded systems.

2.2.1.2 Architectural description languages

The notion of software architecture has been proposed, since the early 1990s by Perry and Wolf [137], as a software system representation composed of a set of components with their interactions and their constraints [135]. A software architecture is designed using an architectural language (may also be referred to as ADL (Architectural Description Language)), which is generally defined as *any form of expression for use in architecture descriptions*, according to the ISO/IEC/IEEE standard [4]. In the context of this thesis, we consider an ADL

as a modeling language that provides concepts to describe the architecture of a software system. Medvidovic and Taylor [123] differentiate, in detail, ADLs from other modeling languages such as programming languages, object-oriented modeling notations and formal specification languages. In general, existing languages, that are commonly referred to as ADLs, are distinguished by the explicit specification of three main architectural elements as follows:

- Components: units of computation or data stores;
- Connectors: interactions among components and rules that govern those interactions;
- Architectural configuration: connection graphs of components and connectors that describe architectural structure.

During the past decades, the topic of the software architecture modeling and analysis has been shown great interest, and an important number of ADLs has been defined by the academic and industrial communities. Recent surveys, such as [116] and [135], discuss tens of existing ADLs in term of language definition, language features and provided tools: Malavolta *et al* [102] maintain an up-to-date list of existing ADLs, available in [117], containing currently more than one hundred ADLs.

An ADL can either be a GPML that is used to specify any type of systems like Wright [13], and Rapide [112], or it may be a DSML for architectural descriptions of a particular domain such as AADL [10] for real-time embedded systems, Koala [165] for consumer electronics and Mobis [118] for mobile systems [135].

2.2.2 Model transformation

One of the most fundamental theme in the MDE context is the model transformation. It is defined as the automatic generation of a target model from a source model, according to a transformation definition [95]. This allows various manipulations of models for different purposes: model abstraction (transformation toward a higher level specification), model refinement (transformation toward a lower level specification), optimization (transformation to improve operational qualities of the model such as scalability), model analysis, code generation (synthesis of the model), reverse engineering (e.g. transformation from the code to the model), model composition (e.g. transformation for merging) and translational semantics [124] [111]. In the following, we highlight the main concepts and classifications behind the topic of the model transformation.

2.2.2.1 Model transformation concepts

A model transformation is composed of a set of artifacts, as shown in Figure 2.1:

- a *source model*, which is written in a *source language*, is transformed into a *target model*, written in a *target language* according to the *transformation definition*;
- the *transformation definition* is a set of transformation rules that together describe how a model in the *source language* can be transformed into a model in the *target language* [95]:
  - a transformation rule is a description of how one or more constructs from the *source language* can be transformed into one or more constructs in the *target language*.
- the *transformation engine/tool* automates the operation of the transformation by executing the *transformation definition* (applying the set of the transformation rules) on concrete models (source and target models).

In general, a transformation may be applicable to multiple source models and/or multiple target models. In addition, the source and target languages may be the same in some situations. A transformation can be described using a transformation language such as ATL (Atlas Transformation Language) [92] and QVT (Query View Transform) [101], in which the transformation is defined between meta-models (structure of models) of the source and target models. Or it can be a direct model manipulation, where the transformation is directly accessed by means of an API (implemented using programming languages) [124].

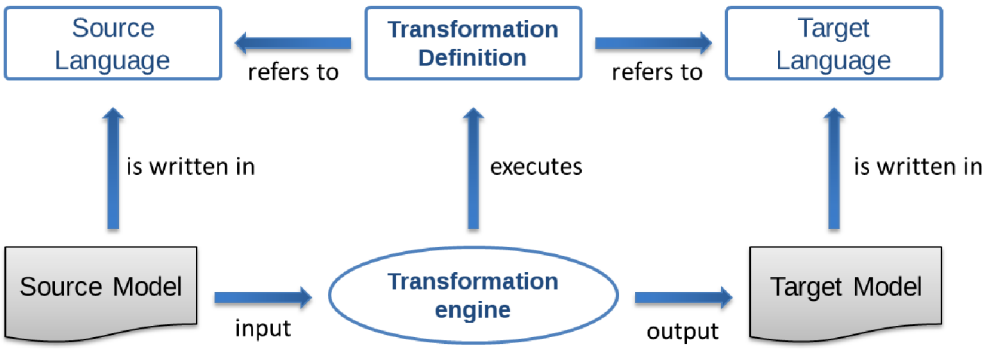


FIGURE 2.1: Main concepts of model transformation

2.2.2.2 Model transformation classifications

In the literature, the model transformations are classified according to many different criteria. In the following, we include some common classifications [124]:

- Endogenous vs. exogenous transformations: this classification is based on the language in which the source and target models are expressed. The transformation whose source and target languages written in the same language are referred to as endogenous (e.g. optimization, merging), whereas the transformation with different source and target languages are referred to as exogenous (e.g. code generation).
- Syntactical vs. semantical transformations: this classification distinguishes between model transformations that just transform the syntax of the source model (e.g. import or export a model in a specific format), and more sophisticated transformations that also consider the semantics (e.g. optimization).
- Model-to-text vs. model-to-model transformations: this criterion concerns the target format, a transformation which allows generating texts from source models is referred to as a model-to-text transformation, whereas, it is a model-to-model transformation.
- Unidirectional vs. bidirectional transformations: the directionality of a transformation can be a criterion to classify unidirectional (only from source to target) transformations and bidirectional (from source to target and from target to source) transformations.

In the context of this thesis, the proposed transformation deal with only one source model and produce one target model for verification ends. It may be categorized as unidirectional, semantical and exogenous model-to-model transformation.

2.2.3 Safety-critical systems

In the literature, several definitions and classifications are used to qualify safety-critical systems. These systems are intuitively distinguished by the consequences associated with the system failure. As mentioned in [96], *If the failure of a system could lead to consequences that are unacceptable, then the system is safety-critical.* That is, a system can be considered to be critical if the failure may cause serious financial, environmental or human losses. This concerns several systems in vital domains like health, transport, aerospace and chemical/nuclear.

The software engineering of safety-critical systems is particularly difficult. A safety-critical system must respect particular constraints throughout its development process to offer guarantees on its critical aspects. It must be documented, followed from conception to implementation and deployment, tested and validated as well as certified by specialized organizations.

2.2.3.1 Certifications and criticality levels

In general, the certification is referred to as to the confirmation of certain characteristics of the product, process or person. In the case of safety-critical domain, the certification of the produced system is indispensable to be used. For this end, engineering organizations have established standards and guidelines for developers to follow them in designing safety-critical systems in several industries. In this context, the avionics industry has succeeded in defining standards for producing avionics systems. For example, the software part of the system must be certified in accordance with a reference document known as RTCA/DO-178B [2] [2], which is a guideline describing the conditions for assurance in designing software in airborne systems. This certification is adopted by the U.S. FAA [3] and the European EUROCAE organizations, as mandatory for design and implementation of airborne systems [100].

A safety-critical system may be assigned a level of criticality. The criticality levels of a system depend on the impact of its failure. For instance, the certification RTCA/DO-178B [2] defines five levels of safety ranging from the *Catastrophic* level A (failure may cause a crash of the aircraft), *Hazardous* level B, *Major* level C, *Minor* level D, to the *No Effect* level E (failure without impact on the operation of the aircraft).

2.2.3.2 Ravenscar profile

Among the approaches for the development of safety-critical systems is the limitation of software constructs that may compromise the fulfillment of certain safety requirements. In this context, patterns such as the Ravenscar profile [43] are dedicated to drive the software design and implementation for reliable system constructions.

The Ravenscar profile [43, 44] is defined to meet safety-critical real-time requirements such as determinism, schedulability analysis and suitability for certification. Initially, the profile was presented as a set of restrictions of the Ada tasking features, which allows the static analysis for high integrity systems certification. Thereafter, the use of the Ravenscar profile was generalized to be adopted in

[2] RTCA: Radio Technical Commission for Aeronautics
[3] FAA: Federal Aviation Administration

early development phases: it may be applied at the design phase to define models respecting the profile requirements (Ravenscar compliant models).

In the context of this thesis, the Ravenscar profile is used to add strong constraints for a real-time task model. More about the profile restrictions and how it is used in the proposed LNT pattern are detailed in chapter 3.

2.2.4 Real-time systems

The notion of real-time systems has appeared since the 1960s and has rapidly spread to be used in our daily life within diverse domains such as transport, telecommunication, multimedia, etc. These systems are continuously in interaction with their environments, and they are often embedded in hardware devices to provide services autonomously or by cooperating with other entities.

A real-time system is a system whose correctness depends not only on the logical results of the computation, but also on the response time at which the results are produced [162, 97]. Such systems should produce correct calculation results (logical correctness) within a specified and finite period (timing correctness). That is, in addition to the functional behavior, a timing model is defined through a set of temporal parameters, and the timing correctness is mainly modeled as deadlines that have to be respected.

Real-time system categories

Real-time systems may be categorized by different criteria. In the following, we distinguish two categories of the real-time systems according to their criticality levels, which are defined by the severity of consequences that may occur when a deadline is missed [162].

- *Soft real-time systems* [45] can accept timing failures that do not cause system failure or catastrophic damages such as video and audio conference systems.
- *Hard real-time systems* [110] have to respect all their deadlines. Otherwise, the failure consequences are unacceptable and may lead to catastrophic damages. These systems may be considered safety-critical systems when they are used to control critical environments such as automotive, aircraft or nuclear power plants.

Real-time architecture

A real-time system is often embedded within an electronic device that allows the interaction with its environment. It is a computer-based system consisting of hardware and software parts.

The hardware platform is a composition of components such as processors, memories and input/output devices. There are several different types of hardware architectures that have an important impact on real-time scheduling and analysis. The number of the available processors is a main factor in real-time systems design, in such a context, studies are commonly classified under two fundamental categories:

- Multiprocessor architecture: the hardware is composed of a set of processors (or cores) sharing memories.
- Uniprocessor: the hardware is based on a unique processor. All tasks share a single CPU (Central Processing Unit) for the execution.

The software part consists mainly of the RTOS (Real-Time Operating System) and application layers. Briefly, the RTOS is the bridge between the application and the hardware parts, it is based on a *kernel* that manages resources access and provides a scheduler for the scheduling of tasks. In addition, the RTOS ensures various services for sharing resources, synchronizations and communications between tasks. The application in a real-time system is often designed as a set of tasks to be executed with the scheduler.

2.2.4.1 Real-time task model

At the design phase, a real-time system can be considered as a set of cooperative and concurrent tasks. The system is then viewed as a task model, specified using a set of temporal parameters, hiding the architectural complexity. The task (may also be referred to as process or thread) represents a logical unit of computation in a processor [18]. It is the basic entity of the real-time system, that contains a set of sequential instructions to be executed by a processor. Every execution of a task is called a job.

We denote by $S = \{\tau_1, ..., \tau_n\}$, the set of n tasks where τ_i is a task identified with a set of conventional temporal parameters, namely: T_i for the period between two successive releases; C_i for the capacity or WCET (Worst Case Execution Time) per period; D_i for the relative deadline: the maximum time, in which the job has to be executed since its release time; P_i for the task priority (for priority-based scheduling); O_i for the offset: the release time of the first job.

In addition, the j^{th} job, denoted by $\tau_{i,j}$, has a set of dynamic parameters as represented in Figure 2.2: $r_{i,j}$ for the release time; $s_{i,j}$ for the start time; $e_{i,j}$ for the completion time; preemption (a job can be interrupted several times by the processor to execute a job of another task); and $d_{i,j}$ for the absolute deadline.

These parameters allow the definition of the task temporal characteristics. According to how it is released, a task may be periodic or non-periodic [110]:

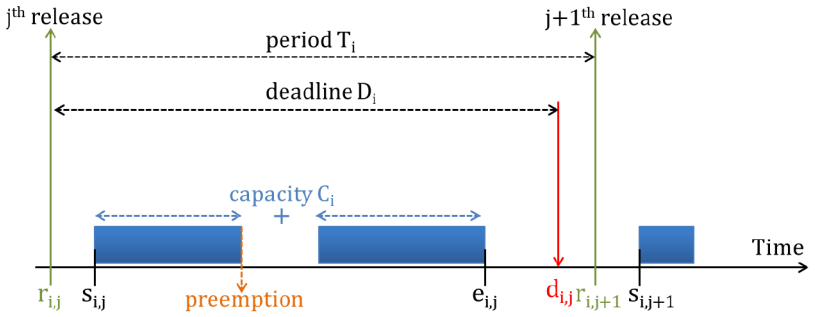


FIGURE 2.2: Main parameters of a real-time task (Gantt diagram)

- A periodic task is regularly released (dispatched or activated) by the processor at fixed time intervals defined by the period T_i parameter. Thus, its release time is equal to $r_{i,j} = O_i + (j-1) * T_i$ and it should meet its absolute deadline $e_{i,j} < d_{i,j} = r_{i,j} + D_i$.
- A non-periodic task is arbitrary dispatched by the processor. If the delay between two successive activations is constrained by a minimum delay, then the task is said to be sporadic. If a task is released at any time, then it is said to be aperiodic, yet when starting the execution, it must complete within a predefined period.

According to its relationship with other tasks, a task may be independent or dependent [18]:

- Independent tasks are independently executed: a task is never blocked by another one.
- Dependent tasks are in interaction with each other, which creates dependency relations between tasks. The tasks can have precedence constraints, when a task should wait for another task results. They may also share some resources (variables, devices, etc.), which do not allow simultaneous accesses, but require their mutual exclusion [46].

According to their deadlines, a set of tasks S may be categorized by implicit, constrained or arbitrary deadlines:

- Implicit-deadlines where the deadlines are exactly equal to the periods $\forall i, D_i = T_i$.
- Constrained-deadlines where the deadlines are less than or equal to the periods $\forall i, D_i \leq T_i$.
- Arbitrary-deadlines where the deadlines and periods are unrelated.

According to their offsets, a set of tasks S may be synchronous or asynchronous:

- Synchronous where all the tasks are simultaneously released $\forall i, O_i = constant$.
- Asynchronous where some of the tasks have different offsets $\exists (i,j), O_i \neq O_j$.

2.2.4.2 Real-time scheduling theory and analysis

Our goal is not to provide an exhaustive study about the scheduling theory and analysis in the real-time domain, but to define some concepts that are required for the comprehension of the scheduling mapping developed in chapter 3. In the following, we give an overview of the concepts of scheduling algorithms and schedulability tests .

Basically, a scheduling problem is the definition of a schedule for the execution of the jobs of a set of tasks, so that they are all completed before their deadlines. A scheduling problem may be defined using three sets: the set of n tasks $S = \{\tau_1, ..., \tau_n\}$, a set of k processors $P = \{P_1, ..., P_k\}$ and a set of m types of resources $R = \{R_1, ..., R_m\}$. The scheduling means assigning processors from P and resources from R to tasks from S in order to complete all tasks under the specified constraints [46]. A scheduling algorithm [18] is then considered (according to some criteria) to describe how tasks are selected for the access to the processor(s) and other shared resources.

A scheduling algorithm (implemented within a scheduler) can be categorized as follows:

- Time vs. event driven scheduling:
  - Time-driven: the scheduling points are determined by the interrupts received from a clock (periodic task scheduling).
  - Event-driven: the scheduler decisions, of which job to execute, are made by certain events (non-periodic task scheduling).
  - Hybrid: the scheduler uses both clock interrupts, as well as event occurrences to make its decisions.
- Preemptive vs. non-preemptive scheduling [46]:
  - Preemptive: the execution of a task may be interrupted at any time to assign the processor to another task.
  - Non-preemptive: a task is executed by the processor until the completion of each job.
- Off-line vs. on-line scheduling [18]:

  - Off-line: all scheduling decisions are pre-calculated before execution of the system.
  - On-line: all scheduling decisions are dynamically calculated during the run-time of the system.

- Fixed-priority vs. dynamic-priority scheduling: the priority-based scheduling is based on the task priorities, in a way that tasks are dispatched in the priority order. At any time, the task with the highest priority is selected by the scheduler to be executed. In this case, two scheduling policies may be used as follows:

  - Fixed-priority: all the priorities of tasks are static and assigned off-line. In this category, we find popular uniprocessor preemptive scheduling algorithms, such as the RM (Rate Monotonic) [110] and DM (Deadline Monotonic) [109] algorithms.
  - Dynamic-priority: all the priorities of tasks are changed during the execution [18]. The EDF (Earliest Deadline First) [85] and the LLF (Least Laxity First) [131] are among the popular scheduling algorithms in this category.

The schedulability analysis decides for a given set of tasks with a scheduling algorithm, whether all the timing constraints will be respected. The problem is addressed in the form of a *schedulability test*: a set of tasks is said to be *schedulable* according to a given scheduling algorithm if none of its tasks, during the execution, will ever miss their deadlines [18].

A *schedulability test* is defined on the basis of a formula which provides a necessary, sufficient or exact condition for a scheduling algorithm to satisfy the timing constraints of a set of tasks:

- A test is defined to be sufficient in the sense that a set of tasks is *schedulable* if it satisfies the test: if the set of tasks fails the sufficient test, it is undecidable whether it can be *schedulable* or not.

- A test is defined to be necessary if all *schedulable* set of tasks satisfy the test: if a set of tasks satisfies the necessary test, it may be *schedulable* but not necessarily. and if it fails the test, then it is definitely decided to be non-*schedulable*.

- A test that is both sufficient and necessary is said to be an exact condition: it is in some sense optimal. A set of tasks is *schedulable* if and only if it does satisfy the exact test.

Many *schedulability tests* have been proposed in the literature (surveyed for example in [154, 58, 163]). As an example, Liu and Layland in [110] propose an exact schedulability test for the RM scheduling: for a synchronous set of tasks S of n independent and periodic tasks under implicit-deadlines, if $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)$, which converges to $\ln 2 (\approx 0.69)$ when $n \to \infty$, then S is RM-schedulable.

2.2.5 Formal methods

The formal methods are mathematically-based techniques used in computer science to describe properties of software and hardware systems. They provide specific framework to rigorously specify, develop and verify systems with consistent, complete and unambiguous solutions [167].

In the software development, the formal methods are particularly advocated in the case of safety-critical systems, and they are explicitly allowed/required in many certifications/standards such as aircraft systems (RTCA DO-178B) [2], military systems [161] and common criteria certification for system security [11].

The formal methods are applied on an abstract mathematical model of the system (called system state space). To apply a vast number of formal techniques, the system is captured by some form of *transition system*, where its behavior is specified as a set of states with a set of transitions (between different states), that may be labeled (by actions/conditions/guards to control the transition from a state to another one). There are several examples of *transition systems*: finite-state machine, labeled transition system, Petri nets, etc. In addition, the systems may be specified using high-level descriptions (known as specification languages), which are translated into a corresponding *transition system*, built according to the specification language semantics, to enable analyzing with verification tools.

2.2.5.1 Formal specification

The specification of a system is known as the step of describing the required behavior of the system. A formal specification is expressed by a formal specification language, in order to describe a system, analyze its behavior and allow the verification of a set of properties. As mentioned above, a modeling language is said to be formal if it is based on a well-defined semantics using mathematical foundations.

Relying on several criteria (structure, semantics, expressiveness, etc.) various classifications of the specification languages have been proposed in many work like [104], [14] and [25], which allows the definition of a set of common categories: state-based languages (e.g. Z [136] and B [37]), transition-based/automata-based

languages (e.g. automata), logic-based languages (e.g. LTL and CTL), process algebras (e.g. CSP, CCS [128]), Petri nets [139] and their extensions, language for real-time systems (e.g. timed-automata [15]), etc. Later in section 2.4, we present and discuss the most popular formalisms used in the formal specification.

State explosion problem

The system state space generation/exploration may face a serious issue known as the state space explosion problem. This problem occurs when the system state space becomes too large (grows exponentially) to be explored by formal tools (exceed resource capabilities of CPU and memory).

In the literature, several solutions have been proposed to address this problem, such as the compositional verification [68], on-the-fly verification [56] and the distribution of the verification between a set of machines or processors to increase the capabilities of verification tools.

2.2.5.2 Formal verification

In the context of formal methods, there are three main approaches commonly used to check a certain kind of properties in a system:

- Static analysis [55] consists of analyzing a program (source code) without actually executing it. It is a set of techniques (e.g. data flow analysis and abstract interpretation) allowing to deduce algorithmically a set of properties used for several purposes such as code optimization, code parallelization, debugging and code understanding.
- Theorem proving [53] consists in formulating the system and the verified property as a formal proof (a mathematical model) that will be solved using a tool called proof assistant/theorem prover (e.g. Coq [24], HOL [73] and PVS [155]). This type of verification often requires human interaction to guide the mapping and the resolution.
- Model-checking [51, 143] is the most popular verification technique, which allows to check if a system satisfies some properties. Two general approaches are used: the equivalence-checking compares the equivalence of two models by applying equivalence or preorder relations; and the model-checking verifies whether a system satisfies a property, often given in temporal logic.

Model-checking

Model-checking was firstly proposed in the early of 1980s [51, 143]. As shown in Figure 2.3, a typical model checking process includes four main steps: construct-

ing a model for the system under verification; defining the properties that the system should satisfy; performing the model-checking; and generating results.

The model-checker verifies whether the system satisfies some requirements/constraints specified as properties (verified properties): if the system definitely holds the properties, the system passes the test, if the system fails the test, a counterexample may be produced, if the time budget or memory is used up during verification, a state explosion problem occurs [93].

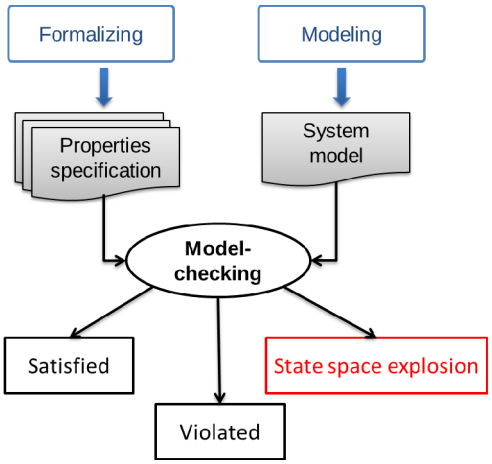


FIGURE 2.3: Basic idea of model-checking [93]

Property specification. The properties to be verified are often expressed using temporal logics [142], which are particular modal logics introduced by Arthur Prior in the late of 1950s. The temporal logic allows to specify a succession of events along the time using time operators such as *eventually*, *until* or *always*. The most important temporal logics used to specify requirements are: LTL (Linear Temporal Logic), CTL (Computation Tree Logic), PSL (Property Specification Language) extending LTL, and modal mu-calculus including subsets of LTL and CTL.

The specification of the properties is based on two main categories, introduced by Lamport [103]: a safety property expresses that something (bad) will not happen during a system execution; and a liveness property asserts that eventually something (good) must happen during the execution. These basic forms allow the expression of different system requirements that may be classified as follows [99]:

- Structural properties are related to the structure of the system such as the connection and consistency between the interfaces of components, invariants to be maintained in the system and fault-tree analysis.
- Qualitative (behavioral) properties deal with the behavior of the system such as the schedulability, liveness, causality and deadlock detection.

- Quantitative properties are used to evaluate performances of the system or to evaluate its behavior considering characteristics such as the probability of actions and response time.

In the context of this thesis, we choose the model-checking technique for the verification of some structural and behavioral properties in real-time systems (see chapter 5).

2.3 Modeling of real-time systems

Several modeling languages have been proposed to assist designers in the software engineering of real-time systems. This field is based on a first generation of languages (in the 1990s) like Wright [13], MetaH [34], Darwin [115] and Rapide [112], that are extended later for more mature languages [135]. Currently, the UML and AADL languages are highly mature languages, which have been extended with powerful modeling tools. Malavolta *et al.* [116] rank both the UML and AADL languages as the top used languages in industry for the software modeling: UML provides an extension mechanism (UML profile) that allows adding extensions to the semantics of the predefined UML concepts for different domains and platforms. A number of UML profiles have been standardized by the OMG organization, including the SYSML (SYStem Modeling Language) profile for system engineering and the MARTE (Modeling and Analysis of Real-Time Embedded systems) language; and AADL (Architecture Analysis and Design Language) is an architectural modeling language standardized by the SAE [4] organization for the modeling of real-time embedded systems.

Being interested by the real-time domain, we study two fundamental standards for the modeling of real-time systems, MARTE UML profile and AADL which are presented and discussed in the rest of this section.

2.3.1 MARTE

MARTE [7] is a UML2 profile proposed by the OMG organization since 2007, to support the model-driven development of real-time embedded systems. The profile is defined for two main goals: modeling of the features of real-time embedded systems; and annotating the UML models so as to support analysis of system properties, which provides a foundation to apply transformations from the UML models into a wide variety of analysis models (e.g. formal models).

[4] SAE: Society of Automotive Engineers

Formalism. MARTE supports multiple levels of abstraction using 14 sub-profiles based on a variety of packages, which are mainly grouped in four parts, as shown in Figure 2.4 [5]:

- Foundations package defines the basic concepts for real-time embedded systems modeling: core elements, non-functional properties modeling, time modeling, generic resource modeling and allocation modeling.
- Design package refines the core concepts to support specific design features like hardware and software platform details.
- Analysis package adds elements to support analysis, especially the schedulability and performance analysis.
- A package of annexes gathers all the MARTE annexes such as the VSL (Value Specification Language) annex that provides concrete syntax for specifying expressions in MARTE.

Based on their needs, designers can select a particular subset of the 14 MARTE sub-profiles to be used in different contexts like the usual specification, design and implementation stages, as well as for analyzing ends (performance and schedulability requirements).

For the architectural design, the profile provides a package GCM (Generic Component Model) to describe a system using the usual concepts of the component-based paradigm. Thus, it is possible to specify the system as a set of components with their interactions. A MARTE component is an autonomous entity of the system that can contain both data and behavior. A component may have properties and ports to explicitly specify its interaction with its external environment. The MARTE port definition has been heavily inspired from the existing architectural languages like AADL.

Tools. Many UML tools support the profile MARTE such as Papyrus which is a free graphical editing tool for UML2, Modelio which is an open source UML tool developed by Modeliosoft and MagicDraw which is a commercial product.

[5] CoreElements: Core Elements, NFP: Non-functional Properties Modeling, Time: Time Modeling, GRM: Generic Resource Modeling, Alloc: Allocation Modeling, GCM: Generic Component Model, HLAM: High-Level Application Modeling, DRM: Detailed Resource Modeling, GQAM: Generic Quantitative Analysis Modeling, SAM: Schedulability Analysis Modeling, PAM: Performance Analysis Modeling, VSL: Value Specification Language, RSM: Repetitive Structure Modeling.

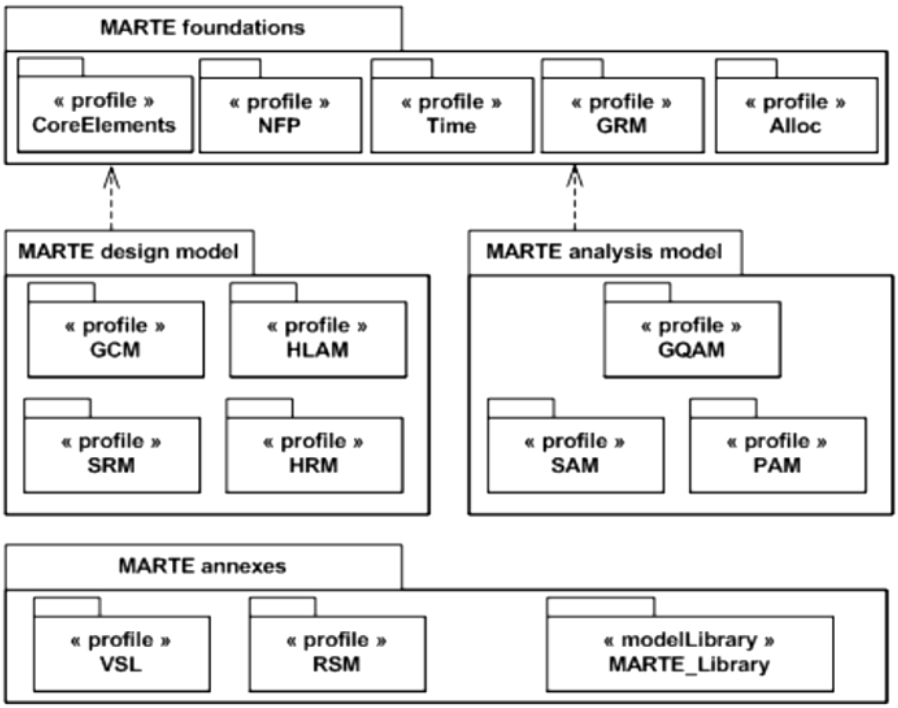


FIGURE 2.4: Architecture of the MARTE Profile [7]

Real-time notations. MARTE allows the description of the temporal concepts of real-time systems using mainly the NFP and Time sub-profiles. The HLAM sub-profile gathers concepts for both quantitative (period, time) and qualitative (behavior, communication, concurrency) real-time characteristics. The SAM sub-profile provides annotations dedicated for the schedulability analysis.

Extension mechanisms. As a UML profile, MARTE may be extended using the UML basic extension mechanisms: the meta-model (profile class) with its basic element the meta-class; and the stereotypes to extend a meta-class to enable the use of a platform or domain specific terminology. For illustration, MARTE extends the UML2 SimpleTime package with a set of stereotypes refining the time events, time observations and duration observations with references to clocks.

2.3.2 AADL

AADL [10] (Architecture Analysis and Design Language) is an architectural description language developed since 2004 for modeling of real-time embedded systems. AADL is based on the component-based paradigm with a rich syntax and semantics for the description of application architectures.

Formalism. The AADL core language allows the definition of the system architecture as a set of software and hardware components and their connections. The modeling of an AADL component consists in describing a type (it declares the component interface elements called features) and zero or more implementations (they present the component internal structure). The AADL components are grouped in three categories: software components (`subprogram`, `subprogram group`, `data`, `thread`, `thread group` and `process`); hardware components ((`virtual`) `processor`, `device`, (`virtual`) `bus` and `memory`); and system composition component (`system`). These components have both textual and graphical notations as shown in Figure 2.5.

AADL allows the modeling of interactions (connections) between components which are drawn between the features (port, parameter, subcomponent access), specified in their interfaces and declared in their implementations. There are three types of connections: port connections (type port), parameter connections (type parameter of subprograms), access connections for shared components (type data access, bus access, subprogram access, virtual bus access).

The AADL ports allow to exchange data and/or events between components. They can be declared as data, event or event data ports and *in* (input), *out* (output) or *in out* (input and output) ports.

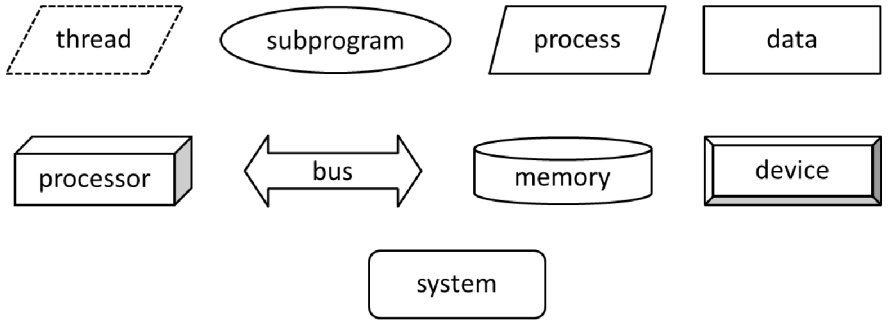


FIGURE 2.5: Main AADL graphical components

Tools. There exist many tools for AADL models processing at different levels (modeling, analysis, scheduling). We mention some free tools such as OSATE2 (plugin Eclipse), Cheddar, Ocarina, TOPCASED and commercialized tools such as Stood and AADL inspector.

Real-time notations. AADL provides a set of predefined properties to specify temporal parameters, concurrency, scheduling, etc. The properties are attributed within the components according to their types. For example, `Period`, `Compute_Execution_Time` and `Deadline` properties allow the specification of the temporal parameters for the `thread` component.

Extension mechanisms. The AADL language can be extended via the definition of new properties (property-set) or annexes. Big and specific additions are specified with separate annexes (separate sub-languages). Some annexes are standardized by the SAE like the Error-Model annex to specify fault behavior and propagation, the ARINC653 annex for avionics modeling and the Behavior annex [9] to add behavioral descriptions in the system architecture.

2.3.3 Discussion

Although MARTE and AADL are standard modeling languages, each one has its own capabilities:

- As a UML profile, MARTE benefits from the advantages offered by the UML ecosystem (meta-models, profiles, tools, etc.). However, as in UML, there is the possibility of the use of a large number of diagrams, which makes difficult to maintain a consistent and semantically-correct specification [59]. In addition, MARTE is a large standard, catering the typical needs of modeling and analysis of real-time embedded systems, yet it represents some practical problems that may restrict its utilization in the industrial context, as discussed in [88]: only a small part of MARTE is needed for a particular purpose; there is no defined methodology for MARTE that guides designers on how to use it; lack of mature modeling tools that are scalable for large industrial systems.

- Being an industry standard is a main advantage of the AADL language, which encourages its utilization in the industrial context (ranked in [116] among the top-used languages in the industry). AADL allows the modeling of a software architecture of a real-time system, comprising of different layers of the system (hardware aspects, network, systems, middleware and applications). Thanks to its analyzability and extensibility, AADL has been studied in several projects [6] for different purposes, namely, analysis (structure, real-time, security, safety and performance), code generation (Ada and C), extensions (property-sets and annexes) and formal verification (simulation, model-checking). However, AADL does not explicitly model some MARTE concepts such as the clock and timer [59].

Based on these observations, we opt for the use of the AADL language in this thesis. This choice is encouraged by the following reasons: AADL is specifically

[6] CESAR (http://www.cesarproject.eu/), ASSERT (http://www-verimag.imag.fr/ASSERT.html?lang=en), TOPCASED (https://www.polarsys.org/topcased), COMPASS (http://www.compass-toolset.org/).

defined for real-time safety-critical systems modeling; it provides a software architecture model (a unique and complete representation of the system), that can be considered as pivot model for model transformation approaches; its ability to specify the real-time software and hardware concepts, especially scheduling and temporal parameters; and its rich ecosystem that contains mature modeling tools such as Ocarina and OSATE. In addition, our work is also extended to support the AADL Behavior annex for the specification of behavioral descriptions. More about the AADL language is presented and discussed in chapter 4.

2.3.3.1 AADL modeling tools

As mentioned before, the AADL language is supported by several tools, either open-source (OSATE, Ocarina, TOPCASED) or commercial (STOOD, AADL Inspector) for modeling, analysis and code generation. In the following, we present some examples and we justify our choice to implement our solution.

OSATE

OSATE (Open Source AADL Tool Environment) [7] consists of a set of plug-ins to the Eclipse platform. It is an extensible open source environment for AADL modeling and analysis, that offers the possibility to automatically generate the graphical representation from the textual one or vice versa. OSATE compiles AADL models into an XMI-based format according to its meta-model specification (EMF (Eclipse Modeling Framework) meta-model for AADL). In addition, OSATE2 provides a set of additional analysis plug-ins, such as the connection consistency, resources allocation and fault-tree analyses. It supports also the compilation of some AADL annexes such as the Behavior and Error-Model annexes.

AADLInspector

AADLInspector [8] is a model processing framework for AADL, developed by Ellidiss-Software company. It comes with a set of analysis plug-ins that can be extended with additional analysis tools and bridges for remote verification tools. This tool supports analysis of AADL models and the Behavior, ARINC653 and Error-Model annexes. It also embeds a set of timing analysis, simulation and code generation tools through the integration of respectively the Cheddar, MARZIN and Ocarina tools.

[7] OSATE2 is the version supporting AADL version 2, http://osate.org/
[8] https://www.ellidiss.com/products/aadl-inspector/

Ocarina

Ocarina [9] [107, 87] is an open source tool suite developed since 2004 and deployed on GitHub in 2013 under the OpenAADL [10] project for AADL modeling. It provides the traditional analysis of AADL models (parsing, semantics checks), the support of ARINC653, Error-Model and REAL annexes with the possibility of the use of extra tools like Cheddar for schedulability analysis and Bound-T for WCET analysis. In addition, it performs automated code generation towards the AADL runtime PolyORB-HI/C and Ada. Ocarina is a stand-alone tool completely written in the Ada language. It uses an AST (Abstract Syntax Tree) as an internal representation of models (AADL, annexes, programming languages, etc.): the model is decomposed into a set nodes connected hierarchically to create the corresponding syntax tree according to the grammar of the language.

Discussion

In the context of this thesis, we choose the Ocarina tool suite to implement the proposed contributions for the following reasons: it is an open-source tool suite; it may be used as stand-alone compiler since it provides different engineering steps (modeling, analysis and code generation); it has a modular and extensible architecture; it can be easily integrated as an external plug-in for other AADL editors (already used through OSATE and AADL Inspector tools).

Compared to the OSATE platform, which is known by its graphical representation of AADL models and its comprehensive set of frontend analysis plug-ins, Ocarina provides both frontend lexical, syntactic and semantic analyses analyses and backend modules for model and code generation. In addition, the Ocarina compiler contains the required routines to rapidly implement a model transformation, that may be implemented without extra training (meta-models, transformation languages, dedicated platforms, etc.). More details about the conceived Ocarina extension are represented in chapter 5.

2.4 Formal specification of real-time systems

The choice of formalisms is a crucial phase in the use of formal methods. In the literature, there is an impressive number of specification languages that have been defined and refined for decades to provide simple, expressive formal models with their associated tools.

In this section, we present three popular formalisms, that we consider as basic models for several modern languages. Our goal is not to provide a detailed study

[9] https://github.com/OpenAADL/ocarina
[10] http://www.openaadl.org/

about the existing formalisms in the formal domain, but to highlight some basic concepts of the specification languages and how they are extended with the notion of time for real-time systems.

2.4.1 Automata

An automaton [144, 94, 42] (called also a finite-state machine/state machine or a finite-state automaton/finite automaton) is a *transition system*, representing an abstract machine with a finite number of states. It is defined by a finite number of states, an initial state and a set of conditions (guards) for each transition. At any time, the automaton can be exactly in one of its states and it can move from one state to another (a transition).

There are many extensions based on the automata theory such as the timed automata for real-time systems. The timed automaton [15] is an extension of classical automata, completed to support time concepts. The automaton transitions are guarded by clocks (clock values can be compared to integers) to constrain the behaviors of the automaton. Among the tools for timed automata modeling, there is the UPPAAL model-checker. UPPAAL [26] is a tool suite for model-checking of safety and bounded liveness properties of network of timed automata extended with structured data types, user defined functions and channel synchronizations. The verified property is expressed as a formula using a subset of TCTL (Timed Computation Tree Logic) logic. Then it is transformed into a timed automaton and composed with the system automaton.

The automata are known by their simplicity for quick design. However, they can be difficult to manage and maintain in the case of larger systems: the states and transitions can cause a fair degree of *spaghetti-factor* when trying to follow the line of the execution.

2.4.2 Petri nets

The Petri nets are *transition system*s originally proposed by Carl Adams Petri in 1962 [139] and they are considered as one of the oldest models for parallel and distributed computation. They are defined by a set of places (nodes), transitions and arcs. The transitions present events (system activities) that may occur and the places (can contain tokens) present conditions (system conditions and resources). The arcs specify which places are preconditions and/or postconditions for which transitions.

During the past decades, many work have extended Petri nets, such as CPN (Colored Petri Net), TPN (Timed Petri Net), PPN (Prioritized Petri Net) and SPN/GSPN ([Generalized] Stochastic Petri Net). The basic idea of the timed

Petri net extension [125] is the association of clocks and time intervals to transitions, in order to control the transition time: a transition is enabled if its clock value respects its time interval. Among the tools for Petri nets modeling there is the TINA model-checker. The TINA (TIme Petri Net Analyzer) [29] tool is an environment for the analysis of Petri Nets and timed Petri nets. It allows the model-checking of general properties like reachability properties, deadlock and liveness with the use of the State/Event-LTL logic, which is an LTL extension to express specific properties based on states and transitions of the specified system.

2.4.3 Process algebra

As mentioned before, systems may be described with high-level languages. In the concurrency theory research area, these specification languages are referred to as process algebras. Since 1970s, process algebras have become an underlying theory of parallel and distributed systems. A process algebra is mainly based on: the definition of parallel processes (behaviors); the representation of the interactions between processes by communications rather than sharing variables; and the definition of algebraic laws for these descriptions (often formulated in terms of a *transition system*) to be manipulated and analyzed by formal tools. A large number of process algebras are based on the SOS semantics to build and compose an LTS (Labeled Transition System) for formal analysis [22].

The basic process algebras are CCS (Calculus of Communicating Systems) [128], CSP (Communicating Sequential Processes) [84] and ACP (Algebra of Communicating Processes) [27], and then several languages have been emerged such as LOTOS (Language Of Temporal Ordering Specification) [1] with its last variant LNT (LOTOS New Technology) [48], pi-calculus [151] and mu-CRL (Representation Common Language) [75].

There are several timed extensions of these process algebras: ATP (Algebra of Timed Processes) [132] (based on CCS and ACP), time CSP [57], timed pi-calculus [150] and timed mu-CRL [74]. A timed process algebra is an extension of a classical process algebra that adds a set of specific constructs (types, functions, variables and references for the time) to represent the timing constraints and its associated notions (delays, timeout, etc.). Hence, the execution of the processes is constrained by the progress of the time (counting units of time).

Many tools for process algebras specification and analysis have been developed. For example, the CWB-NC (Concurrency Workbench of New Century) [52] tool, originally designed for the verification of systems modeled with CCS and then extended to support several formal languages (CSP, Basic LOTOS, etc.). This tool provides model-checking (properties are expressed in the modal mu-calculus and CTL) and equivalence checking of concurrent systems. The CADP (Construction and Analysis of Distributed Processes) [69] toolbox is another example

that uses the process algebraic formalism for the verification of concurrent systems. Early versions of CADP support only LOTOS to describe communication protocols. Then, several languages were added such as FSP, LNT and EXP. The toolbox provides a comprehensive set of tools for model-checking, equivalence checking, interactive simulation and performance evaluation. The last example is the mCRL2 toolset for the analysis of systems modeled with the mCRL2 specification language (the successor of the mu-CRL specification language and toolset). The mCRL2 specifications are linearized and then transformed into LTSs to allow the model-checking using properties expressed in the modal mu-calculus.

The process algebras became an underlying theory for the concurrency modeling and analysis. A good argument supporting their applicability is the impressive number of languages and associated tools developed during the past decades. Yet, this may create confusion for the industrial users, who are unsure about which language to adopt.

2.4.4 Discussion

We have introduced three popular formalisms (with some extensions) for the formal specification. Historically speaking, Petri net is considered as one of the oldest and interesting concurrency models. Research in concurrency theory started with the development of the Petri nets. Before that, researchers only considered sequential processes by means of *transition system*s or automata. Then, the notion of process algebra has appeared with the development of the CCS model as a general theory of concurrency. Since then, new process algebras have been emerged, while automata and Petri nets are extended with concepts like data, time and hierarchy.

Even though these formalisms share many common features, the choice of the formalism depends mainly of the context of work. In the context of this thesis, we aim to formally analyze architectural descriptions of real-time systems, which are based on architecture and real-time concepts such as concurrent tasks, temporal parameters, components, communications, etc. For this end, we opt for the process algebra formalism, since it promotes the specification of the software systems in terms of concurrent processes with their interactions. This choice is encouraged by several reasons, such as the results surveyed in [135], where the author exhibits that process algebras (e.g. pi-calculus, CSP and FSP) are the top preferred formal method by architectural languages, to formalize their formal semantics.

Compared to process algebras, automata and Petri nets are considered as low-level languages, while a process algebra is more related to grammars/languages [164]. Yet, dealing with low-level models may become difficult, especially when specifying large (complex) systems. In the case of Petri nets, the effort for the spec-

ification of large systems is very high (the complexity is exponential compared to linear in the case of process algebras) [83]. A process algebra can be seen as a progressive extension of classical automata. However, instead of describing directly a *transition system*, the system specification is simplified with the help of a high-level language which is quite user-friendly, expressive and design-oriented. In addition, process algebras are originally designed for concurrency modeling (constructs for processes and interactions), and currently they represent well-established concurrency models (CCS, CSP, etc.), that are enhanced by adding the data and time concepts in many extensions or by defining modern languages.

Among the modern specification languages, we choose the LNT (LOTOS New Technology) [48] formal language, which is a process algebra based-language (variant of two standards LOTOS [1] and E-LOTOS [3]) developed since 2005 for the CADP toolbox. Currently, the LNT language is intensively used for specifying and verifying concurrent systems using CADP (progressively replacing the LOTOS standard). Based on a combination of traits from process calculi, functional languages and imperative languages, LNT provides sufficiently expressive operators for data and behavior with a user-friendly notations to simplify writing and extension. The syntax and semantics of the LNT language are presented in chapter 3.

Since we deal with real-time systems, we note that timed extensions exist for the CADP languages (ET-LOTOS [108], RT-LOTOS [54], etc.), but currently, they are not supported by its tools. Nevertheless, the use of the LNT language stills sufficient for our purposes: it provides a rich data part that is used to specify the scheduling algorithms; the time is a part of the proposed LNT mapping (see chapter 3) and it is smartly included (when needed) to provide LNT specifications with reduced state spaces and avoid additional transformations (from timed to untimed models).

CADP [69] (Construction and Analysis of Distributed Processes) is a toolbox for the design and verification of concurrent systems developed since 1986, available with both academic and commercial licenses [11].

Initially, CADP featured a compiler and state space generator for the LOTOS language called *CÆSAR* with an equivalence checker called *ALDEBARAN*. Nowadays, the toolbox offers a comprehensive set of tools for the specification, interactive simulation, verification (model checking, equivalence checking, etc), performance evaluation, etc. To deal with large systems, CADP provides a set of techniques such as the reachability analysis, on-the-fly verification and distributed verification.

CADP explores an LNT specification using its formal operational semantics defined in terms of an LTS. A translation from LNT into LOTOS is firstly applied

[11] http://cadp.inria.fr/

with the *LNT.OPEN*, *LNT2LOTOS* and *LPP* tools. Then, a generation of an LTS is performed by the *CÆSAR* compiler [64]. The LTS represents the system state space that will be explored by different analysis tools. CADP contains three model checkers operating on LTSs: XTL [119] (eXecutable Temporal Language); and *EVALUATOR 3/4* [120, 121], which are two model-checkers based on the MCL (Model Checking Language) temporal logic, a regular alternation-free mu-calculus (an extension of the modal mu-calculus).

In addition to LNT, CADP provides a scripting language SVL (Script Verification Language) [67] for the description of the analysis scenarios. More about the CADP tools and how they are used in this thesis are presented in chapter 5.

2.5 AADL related approaches

The literature is full of formal approaches revolving around the AADL language. Existing AADL framework and tool-chains provide gateways to other tools for advanced analysis, validation or implementation, that are often based on the model transformation into different languages such as Petri nets [146], timed automate [15], LUSTRE [89], IF [12], TLA+ [148], Signal [30], ACSR [158], TASM [169], Fiacre [28], Real-time Maude [134] and BIP [49].

Since we deal with Ravenscar compliant models, we note firstly that some approaches have been proposed for the formal analysis of the Ravenscar systems. In general, these work aim to analyze Ada real-time systems, such as: Kristina *et al.* [113] propose a formal mapping of a Ravenscar compliant runtime kernel for the verification with the UPPAAL model-checker; authors in [80] work on the generation of the Ada Ravenscar code from the AADL models, in which a particular data connector DBX (Deterministic Bridge Exchangers) is manually mapped in LOTOS to verify its deterministic behavior. In addition, they provide a static and dynamic semantics for the generated Ada Ravenscar in [81]; authors in [133] present a transformation of the Ada Ravenscar programs using the IF timed automata. Compared to these approaches, we use the Ravenscar profile to apply a set of strong constraints at the model level for safety-critical systems modeling. We provide an LNT pattern for a Ravenscar compliant task model that can be completed and automated for the Ada code analysis with the CADP toolbox, which presents an important perspective. But, we currently focus on the verification of the architectural models rather than the code analysis.

In the following, we survey about 20 AADL model transformations, classified according to their source and target languages.

2.5.1 Classification according to the source model

According to the source AADL model (with or without annexes), a first classification may be applied for existing AADL model transformations. As included in Table 2.1, we study work dealing with only AADL models, models enriched with the Behavior annex (BA) and models extended by the Error-Model annex (EMA).

The first family includes the transformations of AADL models without annexes. These approaches consider only the AADL semantics described in its standard which is enough to formally simulate the system and verify a set of behavioral properties such as deadlock and liveness. The second family specializes in the error behavior and analysis based on the Error-Model annex specifications. These transformations allow the analysis of the performance and the dependability such as in [149] and [39]. The third family represents analysis approaches for AADL models completed with the Behavior annex specifications. They allow the formal simulation and model-checking of behavioral properties.

TABLE 2.1: Classification according to the source model

	Target language	*Papers*
Only AADL	Real-Time Calculus (RTC)	[157]
	LUSTRE	[89]
	ACSR	[159]
	Timed automata	[90, 91]
	Machine-Readable CSP	[168]
AADL & EMA	GSPN	[149]
	SMV model	[40, 38, 41]
	HiP-HOPS	[126]
AADL & BA	BIP	[49]
	Timed automata	[77, 79]
	Real-Time Maude	[134, 19, 21, 20]
	Fiacre	[28, 36]
	Stateful Timed CSP	[173]
	TASM	[169, 86, 170]
	Signal	[71, 31]
	LNT (our approach)	[129, 130]

2.5.2 Classification according to the target formalism

Table 2.2 sums up a set of AADL transformation approaches, classified in three families according to their target formalisms: automata, Petri nets and other specification languages (e.g. process algebras). We include 18 AADL transformations with their target languages, the tools used for the transformation/formal verification and some objectives.

TABLE 2.2: Classification according to the target formalism

	Target language	*Tools*		*Objectives*	*Papers*
		Transformation	*Verification*		
Automata	Linear hybrid automata	-	TIMES	schedulability analysis, simulation	[76]
	Event-Data Automata	COMPASS	COMPASS	behavioral properties, safety analysis, FDIR and performance evaluation	[40, 38, 41]
	Timed automata	-	UPPAAL	control/data-flow reachability	[90, 91]
	Timed automata	ATL	UPPAAL	behavioral properties	[78]
	Timed automata	-	UPPAAL	AADL mode analysis	[174]
Petri nets	GSPN	ADAPT	SURF-2	dependability analysis	[149]
	SAN	ADAPT-M	MoBIUS	dependability analysis	[82]
	TPN	Ocarina	TINA	behavioral properties	[146, 147]
Formal languages	LUSTRE	aadl2sync	Lurette,Lesar	simulation, behavioral properties	[89]
	BIP	OSATE	BIP framework	behavioral properties	[49]
	TLA+	Topcased	model-checker TLC	temporal analysis	[148]
	Signal	OSATE	Syndex,Polychrony	temporal and schedulability analysis	[172, 30, 171]
	Fiacre	Topcased	TINA	simulation, behavioral properties	[28, 36]
	ACSR	OSATE	VERSA	temporal and schedulability analysis	[158, 159]
	Real-time Maude	OSATE	Maude platform	behavioral properties	[134, 19, 21, 20]
	IF	Topcased	-	safety properties	[12]
	TASM	OSATE,ATL	TASM,UPPAAL	behavioral properties	[169, 86]
	Stateful Timed CSP	OSATE	PAT	behavioral properties	[168]
	LNT (our approach)	**Ocarina**	**CADP**	**behavioral and structural properties**	[129, 130]

The first formalism is the automata. Particularly, we note the use of the timed automata and the UPPAAL model checker. For instance, Hamdane *et al.* [78] describe a tool-chain from AADL into timed automata for the model-checking of deadlock, liveness and reachability properties. Bozzano *et al.* [40, 38, 41] propose a comprehensive platform COMPASS for the analysis of AADL models such as the requirements validation, functional verification, safety analysis, FDIR (Fault Detection, Isolation and Recovery) and performance evaluation. This approach is based on the defined SLIM language, which is an extension of the AADL language and its Error-Model annex. The SLIM model is transformed into an EDA (Event-Data Automata) to be explored with different COMPASS tools.

The second family concerns the Petri nets and their extensions. We mention Renault *et al.* work, in which an AADL subset is firstly transformed into symmetric nets in [147] and then extended into timed and colored Petri nets in [146] for the verification of behavioral properties such as missed deadline or missed `thread` activation, using the TINA formal analysis tool.

The third family transforms different AADL subsets into diverse specification languages. The proposed solution in this thesis is included among this family using the LNT language. In the following, we detail some work form this family that support the Behavior annex and aim mainly to verify the behavioral properties.

Synchronous approaches

Firstly, we note that a set of approaches addresses synchronism by using synchronous languages as target formalisms such as in [89], where authors explain how the synchronous paradigm can be used to describe asynchronous behaviors through the transformation of an AADL subset into the synchronous language LUSTRE.

Other approaches deal with an AADL synchronous subset, for example, the contributions around the Polychrony framework [171, 172, 30, 114] introduce the concept of co-design using an AADL subset (periodic `threads` and data port connections) for modeling and the Simulink language for the behavioral specifications. The verification is based on the transformation into the Signal synchronous language, which allows the exploration of the AADL model with the Polychrony and SynDEx tools. In addition, the Behavior annex is formalized as polychronous automata in [31] and then its mapping is completed towards the Signal language in [71].

Yang *et al.* [169, 86] use the same synchronous subset adding AADL modes and offline non-preemptive scheduling policy to define a formal semantics with the TASM (Timed Abstract State Machine) language. The authors propose mainly a semantics of the AADL synchronous execution model (`thread` execution and communication) with a mapping of the `thread` Behavior annex. The transformation is implemented in the OSATE environment and formally verified by

the Coq theorem prover, in order to verify behavioral properties (deadlock and reachability) with the TASM and UPPAAL tools. The proof is performed by equivalence-checking and based on the equivalence checking of the TTS (Timed Transition System) of both the AADL and TASM models.

In the context of this thesis, we rather handle asynchronous model supported by the AADL language to deal with larger AADL subsets for more realistic applications.

Fiacre

Fiacre (*Format Intermédiaire pour les Architectures de Composants Répartis Embarqués*) is a formal specification language to represent both the behavioral and temporal aspects of real-time systems. The transformation of AADL models into the Fiacre language is addressed by Berthomieu *et al.* in the TOPCASED environment [28]. The verification needs a first transformation into the Fiacre language, and then the Fiacre model is compiled into an abstract timed transition system supported by the TINA tool. This work considers the AADL model as a set of communicating threads and supports periodic/sporadic `thread` and event/data port connections but it is restricted to the non-preemptive scheduling. A second version of this work is presented in [36] dealing with an AADL synchronous subset.

Real time Maude

Another work [134] uses a formal real-time rewriting logic semantics, called Real time Maude, to transform an AADL subset with its Behavior annex to an executable semantics with the Real-Time Maude platform (an AADL simulator and LTL model checker). In addition, authors in [19, 21, 20] are motivated by the PALS (Patterns of Adaptive Learning Scales) pattern that reduces the design and verification of an asynchronous system with its synchronous version. They define a Synchronous AADL sub-language and provide its formal semantics in Real-Time Maude.

BIP

BIP (Behavior Interaction Priority) is a language for the description and composition of components as well as associated tools for analyzing models and generating code. Chkouri *et al.* [49] define a translation from a significant AADL subset with its Behavior annex into the BIP language, and then the BIP model is transformed into a non-timed model to enable the model-checking (*Aldebaran* and *observers* tools) and the simulation with the BIP framework. This work supports periodic/sporadic `threads` and event/data port connections but it uses a simple scheduler without preemption.

2.5.3 Discussion

In general, all the related work aims practically at defining a formal executable semantics for an AADL subset to allow the model-checking of behavioral properties. However, subsets, methodologies and tools are diverse. Compared to the existing approaches, we are distinguished by the following points:

- The AADL2LNT transformation (chapter 4) considers both software and hardware AADL components with the consideration of a significant set of temporal and queuing standard properties. We focus on the AADL `thread` scheduling execution and port connection mechanism with the definition of an explicit scheduler. We support the event-driven preemptive priority-based scheduling and asynchronous communications. The considered subset covers the fundamental real-time features that can be used in more realistic applications rather than synchronous and non-preemptive approaches.

- Many existing work require more than one model transformation to be connected to the analysis tools. We use LNT as a target model which is a direct input language (without additional transformations) for the CADP toolbox. This gateway allows the exploitation of the CADP tools that implement a variety of formal methods (model and equivalence checking, simulation, etc) and provides mature solutions for the state space explosion problem (smart state space reductions, on-the-fly verification, etc.).

- For the soundness of the AADL transformation, [169] and [36] propose a semantics preservation proof based on the formalization of an TTS semantics for a restricted AADL subset. Then, an equivalence relation is checked with the corresponding TTS of the target language using the Coq theorem prover. However, the AADL-TTS semantics definition is supposed to be correct without preservation proving. This semantics gap concerns all the AADL formal transformations. Due to the informal semantics of the AADL language, we can not directly prove the equivalence between the AADL semantics and the formal semantics of the target formalism. In this thesis, we propose an LNT pattern for a task model compliant with the Ravenscar profile (chapter 3), which is considered as a pivot representation. The AADL semantics are abstracted as a standard task model which reduces semantic ambiguity in the transformation.

- Several work use OSATE to implement the AADL transformation. In this thesis, we opt for Ocarina which is an open-source tool suite that can be used as stand-alone compiler or it may be integrated as a backend for other AADL editors such as OSATE, which increases the visibility of our work.

- The Ocarina-CADP tool-chain is totally automated and transparent for the model transformation and verification. The verification phase is ensured by

the SVL script allowing the exploration of the system state space and the model-checking of a set of generic structural and behavioral properties.

2.6 Conclusion

In this chapter, we introduced the general concepts required in our work. A set of modeling and specification languages are studied and discussed to justify the choice of the AADL and LNT languages as a source and target models for the proposed model transformation in this thesis. Finally, we surveyed some AADL model transformations classified according to their source models (with or without annexes) and their target formalisms (Petri nets, automata and process algebras).

The next chapters are devoted to detail the contributions of this thesis.

3

Formal pattern

Contents

3.1 Introduction 48
3.2 LNT language 48
3.2.1 Syntax 48
3.2.2 Some definitions 49
3.2.3 LNT specification 53
3.3 A Ravenscar compliant task model 54
3.4 Formal mapping 56
3.4.1 Task definition 56
3.4.2 Scheduler definition 59
3.4.3 Communication mapping 64
3.4.4 Composition and synchronization 65
3.4.5 Discussion 67
3.5 Conclusion 68

3.1 Introduction

In this chapter, we present our first contribution in the context of the formal specification of real-time systems. We propose a formal pattern for a real-time task model compliant with the Ravenscar profile. This mapping is based on the definition of a formal semantics using the LNT language, containing specific definitions for different real-time concepts, mainly, the task with its temporal parameters, the connections between tasks and the scheduler.

This chapter is organized as follows: firstly, we present the LNT language in section 3.2; then, we define the supported task model in section 3.3; and in section 3.4, we describe and justify the formal pattern through four principal parts (task mapping, scheduler mapping, communication mapping and composition/synchronization) and we conclude with a discussion.

3.2 LNT language

The LNT (LOTOS New Technology) [48] formal language is proposed by the VASY and CONVECS teams from INRIA laboratory in France, for specifying safety-critical concurrent systems. This language combines features from process algebras and programming languages with a dynamic semantics based on the SOS (Structural Operational Semantics) rules. LNT has been developed since 2005, inspired from the proposed improvements for the LOTOS [1] and E-LOTOS [3] (for Extended-LOTOS) standards. These languages were progressively enhanced to remove the limitations about the expressiveness, structuring capabilities and user-friendliness. An historical overview of the evolution of LOTOS and its descendants E-LOTOS and LNT can be found in [70].

3.2.1 Syntax

To provide a practical language, LNT has been defined with an Ada-like syntax, based on a selected set of features borrowed from imperative languages, functional languages and value-passing process calculi.

Syntactically speaking, LNT distinguishes data and control parts. A data part is a fully imperative language in the syntax and semantics. It allows the definition of types and functions used in the LNT program.

A control part defines behaviors. It allows specifying behaviors using the processes definition. This part includes almost all data constructs and adds constructs for the behavior like the non-deterministic choice, process parallelism and communication.

3.2.2 Some definitions

In the section, we introduce some LNT constructs required for the proposed pattern [1]. We consider these identifiers used later in different LNT constructs given in BNF [2]: M: module; Π: process; B: behavior; I: statement; T: type; V: expression; X variable; F: function; G: gate and Γ: channel.

3.2.2.1 Module

The LNT specification consists typically of a set of LNT modules. Each module M (Listing 3.1) is named with the same name as its source file (`*.lnt`). M can import other modules (M_0, ..., M_n), thus, all definitions of M_0, ..., M_n are visible and can be used in M definitions. The module definitions include LNT types, channels, functions and processes. In addition, a set of predefined functions can be specified using the **with** clause, required for different data types. A set of pragmas can also be added to the module definition. In the LNT language, pragmas are used to modify the default setting related to the implementation of the predefined types. For example, by default the `Nat` type is defined with 8 bits. With the `!nat_bits N` pragma, we can specify a new number of bits `N` on which a value of the type `Nat` will be attributed.

Listing 3.1: LNT module definition

```
module M [(M_0, ..., M_n)]
[with predefined_function_0 , ..., predefined_function_n] is
  module_pragma_0 ... module_pragma_p
    definition_0 ... definition_q
end module
```

3.2.2.2 Types and channels

The LNT language provides a set of predefined basic types (Boolean, Natural, Integer, Real, Character and String) with the associated predefined functions (basic operations such as the addition and comparison) that are automatically available. LNT allows likewise the definition of advanced types, as shown in Listing 3.2.

A non-basic type may be a list, a sorted list, a set, an enumerated or a range type (specified within the `type_expression`). In this case, the associated predefined functions (such as `"empty"`, `"length"`, `"member"`, `"access"`, `"delete"` `"remove"`, `"head"`, `"tail"` and `"union"`) are generated according to the specified **with** clause.

[1] A detailed definition of syntax and formal semantics of the LNT language can be found in its reference manual [48]

[2] Descriptions between `"[]"` are optional and the vertical bar `"|"` is used for disjunction.

Listing 3.2: LNT type definition

```
type T is
 type_pragma_0 ... type_pragma_n
   type_expression
 [with predefined_function_0 , ... , predefined_function_m ]
end type
```

A channel (Listing 3.3) is a gate type that defines the types of values to be sent or received during the communication on a given gate. The channel may contain user-declared or predefined basic types.

Listing 3.3: LNT channel definition

```
channel Γ is
  (T_1,1 , ... , T_1,n ) ,
  ...
  (T_m,1 , ... , T_m,l ) ,
end channel
```

3.2.2.3 Function

The LNT function F (Listing 3.4) consists of a function declaration for the gate parameters, formal parameters and a return type T with a function body I_0 for the statements. The body I_0 computes the result value of F and the output parameters. Listing 3.5 includes a set of statements proposed by the LNT language.

Listing 3.4: LNT function definition

```
function F [[gate_declaration_0 , ... , gate_declaration_m ]]
    [( formal_parameter_0 , ... , formal_parameter_n )] [: T] is
 function_pragma_0 ... function_pragma_l
   I_0
end function
```

Listing 3.5: LNT data statements

```
I ::= null                    -- no effect
| I_1 ; I_2 -- sequential composition
| return [V]                  -- return statement
| X := V                      -- assignment
| X [V_0]:= V_1 -- array element assignment
| [X:=] F [[actual_gates]] ( actual_parameter_1 , ... , actual_parameter_n )
| var var_declaration_0 ,... , var_declaration_n in I_0 end var
| case V_0 ,... , V_l in [var var_declaration_0 , ... , var_declaration_n in]
  match_clause_0 -> I_0 | ... | match_clause_m -> I_m end case -- case statement
| if V_0 then I_0 [elsif V_1 then I_1 ... elsif V_n then I_n]
  [else I_n+1] end if -- conditional statement
| loop I_0 end loop -- forever loop
| loop L in I_0 end loop -- breakable loop
| while V loop I_0 end loop -- while loop
| for I_0 while V by I_1 loop I_2 end loop -- for loop
```

3.2.2.4 Process

The LNT process (Listing 3.6) is an object describing a behavior. The process consists of a process declaration for the gate declarations and formal parameters with a process body B for the behaviors. This behavior B includes almost all data statements (except **return**) and adds constructs for the behavior grouped in Listing 3.7.

Listing 3.6: LNT process definition

```
process Π [[gate_declaration_0 , ... , gate_declaration_m ]]
    [( formal_parameter_0 , ... , formal_parameter_n )] is
 process_pragma_0 ... process_pragma_l
    B
end process
```

Listing 3.7: LNT behavior statements

```
1  B ::= stop -- inaction (without continuation)
2  | B_1 ; B_2 -- sequential composition
3  | [only] if V_0 then B_0 [elsif V_1 then B_1 ... elsif V_n then B_n]
4    [else B_{n+1}] end if -- conditional behaviour
5  | Π [[actual_gates ]][( actual_parameter_1 , ... , actual_parameter_n )]
6  | G [ (O_0 ,... ,O_n) ][where V] -- communication
7  | select B_0 [] ... [] B_n end select -- nondeterministic choice
8  | par [G_0 , ... , G_k in]
9          [G_(0,0) , ... , G_(0,n_0) -> ] B_0
10         ||
11         [G_(i,0) , ... , G_(i,n_i) -> ] B_i
12         ||
13         [G_(m,0) , ... ,G_(m,n_m) -> ] B_m
14   end par -- parallel composition
15 | hide gate_declaration_0 , ... , gate_declaration_n in B end hide -- hiding
```

3.2.2.5 Gates and parameters

An LNT gate is used for input/output communication or synchronization (Listing 3.7: line 7). This behavior is only allowed within the process body, since functions perform only local calculations from the data part. The LNT language provides predefined gates such as internal (or invisible) gate `"i"`, which can not be actually used for communication or synchronization.

The set of the formal gate parameters may be used at the process and function declarations. A formal gate parameter G_i is typed by a channel Γ as shown in Listing 3.8. In a process call (Listing 3.7: line 6), a compatibility relation between gates is defined, to determine when a formal gate parameter G_i can be instantiated by an actual gate G_j (G_i and G_j are compatible if they are both declared with the same channel Γ).

A formal parameter consists of the variable parameters (X_1,..., X_n) associated to the same mode parameter (**in**, **in var**, **out** , **out var** or **in out**) and the same type T as shown in Listing 3.8. The different modes are used as follows:

- A value parameter declared with the **in** mode (default mode) denotes a constant parameter, so it can not be changed with the process or function body.
- A value parameter declared with the **in var** mode denotes a constant parameter, so it can be changed only as a local value within the process or function body, but this change is invisible to the caller.
- A value parameter declared with the **out** mode is a result parameter, so it is not read within the process or function body, but it should be assigned and its value is visible after the function or process call.
- A value parameter declared with the **out var** mode is a result parameter, as the **out** mode, it should be assigned and thereafter its value can be read within the process or function body.
- A value parameter declared with the **in out** is a modifiable parameter, it has an initial value that may be modified within the process or function body, the assigned value is visible after the process of function call.

Listing 3.8: LNT formal gates and parameters

```
1 gate_declaration_i ::= G_0 , ... , G_n : Γ
2 formal_parameter_i ::= [ in ] | in var | out | out var | in out X_0 , ... , X_n :T
```

3.2.2.6 Statement

In the following, we detail some statements that are frequently used in this thesis.

Variable declaration

The LNT **var** statement (Listing 3.5: line 8) allows the definition of local variables by their names and their types. The scope of each variable is the statement I_0.

Communication

The LNT processes can communicate (Listing 3.7: line 7) through their declared gates. A gate allows exchanging data by a *rendezvous* that blocks the sender and the receiver in the communication until it takes place. A *rendezvous* on a same gate may allow multiple sending and receiving of data (Listing 3.9) at the same time.

Listing 3.9: LNT communication definition

```
1 O_i ::=
2   [X =>][!]V   -- output offer
3 |[X =>] ? P    -- input offer
```

Non-deterministic choice

The LNT **select** statement (Listing 3.7: line 8) allows a non-deterministic choice between behaviors $B_0, ..., B_n$.

Parallel composition

The LNT **par** statement (Listing 3.7: lines 9..15) is used for behaviors $(B_0, ..., B_m)$ parallelism and gates $(G_0, ..., G_k$ and $G_{(i,0)}, ..., G_{(i,n_i)})$ synchronization. The behavior B_i represents often a process instantiation ($\Pi\,[G_0, ..., G_k, G_{(i,0)}, ..., G_{(i,n_i)}](..)$). The **par** composition allows two types of synchronization: global and interface. The global synchronization is defined with $G_0, ..., G_k$, this communication can happen only if all behaviors $(B_0, ..., B_m)$ can make it simultaneously. The interface synchronization is defined with $G_{(i,0)}, ..., G_{(i,n_i)}$ gates. In this case, if a B_k is waiting for a communication on gate which belongs to its synchronization interface list (e.g. $G_{(i,j)}$), this communication can happen only if all processes synchronized on the same gate (contain $G_{(i,j)}$ in their synchronization interface lists) can make it simultaneously.

3.2.3 LNT specification

In the LNT language, the system is represented by a set of concurrent processes communicating through gates typed with channels. As shown in Figure 3.1, a `root process` named `MAIN` should be added to define an entry point of the whole specification.

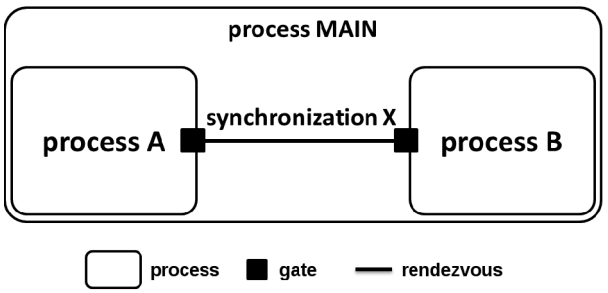


Figure 3.1: Example of LNT specification: graphical representation

The specification represents an executable semantics in which all parallel processes start execution and terminate at the same time with the possibility of synchronization by *rendezvous*.

3.3 A Ravenscar compliant task model

In this thesis, we rely on a conventional tasking representation inspired from the Liu and Layland task model [110]. We define a set of tasks S, consisting of n tasks denoted by $S = \{\tau_1, ..., \tau_n\}$, with $\tau_i = (T_i, C_i, D_i, P_i, O_i)$, based on the tasking descriptions from section 2.2.4.1 (chapter 2).

A set of hypotheses are assumed as follows: the supported architecture is based on the uniprocessor scheduling; S is a synchronous set of tasks, whose tasks have implicit-deadlines, fixed-priorities with the support of preemptions; i represents the priority P_i and $\forall i : D_i = T_i, C_i \leq T_i, O_i = constant = 0$. Thus, the task may be simply represented as $\tau_i = (T_i, C_i)$.

In addition, strong constraints are also considered since we deal with safety-critical systems. We rely on the Ravenscar profile (see chapter 2: section 2.2.3.2), that can be applied in real-time tasking description as a subset composed of a static set of tasks in interaction, running by one preemptive fixed-priority scheduler. Precisely, to be Ravenscar compliant, the task model should mainly respect the following restrictions:

- All tasks must be:
  - either periodic tasks;
  - or sporadic tasks: they have no fixed first activation, they are activated in response to asynchronous events (invocation-events) with a fixed minimal delay between two successive releases (T_i).
- All tasks are created at initialization and then activated and executed according to their priorities;
- All communications and synchronizations between tasks are achieved using the protected objects [3] with these constraints:
  - at most one task can wait on each object;
  - sending and receiving operations occur atomically through the protected object procedures.

[3] A protected object is a construction based on the well-known concept of monitors for synchronizations

- Scheduling is based on FIFO_Within_Priorities policy as follows:
  - each task has a static priority;
  - a task may preempt a task of a lower priority.

The life-cycle of a task can be represented as the state automaton drawn in Figure 3.2. At any time, the task can be in one of these states (READY, RUNNING and BLOCKED) as follows:

- READY state: the task is able to be executed by the processor. The READY task can be resumed (selected by the scheduler), thus it moves to the RUNNING state.
- RUNNING state: the task is actually executing. While running, the task can be preempted, thus it returns to the READY state or it can complete its execution and becomes in the BLOCKED state.
- BLOCKED state: the task can not execute until an external event occurs. A BLOCKED task may become READY by a temporal event (dispatch for a new period) or an invocation-event (for sporadic task).

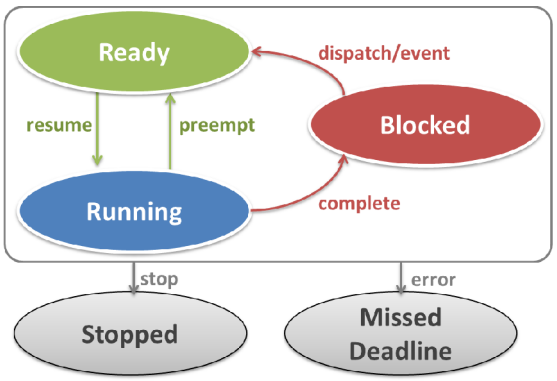


FIGURE 3.2: Task state automaton

In this thesis, we consider the RM (Rate Monotonic) scheduling with a negligible context-switch time. In this case, the indexes $\tau_{i..n}$, representing task priorities, are assigned according to their period values $T_{1..n}$ as the task with the shortest period takes the highest priority. During the scheduling, each $\tau_{i,j}$, representing the j^{th} job of task τ_i, is defined by $\tau_{i,j}$=$(r_{i,j}, s_{i,j}, e_{i,j}, d_{i,j})$ as described in section 2.2.4.1 (chapter 2). The execution of the set of tasks S is simulated while an hyperperiod interval. The hyperperiod $(H(\tau_{1..n}))$ interval represents the minimum interval of time until the schedule repeats in a cycle of task execution [46]. $H(\tau_{1..n})$ is calculated as the least common multiple of all task periods ($H(\tau_{1..n})$ =LCM (T_1,

.., T_n)) [50], which is considered a sufficient interval of time to study periodic tasks synchronously activated at $O_i = 0$ under a priority-based uniprocessor scheduling. For sporadic tasks, the interval $H(\tau_{1..n})$ is multiplied, as possible, for exhaustive analysis.

3.4 Formal mapping

In this section, we detail the formal pattern of the defined real-time task model. As explained in section 3.2, the LNT syntax combines features of programming languages with concurrency primitives adopted from process algebras. This makes it suitable to specify concurrent tasks and handle scheduling calculations. The formal representation should simulate the scheduling, the execution and the interaction of tasks. Ideally, the task and the scheduler are separately specified to provide an extensible and modular mapping.

Note that the LNT language is not a specific real-time process algebra: there are no time operators; all parallel processes start execution and terminate at the same time with the possibility of *rendezvous* synchronizations. Therefore, we define a `COUNTER` variable to represent time (a timer to count units of time) used as required to perform temporal calculations (e.g. release time, start and completion time, preemptions). In addition, we define a `HYPERPERIOD` variable that represents $H(\tau_{1..n})$, thus, the timer `COUNTER` is bounded as ($0 <$ `COUNTER` $<$ `HYPERPERIOD`).

Generally speaking, tasks are mapped into LNT processes to be concurrently executed, each τ_i is represented by one LNT process, named `TASK`. These `TASK`s can communicate with each other, through a particular process, named `CONNECTOR`. They are scheduled by a main LNT process, named `SCHEDULER`, that represents the scheduler and implements the scheduling algorithm: `TASK`s are synchronized (through the LNT gates and channels) with the `SCHEDULER` to be activated. In the following, we develop this pattern through four main parts: `TASK` definition, `SCHEDULER` definition, inter-task communication mapping and composition/synchronization.

3.4.1 Task definition

The `TASK` process is designed to represent the task as a schedulable concurrent unit with a potentially infinite sequence of jobs by the scheduler. In Listing 3.10, we include the `TASK` LNT skeleton. The process declares an LNT gate, named `ACTIVATION`, to be synchronized with the `SCHEDULER`. The `TASK` behavior is an infinite `loop` whose body is a non-deterministic choice `select` in order to separate *execution*, *error* and *termination* behaviors. The selected behavior

is determined by the ACTIVATION communication with its different possible values.

The TASK-SCHEDULER dialogue is defined through a set of activation orders according to the considered scheduling policies. In the case of the RM preemptive scheduling, we define a set of activation orders distinguishing different alternatives of preemptive scheduling, mapped with an LNT enumerated type (T_Dispatch_Preemption, T_Preemption_Completion, T_Preemption, T_Dispatch_Completion, T_Completion, T_Error and T_Stop).

The task life-cycle, as described with the task state automaton in Figure 3.2, is mapped in the TASK *execution* behavior part. The ACTIVATION communication defines the TASK states: the current state is defined according to the received SCHEDULER order. Initially, TASK is supposed to be in the READY state. It is suspended until the reception of the SCHEDULER order on ACTIVATION gate. All transitions (resume, preempt, dispatch and complete) between the states of the automaton of Figure 3.2 are translated by suspensions on the ACTIVATION *rendezvous* with the SCHEDULER. At the reception of an activation order (T_Dispatch_Completion, T_Dispatch_Preemption, T_Preemption, and T_Preemption_Completion), the TASK moves to the RUNNING state.

After the execution, TASK sends the label T_Completion to the SCHEDULER meaning that TASK has accomplished the activation order and it is no more in the RUNNING state. At this level, depending on the received order, TASK may switch state as follows:

- T_Dispatch_Completion: TASK starts and completes the execution of the current period and enters to the BLOCKED state;
- T_Dispatch_Preemption: TASK starts the execution in the current period but with a preemption, thus, it returns to the READY state;
- T_Preemption: TASK progresses in its execution but without reaching the completion time, so it returns to the READY state;
- T_Preemption_Completion: TASK finishes the execution of the current period and enters to the BLOCKED state.

TASK can also receive a T_Error and T_Stop orders which are used respectively to mark a missed deadline and to stop the system simulation. This concerns the *error* and *termination* behaviors, which allows the definition of two additional task states MISSED_DEADLINE and STOPPED included in the task state automaton as shown in Figure 3.2, used for verification ends.

Note that periodic and sporadic tasks are represented with the same LNT skeleton while the difference between them will be in the releasing mode controlled by the

SCHEDULER. All temporal calculations are encapsulated within the SCHEDULER which maintains that periodic tasks are released with regular-orders, while sporadic tasks receive irregular-orders according to the reception of invocation-events.

Listing 3.10: TASK LNT skeleton

```
1  process TASK [ACTIVATION: LNT_Channel_Dispatch,
2     G : LNT_Channel_Data,
3     -- other gate declarations
4  ] is
5     loop
6        select
7           -- execution behavior
8           select
9              -- a complete execution time
10             ACTIVATION (T_Dispatch_Completion);
11             G (DATA)
12             []
13             -- preemption
14             ACTIVATION (T_Dispatch_Preemption)
15             []
16             ACTIVATION (T_Preemption)
17             []
18             ACTIVATION (T_Preemption_Completion)
19          end select;
20          ACTIVATION (T_Completion)
21          []
22          -- error behavior
23          ACTIVATION (T_Error)
24          []
25          -- termination behavior
26          ACTIVATION (T_Stop)
27       end select
28    end loop
29 end process
```

In this thesis, we support inter-task communications, as described in the Ravenscar profile. In this case, the TASK may declare gates as required for its connections (Listing 3.10: line 2). The exchanged data and events are generically mapped using an LNT enumerated type (label DATA). TASK interactions are also controlled through the SCHEDULER orders that fix the input and output times as follows:

- T_Dispatch_Preemption represents the start time of the execution, TASK may receive the inputs;
- T_Preemption_Completion represents the completion time, TASK may send the outputs;
- T_Dispatch_Completion represents a complete execution time, so the

TASK can receive the inputs at the start time and send the outputs at the completion time.

3.4.2 Scheduler definition

The SCHEDULER process implements the scheduling algorithm to simulate the execution of the tasks. It is synchronized with all TASKs through their ACTIVATION gates. The SCHEDULER construction depends on the set of tasks S and the considered scheduling algorithm. In Listing 3.11, we include the SCHEDULER LNT skeleton. The process declaration has n gates (ACTIVATION_1, .. , ACTIVATION_N) with k additional gates (NOTIFICATION_1, .. , NOTIFICATION_K) when S contains sporadic tasks (k is the number of sporadic tasks with $k < n$). The SCHEDULER behavior consists of three parts as follows:

- *Initialization part*: SCHEDULER begins with a set of initializations needed for the temporal calculations, mainly, the COUNTER and the set of tasks S.
- *Operational part*: this part implements the scheduling algorithm. While COUNTER has not reached the HYPERPERIOD, SCHEDULER simulates the execution of tasks using Algorithm 1 [4] (illustrated in Figures 3.4 and 3.3).
- *Stopping part*: the termination of tasks is not allowed in the Ravenscar profile, but in the context of formal verification, we define a global system termination when COUNTER = HYPERPERIOD. Therefore, SCHEDULER sends the T_Stop order for all tasks to mark the end of the simulation.

Following the Ravenscar profile, the task execution is ensured by the priority-based preemptive scheduling: the task priorities are statically assigned; the scheduler runs always the READY task with the highest priority; at any time, if a task with a higher priority becomes READY, the scheduler performs a context-switch preempting the current RUNNING task to enable the higher priority task to resume execution.

The set of tasks S is statically included using an LNT array type. Tasks are indexed by their fixed priorities according to their periods ($T_{1..n}$): τ_1 (index 1) has the highest priority and task τ_n (index n) has the lowest one. This array represents the ready-queues in real systems: ready tasks are inserted/deleted at the head/tail according to their priorities in the ready-queue and at any time the scheduler selects the task with the highest priority for the execution. In our mapping, we use a static structure where tasks have fixed indexes to mark their priorities, while their states (job parameters) are modifiable by the SCHEDULER itself.

[4] The time complexity of this algorithm is about $O(n^2)$

Each τ_i is also represented with an LNT array containing the set of its temporal parameters and its job dynamic parameters (12 elements), mainly τ_i=(C_i, T_i,j, $r_{i,j+1}$, $d_{i,j+1}$, $s_{i,j}$, $e_{i,j}$) and it is initialized as τ_i=(C_i, T_i,$j = 1$, $r_{i,1} = 0$ $d_{i,1} = T_i$, $s_{i,1} = 0$, $e_{i,1} = 0$).

Listing 3.11: SCHEDULER LNT skeleton

```
process SCHEDULER [
ACTIVATION_1 : LNT_Channel_Dispatch
, ... ,
ACTIVATION_N : LNT_Channel_Dispatch ,
NOTIFICATION_1 : LNT_Channel_Event
, ... ,
NOTIFICATION_K : LNT_Channel_Event ]
is
  -- initialization part
  var
     S : LNT_Type_Task_Array ,
     ..
  in
     S[i] := LNT_Type_Task (..);
     ..
     loop
        if (Counter < HYPERPERIOD) then
        -- operational part
           -- time allocation
           -- task state updating
           -- task activation
           -- notification for sporadic task
        else
        -- termination part
           ACTIVATION_1 (T_Stop)
           ...
           ACTIVATION_N (T_Stop)
        end if
     end loop
  end var
end process
```

During the scheduling, the SCHEDULER visits tasks in loops (task-loop) in order of priorities to find and run the Ready tasks.

From the highest to the lowest priority, each task is handled to determine its current state (Ready or Blocked), thus the first task τ_i fixed to be in Ready state, is always the ready task with the highest priority as shown in Figure 3.3.

Note that sporadic tasks are ignored until the reception of an invocation-event (considered in the Blocked state). Thereby, the SCHEDULER should be notified for every invocation-event which is ensured by the NOTIFICATION_i gates.

For each τ_i, SCHEDULER compares $r_{i,j}$ and $d_{i,j}$ with the COUNTER value. Thus, it decides the τ_i state as follows:

Algorithm 1: SCHEDULER algorithm: *Operational part*

```
1  begin
2    while (i ∈ S) do
3      if (BLOCKED or MISSED_DEADLINE τ_i) then
4        Move to τ_{i+1};
5      else if (READY τ_i) then
6        Calculate Allocated_Time of τ_i;
7        hp(i) = 1..(i − 1);
8        while (h ∈ hp(i)) do
9          if (τ_i reaches r_{h,*}) then
             /* τ_i is preempted by τ_h                          */
10           Update Allocated_Time of τ_i;
11         end if
12       end while
         /* if τ_i respects all r_{hp(i),j+1}, the Allocated_Time
            contains the required time to complete the execution
            */
13       Update τ_i state;
14       Activate τ_i;
15       Check sporadic tasks;
         /* if we have a preemption, we return to τ_h; if we have
            a notification from a sporadic task, we return to
            τ_1; else we move to τ_{i+1}                         */
16   end while
17 end
```

- BLOCKED: τ_i is a not awake sporadic task or it is a periodic task awaiting for the next dispatch ($r_{i,j}$ >COUNTER);
- MISSED_DEADLINE: τ_i has missed a deadline ($d_{i,j}$ <COUNTER), in this case, the T_Error label will be sent to the task;
- READY: τ_i is initialized, preempted or dispatched ($r_{i,j}$ <=COUNTER< $d_{i,j}$).

The BLOCKED or MISSED_DEADLINE tasks are ignored. For the first τ_i asserted to be in the READY state, the SCHEDULER decides about its execution, in order to be resumed to the RUNNING state.

At this point, the SCHEDULER can execute τ_i with the whole capacity C_i, and then, it may move to handle other tasks. This can be sufficient for a non-preemptive scheduling of a set of periodic tasks. However, in our context, a task with a higher priority can become READY at any time. Similarly, a sporadic

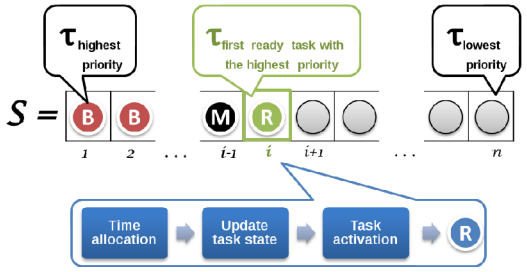


FIGURE 3.3: SCHEDULER algorithm: ready task

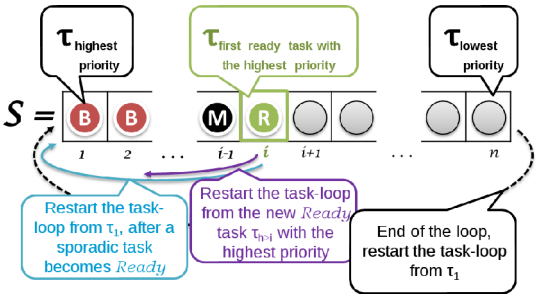


FIGURE 3.4: SCHEDULER algorithm: task-loops

task with a higher priority can become READY by the reception of an invocation-event. These cases can lead to preempt the execution of the current RUNNING task, so they should be considered during the scheduling. To consider the task preemption, we change the SCHEDULER ordered task-loop (τ_i then τ_{i+1} then ... τ_n). The general idea consists of interrupting the loops and restarting the task-loop to consider new READY tasks as shown in Figure 3.4. In the rest of this section, we explain this algorithm through three steps achieving the execution of a given READY task τ_i: *time allocation*, *task state updating* and *task activation*. In addition, we include a *non-periodic task checking* section added if S contains sporadic tasks.

3.4.2.1 Time allocation

This part ensures the calculation of the execution time of the current period for τ_i and prepares an activation order that will be sent in the *task activation* step. We remind that τ_i is the current READY task with the highest priority. We define a `Allocated_Time` variable to compute the execution time. It can return the required time C_i to achieve the execution in the current period, or it can return a part of the execution time, since τ_i can be preempted by another task $\tau_{h<i}$ which is a new ready task with a higher priority. For this reason, while calculating the `Allocated_Time` value, SCHEDULER should always check the states of the tasks with higher priorities. We define $hp(i) = \{1..i-1\}$, a set of indexes of the tasks with higher priorities than τ_i. Simply, SCHEDULER checks $r_{hp(i),j+1}$ values which are the next release times of $\tau_{hp(i)}$ by comparing COUNTER + `Allocated_Time` value with all $r_{hp(i),j+1}$ values as shown in Algorithm 1 (lines 7-12). Three alternatives can be presented leading to fix different activation orders:

1. A complete execution time C_i is allocated, if τ_i respects all $r_{hp(i),j+1}$ (COUNTER + `Allocated_Time` $< r_{hp(i),j+1}$): `T_Completion_Execution` order
2. A preemption is imposed, if `Allocated_Time`+COUNTER reaches an $r_{h<i,j+1}$ of $\tau_{h<i}$, thus (`Allocated_Time`=$r_{hp(i),j+1}$-COUNTER) with two behaviors:

2.1. A preemption at the start time of the execution: `T_Dispatch_Preemption` order

2.2. A preemption in the middle of the execution: `T_Preemption` order

3. A complete time is allocated when τ_i is already preempted: `T_Preemption_Completion` order

Note that in the case of a preemption (alternative 2), τ_i is preempted by $\tau_{h<i}$ so the `SCHEDULER` restarts its task-loop from h to handle $\tau_{h<i}$, the new READY task with the highest priority as shown in Figure 3.4.

3.4.2.2 Task state updating

At this level, τ_i is considered in the RUNNING state, `SCHEDULER` increments `COUNTER` with the `Allocated_Time` value and updates the τ_i array for the next activation. In the case of a preemption, `SCHEDULER` conserves the task state and saves the executed time of τ_i in order to complete the rest later. Else, `SCHEDULER` prepares the tasks for a new period. The periodic task becomes τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} = j * T_i$, $d_{i,j} = d_{i,(j-1)} + T_i$, $s_{i,j+1} = 0$, $e_{i,j+1} = 0$). On the contrary, the parameters of a sporadic task can not be predicted. `SCHEDULER` has no values for its next activation or deadline. Currently, the sporadic task is viewed as τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} >= d_{i,(j-1)}$, $d_{i,j} = *$, $s_{i,j+1} = 0$, $e_{i,j+1} = 0$) and it will be ignored in the scheduling until the reception of a new notification.

3.4.2.3 Task activation

`SCHEDULER` sends to τ_i its current order with the `ACTIVATION_i` gate. It waits for a `T_Completion` response from τ_i and then it moves to τ_{i+1} calculation. In the case of a missed deadline, a `T_Error` label will be the last order sent to τ_i, since it will be ignored in the rest of the simulation.

3.4.2.4 Sporadic task checking

The global state of the set of tasks S may change after each task activation (exchanging events and data, increase of `COUNTER`, etc). Particularly, the sporadic tasks may be waked by the invocation-events and their states may change (to the READY state), so they should be considered in the scheduling with other periodic tasks. For this end, after each task activation, `SCHEDULER` consults all the `NOTIFICATION_i` gates. When receiving a notification at $t_{i,j}$, `SCHEDULER` applies the following steps:

1. marking $t_{i,j}$ equal to the current value of `COUNTER`;

2. verifying $t_{i,j} >= t_{i,j-1} + T_i$: a notification can be ignored. Since we consider T_i as the minimal delay between two successive activations, τ_i can not be reactivated before $t_{i,j} + T_i$ ($t_{i,j+1} >= t_{i,j} + T_i$);

3. if (2) is verified, then, τ_i moves to the READY state with these parameters τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} = t_{i,j}$, $d_{i,j} = r_{i,j} + T_i$, $s_{i,j} = 0$, $e_{i,j} = 0$) Thus, τ_i is considered in the scheduling and served according to its priority;

4. if (2) is verified, then, the SCHEDULER restarts the task-loop (return to τ_1) as shown in Figure 3.4: SCHEDULER should recheck the set of tasks to find the new READY task with the highest priority.

3.4.3 Communication mapping

The tasks can be connected to each other to asynchronously exchange data or events. In sporadic case, each task has at least one connection needed for its activation: the reception of an invocation-event wakes the sporadic task that may move to the READY state.

The LNT processes communicate by symmetrical (bidirectionally) blocking *rendezvous* on the gates. The LNT *rendezvous* on a gate allows the synchronization of processes (several sending and receiving offers at the same time). In our case, we do not need such advanced synchronizations between processes. We consider gates unidirectionally and we use only synchronization between pair of processes (sender and receiver). The asynchronous inter-task connections can not be represented directly with synchronizations of LNT gates since they are blocking *rendezvous*. For this reason, we add an auxiliary process CONNECTOR to represent the connection by means of the Ravenscar protected objects. CONNECTOR has two main gates (INPUT and OUTPUT) and a variable to save data/event.

In Listing 3.12, we give the CONNECTOR LNT skeleton. The CONNECTOR behavior consists of three parts through an infinite loop which the body is a select statement to separate *sending* and *receiving* data/event and *sporadic notification*. Thus, only one operation can be executed at any time and the choice is solved by the possibility of communication on the gates.

Each connection between two TASKs is mapped with a CONNECTOR synchronized (*rendezvous* point) with the sender on the INPUT gate and the receiver on the OUTPUT gate, which assumes the atomicity of the two operations (sending and receiving) and the TASK unicity in awaiting at any time (respectively on the INPUT and OUTPUT gates).

Data are saved and kept until the next reception. Each time a new data is received, the last one is overwritten. While, events are queued in an LNT list type with a definite size. We use non-blocking FIFO, in which new incomings

overwrite previous events in the case of an overflow. In the case of an empty FIFO, TASK receives an EMPTY label without blocking.

Since we consider a special invocation-event for sporadic task activation, we add a third gate NOTIFICATION to the CONNECTOR process to be synchronized with the SCHEDULER. We use this gate to notify the SCHEDULER of every new reception, thus, it considers the concerned task in the scheduling.

Listing 3.12: CONNECTOR LNT skeleton

```
process CONNECTOR [
INPUT: LNT_Channel_Data, OUTPUT: LNT_Channel_Data,
NOTIFICATION: LNT_Channel_Event]
(Queue_Size: Nat)
is
   loop
      select
         -- inputs of event/data part
         INPUT ()
         []
         -- output of event/data part
         OUTPUT ()
         []
         -- sporadic part
         -- needed to notify SCHEDULER
         -- when receiving a new event
         NOTIFICATION ()
      end select
   end loop
end process
```

3.4.4 Composition and synchronization

We complete the proposed pattern with two indispensable steps: composition and synchronization. All described LNT processes should be structured (connected) to form the main system. This is ensured by the LNT par composition for parallelism and synchronization. Thus, we assemble the whole system in the MAIN process. Note that a set of LNT types and channels are used for the TASK-CONNECTOR, CONNECTOR-SCHEDULER and TASK-SCHEDULER synchronizations. For example, we include, in Listing 3.13, the type and the channel for the TASK-SCHEDULER synchronization.

For further explanations, we include an example of a task model whose MAIN process is included in Listing 3.14 and graphically presented in Fig. 3.5. The initial task model (τ_1, τ_2) consists of a periodic task connected to another sporadic task running on a scheduler (*Producer-Consumer* system). The obtained LNT specification contains five processes synchronized in the MAIN process. The par composition is globally used for the following synchronizations:

- The TASK_CONSUMER and CONNECTOR on the RECEIVE_A gate;
- The CONNECTOR and TASK_PRODUCER on the SEND_A gate;
- The CONNECTOR and SCHEDULER on the NOTIFICATION_1 gate;
- The TASK_CONSUMER, TASK_PRODUCER and SCHEDULER on the ACTIVATION_1 and ACTIVATION_2 gates.

Listing 3.13: LNT types and channels for TASK-SCHEDULER synchronization

```
type LNT_Type_Dispatch is
    T_Dispatch_Completion,
    T_Dispatch_Preemption,
    T_Preemption,
    T_Preemption_Completion,
    T_Stop,
    T_Error,
    T_Completion
end type

channel LNT_Channel_Dispatch is
    (LNT_Type_Dispatch)
end channel
```

Listing 3.14: *Producer-Consumer*: LNT MAIN

```
process MAIN [
    ACTIVATION_1: LNT_Channel_Dispatch,
    ACTIVATION_2: LNT_Channel_Dispatch,
    NOTIFICATION_1: LNT_Channel_Event,
    SEND_A: LNT_Channel_Port,
    RECEIVE_A: LNT_Channel_Port
] is
    par
     ACTIVATION_1, RECEIVE_A->
     TASK_CONSUMER[ACTIVATION_1, RECEIVE_A]
     ||
     NOTIFICATION_1, SEND_A, RECEIVE_A->
     CONNECTOR[SEND_A, RECEIVE_A, NOTIFICATION_1]
     ||
     SEND_A, ACTIVATION_2->
     TASK_PRODUCER[ACTIVATION_2, SEND_A]
     ||
     ACTIVATION_1, ACTIVATION_2, NOTIFICATION_1->
     SCHEDULER[ACTIVATION_1, ACTIVATION_2, NOTIFICATION_1]
    end par
end process
```

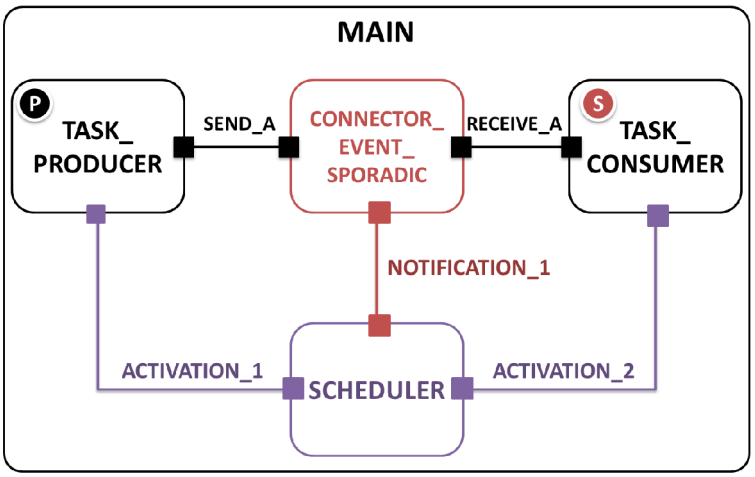


FIGURE 3.5: *Producer-Consumer*: LNT graphical MAIN

3.4.5 Discussion

The defined LNT specification (the set of LNT definitions composed in the MAIN process) represents a formal executable semantics for a real-time task model, where TASKs are connected through CONNECTORs and scheduled by the SCHEDULER.

The proposed LNT mapping is flexible enough to support various task models with minor changes: periodic/sporadic tasks, independent/communicated tasks and preemptive/non-preemptive tasks. In addition, real-time features are modularly designed which increases the mapping extensibility: each component can be separately completed or extended. For example, the TASK skeleton can be easily enriched with more behavior. Additional code can be added using LNT functions or processes, and then they are just called (process instances, function calls) in the TASK process. Similarly, the scheduling mapping can be extended with other scheduling algorithms since we specify an explicit SCHEDULER that encapsulates all scheduling calculations.

Thanks to its programming ability, many scheduling algorithms may be implemented using the LNT language. For instance, we have developed a scheduler based on the Earliest Deadline First EDF [110], which is the most common preemptive dynamic-priority scheduling for real-time systems. The SCHEDULER skeleton is conserved. The modifications can be limited in the *operational part*, mainly, the *time allocation* operation. The other manipulations (*task state updating*, *task activation* and *non-periodic task checking*) can be conserved and thus TASK and CONNECTOR processes need no changes (see section 4.4 of chapter 4).

3.5 Conclusion

In this chapter, we presented the first contribution in this thesis about the definition of a formal pattern of a real-time task model using the LNT language. The proposed LNT pattern is generically specified so that it can be considered as the base of model transformations based on real-time architectural languages.

The next chapter presents an application of this pattern in the case of an AADL model-based approach.

4

AADL model transformation

Contents

4.1 Introduction 70

4.2 AADL in a nutshell 70

4.2.1 Core language 70

4.2.2 Behavior annex 74

4.2.3 AADL subset 75

4.3 Model transformation 76

4.3.1 Scheduling mapping 78

4.3.2 Communication mapping 81

4.3.3 Hierarchical mapping 85

4.3.4 Discussion 88

4.4 Transformation of a larger AADL subset 88

4.5 Behavioral mapping 90

4.5.1 Data mapping 90

4.5.2 Port and port connection new mapping 91

4.5.3 Behavior specification mapping 93

4.6 Conclusion 98

4.1 Introduction

In this chapter, we discuss the applicability of the proposed LNT pattern as a software engineering practice. We aim at integrating the formal methods in an AADL model-based development process. During the design phase, the formal verification of the AADL model seems useful and complementary to traditional syntactic and semantic analyses. For this end, we define a model transformation (AADL2LNT) from an initial AADL model into an LNT specification based on the proposed LNT pattern. This transformation achieves the automatic generation of an LNT specification compliant with the CADP toolbox.

This chapter is organized as follows: the AADL language and the supported subset are introduced in section 4.2; the AADL2LNT model transformation is detailed in section 4.3 through its different levels (scheduling, communication and hierarchical); a larger AADL mapping is discussed in section 4.4; and finally, the transformation is extended with the Behavior annex mapping in section 4.5.

4.2 AADL in a nutshell

AADL (Architecture Analysis and Design Language) [10, 60] is an SAE [1] standard (SAE AS 5506), proposed since 2004. Its second version was published in 2009 and recently revised in 2017 [2]. AADL is based on the MetaH language. It is an architectural language for real-time embedded systems modeling in safety-critical domains such as avionics, automotive electronics and robotics. AADL has a textual syntax as well as a graphical representation of some elements (see Figure 2.5 section 2.3.2). The AADL syntax and static/dynamic semantics are described in its standard [10], which defines the core language that is designed to be extensible with property-sets and annexes. In this thesis, we focus on the AADL core language and the Behavior Annex.

4.2.1 Core language

The AADL core language describes mainly the software and hardware components of a system, and how they are assembled to form a complete system. Different concepts and key elements of the AADL language are summarized in Figure 4.1. In the following, we detail some AADL elements required for the AADL2LNT model transformation.

[1] SAE: Society of Automotive Engineers

[2] The AADL committee has started work on a major revision AADL V3 based on industrial experience, using AADL V2.2 as baseline.

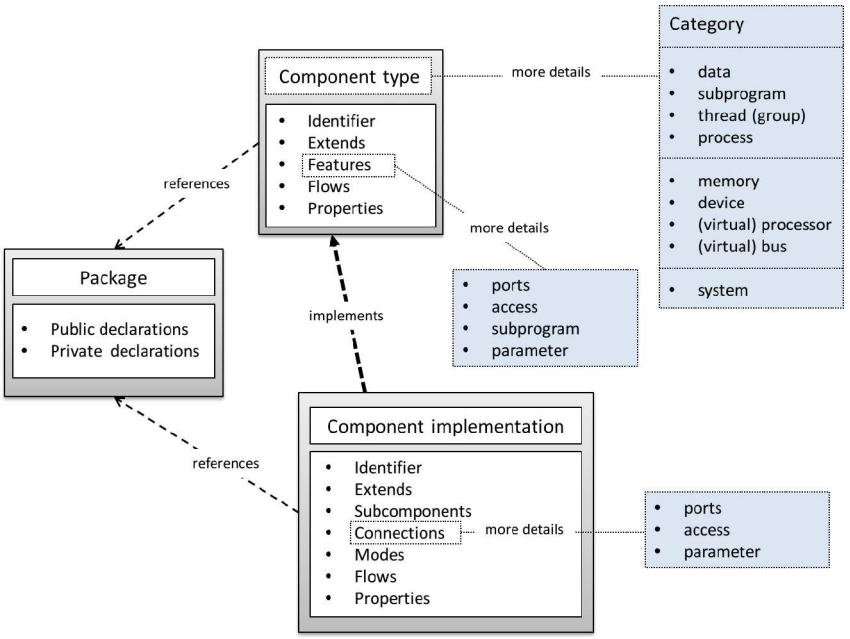


FIGURE 4.1: Summary of AADL elements [61]

4.2.1.1 Components

An AADL component is defined through a type (it declares the component interface elements called features) and zero or more implementations (they represent the component internal structure), as shown in Figure 4.1. AADL defines three categories of components: software components (`subprogram`, `subprogram group`, `data`, `thread`, `thread group` and `process`); hardware/execution platform components ((`virtual`) `processor`, `device`, (`virtual`) `bus`, and `memory`); and system composition component (`system`).

Software components

This part models the applicative part of the system, including source text, virtual address spaces, concurrent tasks and their interactions. It contains mainly the following components:

Data. The `data` component represents static data and data types within a system.

Subprogram. The `subprogram` component represents sequential executed source text, which can be coded in programming languages like the C and Ada lan-

guage. The subprograms are callable from within thread and subprogram components.

Thread. A concurrent schedulable unit of sequential execution code. The thread should be declared within a process component. It is scheduled by a scheduler (processor component). It can contain subprogram and data components, and it can provide or require access to data components.

The dynamic semantics of a thread is defined using an hybrid automaton, including its different states (e.g. suspended, ready and running). The transitions between these states occur as a result of dispatch requests, faults and runtime service calls. The dispatch semantics of a thread is given for the standard dispatch protocols such as periodic, sporadic, aperiodic, timed, hybrid, and background threads. More details about the thread syntax and semantics are given through the AADL2LNT transformation description in section 4.3.1.

Process. The process component is a virtual address space containing data, thread and subprogram associated with the process and with its subcomponents.

Hardware components

This part specifies the computing hardware and the physical environment, that are capable of the scheduling of threads, storing source text and data and performing communication. It contains mainly the following components:

Bus. The bus component represents hardware and associated communication protocols (communication channels) to exchange control/data among other execution platform components.

Processor. The processor component is an abstraction of hardware and software for the scheduling and execution of threads. The processors can support different scheduling protocols (e.g. RM and EDF).

Device. The device component is considered as an entity to interface with the external environment (such as sensors). A device can interact with both hardware and software components (e.g. using port connections).

System component

This part models the hierarchical composites of software and hardware components through the system component:

System. The `system` component is a composite of `system` (subsystems) or of the software and hardware components (see Figure 4.2 below).

4.2.1.2 Connections

The AADL connections are linkages established between the component features to exchange data and control. There are four categories of features, as shown in Figure 4.1: port, subprogram, parameter and subcomponent access. They enable three types of connections: port connection, parameter connection and access connection.

A semantic connection is represented by a set of one or more connection declarations that follow the component hierarchy from the ultimate connection source to the ultimate connection destination (for example, the port connections `C1`, `C2` and `C3` in Figure 4.2 below).

Port connection

The `thread` components may declare ports to be in interaction with other components (`thread`, `device`, `processor`). They can be declared to be data, event or event data ports. They are directional: *in*, *out* or *in out*. A port connection allows the transfer of data and/or event between two components, explicitly declared between two ports at the `process` or `system` levels. More details about the port and port connections semantics are given through the AADL2LNT transformation description in section 4.3.2.

4.2.1.3 Properties

AADL properties provide additional information about the AADL elements. The standard [10] defines a set of predeclared properties and property types (standard properties). These properties can be associated with the component types, component implementations, subcomponents, features, connections and subprogram calls.

4.2.1.4 AADL system modeling

A system model consists of an application software mapped to an execution platform. It is modeled through a set of nested components whose top-level is the `system` component as included in Listing 4.1 and graphically depicted in Figure 4.2: the `thread` component is declared within a `process` component; the `system` contains mainly a set of `subcomponents` that may be `data`, `process`, `processor`, `bus` or `device`, a set of `connections` to declare

port connections of `processes` and `devices` and a set of `properties` in which the `Actual_Processor_Binding` property is used to bind `processes` with the `processor`.

A system instance represents a complete component hierarchy as specified recursively through the subcomponents of each component: from the `system` subcomponents down to the lowest level defined in the architecture specification.

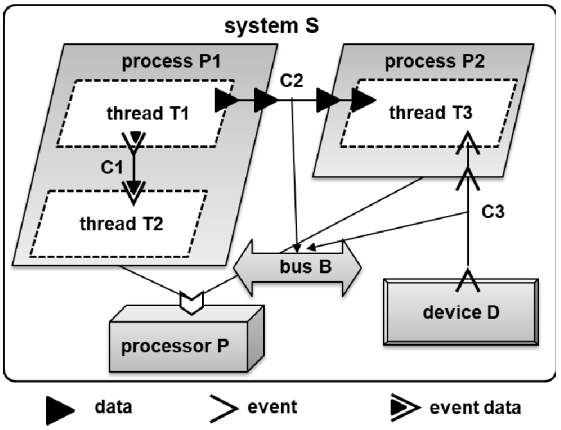


FIGURE 4.2: Example of AADL system model: graphical representation

LISTING 4.1: Example of AADL system model

```
system implementation System_Example.Impl
subcomponents
 CPU             : processor P.Impl;
 Process_P1      : process P1.Impl;
 Process_P2      : process P2.Impl;
 Device_Example  : device D.Impl;
 Bus_Example     : bus B.Impl;
connections
 C2: port Process_P1.Port_A -> Process_P2.Port_B
    {Actual_Connection_Binding=>(reference (Bus_Example));};
 C3: port Device_Example.Port_C -> Process_P2.Port_D
    {Actual_Connection_Binding=>(reference (Bus_Example));};
properties
 Actual_Processor_Binding=>(reference (CPU)) applies to Process_P1;
 Actual_Processor_Binding=>(reference (CPU)) applies to Process_P2;
end System_Example.Impl;
```

4.2.2 Behavior annex

The Behavior annex [9] (called also Behavior model annex) is an SAE standard (SAE AS 5506/3) originally published in 2011. Then, it is revised in 2017 with a

number of errata and improvements to align with the AADL V2.2 core language. The aim of the Behavior annex is to refine the implicit behavior specifications that are specified by the AADL core language as follows: describing the internal behavior of the component implementations; extending the default AADL execution model semantics, such as `thread` dispatch protocols; providing more precise `subprogram` calls synchronization protocols.

A `Behavior_Specification` clause may be added within the `subprogram`, `thread` and `device` components to describe the behavior as a *transition system* with guards and actions. Its dynamic behavior is based on the transitions between states. A transition is a change of the current state from a source state to a destination state when its condition (guard) is satisfied. The corresponding set of actions is consequently performed.

More details about the Behavior annex syntax and semantics are explained in the AADL2LNT transformation description in section 4.5.

4.2.3 AADL subset

The AADL language describes different concepts of real-time embedded systems with a rich semantics detailed in its standard [10]. The language covers important aspects (timing requirements, fault and error behaviors, time and space partitioning, safety properties, etc.) that can not be wholly analyzed in one work. According to our verification purposes, we define an AADL subset whose components are depicted in Figure 4.3 (AADL part). Moreover, the consideration of the Ravenscar profile requires some additional restrictions applied at the model level, meaning that the AADL subset should be Ravenscar compliant. Mainly, the profile claims that `threads` are either sporadic or periodic and they are schedulable according to the RM scheduling.

In this thesis, we aim at defining an executable formal semantics of the AADL model viewed as a set of communicating tasks. This abstraction allows different alternatives of verification, such as the schedulability analysis, `thread` execution simulation, `thread` behavior analysis and the verification of a set of communication properties (see chapter 5).

At the model level, we consider the `system` as a set of `threads` in communication through port connections. Thus, the following components are supported: `data`, `thread`, `process`, `processor` and `device`. We do not support neither AADL shared access nor AADL flows/modes. For behavioral descriptions, we consider the Behavior annex, especially, for the `thread` components to specify its behavior as a *transition system*. So the `Behavior_Specification` clauses are part of the transformation. Finally, the model is completed by a set of AADL standard properties attributed to different components. We distinguish the following sets:

- Properties specifying constraints for the software-hardware binding such as the `Actual_Processor_Binding` property that can list only one `processor`;
- Properties specifying the temporal and scheduling information like `Deadline`, `Scheduling_Protocol`, `Dispatch_Protocol`, `Period`, `Priority`, `Compute_Execution_Time` and `Dispatch_Offset`;
- Properties specifying information for the port connections and queuing such as `Input_Time` , `Output_Time`, `Queue_Size`, `Dequeue_Protocol`, `Queue_Processing_Protocol` and `Overflow_Handling_Protocol`.

In Listing 4.2, we provide a simple AADL example using the considered subset: a `thread` component with the port connections, scheduling properties and Behavior annex specification. This `thread` compares periodically (every 10s) two arriving data `a` and `b`. If they are equal, then an event is triggered through the `success` port, else an event is triggered through the `fail` port.

Listing 4.2: Example of AADL `thread` component

```
thread compare
  features
    a: in data port Base_Types::Integer;
    b: in data port Base_Types::Integer;
    fail: out event port;
    success: out event port;
  properties
    Dispatch_Protocol => Periodic;
    Compute_Execution_Time => 1s .. 5s;
    Period => 10s;
end compare;
thread implementation compare.impl
  annex behavior_specification {**
    states
      s0: initial complete final state;
    transitions
      s0 -[on dispatch]-> s0
      {if (a = b) success! else fail! end if};
  **};
end compare.impl;
```

4.3 Model transformation

The AADL2LNT transformation is based on the proposed LNT pattern (chapter 3). In addition, it requires a set of new LNT definitions to cover the considered AADL subset.

The AADL2LNT transformation is described with a set of corresponding rules between the AADL and LNT languages. The main idea of the transformation consists in mapping and encapsulating each AADL component into an LNT process to obtain a modular specification. According to the component categories, we define a set of LNT definitions. The AADL2LNT transformation overview is given in Figure 4.3, highlighting main corresponding rules between both the AADL and LNT languages as follows: the AADL `thread` is transformed into the `TASK` process, that represents the `thread` component with its supported features, subcomponents and behavior specification; the AADL ports are transformed into LNT gates; the AADL port connections are mapped using the `CONNECTOR` processes; the AADL `processor` becomes the `SCHEDULER` process, which implements the `processor` scheduling protocol; and the AADL `device` component is mapped with a new defined `DEVICE` process. Finally, all processes are composed and synchronized to form the whole system in the `MAIN` process.

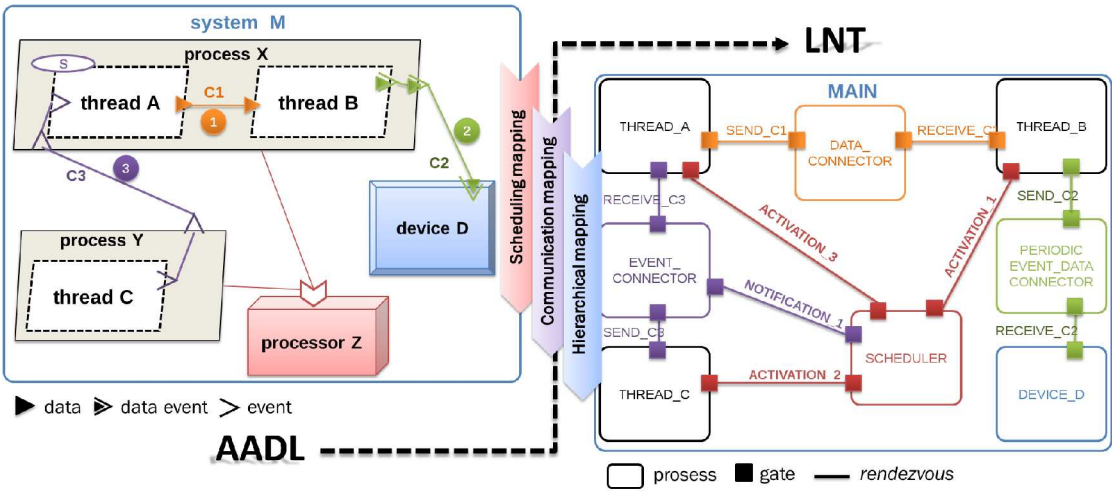


FIGURE 4.3: Overview of the AADL2LNT transformation

The AADL2LNT transformation can be represented at three levels, allowing the mapping of different kind of AADL models as follows:

- Scheduling mapping ensures the transformation of models with independent `threads`;
- Communication mapping completes the mapping by considering the port connections between the `thread` and `device` components;
- Hierarchical mapping concerns mainly the `system` component (with some additional rules) to achieve the mapping of the whole system within the obtained `MAIN` process.

In the following, we develop and justify different AADL2LNT transformation levels: corresponding rules are graphically represented and the obtained LNT definitions are included. Later in this chapter, we present the behavioral mapping level based on the transformation of the Behavior annex.

4.3.1 Scheduling mapping

At this level, we present a set of the execution and scheduling rules, which concerns only the AADL `thread` (independent) and `processor` components mapped respectively into the LNT `THREAD` and `SCHEDULER` processes.

4.3.1.1 Thread mapping

The `thread` component becomes the `TASK` process described in section 3.4.1 (chapter 3). The `thread` rule is illustrated in Figure 4.4. The `TASK` process takes the `thread` implementation name prefixed by "`THREAD_`" and declares the default `ACTIVATION` gate, as included in Listing 4.3.

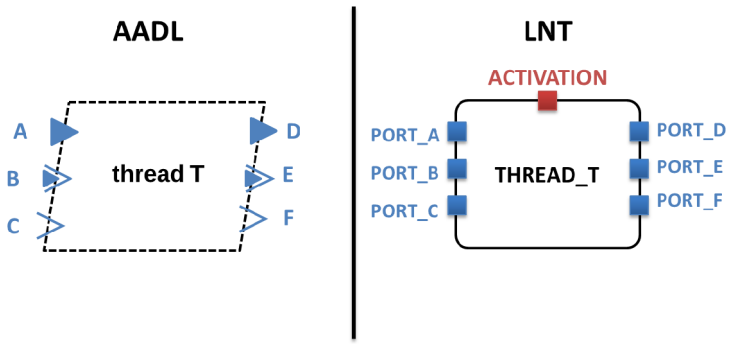


FIGURE 4.4: AADL `thread` transformation rule

The supported standard properties are used to specify the temporal parameters of the `thread`, as follows: the `Dispatch_Protocol` property represents its dispatch model; the `Period` property represents its period T_i; and the `Compute_Execution_Time` property represents its capacity C_i. In the AADL language, the dispatch semantics is given for standard dispatch protocols such as periodic, sporadic, aperiodic, timed, hybrid, and background `threads`. Since we consider a Ravenscar compliant model, we support periodic and sporadic dispatch models, as described in the AADL standard:

- Periodic `threads`: they are periodically dispatched at time intervals of the specified `Period` property value.

- Sporadic threads: they are activated as the result of an event/event data (invocation-event) arriving at an event/event data port of the thread. The time interval between two successive dispatch requests will never be less than the associated Period property value.

Listing 4.3: LNT THREAD for AADL independent thread

```
process THREAD_* [ACTIVATION: LNT_Channel_Dispatch] is
   loop
      select
         select
            ACTIVATION (T_Dispatch_Completion)
            []
            ACTIVATION (T_Dispatch_Preemption)
            []
            ACTIVATION (T_Preemption)
            []
            ACTIVATION (T_Preemption_Completion)
         end select;
         ACTIVATION (T_Completion)
         []
         ACTIVATION (T_Error)
         []
         ACTIVATION (T_Stop)
      end select
   end loop
end process
```

As described in section 3.4.1, the thread dispatching is ensured by the SCHEDULER through the defined activation orders (T_Completion, T_Dispatch_Preemption, T_Preemption_Completion, T_Dispatch_Completion and T_Preemption).

The LNT processes are suitable for representing AADL thread described in the AADL standard as a schedulable unit that can execute concurrently with other threads. The dynamic semantics of the AADL thread are defined using a hybrid automaton (*thread scheduling and execution states* automaton) included in Figure 4.5. THREAD covers an important part of the AADL thread semantics for the thread dispatching, scheduling and execution. Compared to the standard *thread scheduling and execution states* automaton, our state automaton of Figure 3.2 (chapter 3) excepts the awaiting states (shared resources, subprogram calls and background thread) which are not supported in this thesis. In addition, the standard defines a suspended AWAITING DISPATCH state for the threads when completing the execution of the current dispatch, this state corresponds to the BLOCKED state in the proposed LNT pattern.

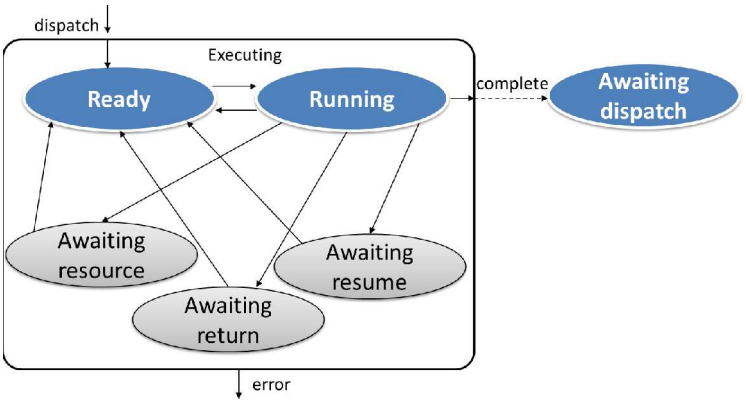


FIGURE 4.5: AADL *thread scheduling and execution states* automaton [10]

4.3.1.2 Processor mapping

The `processor` component becomes the `SCHEDULER` process developed in section 3.4.2 (chapter 3). The `processor` rule is depicted in Figure 4.6. The AADL `processor` is a hardware component that ensures the scheduling and execution of `threads`. The `SCHEDULER` process represents the `processor` component by implementing its scheduling algorithm.

The generation of the `SCHEDULER` requires the extraction of the task model of the AADL system, based on tasking descriptions in section 3.3 (chapter 3). Such an abstraction is feasible since the AADL language is defined to design and analyze real-time systems. In fact, the AADL model can be abstracted as a task model: a set of n `thread` instances $S = \{\tau_1, ..., \tau_n\}$, bound to one `processor` supporting the RM scheduling. Different temporal parameters are specified using the set of the supported standard properties as follows: `Dispatch_Protocol` (periodic or sporadic), `Compute_Execution_Time` (C_i), `Period` (T_i), `Deadline` (D_i) and `Priority` (P_i or simply the index i), `Dispatch_Offset` (O_i).

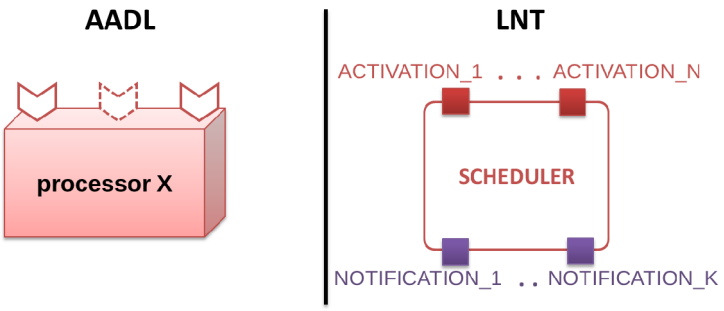


FIGURE 4.6: AADL `processor` transformation rule

For the `SCHEDULER` process declaration, each `thread` (τ_i) is bound to the

processor through the Actual_Processor_Binding property corresponds to the declaration of an ACTIVATION_*i* gate in the SCHEDULER process to be synchronized with the equivalent THREAD, as shown in Figure 4.3. In the case of a sporadic thread, an additional NOTIFICATION_i gate is also declared. Based on the developed mapping in section 3.4.2 (chapter 3), the SCHEDULER process body is composed of three parts, which are generated as follows:

- The *initialization part*: this part declares and initializes the task array, so it needs all information about the set of threads and their scheduling properties.
- The *Operational part*: this is a generic part depending only of the considered scheduling algorithm, which is in our case the RM scheduling algorithm.
- The *Stopping part*: this part requires only the number of thread instances to include different communications for the T_Stop order.

4.3.2 Communication mapping

The communication mapping level concerns the AADL models with dependent threads. We support port connections between threads and devices, that are declared at the process and system levels and typed with the data components.

At the communication mapping level, we aim to draw the thread connections without the consideration of the data contents. We simply map data/event into an LNT enumerated type (AADLDATA label) exchanged between different THREADs through the corresponding channel, as included in Listing 4.4.

Listing 4.4: LNT type and channel for AADL data component

```
type LNT_Type_Data is
   EMPTY, AADLDATA
end type

channel LNT_Channel_Data is
   (LNT_Type_Data)
end channel
```

4.3.2.1 Port mapping

At the THREAD level, the port declarations are transformed into LNT gate declarations as shown in Figure 4.4. The process body of the THREAD process (Listing 4.3) is completed with an *initialization* part using the LNT var statement. As shown in Listing 4.5, for each declared port, named P, we proceed as follows:

- A gate declaration (`PORT_P: LNT_Channel_Data`) is added;
- A variable (`P: LNT_Type_Data`) is declared and initialized in the *initialization* part;
- A corresponding communication is added as follows:
  - For *out* port: `PORT_P (!P)`
  - For *in* port: `PORT_P (?P)`

Note that data or event/event data ports are exactly mapped at the `THREAD` level, while the difference between these types (reception, queuing, etc) is ensured by the communication mechanism using the `CONNECTOR` processes.

Listing 4.5: LNT `THREAD` for AADL `thread` with port connections

```
process THREAD_* [ACTIVATION: LNT_Channel_Dispatch,
   PORT_IP: LNT_Channel_Data, -- in port
   PORT_OP: LNT_Channel_Data] -- out port
is
   var IP : LNT_Type_Data, OP : LNT_Type_Data in
   IP := EMPTY;
   OP := EMPTY;
   loop
      select
         select
            ACTIVATION (T_Dispatch_Completion)
            PORT_IP (?IP)
            -- --
            PORT_OP (!OP)
            []
            ACTIVATION (T_Dispatch_Preemption)
            PORT_IP (?IP)
            []
            ACTIVATION (T_Preemption)
            []
            ACTIVATION (T_Preemption_Completion)
            PORT_OP (!OP)
         end select;
         ACTIVATION (T_Completion)
         []
         ACTIVATION (T_Error)
         []
         ACTIVATION (T_Stop)
      end select
   end loop
end process
```

4.3.2.2 Port connection mapping

The port connections are transformed through the synchronizations with the CONNECTOR process as developed in section 3.4.3 (chapter 3). We conserve the CONNECTOR definition, as it allows unidirectional communication, so we consider only 1-to-1 connections with no *in out* ports.

Each port connection becomes a CONNECTOR instance as shown in Figure 4.7. The CONNECTOR should be synchronized on its INPUT and OUTPUT gates between two THREADs equivalent, respectively, to the threads of *in* and *out* ports.

The CONNECTOR represents the AADL semantics port connection which includes all port connection declarations (at the process and system levels) that follow the component containment hierarchy in the instantiated system from a source thread (*out* port) to a destination thread (*in* port). Thus, all port connection declarations are abstracted in the CONNECTOR synchronizations. As shown in Figure 4.3, different port connections, at the process level (inter-thread connection ❶) or at the system level (thread-device connection ❷ and process-process connection ❸), are similarly transformed into the CONNECTOR instances.

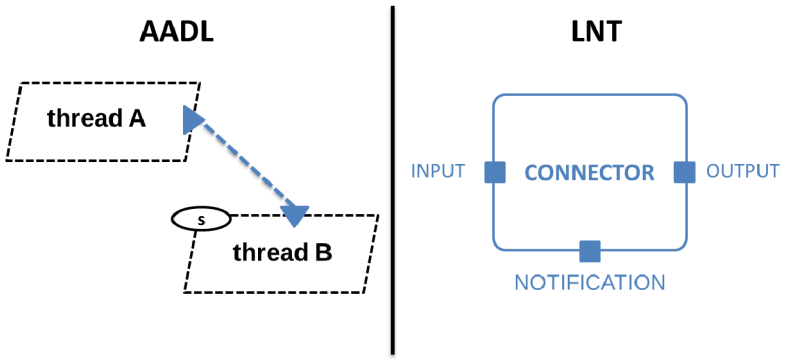


FIGURE 4.7: AADL port connection transformation rule

In this thesis, we consider asynchronous port connections, whose determinism is ensured by the Ravenscar constrained protected object as developed in section 3.4.3 (chapter 3). During the port connection, the content of incomings is frozen during the thread execution: the port variable content is not affected by the arrival of new incomings. Data and events arriving through *in* ports are available to the thread at a specified input time, fixed by the Input_Time property. This communication model is assumed in the LNT specification through the INPUT synchronization (reading the port content) between the CONNECTOR and the THREAD corresponding to the thread of *in* port. According to our LNT pattern, the INPUT *rendezvous* is fixed at the start time of each period equivalent to the Start_Time value of the Input_Time property. After the INPUT

rendezvous any new incoming data or event becomes available only at the next start time.

In addition, the AADL port output is transferred to other components at an output time specified by the `Output_Time` property. The transfer of data or event corresponds to the `OUTPUT` synchronization with the `thread` of *out* port, which is fixed at the completion time of each period equivalent to the Completion_Time value of the `Output_Time` property.

At the `THREAD` level, the communications are also controlled through the orders from the `SCHEDULER`, that fix the input and output times as follows:

- `T_Dispatch_*` represents the start time, `THREAD` receives the inputs;
- `T_*_Complete` represents the completion time, `THREAD` sends the outputs.

According to the port type (data or event) and the `thread Dispatch_Protocol` property (periodic or sporadic), we define three `CONNECTOR` types. In the case of the periodic `threads` with data port connections, a `Data_CONNECTOR` (Listing 4.6) is used without queuing or event notifications. It keeps the data until the next reception, each time a new data is received, the last one is overwritten. When exchanging events or event data, we use `Event_CONNECTORs` I and II (Listings 4.7 and 4.8) with a list of inputs with a definite size. Event buffers implement the set of queuing properties as follows: `Queue_Size` (`Queue_Size` parameter), `Overflow_Handling_Protocol` (drop oldest, drop newest), `Queue_Processing_Protocol` (FIFO, LIFO) and `Dequeue_Protocol` (one item).

Listing 4.6: LNT `CONNECTOR` for AADL data port connection

```
process Data_CONNECTOR[
   Input: LNT_Channel_Data, Output: LNT_Channel_Data]
is
   var
      Data : LNT_Type_Data in
      Data := EMPTY;
      loop
         select
            Input (?Data)
            []
            Output (Data)
         end select
      end loop
   end var
end process
```

As detailed in section 3.4.3 (chapter 3), the `SCHEDULER` needs to be notified for every new incoming invocation-event for the sporadic `threads`. This is ensured by the `NOTIFICATION` gates to synchronize the `EVENT_CONNECTORs` with the `SCHEDULER` (Figure 4.3). Thus, at the reception of a new event, the

EVENT_CONNECTOR notifies the SCHEDULER, to consider the concerned THREAD in the scheduling.

Listing 4.7: LNT Event_CONNECTOR I

```
process Event_CONNECTOR[
   Input: LNT_Channel_Data,
   Output: LNT_Channel_Data](
   Queue_Size: Nat)
is
   var
      Data : LNT_Type_Data,
      FIFO : LNT_Type_Data_FIFO,
      Is_New : bool
   in
      FIFO := {};
      Data := EMPTY;
      Is_New := false;
      loop
         select
         Input (?Data);
         Is_New := true;
         if length (FIFO)>=Queue_Size
         then
            FIFO := tail (FIFO)
         end if;
         FIFO := append (Data, FIFO)
         []
         if (FIFO != {}) then
            Output (Head (FIFO));
            FIFO := tail (FIFO)
         else
            Output (EMPTY)
         end if
         end select
      end loop
   end var
end process
```

Listing 4.8: LNT Event_CONNECTOR II

```
process nonperiodic_Event_CONNECTOR[
   Input: LNT_Channel_Data,
   Output: LNT_Channel_Data,
   Notification: LNT_Channel_Notif](
   Queue_Size: Nat)
is
   var
      Data : LNT_Type_Data,
      FIFO : LNT_Type_Data_FIFO,
      Is_New : bool
   in
      FIFO := {};
      Data := EMPTY;
      Is_New := false;
      loop
         select
         Input (?Data);
         Is_New := true;
         if length (FIFO) >= Queue_Size
         then
            FIFO := tail (FIFO)
         end if;
         FIFO := append (Data, FIFO)
         []
         if (FIFO != {}) then
            Output (Head (FIFO));
            FIFO := tail (FIFO)
         else
            Output (EMPTY)
         end if
         []
         if (Is_New) then
            Notification (Incoming_Event);
            Is_New := false
         else
            Notification (No_Event)
         end if
         end select
      end loop
   end var
end process
```

4.3.3 Hierarchical mapping

The hierarchical mapping represents the last phase of the AADL2LNT transformation. At this level, we transform the `system` component in order to instantiate and synchronize all the LNT definitions. In addition, we define transformation rules for some AADL components, which have no equivalent in the proposed LNT pattern (chapter 3).

4.3.3.1 System mapping

All AADL components are hierarchically structured in the `system` component. As shown in Figure 4.3, the `system` component becomes the LNT `MAIN` process. This generation is based on the composition and synchronization phase developed in section 3.4.4 (chapter 3). The `MAIN` generation can be resumed in three steps:

- Extracting of the `THREAD` list: all `system` subcomponents are visited to form the list of the `THREAD` instances.
- Preparing of the `CONNECTOR` synchronizations: for each port connection, a `CONNECTOR` instance is created. We use the connection identifier, which is prefixed with "`SEND_`" and "`RECEIVE_`" to represent two gates. For example, the connection `C1` from Listing 4.3 is represented by the `SEND_C1` and `RECEIVE_C1` gates. These gates are used for the `THREAD` and `CONNECTOR` synchronizations, as shown in Figure 4.3:
  - `RECEIVE` synchronization represents the reading of the port content by the `thread` of *in* port.
  - `SEND` synchronization represents the transfer of the data or event from the `thread` of *out* port.
- Global composition: all `THREAD`s are synchronized with the `SCHEDULER` on the `ACTIVATION_i` gates. Similarly, all `CONNECTOR`s are firstly synchronized with the `THREAD`s on the `RECEIVE` and `SEND` gates, then they are synchronized with the `SCHEDULER` on `NOTIFICATION_i` gates as shown in Figure 4.3.

4.3.3.2 Other rules

Throughout previous mapping levels, the proposed LNT pattern is used in the definition of AADL2LNT transformation with minor changes. However, the AADL language represents additional architectural features that are supported in this thesis. The transformation should then be completed by rules concerning other AADL components such as `processes` and `devices`.

Process mapping

In the AADL language, a `thread` should be declared within a `process` component. The `process` component represents a protected virtual address space that can be ignored in the verification. So the `processes` have no equivalent in the produced LNT specification and its dispatch semantics is omitted.

The AADL model may contain a `process` with a composition of `threads`. In this case, the corresponding `THREAD` instances are directly added in the `MAIN` process.

Device mapping

The `device` components are supported since they can be in interaction with `thread` components. They are not mapped at the scheduling mapping level since they are not scheduled, also their internal behavior is not considered. thus, we opt to define a simple LNT specification for the `devices` sufficient for the `thread-device` port connections.

The `device` becomes an LNT process prefixed by "`DEVICE_`", as shown 4.8. Unlike `THREAD`, `DEVICE` has no activation gate, but their port declarations are similarly mapped. `DEVICE` behavior consists simply of a loop-select statement gathering `PORT_*` communications as shown in the example of Listing 4.9.

In addition, `device` port connections are similarly mapped as the `thread` port connections through the `CONNECTORs` at the `MAIN` level, as depicted in Figure 4.3.

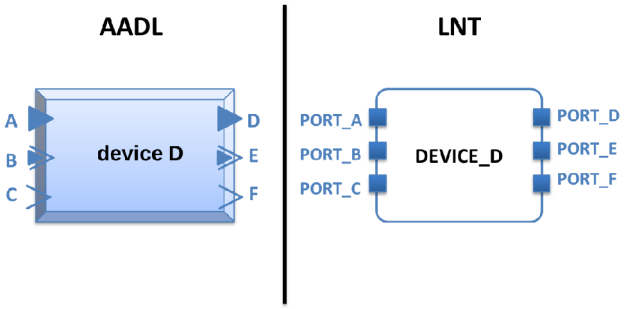


FIGURE 4.8: AADL `device` transformation rule

LISTING 4.9: LNT `DEVICE` for AADL `device` component (example)

```
process Device_D [
   PORT_A: LNT_Channel_Event, PORT_B: LNT_Channel_Event]
is
   var
      A: Bool,
      B: Bool
   in
      A:= false; B:= false;
      loop
         select
            PORT_A (?A)
            []
            PORT_B (!B)
         end select
      end loop
   end var
end process
```

4.3.4 Discussion

The defined AADL2LNT is the result of several adjustments (at the AADL subset) and refinements (in the LNT pattern) to obtain LNT specifications exploitable with verification tools. In fact, an LNT specification may fail its state space generation when the number of states or transitions explodes, known as the space explosion problem (see section 2.2.5.1 of chapter 2). In this case, the formal verification phase will be interrupted and the AADL model transformation becomes obsolete.

For this reason, the proposed LNT pattern is based on the encapsulation of the temporal calculations within the `SCHEDULER`, thus the synchronizations between the processes of the LNT specification are restricted to a set of enumerated labels (activation orders, data labels, etc.), which allows to reduce significantly the generated system state space at the verification phase.

In addition, restrictions were applied in the considered AADL subset, especially at the communication mapping level. The AADL language provides a rich semantics for ports and their connections in all aspects (topologies, directions, timing, etc.). Since it provides bidirectional gates and multiple synchronizations, the *in out* ports, the n-to-n port connections and the multiple-items dequeuing were mapped with the LNT language at a first transformation definition [129]. However, the experiments show that the obtained specifications explode rapidly especially with highly connected models. Thus, these restrictions are considered to obtain formal models giving realistic state space size that can be explored by verification tools.

Despite the considered restrictions, we provide a scalable solution (see evaluations in section 5.6 of chapter 5) without restricting our main purpose of the mapping of a Ravenscar compliant AADL model (preemptive priority-based scheduling, asynchronous communication, etc.).

4.4 Transformation of a larger AADL subset

The AADL2LNT transformation may be extended, especially at the scheduling mapping level, by supporting more AADL real-time features beyond the Ravenscar restrictions. Firstly, more `thread` dispatching model can be considered, we support timed and hybrid models as defined in the AADL standard:

- Timed `threads`: they are dispatched as the result of an event/event data or it is issued a time interval specified by the `Period` property value since the last dispatch. The `Period` property value represents a time-out value ensuring that a dispatch occurs after a period if no event/event data have arrived.

- Hybrid `threads`: they are dispatched as the result of an event/event data, as well as periodic dispatch at a time interval specified by the `Period` property value.

We remind that `THREAD`s are generically designed for all dispatch protocols, so modifications are only performed in the `SCHEDULER` process. To support the timed and hybrid `threads`, we extend the `SCHEDULER` *Operational part*. In fact, the *non-periodic task checking* and *task state updating* parts are updated to consider new dispatching characteristics. For the timed `threads`, `THREAD` may periodically progress since it must be activated after a time-out interval. Thus, it is updated based on last release and deadline times: τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} = r_{i,(j-1)} + T_i$, $d_{i,j} = d_{i,(j-1)} + T_i$, $s_{i,j+1} = 0$, $e_{i,j+1} = 0$). At the reception of an event/event data, timed `THREAD` is handled as the sporadic `THREAD`s: τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} = t_{i,j}$, $d_{i,j} = r_{i,j} + T_i$, $s_{i,j} = 0$, $e_{i,j} = 0$).

For the hybrid `threads`, `THREAD` must be periodically activated. Thus, it must be updated as a periodic `thread`: τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} = j * T_i$, $d_{i,j} = d_{i,(j-1)} + T_i$, $s_{i,j+1} = 0$, $e_{i,j+1} = 0$). At the reception of an event/event data, hybrid `THREAD` is handled as the sporadic `THREAD`s: τ_i=(C_i, T_i, $j = j + 1$, $r_{i,j} = t_{i,j}$, $d_{i,j} = r_{i,j} + T_i$, $s_{i,j} = 0$, $e_{i,j} = 0$).

In addition to the RM scheduler, we have developed an EDF scheduler based on the preemptive dynamic-priority scheduling. According to the EDF algorithm, the jobs of a task may have different priorities. At any time, the scheduler executes the task with the highest priority, which corresponds to the READY task with the earliest absolute deadline ($d_{i,j}$).

The EDF `SCHEDULER`, graphically represented in Figure 4.9, is based on the `SCHEDULER` skeleton detailed in section 3.4.2. It is almost conserved using the same task array and the same activation orders (based on preemptive scheduling). Thus, the `TASK` and `CONNECTOR` processes need no changes. The required modifications are limited in the *operational part*, mainly, in the *time allocation* operation. Briefly described, during the scheduling, the `SCHEDULER` makes a complete task-loop to select the READY task with the earliest absolute deadline. This operation is achieved using an auxiliary array for the absolute deadlines, as shown in Figure 4.9. The selected task is then assigned to the RUNNING state to be executed through the *task state updating* and *task activation* steps. The `SCHEDULER` repeats regularly its task-loop at each unit of time in order to control all the tasks states since the absolute deadlines of the tasks may change at any time.

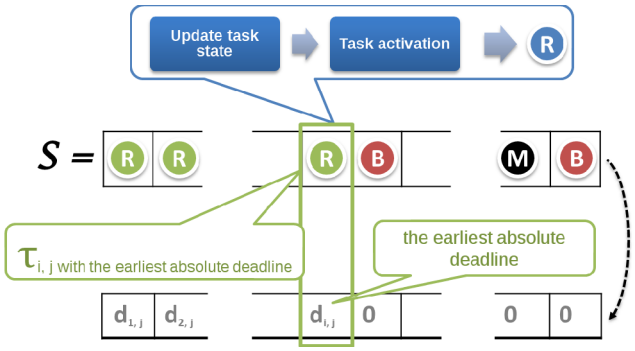


FIGURE 4.9: SCHEDULER algorithm: EDF scheduling

4.5 Behavioral mapping

At this level, the AADL model is completed with behavioral descriptions using the Behavior annex [9]. The annex extends the AADL standard by further syntax and semantics to specify the component behavior. The goal is not the specification of a complicated behavior executed by the `threads`, we specifically support the Behavior annex to enrich the communication mechanism: the annex can be used to handle the inputs and prepare the outputs.

The Behavior annex transformation is directly based on the semantics described in its standard [9], as its behavioral descriptions are not considered in the proposed LNT pattern. In addition, the Behavior annex mapping requires a new abstraction level of the AADL model where the `thread` behavior and the `data` content are considered. The transformation rules of the `data`, `thread` and port connection are redefined, especially, at the communication mapping level.

In this section, we illustrate the Behavior annex mapping with a complete `THREAD` example, given in Listing 4.10, which corresponds to the mapping of `thread compare` in Listing 4.2 from section 4.2.3.

4.5.1 Data mapping

In the case of the data or event data port connection, the ports are typed with the `data` components. In the previous levels, the `data` components were generically mapped using an LNT enumerated type: the consideration of the data contents was useless since they are not handled within components. Now, the inputs and outputs can be used in the Behavior annex description and the port contents may be exploited in the calculations. Thus, the `data` abstraction should be overridden and the `data` type should be considered during the transformation. Currently, we deal with basic *data* types. Each type (Boolean, Integer, Natural, Float, Character and String) from the AADL *Base_Types* package is mapped into

Listing 4.10: LNT THREAD for AADL thread with Behavior annex (example)

```
process THREAD_COMPARE_IMPL[ -- process declaration
   DISPLAY_STATE: Channel_STATE,
   ACTIVATION: LNT_Channel_Dispatch,
   PORT_A: LNT_Channel_Data, PORT_B: LNT_Channel_Data,
   PORT_FAIL: LNT_Channel_Event, PORT_SUCCESS: LNT_Channel_Event]
is -- process body
   var -- initialization part
      STATE : LNT_Type_States, -- thread current state
      A : LNT_Type_Data, B : LNT_Type_Data
   in
      STATE := S0;
      A := DATA_INIT; B := DATA_INIT;
      loop
         select
            select -- execution behavior
              ACTIVATION (T_Dispatch_Preemption)
              []
              ACTIVATION (T_Preemption_Completion)
              []
              ACTIVATION (T_Dispatch_Completion);
              PORT_A (?A); PORT_B (?B); -- inputs
              -- Behavior annex mapping
              if (STATE == S0) then
                 STATE := S0;
                 if (A == B) then PORT_SUCCESS (true)
                 else PORT_FAIL (true) end if
              end if;
              DISPLAY_STATE (Thread_Compare, STATE)
              []
              ACTIVATION (T_Preemption)
            end select ;
            ACTIVATION (T_Complete)
            []
            ACTIVATION (T_Error)
            []
            ACTIVATION (T_Stop)
   end select end loop end var
end process
```

a suitable LNT type. The LNT_Type_Data is no longer a generic type, now it depends on the data component: LNT_Type_Data represents the data type and may be declared as Bool, Nat, Int, Real, Char or String. The boolean type (*Bool*) is used to map events (the *true* value marks a new incoming event). In addition, port types are now differently considered, a set of channels is then defined for the gate synchronization as shown in Table 4.1. For each data, event or event data port, a separate LNT channel is used based on the defined LNT_Type_Data type.

4.5.2 Port and port connection new mapping

At the behavioral mapping, the consideration of the data content requires the extension of the port and port connection transformation rules. In the THREAD

TABLE 4.1: LNT channels for the behavioral mapping

	Channel
data port	`channel LNT_Channel_Data is (LNT_Type_Data) end channel`
event port	`channel LNT_Channel_Event is (Bool) end channel`
event data port	`channel LNT_Channel_Event_Data is (LNT_Type_Data, Bool) end channel`

process, we prepare a set of the required variables and statements grouped in Table 4.2 (for an AADL port named `P`) to be incorporated within its different parts according to the port types and directions. The `THREAD` declaration part is completed with the gate declarations using the suitable channel from Table 4.1.

TABLE 4.2: AADL port transformation rule

	THREAD			
	gate declaration	variables	communication	
			in port	*out port*
data port	`PORT_P: LNT_Channel_Data`	`P: LNT_Type_Data`	`PORT_P (?P)`	`PORT_P (!P)`
event port	`PORT_P: LNT_Channel_Event`	`EVENT_P: Bool`	`PORT_P (?EVENT_P)`	`PORT_P (true)`
event data port	`PORT_P: LNT_Channel_Event_Data`	`P: LNT_Type_Data; EVENT_P: Bool`	`PORT_P (?P,?EVENT_P)`	`PORT_P (!P,true)`

TABLE 4.3: AADL port connections transformation rule

	CONNECTOR	
	periodic threads	*non-periodic threads*
data port	`process LNT_Data_Connector [INPUT: LNT_Channel_Data, OUTPUT: LNT_Channel_Data]`	`process LNT_Data_Connector [INPUT: LNT_Channel_Data, OUTPUT: LNT_Channel_Data]`
event port	`process LNT_Event_Connector [Input: LNT_Channel_Event, Output: LNT_Channel_Event] (Queue_Size: Nat)`	`process LNT_Event_Connector [Input: LNT_Channel_Event, Output: LNT_Channel_Event, Notification: LNT_Channel_Notif] (Queue_Size: Nat)`
event data port	`process LNT_Event_Data_Connector [Input: LNT_Channel_Event_Data, Output: LNT_Channel_Event_Data] (Queue_Size: Nat)`	`process LNT_Event_Data_Connector [Input: LNT_Channel_Event_Data, Output: LNT_Channel_Event_Data, Notification: LNT_Channel_Notif] (Queue_Size: Nat)`

The THREAD behavior part is completed to represent port variables (declaration and initialization) and communications. For illustration, the THREAD of Listing 4.10 includes the mapping of the data in ports (a and b) and the event *out* ports (success and fail).

In addition, considering different port types (data, event or event data) and thread Dispatch_Protocol property (periodic, sporadic, hybrid or timed), we redefine six CONNECTORs as shown in Table 4.3.

4.5.3 Behavior specification mapping

The thread component is being transformed at the scheduling mapping level, as well as all the required constructions (types, channels and CONNECTOR processes) to ensure its port connections. Now, we can complete the THREAD process with the thread behavior specified with the Behavior annex.

The Behavior_Specification clause, included in Listing 4.11, is composed of three sections (*variables*, *states* and *transitions*) used to describe the behavior automaton which is a collection of states with guarded transitions and actions. In the following, we describe how these different sections are respectively mapped within the THREAD *execution behavior* (Listing 4.5).

Listing 4.11: Behavior_Specification

```
annex Behavior_specification {**
variables
   V1: T1; ... Vm: Tm;
states
   S0: initial state;
   Si, Sj: complete state;
   Sl: state;
   Sn: final state;
transitions
        -- dispatch condition
   S0 -[on dispatch dispatch_trigger_condition]-> Si;
        -- execution condition
   Sl -[logical_value_expression]-> Sn {
   -- actions
   };
**};
```

4.5.3.1 Variables and states

The *variables* section allows the declaration of the local variables that can be used within the scope of the Behavior_Specification clause. The set of local variables (V_1 .. V_m) is added in the THREAD process within its *initialization* part, they are declared with the same name and the corresponding type.

The *states* section allows the declaration of the behavior automaton states. A state may be declared as:

- *initial* state (before the initialization of the `thread`);
- *final* state (after the finalization of the `thread`);
- *complete* state;
- A state without qualification is referred to as an intermediate execution state (discrete state of execution behavior).

Note that one state may play the role of a `initial`, `complete` and `final` state at the same time. A behavior automaton starts from an `initial` state and terminates in a `final` state. As shown in Listing 4.12, all states are added in the LNT specification using an LNT enumerated type `LNT_Type_STATES`.

Listing 4.12: LNT `Type_STATES` type for Behavior annex states

```
1 type LNT_Type_STATES is
2     S0, .., SN
3 end type
4
5 channel LNT_Channel_CURRENT_STATE is
6     (LNT_Type_NAMES, LNT_Type_STATES)
7 end channel
```

4.5.3.2 Transitions

The *transitions* section describes the behavior of the state machine. The behavior is created by linking the states using the guarded transitions (S_i-`[guard]`->S_j {`actions`}). The transition specifies the behavior as a change of the current state from a source state S_i to a destination state S_j which can be guarded by the conditions (dispatch or execute).

Different supported transitions are grouped in Table 4.4 with the corresponding LNT statements. In the `THREAD` process, we map explicitly the current state of the behavior automaton using a `STATE` variable of the type `LNT_Type_STATES`. This variable is initialized by the `initial` state, and then, it can change the value (S_0, .., S_n) according to the `THREAD` behavior. For verification ends, a channel `LNT_Channel_CURRENT_STATE` (Listing 4.12) and a gate `DISPLAY_STATE` (line 2 of Listing 4.10) are added to mark the current state of each `THREAD`.

Complete states. A `complete` state, as defined in the Behavior annex standard [9], acts as a suspend/resume state out of which `threads` are dispatched. In consistence with the core AADL semantics, a `thread` in the Awaiting Dispatch state from the *thread scheduling and execution states* automaton (Figure 4.5) may be in one of its `complete` states. A transition out of a `complete` state is initiated by a dispatch once its condition is satisfied. This behavior corresponds to updating of the `STATE` variable, that can take one of the `complete` states after every `T_Dispatch_Completion` activation order. In this way, we assume that the behavior automaton is embedded within the `THREAD` state automaton: in the RUNNING state, the `THREAD` can move into one of the `complete` states.

The `thread` behavior automaton may: suspend itself at a `complete` state; reactivate from the `complete` state repeatedly based on temporal events or the arrival of invocation-events; and involve transitioning to intermediate execution states until attaining a new `complete` state. In the `THREAD` process, different transitions are specified using the LNT conditional statements that are directly placed after the `ACTIVATION` communication. A transition, between two `complete` states S_i and S_j, has the following form: **`if`** `((STATE==`S_i`)` **`and`** `(`*`conditions`*`))` **`then`** *`actions`* **`end if`**, in which, the `STATE` variable is assigned to the new `complete` state (STATE:=S_j;) and then the corresponding actions are included. This `if` statement embeds all transitions into the intermediate execution states, as shown in Table 4.4 (transitions with execute conditions), the intermediate state S_x is hidden in the equivalent `if` statement.

Conditions. A condition determines whether a transition is taken and then the corresponding actions are performed. The transitions can be guarded by the dispatch (`on dispatch`) or execute conditions as shown in Listing 4.11.

The dispatch conditions specify explicitly dispatch trigger conditions out of a `complete` state. The dispatch condition means that the `thread` controls its state when it is dispatched. The condition specifies a dispatch trigger condition, which is a boolean expression describing a logical combination of the triggering events (the arrival of events or event data).

The periodic `threads` are always considered to be unconditionally handled by the dispatch conditions without the `dispatch_trigger_condition`. In the case of a sporadic or timed `thread`, the invocation-events can be used in the `dispatch_trigger_condition` expression, which refines the AADL dispatch model by defining different behaviors for each incoming event in the case of several event port declarations. The `dispatch timeout` condition must only be declared within the timed `threads`, and must be declared in only one transition out of a `complete` state to specify the behavior when the `thread` is dispatched after a period without invocation-event.

In the `THREAD` process, the semantics of `on dispatch` conditions is implicitly

TABLE 4.4: Behavior annex transitions transformation rule

	Behavior annex	**LNT**

dispatch condition	S_i −[**on dispatch**]−> S_j	**if** (STATE == Si) **then**
		STATE := Sj;
		end if;
		DISPLAY_STATE (STATE)

dispatch trigger condition	S_i−[**on dispatch** ip_{e1} **and**..**or** ip_{en}]−>S_j
	S_i−[**on dispatch** ip_{el} **and**..**or** ip_{em}]−>S_k
		if((STATE == Si) **and** (EVENT_IPE1) **and** .. **or** (EVENT_IPEN)) **then**
		STATE := Sj;
		elsif((STATE == Si) **and** (EVENT_IPEL) **and** .. **or** (EVENT_IPEM)) **then**
		STATE := Sk;
		end if

for timed `thread`	S_i−[**on dispatch** timeout]−>S_j
	S_i−[**on dispatch** ip_{en}]−>S_k
	S_i−[**on dispatch** ip_{el}]−>S_l
		if (STATE == Si) **and** (EVENT_IPEN) **then**
		STATE := Sj;
		elsif (STATE == Si) **and** (EVENT_IPEL) **then**
		STATE := Sl;
		elsif (STATE == Si) **then**
		STATE := Sk;
		end if

execute condition	S_i−[**on dispatch**]−> S_x
	S_x−[ip_{d1} **and**..**or** ip_{dn}]−>S_j
	S_x−[ip_{dl} **and**..**or** ip_{dm}]−>S_k
		if (STATE == Si) **and** (IPD1) **and** .. **or** (IPDN)) **then**
		STATE := Sj;
		elsif (STATE == Si) **and** (IPDL) **and** .. **or** (IPDM)) **then**
		STATE := Sk;
		end if

- ip_e: name of event or event data *in* port
- ip_d: name of data or event data *in* port
- $S_{i,j,k}$: *complete* states
- S_x: intermediate execution state

assumed since we include the behavior mapping after the `ACTIVATION` communication, thus at each activation, the `THREAD` controls its state and executes the corresponding actions. Similarly, in the case of a timed `thread`, the `on dispatch timeout` condition is implicitly ensured. As shown in Table 4.4, all possible event or event data ports are checked using the `if` statement before handling the timeout case. If there are no new incoming events, the `thread` is

activated after a period by the SCHEDULER and all ports variables remain at the *false* value. So the THREAD executes the behavior of the last alternative, that of the timeout condition.

The dispatch_trigger_condition is an expression of event or event data port names. Thus, the if condition checks the equivalent EVENT_P variables as included in Table 4.4. These dispatch_trigger_condition expressions are easily mapped in LNT, since the language provides all logical disjunction and conjunction operators (and, and then, or, or else),

The execute condition signifies that the transition is guarded by a logical expression based on the input values. It allows the selection between multiple transitions out of a given state to other states. The logical_value_expression is included in the if condition using the variables of the data or event data ports.

4.5.3.3 Actions

The behavior action blocks are associated with a transition (between {}) and performed when the transition is taken. The actions consist of the control structures (basic actions, conditionals, finite loops, etc) grouped in sequences or sets.

TABLE 4.5: Behavior annex actions transformation rule

	Behavior annex	**LNT**
Assignment action	op := v op := any	OP := *V* OP := **any** LNT_Type_Data
Communication action: output	op! op!(v)	PORT_OP(!OP) PORT_OP(Type_Data(*V*))
Communication action: input	ip? ip?(x)	PORT_IP(?IP) PORT_IP(?*X*)
Conditional execution of alternative actions	**if** (L) actions **elsif** (L) actions **else** actions **end if**	**if** (*L*) **then** *B* **elsif** (*L*) *B* **else** *B* **end if**
Conditional repetition of actions	**while** (L) {actions}	**while** *L* **loop** *B* **end loop**

- op: name of *out* port
- ip: name of *in* port
- v: value
- x: variable
- L: logical value expression
- V: value of type LNT_Type_Data
- B: behavior statement

In this thesis, we support sequences of actions that are executed in a given order (contrary to action sets that can be executed in any order). These actions are transformed one by one (separated by a semicolon) within the `if` transition statement to be sequentially executed when the `if` condition is satisfied. The supported actions are transformed in the LNT language using the suitable statements as shown in Table 4.5. As illustrated in Listing 4.10 (lines 25..35), the `THREAD_COMPARE_IMPL` represents a complete `Behavior_specification` clause mapping.

4.6 Conclusion

In this chapter, we described the AADL2LNT transformation, taken an AADL model as a source model, to obtain an LNT specification. We detailed firstly how the proposed LNT pattern (chapter 3) is used to transform the AADL model execution, port connections and system hierarchy. Then, we proposed an LNT mapping for the Behavior annex. The model transformation is achieved through its different levels (scheduling, communication and behavioral mapping) to support different kinds of AADL models: models with independent `threads`, models with connected `threads` and models completed with the Behavior annex.

The produced LNT specification is ready for the verification phase with the CADP toolbox, developed in the next chapter 5, in which we present the implementation and validation of our contributions.

5

Implementations and validation

Contents

5.1 Introduction 100
5.2 Ocarina architecture 100
5.3 Ocarina extensions 102
5.3.1 Behavior annex parsing 102
5.3.2 LNT code generation 103
5.3.3 SVL script generation 105
5.4 Tool-chain 106
5.5 Case studies 108
5.5.1 AADL modeling 108
5.5.2 LNT code generation 115
5.5.3 Formal verification 117
5.5.4 Analysis results 123
5.5.5 Manual verification 124
5.6 Scalability 126
5.6.1 Test suite 126
5.6.2 Results and interpretations 126
5.7 Conclusion 129

5.1 Introduction

This chapter is dedicated to detail the implementation and validation of the proposed theoretical contributions and research activities performed in the last chapters (3 and 4), in the context of the formal verification of the AADL language. We describe firstly how the model transformation and formal verification phases are automated by means of our Ocarina extension. Then, we detail our experiments to validate the proposed contributions.

This chapter is organized as follows: section 5.2 represents the Ocarina architecture; our implementations are detailed in section 5.3; The resulting tool-chain is represented in section 5.4; Experiments on three case studies are detailed in section 5.5; A scalability study is discussed in section 5.6.

5.2 Ocarina architecture

Ocarina is a model processor for the AADL language. Its compiler is designed with the Ada language with a modular architecture, as depicted in Figure 5.1. As mentioned before (section 2.3.3.1 of chapter 2), analyses and generations are handled using ASTs (Abstract Syntax Tree) which are the internal representation of models (AADL, annexes, programming languages, etc.). Based on the language grammar rules, the model is decomposed into a set nodes hierarchically connected to create the corresponding syntax tree. These ASTs are manipulated in three distinguished parts:

- Central library: this part consists of a set of routines allowing the AST construction and manipulations of the AADL models (*Generic routines*) and other languages such as Ada, C and Petri nets (*Specific routines*). These routines are used to manipulate files and strings and to facilitate the access and update of the nodes of the ASTs (functions for node builder and finder).
- Frontends: this part ensures lexical, syntactic and semantic analyses of AADL models. From the syntactic phase, the AADL AST is created using routines of the central library, which is then semantically checked according to the described semantics in the AADL standard [10]. At the instantiation phase, the tree is decorated with information about properties and relations between components, to obtain a complete AADL AST. This tree represents a hierarchical view of the AADL instance model, whose root is the top-level `system` component of the model describing the entire application topology (access for all subcomponents of the AADL model). In addition to AADL, other secondary frontend modules are developed in order to support the

syntax of some annexes (ARINC653, EMV2 and REAL). These annexes are handled with separate lexical, syntactic and semantic analyzers according to their grammars to produce specific ASTs.

- Backends: this part provides different automatic code generations. Using central library routines, the instantiated frontend ASTs are expanded and then used for model and code generation. The expansion phase aims to simplify or add some constructions of the ASTs (such as adding special port for dispatching in the case of `thread` components). During this phase, high-level errors can be detected like unsupported dispatch protocol or missed connections. The code generation phase is modularly designed for maintainability/extensibility and optimization ends: the AADL model considered as a pivot model from which a set of model can be separately generated using specific modules. Each code generation has its specific modules that implement transformation rules to construct an intermediate tree of the target language (*Tree conversion*), which will be scanned (*Code printing*) to finally generate source code files.

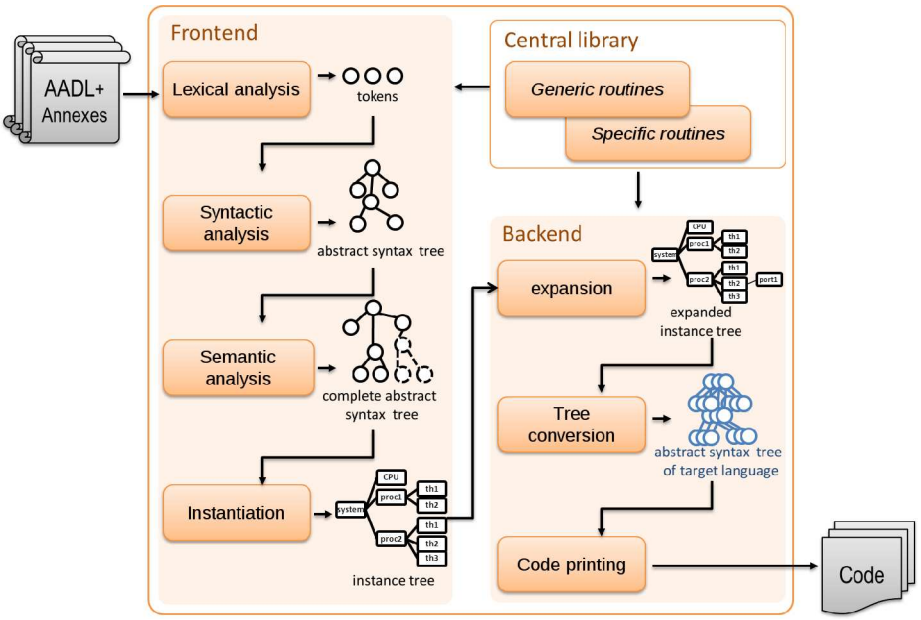


FIGURE 5.1: Ocarina compiler architecture

In this thesis, we contribute to the Ocarina project by adding extensions to implement the AADL2LNT transformation, in order to draw a gateway to the CADP toolbox. We extend the compiler with a set of modules in its different parts. Since we support the Behavior annex at the behavioral mapping level, this annex is considered as input and should be analyzed in the frontend part. The central library is enriched with the required routines for both Behavior annex and

LNT ASTs, based respectively on the Behavior annex [9] and LNT [48] grammars. Finally, the AADL2LNT transformation rules are applied in the backend part to produce the LNT specification.

5.3 Ocarina extensions

In this section, we describe our implementations within the Ocarina compiler to analyze the Behavior annex specifications and produce the final LNT specification for a given AADL model.

5.3.1 Behavior annex parsing

From the `Behavior` grammar described in the Behavior annex standard [9], we add routines for the Behavior annex syntax tree (BA AST) construction, whose root corresponds to the `behavior_annex` rule in the Listing 5.1. Then, the tree is composed of lists for *variables*, *states* and *transitions* sections.

Listing 5.1: `Behavior_annex` grammar rule [9]

```
1 behavior_annex ::=
2 [ variables { behavior_variable } + ]
3 [ states { behavior_state } + ]
4 [ transitions { behavior_transition } + ]
```

The frontend handles `Behavior_specification` clauses following Ocarina analysis phases as shown in Figure 5.1:

- Behavior lexical analysis: the specification is parsed to distinct tokens and identify the Behavior annex keywords.
- Behavior syntactic analysis: the set of tokens is analyzed according to the Behavior annex grammar. For each `Behavior_specification`, a BA AST is created and hierarchically constructed and then attached to its corresponding AADL component (`thread`, `device`, `subprogram`, etc.). Thus, the AADL AST is completed with BA ASTs to obtain a final AADL-BA AST.
- Behavior semantic analysis: the BA ASTs are analyzed after the semantic analysis of the AADL model. Then, each BA AST is visited to check semantics rules detailed in the Behavior annex standard [9] as follows:
  - Naming rules such as the unicity of identifiers in the scope of the `Behavior_specification` clause;

- Internal semantic rules to validate the specification descriptions such as transitions from `final` states are not allowed;
- External semantic rules to validate dependencies between the `Behavior_specification` clauses and the core AADL language such as local variables must be explicitly typed with a declared `data` component, and the compatibility rules with the `Dispatch_Protocol` property values (e.g. the `dispatch_trigger_condition` can not be declared within the behavior of a periodic `thread`)

Note that, throughout these phases, if a problem is detected, an error message will be displayed to explain the problem and the compilation will be interrupted.

The Behavior annex is defined to refine the implicit behavior specifications that are specified by the AADL core language (e.g. extending the execution and dispatch semantics of `threads`). Thus, the Behavior annex descriptions depend on the AADL component which requires the respect of a set of semantics dependency rules (external semantic rules). For this reason, the semantic analysis is applied on the complete AADL-BA AST, so that, these rules are easily verified on both AADL and BA ASTs.

To verify the Behavior annex parser, we implemented an Behavior annex backend generator, which gathers a set of functions regenerating `Behavior_specification` code from an BA AST. A test suite is also defined using the Behavior annex standard examples to check different syntactic and semantic rules.

5.3.2 LNT code generation

In this section, we describe the LNT generator implemented within the Ocarina compiler in order to produce an LNT specification compliant with the CADP toolbox. Based on the LNT grammar described in its manual [9], a set of LNT routines are firstly added in the central library for the LNT AST construction. Then, the defined AADL2LNT transformation with its different mapping levels is implemented within the Ocarina backend.

We remind that only *instantiable* `systems` are accepted for the AADL2LNT transformation, which means that the model is successfully analyzed (lexically, syntactically and semantically) and all the components are bound to each other, as when all `threads` are bound to the `processor`. In addition, we note that a set of standard properties should be used as described in section 4.2.3 (chapter 4) and the `data` components should be typed with the standard `Base_Types` package. Else, an error or warning message is displayed. Firstly, the AADL AST is created and instantiated in the frontend and becomes ready for the manipulations with the LNT generator. Then, depending on the AADL model kind, the appropriate transformation level is applied in the Backend. Note that non-supported elements

(e.g. `bus`, `shared access`) are automatically ignored. And when using the Behavior annex, the transformation is applied on the complete AADL-BA AST.

5.3.2.1 Model transformation

Following the Ocarina backend modular architecture, we implement the AADL2LNT transformation in two phases. The transformation rules are directly applied on the AADL AST to simultaneously build the LNT AST (*AADL-LNT tree conversion*). Then, the LNT AST is scanned by the LNT code generator (*LNT printing*) in order to produce source code files (`*.lnt`).

To obtain an LNT modular specification, we generate a set of LNT modules, which corresponds to the construction of a set of LNT ASTs. Two main modules, *Types* and *Main*, are always generated for all LNT specifications. Then, according to the mapping level, a set of LNT modules is added as follows:

- *Threads* module consists of a set of the `THREAD` and `DEVICE` declarations, whose generation depends mainly on the port declarations in the `thread` or `device` components.
- *Processor* module contains the `SCHEDULER` process with a set of LNT function definitions required for the `thread` scheduling and execution. The `SCHEDULER` generation needs the extraction of a set of `thread` information (task model) to be included in its *initialization part*, such as the list of `thread` instances and the set of values of each `thread` properties.
- *Port_Connections* and *Port_Connections_BA* are generic modules included within the compiler resources. They are copied in the work directory as required for the communication or behavioral mapping. Each module consists of a set of `CONNECTOR` declarations that will be instantiated for each port connection.

All declared processes in different modules should be instantiated and synchronized to form the whole system, which is based on the generation of two modules:

- *Types* module consists of a set of LNT types, functions and channels required for different temporal calculations and process synchronizations: `THREAD-CONNECTOR`, `CONNECTOR-SCHEDULER` and `THREAD-SCHEDULER`. This module depends largely on the mapping level: some generic definitions at the communication mapping level, are replaced at the behavioral mapping level.

- *Main* module imports all the other modules to specify the MAIN process based on different definitions (types, channels, processes THREAD, SCHEDULER, etc.). This module is the entry-point of the whole LNT specification, which will be later used in the verification phase.

During the AADL2LNT generation, a set of naming rules are applied, for traceability ends. At the THREAD level, all AADL port identifiers are conserved and prefixed by "PORT_". Also, all port content variables, Behavior annex variables and states identifiers are conserved in the behavioral mapping. Similarly, at the MAIN level, the generation conserves AADL component implementation identifiers as follows: the thread identifier is prefixed by "THREAD_"; the device identifier is prefixed by "DEVICE_". For each AADL port connection, two identifiers are prepared based on the connection name which is prefixed with "SEND_" and "RECEIVE_" to represent two gates (see in section 4.3.3.1 of chapter 4).

5.3.3 SVL script generation

In addition to LNT modules, we note that a second input is provided for analyzing with the CADP toolbox. A script file (demo.svl) containing a set of operations with SVL (Script Verification Language) [67] language is also generated to automate the verification phase. In the following, we present the SVL language, then we propose an SVL script for the AADL model verification.

5.3.3.1 SVL language

The CADP tools are traditionally invoked (with many options) from the command line. Using the SVL language, the use of several tools is simplified within a single script to orchestrate verification phases. SVL is both a high-level language for the description of complex verification scenarios and a compiler dedicated to this language.

Briefly, the SVL language offers a way to describe verification phases, under the form of sequences of statements based on expressions. The SVL statements are either: assignments (to produce a file containing a representation of an expression); behavior comparisons; temporal logic verifications (verifying a temporal logic formula in an expression); deadlock/livelock checks; property definitions; or property checks. The expressions represent the state spaces and the operators for the composition, generation, abstraction, etc.

5.3.3.2 SVL script for AADL model

In this thesis, SVL is used to describe the AADL model verification based on the model-checking of a set of behavioral properties. These properties are spec-

ified using the SVL property statement embedding a temporal logic verification statement, that contains the temporal logic formula.

This file is directly created for each AADL system (a model-text transformation without an SVL AST). The script skeleton is included in Listing 5.2, having two main parts: compilation of the LNT specification (assignment statement) and model-checking of a set of generic properties (property definitions and checks).

A property definition consists of a name, optional parameters, comments and possibly an expected result (`excepted TRUE/FALSE`) that must be attached to each embedded verification statement (temporal logic verifications, deadlock-/livelock checks, etc.).

In the case of a verification statement, a property is defined to evaluate a *formula* (temporal logic formula) on a B behavior, with the T tool using the M method, as shown in line 21 of Listing 5.2. This structure is used to specify a set of properties to evaluate the LNT specification with the CADP model-checkers (see section 5.5.3.2 of chapter 5).

Listing 5.2: SVL script skeleton for AADL models

```
1  % DEFAULT_MCL_LIBRARIES="standard.mcl"
2  -- Compilation of LNT specification
3  "Main.bcg"= generation of "Main.lnt";
4  -- Verification of a set of generic properties
5  -- deadlock/livelock checking statements
6  property DEADLOCK_FREEDOM is
7     deadlock of "Main.bcg";
8     expected FALSE;
9  end property;
10 property LIVELOCK is
11    livelock of "Main.bcg";
12    expected FALSE;
13 end property;
14 -- property definition
15 property property_name (parameters) "comment" is
16    B |= [using M] [with T] formula;
17    expected TRUE;
18 end property;
19 -- model-checking
20 check property_name (parameter_1);
21 check property_name (parameter_i);
22 ...
23 check property_name (parameter_n);
```

5.4 Tool-chain

The implemented AADL2LNT transformation allows the definition of a tool-chain based on Ocarina for architectural modeling and CADP for formal verification,

as depicted in Figure 5.2. Note that the AADL2LNT extension has been validated and integrated in the official Ocarina GitHub repository to be available for academic and industrial users [1]. In addition, the CADP toolbox can be downloaded with both academic and commercial licenses from its official web site [2].

As input, the tool-chain takes an AADL model that may contain Behavior annex specifications. This model is handled by different frontend analysis phases. Then it is transformed in the backend modules. As result, the following outputs are generated: the LNT specification composed of 5 LNT modules in separate files (*Threads*, *Processor*, *Ports*, *Types* and *Main*); and the script file (*demo.svl*).

The provided tool-chain ensures an automatic and transparent AADL model transformation and formal verification. The transformation is proceeded using the Ocarina command line, then, the generated SVL script is simply invoked to begin the verification with the CADP toolbox.

In addition, the produced LNT specification can be considered as a good base of a manual verification. In this case, designers may complete different LNT modules and verify new properties (to be added in *demo.svl*) for a specific case study (see section 5.5.5 of chapter 5).

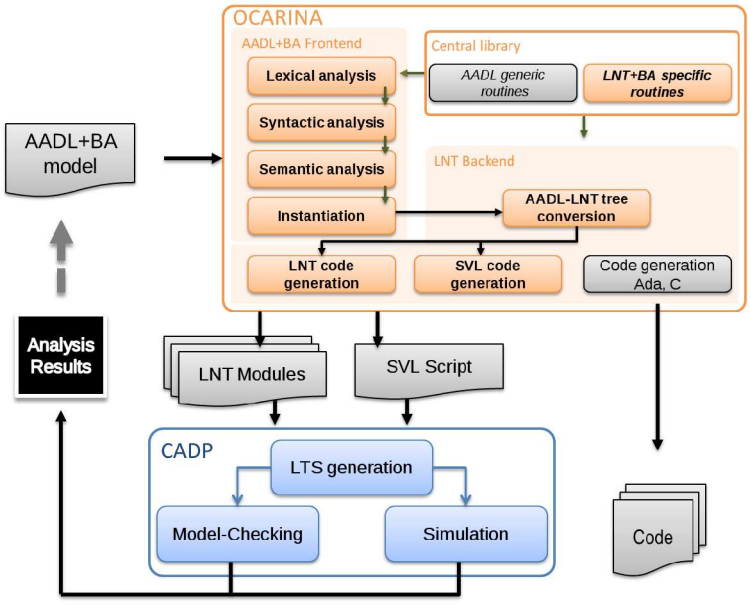


FIGURE 5.2: Ocarina-CADP tool-chain

[1] Ocarina GitHub: https://github.com/OpenAADL/ocarina
[2] CADP web site: http://cadp.inria.fr/

5.5 Case studies

The proposed AADL2LNT transformation has been tested with various real-time systems. In this section, we present experiments performed on three case studies (flight control system, line follower robot and pacemaker device) through different development phases: modeling, model transformation and formal verification. In addition, we discuss the usability of the analysis results and the possibility of a manual verification.

5.5.1 AADL modeling

In the design phase, we model the considered real-time case studies with the AADL language with its Behavior annex. As shown in Table 5.1, the *FCS* (flight control system) case study allows the illustration of both scheduling and communication mapping levels. While, the *Robot* (line follower robot) and *Pacemaker* systems cover different transformation levels including the behavioral mapping.

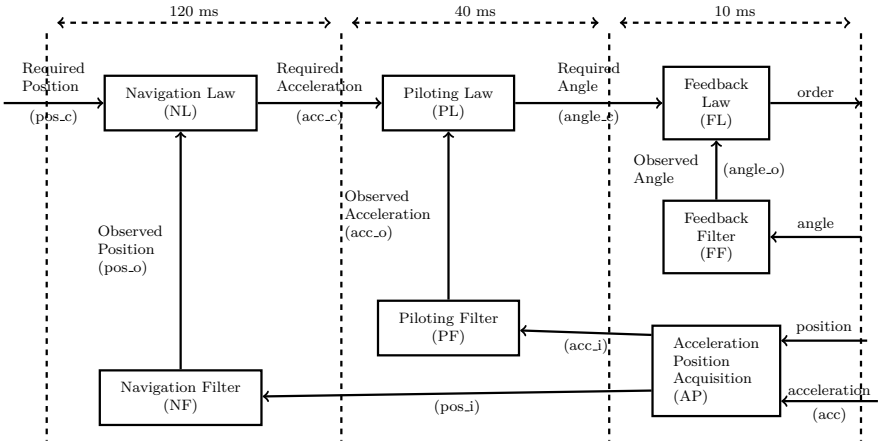
TABLE 5.1: Case studies

Mapping level		FCS	Robot	Pacemaker
Scheduling	`thread`	periodic	sporadic + periodic	sporadic + timed
	`Processor`	RM + EDF	RM	RM
Communication	port connection	data	event data	event
Behavioral	states	-	intermediate + `complete`	`complete`
	conditions	-	dispatch + execute	dispatch + timeout

5.5.1.1 Flight control system

The flight control system (*FCS*) is a safety-critical avionics system for aircraft controlling. This system controls the altitude, trajectory and speed of an airplane. We consider a simplified version, composed of 7 periodic tasks which collaborate in order to send a feedback to the control display system. As shown in Figure 5.3, the fastest subsystem executes at 10 ms, it acquires the state of the system (angles, position, acceleration) and computes the feedback law of the system. The order is then sent to the flight control surfaces. The intermediate subsystem is the piloting loop, it executes at 40 ms and determines the acceleration to apply. The slowest subsystem is the navigation loop, it executes at 120 ms and determines the position to reach. The required position of the airplane is acquired at the slow rate.

FIGURE 5.3: Flight control system



The *FCS* AADL model, depicted in Figure 5.4, composed of 7 periodic `threads` (`FL`, `PL`, `PF`, `NL`, `NF`, `AP` and `FF`) grouped in one `process` (`process_FCS`) bound to one `processor`. The model contains a set of inter-`thread` port connections (declared at the `process` level) and a set of `thread-device` port connections with 4 `devices` (declared at the `system` level).

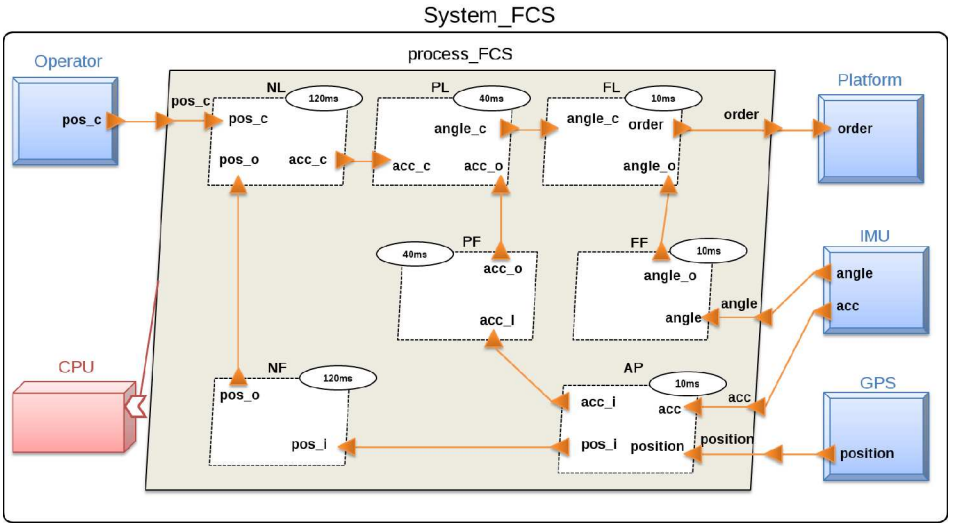


FIGURE 5.4: *FCS* AADL model

5.5.1.2 Line follower robot

The *Robot* system is a machine that follows a black line on a white area. It uses sensors to detect the line and control units to make movement decisions and

command motors (wheels). The sensors control regularly the follow-up of the black line and send information to control units. The motors are commanded (turn on/off) only if the *Robot* loses the line, which is obviously a non-periodic action. So the motor tasks would be modeled using sporadic `threads`.

The *Robot* AADL model, depicted in Figure 5.5, is composed of right and left similar sides. Each *Robot* side is represented with 3 `threads`: a periodic `Sensor thread` for sensing; a periodic `Control thread` for controlling; and a sporadic `Motor thread` for turning off/on the motor. These `threads` are in communication (*sensor-control*/*control-motor*) through event data port connections. At the `system` level, all the considered `threads` are declared in `process` components that are bound to one `processor`. The `system` may be completed with other hardware components to represent `devices` for the sensors and motors, yet the current software version is sufficient for our purposes.

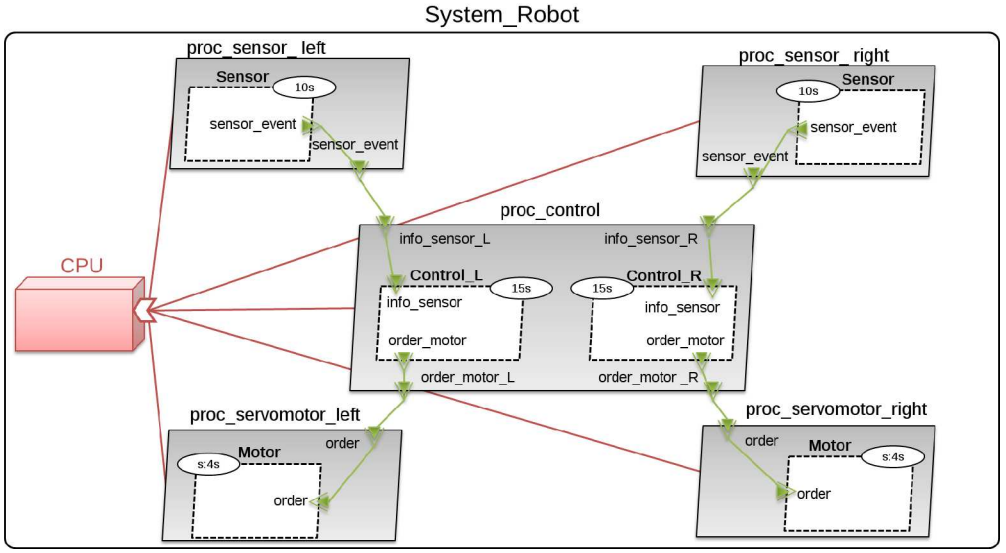


FIGURE 5.5: *Robot* AADL model

The *Robot* AADL model is extended with a set of Behavior annex specifications describing the control behavior to keep following of the black line. This behavior concerns mainly the `Control thread`. Each period, the `Control thread` receives information (with the `info_sensor` port) from the `Sensor thread` and sends, when needed, an order (with the `order_motor` port) to the `Motor thread`. If the sensor detects a deviation, the corresponding motor should be stopped to make the *Robot* turn and find again the black line. If the sensor detects the line again, then the control unit re-activates the motor to maintain the advancement of the *Robot*.

The *Robot* `threads` communicate through event data port typed with Boolean as follows: in the *sensor-control* connection, the *true* value means that the *Robot*

is on the line and the *false* value signifies that the *Robot* goes out of the line; and in the *control-motor* connection, the *true* value represents an order to the motor to turn-on and contrary the *false* value is an order to turn-off.

The described behavior is specified within the `Behavior_Specification` of the `Control thread`, as included in Listing 5.3. This specification is mainly composed of two `complete` states `s_online` and `s_outline` to model the *Robot* states, respectively, on and out of the black line. Initially, the *Robot* is considered in the `s_online` state (declared as `initial`). Every dispatch, the *Robot* controls its state. Then, according to the `info_sensor` port variable value, it may change the state. Being in one of its `complete` states (`s_online` or `s_outline`), the *Robot* may stay in the same state or it may move to another `complete` state, which is designed using the execute conditions (via intermediate execution states `s1` and `s2`). While the `info_sensor` port value is not changing, the *Robot* keeps the same state (lines 9 and 14 in Listing 5.3). If it is changed, the *Robot* moves into another state (lines 10 and 15 in Listing 5.3). In this case, the `order_motor` port variable is updated and an event is sent to the corresponding `Motor thread`.

LISTING 5.3: *Robot*: `Control Behavior_Specification`

```
1  thread implementation Control.Impl
2  annex Behavior_specification {**
3    states
4      s_online : initial complete final state;
5      s1, s2 : state;
6      s_outline : complete state;
7    transitions
8      s_online -[on dispatch]-> s1;
9      s1 -[info_sensor ]-> s_online;
10     s1 -[not info_sensor]-> s_outline {
11       -- order for motor to turn off
12       comm_motor !(false)};
13     s_outline -[on dispatch]-> s2;
14     s2 -[not info_sensor]-> s_outline;
15     s2 -[info_sensor ]-> s_online {
16       -- order for motor to turn on
17       comm_motor ! (true)};
18 **};
19 end Control.Impl;
```

5.5.1.3 Pacemaker

The *Pacemaker* [106, 98] is a real-world case study. It is a medical device inserted in the body of a patient, to regulate his/her heart beating with electrical impulses, as shown in Figure 5.6. The *Pacemaker* is used in the case of heart rhythm problems (inability to maintain a normal heart rate).

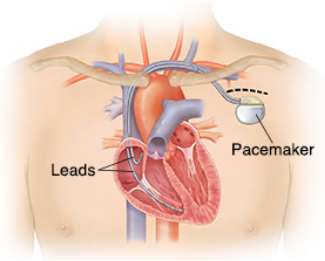


FIGURE 5.6: A pacemaker implantation (taken from [47])

Depending on the heart problem, the *Pacemaker* provides different modes that perform different kinds of therapeutic behavior. Modern pacemakers are designed to pace only when necessary. The mode determines the type of pacing that must be implemented. In this thesis, we consider the VVI (Ventricle-Ventricle-Inhibited) mode which is a single-chamber pacing mode: only ventricular chamber is sensed and paced; and the pace is inhibited when the heart is beating fast enough on its own. In this mode, two parameters are required: the LRL (Lower Rate Limit) which is the minimum rate that must be guaranteed by the pacemaker; and the VRP (Ventricular Refractory Period) which is a refractory period after stimulation (pace or beat) in which senses are ignored.

The *Pacemaker* AADL model [3] is depicted in Figure 5.7. Structurally speaking, it consists of two main parts: the device controller monitor (`DCM system`), which embeds the software implementing the therapeutic behavior to monitor; and a pulse generator (`PG device`) with tow electrical leads for ventricle and atrium heart chambers. The `DCM system` consists of a `Pacemaker_SW process` bound to a `processor` component. Since we deal with the VVI mode, atrium heart chambers are not designed. Thus, `PG device` represents only ventricle lead. The `Pacemaker system` is included in Listing 5.4. The `DCM system` and `PG device` are in communication through event port connections to sense the heart beating (`sense` *in* port) and to send a pacing order (`pace` *out* port).

LISTING 5.4: *Pacemaker*: `Pacemaker system implementation`

```
1 system implementation Pacemaker.Impl
2 subcomponents
3   PG : device Pulse_Generator.impl;
4   DCM : system Device_Controller_Monitor.Impl;
5 connections
6   connection_00 : port PG.ventricle_sense -> DCM.ventricle_sense;
7   connection_01 : port DCM.ventricle_pulse -> PG.ventricle_pulse;
8   connection_02 : port DCM.normal_Beat -> PG.normal_beat;
9 end Pacemaker.Impl;
```

[3] This model is inspired from the AADL pacemaker model published by Ellidiss technologies: http://www.ellidiss.fr/public/wiki/attachment/wiki/AADL/Pacemaker.aadl#L207

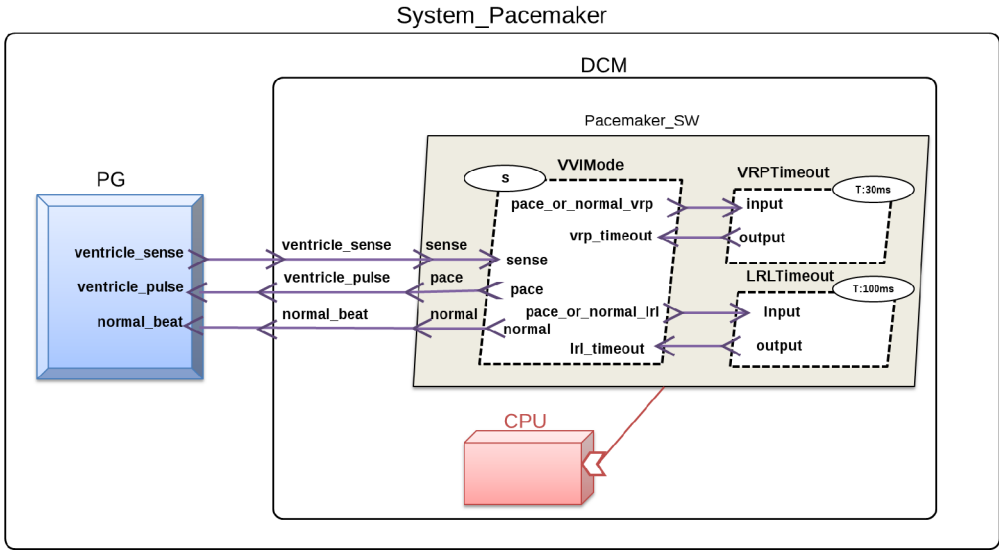


FIGURE 5.7: *Pacemaker* AADL model

The VVI therapeutic behavior is accomplished using 3 `threads` (within the `Pacemaker_SW process`) interconnected and enriched with Behavior annex specifications. The VVI pacing is not a periodic action, it depends on sensing and VRP/LRL periods. Thus, we use a sporadic `thread` to present the VVI mode (`thread VVIMode`). In addition, VRP and LRL periods are considered as two timers that should be reinitialized after each pacing or beating event. So they are consequently designed as timed `threads` (`Dual_Or_Timer_VRP` and `Dual_Or_Timer_LRL`).

The defined `thread` components are then completed with Behavior annex descriptions. In Listings 5.5 and 5.6, we include the `Behavior_Specification` of the `VVIMode` and `Dual_Or_Timer_VRP threads`. The `VVIMode` specification consists of one `complete` state and 3 transitions to describe different VVI mode behaviors as follows: when the heart has no beat during an LRL period, `VVIMode` causes a pace (an event on the `pace` port) (transition line 21 in Listing 5.5); if the sense (event on the `sense` port) comes too soon after a beat (during the VRL period), it will be ignored (transitions lines 9 and 13 in Listing 5.5); and when the heart is beating regularly, `VVIMode` detects a normal beating rhythm (an event on the `normal` port)(transition line 13 in Listing 5.5). The `Dual_Or_Timer_VRP` specification is mainly based on a timeout transition (line 7 in Listing 5.6), in which an event is sent to the `VVIMode thread` to mark the end of the VRP period (received on the `vrp_timeout` event port).

Listing 5.5: *Pacemaker*: VVIMode Behavior_Specification

```
-- vrp data subcomponent saves the VRP state,
-- it takes the value 1 during the VRP and 0 otherwise
thread implementation VVIMode.Impl
annex Behavior_Specification {**
  states
    s1 : initial complete final state;
  transitions
    -- out of VRP
    s1 -[ on dispatch vrp_timeout ]-> s1 {
       vrp := 0
    };
    -- normal heart rate rhythm is detected
    s1 -[ on dispatch sense ]-> s1 {
       if (vrp = 0)
          normal!; -- sent an event in normal port
          pace_or_normal_vrp !;
          pace_or_normal_lrl !;
          vrp := 1
       end if
    };
    -- out of LRL, a pace is sent
    s1 -[ on dispatch lrl_timeout ]-> s1 {
       pace!;
       pace_or_normal_vrp !;
       pace_or_normal_lrl !;
       vrp := 1
    };
**};
end VVIMode.Impl;
```

Listing 5.6: *Pacemaker*: Dual_Or_Timer_VRP Behavior_Specification

```
thread implementation Dual_Or_Timer_VRP.impl
annex Behavior_Specification {**
  states
    s1 : initial complete final state;
  transitions
    s1 -[on dispatch input ]-> s1;
    s1 -[on dispatch timout ]-> s1 {
      output!
    };
**};
end  Dual_Or_Timer_VRP.impl;
```

5.5.2 LNT code generation

The model transformation of our case studies are performed by the AADL2LNT Ocarina extension. Table 5.2 sums up the transformation metrics. We obtain three LNT specifications (with their SVL scripts) as follows:

- *FCS* LNT specification counts 905 code lines, 100% generated by our extension. It contains 24 processes, instantiated and synchronized at the `MAIN` process: the `thread-device` port connections are similarly mapped as the inter-`thread` port connections.
- *Robot* LNT specification counts 648 code lines, generated by our extension with some manual additions. The `MAIN` process consists of 11 processes. Different `Behavior_specifications` are mapped within the corresponding `THREAD`s. In Listing 5.7, we include extracts from the `THREAD_CONTROL_IMPL` process equivalent to the `Control thread`, we note that:
  - Since `Bool` is reserved to represent events, the `data` Boolean type is mapped by an LNT range type (`range (0..1) of Nat`), where 0 represents the *false* value and 1 represents the *true* value.
  - Different intermediate states are hidden and their transitions are embedded within the `if` transition statement. Starting from a specific `complete` state, all possible alternatives (transitions) are added in the `if` statement to reach a new `complete` state. For example, from the `S_ONLINE` state, there are two alternatives to reach the `S_ONLINE` and `S_OUTLINE` states (lines 4 and 8 in Listing 5.7).
- *Pacemaker* LNT specification counts 713 code lines, generated by our extension with some manual additions. The `MAIN` process instantiates 9 processes. Different `Behavior_specifications` are mapped within the corresponding `THREAD`s. We include in Listing 5.8 and 5.9 extracts from the `THREAD_VVIMode_IMPL` and `THREAD_DUAL_OR_TIMER_VRP_IMPL` processes equivalent to the `VVIMode` and `Dual_Or_Timer_VRP` threads. We note that:
  - `EVENT_*` variables are used for the event port mapping.
  - The `THREAD_VVIMode_IMPL` process includes all transitions from the AADL initial model (3 transitions of Listing 5.5) since they are out of `complete` states.
  - The `THREAD_DUAL_OR_TIMER_VRP_IMPL` contains the `if` transition statement of an `on dispatch timeout` condition which is implicitly ensured after checking the `EVENT_INPUT` event port (lines 6 .. 12 in Listing 5.9).

TABLE 5.2: Case studies transformation and verification metrics

FCS		Robot		Pacemaker	
AADL	LNT	AADL	LNT	AADL	LNT
source code lines		source code lines		source code lines	
188	905	138	648	108	713
transformation		transformation		transformation	
7 threads	7 THREADs	6 threads	6 THREADs	3 threads	3 THREADs
4 devices	4 DEVICEs	-	-	1 device	1 DEVICE
1 process	-	5 processes	-	1 process	-
17 data connections	12 Data_CONNECTORs	10 event data connections	4 Event_Data_CONNECTORs	13 event connections	5 Event_CONNECTORs
1 processor	1 SCHEDULER	1 processor	1 SCHEDULER	1 processor	1 SCHEDULER
1 system	1 MAIN	1 system	1 MAIN	1 system	1 MAIN
LTS		LTS		LTS	
-	2439 states 14570 transitions	-	918 states 953 transitions	-	5688 states 5691 transitions

The LNT specification complexity depends on the `threads` and port connections numbers. From models with independent `threads`, we obtain simple LNT models without `CONNECTOR` synchronizations. While with highly-connected models, the processes number increases significantly.

LISTING 5.7: *Robot*: extract of `THREAD_CONTROL` process

```
process THREAD_CONTROL_IPML [
...
   ACTIVATION (T_Dispatch_Completion);
   PORT_INFO_SENSOR (?INFO_SENSOR,?EVENT_INFO_SENSOR);
   if (STATE == S_ONLINE) and (INFO_SENSOR == LNT_Type_Data (0)) then
      STATE := S_OUTLINE;
      COMM_SERVO := LNT_Type_Data (0)
   elsif (STATE == S_ONLINE) and (INFO_SENSOR == LNT_Type_Data (1)) then
      STATE := S_ONLINE -- no actions
   elsif (STATE == S_OUTLINE) and (INFO_SENSOR == LNT_Type_Data (1)) then
      STATE := S_ONLINE;
      COMM_SERVO := LNT_Type_Data (1)
   elsif (STATE == S_OUTLINE) and (INFO_SENSOR == LNT_Type_Data (0)) then
      STATE := S_OUTLINE
   end if;
   PORT_COMM_SERVO (COMM_SERVO, true);
   DISPLAY_STATE (Thread_Control, STATE)
...
end process
```

The provided Ocarina extension automates an important part of the defined AADL2LNT transformation and eliminates its complexity [4]. Especially, in the `SCHEDULER` mapping which is less generic (extraction of the task model) compared with others `THREAD`, `DEVICE` and `CONNECTOR` process generation. For example, the *Robot Processor* module counts 273 lines (nearly 50% of code). Another difficulty resides at the hierarchical mapping level. The mapping of the `MAIN` process seems tricky since we deal with a lot of process instances and gates. Without forgetting that all port connections, following component hierarchical containment (`process` and `system` levels) should be abstracted at the `MAIN`

[4] Note that manual additions are only at the behavioral mapping level

level. For instance, to map the 7 `threads` of the *FCS* case study, we should synchronize 24 process instances on 31 different gates. In addition, the mapping of *Types* module depends on the mapping levels and may count up to 20 definitions between types, functions and channels in the case of the behavioral mapping.

LISTING 5.8: *Pacemaker*: extract of `THREAD_VVIMode` process

```
process THREAD_VVIMODE_IMPL [ ...
   ACTIVATION (T_Dispatch_Completion);
   PORT_SENSE (?EVENT_SENSE);
   PORT_LRL_TIMEOUT (?EVENT_LRL_TIMEOUT);
   PORT_VRP_TIMEOUT (?EVENT_VRP_TIMEOUT);
   -- on disptch VRP_TIMEOUT
   if (STATE == S1) and (EVENT_VRP_TIMEOUT) then STATE := S1; VRP := 0
   -- on disptch lrl_timeout
   elsif (STATE == S1) and (EVENT_LRL_TIMEOUT) then
      STATE := S1;
      PORT_PACE (true);
      PORT_PACE_OR_NORMAL_LRL (true);
      PORT_PACE_OR_NORMAL_VRP (true);
      VRP := 1
   -- on disptch sense
   elsif (STATE == S1)  and (EVENT_SENSE) then
      STATE := S1;
      if (VRP == 0) then
         PORT_NORMAL (true);
         PORT_PACE_OR_NORMAL_LRL (true);
         PORT_PACE_OR_NORMAL_VRP (true);
         VRP := 1
      end if
   end if ...
end process
```

LISTING 5.9: *Pacemaker*: extract of `THREAD_DUAL_OR_TIMER_VRP` process

```
process THREAD_DUAL_OR_TIMER_VRP_IMPL [
...
   ACTIVATION (T_Dispatch_Completion);
   PORT_INPUT (?EVENT_INPUT);
   -- on dispatch event_port
   if (STATE == S1) and (EVENT_INPUT) then
      STATE := S1
   -- on dispatch timout
   elsif (STATE == S1) then
      STATE := S1;
      PORT_OUTPUT (true)
   end if;
   DISPLAY_STATE (DUAL_OR_TIMER_VRP, STATE)
...
end process
```

5.5.3 Formal verification

After AADL2LNT transformation, various analyses can be performed by the CADP toolbox. In this thesis, we deal with two important formal techniques: simulation and model-checking. The LNT specification can be directly simulated

using the CADP simulators like the *OCIS* (Open/Cæsar Interactive Simulator) simulator enabling step-by-step simulation with backtracking.

In addition, an automatic verification phase can be carried out by the generated SVL script. In Listing 5.10, we include an extract from the *Pacemaker* SVL script. We remind that this script allows two steps, the compilation and then the verification of the LNT specification, respectively detailed in the next sections.

Listing 5.10: *Pacemaker*: mini SVL script

```
% DEFAULT_MCL_LIBRARIES="standard.mcl"
"Main.bcg"= divbranching reduction of "PACEMAKER_DCM_Main.lnt";
property Scheduling_Test (TH_NAME, ID)
   "Thread $TH_NAME scheduling test"
is
   "Main.bcg" |= with evaluator3
   NEVER ("ACTIVATION_$ID !T_Error");
   expected TRUE;
end property;
check Scheduling_Test (VVIMODE, 1);
property Connection (ID)
   "After a SEND_$ID action, a RECEIVE_$ID is eventually reachable"
is
   "Main.bcg" |= with evaluator3
   AFTER_1_INEVITABLE_2 ('SEND_$ID !*', 'RECEIVE_$ID !*');
   expected TRUE;
end property;
check Connection (CONNECTION_00);
check Connection (CONNECTION_01);
check Connection (CONNECTION_02)
```

5.5.3.1 Compilation: state space generation

This is an imperative step to enable the CADP verification of the LNT specifications. A translation from LNT into LOTOS is firstly applied with a set of the CADP tools, as follows:

- *LNT.OPEN* is a script used to automate the conversion of LNT programs to LOTOS code. It provides a connection between *LNT2LOTOS* and the *OPEN/CÆSAR* framework [65] [5].
- *LPP* is an LNT preprocessor that helps translating the LNT notations (for numbers, lists, etc.) into LOTOS models.

[5] *OPEN/CÆSAR* is an environment that allows user-defined programs for simulation, execution, verification and test generation to be developed in a simple and modular way. This framework includes various modules such as the OCIS simulator, the EVALUATOR model-checker, etc.

- *LNT2LOTOS* translates the LNT program into LOTOS. The input specification must be a valid LNT program (compilable) according to the language syntax described in its manual [48].

Thereafter, an LTS is generated from the LOTOS program by the *CÆSAR* compiler [64] and saved with the BCG (Binary-Coded Graphs) format. The LTS represents the state space of the LNT specification that will be explored later in model-checking. LTSs can be explicitly manipulated as BCG graphs, which is both a format for LTSs representation and a set of libraries and programs dealing with LTSs (information (*BCG_INFO*), display (*BCG_DRAW*), edition (*BCG_EDIT*), minimization (*BCG_MIN*), etc.).

In addition, the LTS generation can be smartly reduced to improve the verification performance. It can also be personalized using the SVL language (hiding, cutting, renaming labels, etc.). As illustrated in Listing 5.11, we include the generation of the *control_thread_states.bcg* graph from which we obtained an LTS which is depicted in Figure 5.8. It is a `divbranching` reduction [68] of the *Robot* case study that hides all LTS labels except the `DISPLAY_STATE` and `INFO_SENSOR` gates, in order to highlight the Behavior annex states and transitions of the `Control thread`. The resulting *control_thread_states.bcg*, drown in Figure 5.8 by *BCG_DRAW* tool, represents a reduced LTS of the *Robot* LNT specification.

Listing 5.11: *Robot*: smart generation with the SVL language

```
1 "control_thread_states.bcg"=
2    divbranching reduction of
3       hide all but
4          DISPLAY_STATE, INFO_SENSOR in "ROBOT_Main.lnt"
5 end hide;
```

Table 5.2 includes LTS metrics of our case studies. These results show the effectiveness of our contribution and the improvements achieved for our work and for the Ocarina formal verification in general:

- FCS, *Robot* and *Pacemaker* systems are presented with small state spaces, compared to the considered AADL subset (event-driven `threads`, Behavior annex, preemptive scheduling, etc.).

- Considering the scheduling mapping level, the obtained activation graph can be compared to analysis results performed with existing schedulability analysis tools. For instance, we choose Cheddar [156] tool, since it supports AADL as input model. The generation of the *FCS* LTS can be personalized by cutting the `T_Stop` and `T_Completion` labels, thus we obtain the activation graph counting 52 states corresponding to the same number of context switches found by Cheddar when analyzing the *FCS* AADL model.

- Note that AADL `device` presence only increases the transitions number without changing the activation graph.
- The *FCS* was experimented in [129] using a first version of the AADL2LNT transformation. Regarding the state explosion problem met with this old mapping, we note a significant reduction (up to 100%) in the state space with the current transformation version, compared to statistics given in [129].
- Note that the Ocarina tool suite is extended in [145, 146, 147] by the generation of a Petri net model for formal analysis with the Tina tool. This work is illustrated with the same *Robot* case study in [145]. Compared to our experimental results, we note a significant reduction in the state space metrics: about 918 states and 953 transitions for the *Robot* LNT specification, compared to 65 527 states and 425 985 transitions for a Petri net model without a timer.

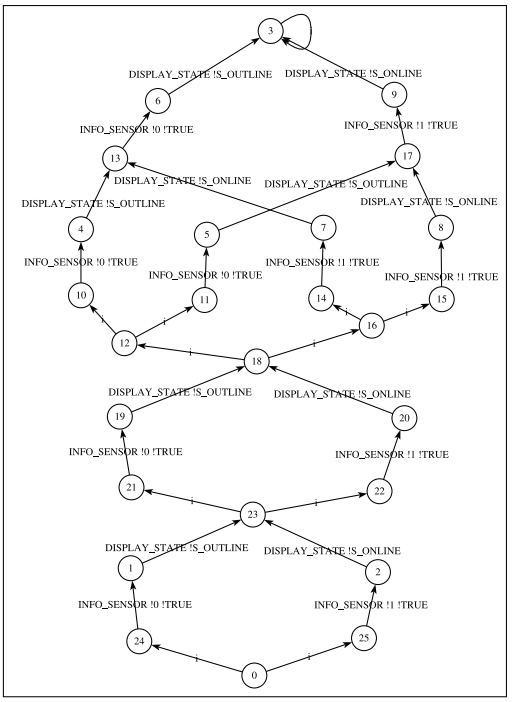


FIGURE 5.8: Generated LTS corresponding to Listing 5.11

5.5.3.2 Verification: model-checking

After state space generation, we reach the verification phase based on the model-checking of a set of structural and behavioral properties. In this thesis, we work with the *EVALUATOR* [120, 121] model-checkers to verify properties expressed

in the MCL temporal logic [6]. In our case, the verified properties are integrated in the SVL script using property statements, which embed MCL temporal logic formulas, as developed in section 5.3.3. To specify different properties, we use a set of standard macros (*standard.mcl* library) for the temporal operators parameterized by action and/or state formulas, such as `SOME (R)` (there exists at least one action sequence R), `NEVER (R)` (there is no action sequence R) and `AFTER_1_INEVITABLE_2 (A, B)` (after action A, the action B is inevitably reachable). We include below the set of generic properties for different mapping levels:

- Scheduling mapping
  - `Scheduling_Test` property (Listing 5.12): this property may be applied at the system level so it will analyze the schedulability of the whole task model, in addition, it can be parametrized to indicate if a given `thread` has respected all its deadlines (absence of the `T_Error` label);
  - `Is_Preempted` property (Listing 5.13): a `thread` may be preempted by the `SCHEDULER`, this property detects if a given `thread` has been preempted during the scheduling. The absence of all the `T_Preemption`, `T_Dispatch_Preemption` and `T_Preemption_Completion` labels means that the `thread` is never preempted.
- Communication mapping
  - `Connection` property (Listing 5.14): this property verifies if a port connection is well established, through a given port connection `AB`, after the transfer of the data or event (a *rendezvous* on `SEND_AB` gate), there is at least one reading of the port content (a *rendezvous* on `RECEIVE_AB` gate);
  - `Data_Loss` property (Listing 5.15): this property detects the loss of data through a given data port connection `AB`, it detects the occurrence of two successive transfers of data (*rendezvous* on `SEND_AB` gate), without a reading of the port content, in this case, the oldest input is overwritten by the newest one;
  - `Overflow_FIFO_N` property (Listing 5.16): this property detects if a list (FIFO with `N` size) of an event/event data port is overflowed, it detects the occurrence of `N+1` successive transfers of events, without any reading of the port content, in this case, the oldest input is overwritten by the `N+1`th one.

[6] The MCL language is based on the alternation-free fragment of the modal mu-calculus [120] for temporal logic formula specification.

- Behavioral mapping
  - `Transition` property (Listing 5.17): verifies the transitioning between two given `complete` states S_I and S_J through the `DISPLAY_STATE` communication for a given `thread`.

Listing 5.12: SVL schedulability property

```
property Scheduling_Test (ID) is
   "Main.bcg" |= with evaluator3
   NEVER ("ACTIVATION_$ID !T_Error");
   expected TRUE;
end property;
```

Listing 5.13: SVL preemption property

```
property Is_Preempted (ID)
   "Preemption test"
is
   "Main.bcg"|= with evaluator3
   SOME (true* . "ACTIVATION_$ID !T_DISPATCH_PREEMPTION" .
         true* . ("ACTIVATION_$ID !T_PREEMPTION")* .
         true* . "ACTIVATION_$ID !T_PREEMPTION_COMPLETION");
   expected TRUE;
end property;
```

Listing 5.14: SVL connection property

```
property Connection (ID)
   "After a SEND_$ID action,
   a RECEIVE_$ID is eventually reachable"
is
   "Main.bcg" |= with evaluator3
   AFTER_1_INEVITABLE_2 ('SEND_$ID !*', 'RECEIVE_$ID !*');
   expected TRUE;
end property;
```

Listing 5.15: SVL data loss property

```
property Data_Loss (ID)
   "Between two consecutive SEND_$ID actions,
   there must be a RECEIVE_$ID ation"
is
"Main_$ID.bcg" =
   hide all but "RECEIVE_$ID", "SEND_$ID" in "Main.bcg"
   |=  with evaluator3
    NEVER (true* . "SEND_$ID !AADLDATA" .
           (not "RECEIVE_$ID !AADLDATA")* .
           "SEND_$ID !AADLDATA");
       expected TRUE;
end property;
```

Listing 5.16: SVL FIFO property

```
property Overflow_FIFO_3 (ID)
   "Between 3 consecutive SEND_$ID actions,
   there is a RECEIVE_$ID action"
is
   "Main.bcg"    |=  with evaluator3
   NEVER (true*."SEND_$ID !AADLDATA".
                "SEND_$ID !AADLDATA".
                "SEND_$ID !AADLDATA".
                (not "RECEIVE_$ID !AADLDATA")*.
                "SEND_$ID !AADLDATA");
   expected TRUE;
end property
```

Listing 5.17: SVL transition property

```
property Transition (TH_NAME, Si, Sj)
   "Thread $TH_NAME, being in state $Si,
   the corresponding $Sj state is eventually reachable"
is
   "Main.bcg" |= with evaluator3
   AFTER_1_INEVITABLE_2 ("DISPLAY_STATE !$TH_NAME !$Si",
                         "DISPLAY_STATE !$TH_NAME !$Sj");
   expected TRUE;
end property;
```

At this level, we can explain the necessity of additional synchronizations like the `DISPLAY_STATE` communication and the `T_Error` label in the `ACTIVATION` communication as detailed in section 3.4.1 (chapter 3) and section 4.5.3.2 (chapter 4). These synchronizations are visible in the resulting LTS, that so, the `thread` behavior can be traced and any Behavior annex state can be inspected.

These properties allow the detection of serious problems at the model level. Since we deal with real-time systems, it is necessary to validate temporal and communication parameters such as the deadlock detection, schedulability test and the detection of connection failures (FIFO overflow, loss of data, broken links) at early phases of the development process.

5.5.4 Analysis results

An import issue in the AADL formal approaches concerns the usability of the analysis results produced by the formal tools. In our work, we provide a user-friendly (simple) output form that is easily interpreted with non formal-expert designers.

Based on a traceable mapping of the LNT specification and SVL properties, we preserve information from the initial AADL model through model transformation and verification so that they are kept until the generation of analysis results.

The traceability is firstly assumed during the model generation as described in section 5.3.2.1: the LNT specification preserves the identifiers of the AADL components, ports, port connections and Behavior annex variables/states, which are used in the naming rules of the LNT processes, gates and variables. Thus, elements from AADL model and Behavior specification can be identified in the produced LNT specification. In addition, the use of parameterized and commented properties furthers the traceability and gives understandable results. The parameters are used to present connections, `threads` and Behavior states by their initial AADL names. Thus, each `thread`, port connection or `complete` state can be separately verified and so failures are rapidly localized in the AADL model.

The *FCS*, *Robot* and *Pacemaker* case studies are successfully checked (compilation and verification). We note that, in general, all generated LNT specification are compilable with CADP. However, an unicity error may occur when using same port connection names at different levels (`process` and `system`), since all port connections are abstracted, with their AADL model names, at the `MAIN` process. To avoid this error, designers should simply use different names (pairwise distinct) for all port connections in the AADL model.

The displayed analysis results are simple and meaningful: each verified property is displayed with a `Pass` or `Fail` decision. For illustration, we include an extract from the *Pacemaker* analysis output in Figure 5.9, which corresponds to the SVL script of Listing 5.10: `Scheduling_Test` and `Connection` properties are applied in the *Pacemaker* case study. Results for `VVIMode thread` and all port connections of Listing 5.4 are obviously identified in Figure 5.9.

Note that different case studies, *FCS* (48 properties), *Robot* (28 properties) and *Pacemaker*(27 properties), are verified in few second in a basic machine. They are well scheduled and deadlock-free. All connections are well established. We test the *Robot* system with different FIFO sizes and we find that *sensor-control* queue size should be >= 2 to avoid overflow problem.

5.5.5 Manual verification

The automatic generation of the LNT specification is an advantageous achievement in this thesis, allowing an important formal verification phase. Yet, case studies have their specific details, that may not be modeled in the AADL language or not considered in our subset. In this case, the produced specification misses information which may be essential in the verification phase. While defining the AADL2LNT transformation, we opted to design a modular and traceable mapping (see section 3.4.5 of chapter 3) to obtain a comprehensible LNT specification favoring the manual extensions for a specific case study.

For illustration, we apply advanced analysis on the *Pacemaker* case study: the `DEVICE_PULSE_GENERATOR_IMPL` is completed to be dispatch by the `SCHEDULER`.

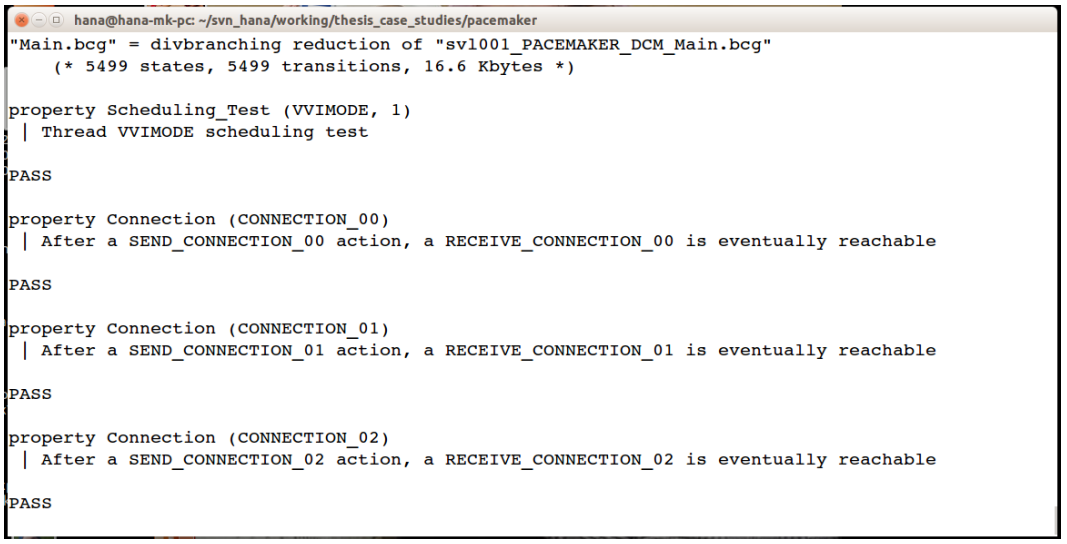


FIGURE 5.9: Analysis results corresponding to Listing 5.10

Thus, we can verify the therapeutic behavior of the pulse generator. These changes consider the DEVICE_PULSE_GENERATOR_IMPL and SCHEDULER processes with some modifications in the *Types* module.

Now, we can check, for example, the normal rhythm detection as follows: firstly, the DEVICE_PULSE_GENERATOR_IMPL should be periodically dispatched to sends event on sense port every 90 ms; and secondly, we define a new SVL property (Listing 5.18) that verifies if the THREAD_VVIMODE_IMPL puts an event on the normal (connection_02 port connection) port without any pacing event (no event in the connection_01 port connection).

LISTING 5.18: *Pacemaker*: SVL normal rhythm property

```
property Normal_Rhythm is
    "Main.bcg" |= with evaluator3
    SOME (true* . "RECEIVE_CONNECTION_02 !TRUE") and
    NEVER ("RECEIVE_CONNECTION_01 !TRUE");
    expected TRUE;
end property;
```

In addition, we add a reachability property, included in Listing 5.19, for the *FCS* case study to verify if an order is finally sent by the FL thread (connection V14 at the system level).

LISTING 5.19: *FCS*: SVL order reachability property

```
property ORDER is
    "Main.bcg" |= with evaluator3
    SOME (true* . "SEND_V14 !AADLDATA");
    expected TRUE;
end property;
```

5.6 Scalability

The proposed tool-chain has been initially tested with a set of case studies (flight control system, pacemaker, robot, door management system, etc.) from the AADLib [7] library to validate the correction of the AADL2LNT generation. Such examples with certain scale are easily verified in few minutes with basic machines. In addition, we carry out a set of advanced experiments to evaluate the scalability of our solution.

In a formal context, the state space explosion is a serious issue that discourages the application of formal methods. As defined in section 2.2.5.1 (chapter 2), the state space explosion problem occurs when the system state space becomes too large to be verified. In our work, we deal with real-time systems based on parallel concurrent behaviors, which often lead to generate large state spaces. To avoid the pitfall of explosion problem, a set of adjustments (discussed in section 4.3.4 of chapter 4) were applied on the LNT mapping in order to obtain small state spaces. The resulting LTS size depends on many factors related to the given AADL model in different points (tasking model, scheduling protocol, communication, etc.). In this thesis, we propose a scalability study based on two main factors which are the number of `threads` and the simulation interval $H(\tau_{1..n})$.

5.6.1 Test suite

A test suite is defined, composed of 100 AADL models, to evaluate three model families: (i) models with independent `threads`; (ii) models with periodic `threads` and data port connections; (iii) models with sporadic `threads` and event port connections. The test is based on a set of generalized *Produce-Consumer* system: we increase progressively the number of *Produce-Consumer* couples with different `thread` periods to also increase the $H(\tau_{1..n})$ value.

Starting from models with only 2 `threads`, we reach 40 and 70 `threads` to be tested during thousands of units of time (a unit may be any time interval e.g. 10ms). The prepared AADL *Produce-Consumer* models are transformed into LNT specifications by Ocarina and then compiled into LTSs by CADP.

5.6.2 Results and interpretations

Tables 5.3, 5.4 and 5.5 summarize some of the resulting LTS metrics. Table 5.3 concerns a set of exhaustive tests of family (i): the $H(\tau_{1..n})$ value is between 100 and 30 000 units of time (row 1) and the number of `threads` is between 50 and

[7] AADLib is a library of reusable AADLv2 models under the OpenAADL project (https://github.com/OpenAADL/AADLib)

70 (column 1). Tables 5.4 and 5.5 regroups respectively results of families (ii) and (iii): the $H(\tau_{1..n})$ value is between 100 and 5 000 units of time (row 1) and the number of `threads` is between 2 and 40 `threads` (column 1). For each test, we give the number of states of the generated LTS.

In general, the LTS size grows as the number of `threads` and units of time ($H(\tau_{1..n})$) increases, yet each family test has some particular observations:

(i) This family includes the state spaces generated from independent-`thread` AADL models. As shown in Table 5.3, LTSs are quite small compared to the important number of `threads`. The exhaustive experiments reach 50 `threads` tested until 30 000 units of time without explosion. We notice an interesting results with 50 and 60 `threads`, simulated during 100 units of time, which are tested in pretty small spaces (about one hundred states). Note that, beyond 70 `threads`, LTSs grow exponentially and the state explosion problem is more frequent which makes models, with such a scale, hard to be verified.

(ii) When adding port connections, the corresponding LNT processes (`CONNECTOR`) are added in the produced LNT specification which increases the size of generated LTS. For this reason, tests are limited to 40 `threads`, and beyond that, the state explosion problem is more frequent. Nevertheless, we still obtain interesting results: starting with only 29 states for 2 `threads` and 100 units of time, reaching 2 020 states for 40 `threads` and 5 000 units of time (Table 5.4).

(iii) This family includes models with sporadic `threads` and event or event data ports. In the *Produce-Consumer* system, the consumer `thread` becomes sporadic with an event *in* port. Being tested in the same scope as the family (ii), the resulting LTSs of this family are much bigger and lead to some explosions (mainly with 40 `threads`). This is explained by the fact that additional synchronizations are required for the sporadic `threads` scheduling (`NOTIFICATION` synchronizations between the `CONNECTOR`s and `SCHEDULER`). Yet, we still obtain reasonable results. For example, models with 10 and 20 `threads` are successfully simulated with small state spaces.

This test suite has been carried out on a machine with high-performances [8] using the 2017-J "Sophia Antipolis" CADP version. The generation of all LNT specifications needs a few seconds. While, the analysis time (LTS generation and model-checking of properties) depends evidently of the test and may require some minutes or hours. The LTS generation of all tests of family (ii) needed 28 minutes,

[8] Processor Intel Xeon(R), 2.20 GHz x32, 63GB RAM, running Linux MATE 1.12

Table 5.3: State spaces results of family (i)

	periodic independent threads			
	100	5 000	10 000	30 000
50	114	1 716	2 859	16 001
60	134	1 804	∞	∞
70	∞	1 884	2 610	∞

Table 5.4: State spaces results of family (ii)

	periodic threads with data port connections						
	100	300	500	1 000	1 500	3 000	5 000
2	29	37	23	30	37	113	89
8	121	140	196	266	326	116	386
10	153	175	115	386	175	145	523
20	305	350	230	498	350	395	1 003
40	428	269	774	836	700	2 290	2 020

Table 5.5: State spaces results of family (iii)

	sporadic threads with event port connections						
	100	300	500	1 000	1 500	3 000	5 000
2	136	563	997	2 052	2 904	5 910	9 782
8	1 590	7 028	12 116	27 318	39 254	85 212	134 064
10	2 802	9 875	17 820	39 788	58 401	134 750	212 148
20	7 960	36 655	63 192	129 619	211 519	504 365	835 786
40	38 820	∞	∞	∞	∞	∞	∞

which is a satisfying time for a test suite composed of 43 models counting up to 40 `threads`. Regarding family (iii), the analysis time of a *Producer-Consumer* model with 10 `threads` simulated during 1 500 units of time is about few seconds (with a basic machine, it may take 4 minutes). While, it takes about 1 hour for the *Producer-Consumer* model with 20 `threads` simulated during 1 000 units of time.

To conclude, the size of LTS depends directly on the number of `threads`, but also on other minor factors such as the scheduling mapping and the obtained activation graph: sporadic and preemptive `threads` are more expensive (in size) than independent, periodic or non-preemptive `threads`; similarly, highly-connected `threads` increase significantly the size of the LTSs. Compared to the considered subset, the proposed AADL model verification requires reasonable resources in memory and time. These experimental results are promising, showing the effectiveness of our solution to verify real-time systems with respectable scale.

5.7 Conclusion

In this chapter, we firstly presented the AADL2LNT Ocarina extension conceived and implemented as part of research activities conducted in this thesis. Based on this extension, an automatic tool-chain is obtained connecting two powerful tools: Ocarina for AADL modeling and CADP for formal verification. The usefulness and the feasibility of this tool-chain have also been highlighted through three case studies: flight control system, line follower robot, pacemaker device. Different design and verification phases are explained and discussed to evaluate the proposed model transformation and formal verification phases of AADL models. Finally, we end this chapter with a scalability study.

The proposed model verification brings useful analysis results that help designers in the AADL model correction and improvement. This operation may be iteratively applied after each model modification, throughout the system development process, until the generation of the final application.

6

Conclusion and perspectives

6.1 Conclusions

In this thesis, we reported our experience in the context of the formal verification of real-time systems. The main problem concerns the integration of formal methods in an MDE approach, that requires a formal expertise and significant effort for the specification and verification activities. In this context, a common solution is the use of model transformation techniques to connect MDE platforms with existing analysis tools, which raises several challenges about the semantic gap between design and formal models, the complexity and the correction of the model transformation, the usability of the analysis results and the applicability in the case of large scale systems (state space explosion problem).

6.1.1 Reminder of the contributions and results

Our main objective is to assist designers in the formal verification activities of real-time systems. We proposed an MDE approach integrating an automatic formal verification phase, hiding all formal aspects for non-formal expert designers. This characteristic is a key feature to encourage formal methods practice in software engineering.

A first theoretical contribution (chapter 3) in this thesis consists of the definition a formal pattern for a Ravenscar task model. The considered real-time task model is formally mapped through the LNT language based on the process algebra concept suitable for concurrent models and a rich data part sufficient for the mapping of scheduling policies. This pattern is described, justified and refined several times to be modular and extensible. It provides a formal executable semantics considering mainly the scheduling execution and communication which

are indispensable for a useful analysis of real-time systems. The main idea of this pattern consists in the encapsulation of the temporal calculations (execution times, preemptions, etc.) within the scheduler, in order to restrict the synchronization between processes, which reduces significantly the generated system state space. This pattern is generically designed to be used as a pivot model for other real-time transformation applications.

The proposed LNT pattern has been applied and tested in an MDE development process based on the AADL modeling language (chapter 4). A model transformation AADL2LNT is defined to translate a given AADL model into an LNT specification. The transformation supports an important AADL subset composed of a set of software and hardware components (`data`, `thread`, `process`, `processor` and `device`), port connections and a set of temporal and communication properties. This proposition is also extended to consider the Behavior annex that adds behavioral descriptions for the `thread` components. The AADL2LNT transformation is implemented within the Ocarina tool suite, allowing the generation of the LNT specification with an SVL script. These outputs are used by the CADP toolbox to achieve the formal verification by model-checking of a set of structural and behavioral properties for the schedulability analysis, `thread` execution simulation, `thread` behavior analysis and the verification of a set of communication properties.

The AADL2LNT Ocarina extension (chapter 5) can be downloaded from the Ocarina GitHub repository [1] to be used as a stand-alone compiler or to be integrated in OSATE through the OSATE2-Ocarina-plugin [2]. We provided a tool-chain that covers all the required verification phases and issues (formal specification, model-checking, analysis results and state space explosion problem), to take advantage of existing well-experimented analysis tools such as CADP. The proposed solution has been illustrated with various real-time systems form the AADLib library. In this manuscript, we presented experiments performed on three case studies: flight control system, line follower robot and pacemaker device (chapter 5). In addition, a scalability study is carried out to prove the efficiency and the applicability of our work in the verification of real-time systems with a respectable scale. A test suite is performed to evaluate the different kinds of the AADL models supported in our work: models with independent `threads`, models with periodic `threads` and data port connections and models with sporadic `threads` and event port connections.

The proposed solution brings promising results face to the formal verification challenges (analysis time and state space explosion), which is encouraging for more advanced mapping and analysis. We believe that our solution is a representative example of how formal methods can be smoothly integrated into the

[1] Ocarina GitHub: `https://github.com/OpenAADL/ocarina`

[2] Since the release of OSATE 2.0.9, the OSATE2-Ocarina-plugin is part of the distribution.

development of safety-critical systems, by defining appropriate formal patterns and model transformations between high-level languages.

6.2 Perspectives

Although the contributions of this thesis have yielded interesting results, there is still scope to supplement and improve them. In the following, we identify some perspectives concern the different phases of our work.

LNT real-time pattern

A promising perspective concerns the formal analysis of the Ravenscar tasking profile for Ada programs. We provided an LNT pattern for a Ravenscar compliant task model that can be adapted for code analysis ends. In this case, the attention turns to the implementation phase to automate verification activities. It is possible to integrate this proposition within the Ocarina tool suite and even suitable since it already provides a code generator for the Ada language [87].

In addition, the proposed pattern can be archived as a Ravenscar library in the LNT language. This library includes the LNT definitions for the Ada tasking sub-language such as tasks and protected procedures. The scheduler definition may also be added for different scheduling policies (RM, EDF, etc.).

AADL2LNT model transformation

The AADL2LNT transformation is defined to formally verify the AADL models compliant with the Ravenscar profile for safety-critical systems. Thus, only periodic and sporadic `threads` with preemptive fixed-priority scheduling were used at first. In addition, a larger subset, beyond the Ravenscar restrictions, is also considered in section 4.4 (chapter 4) adding mainly timed and hybrid dispatch protocols with the EDF scheduling policy. In the same direction, the AADL2LNT transformation can be extended in future work through its different levels as follows: the scheduling mapping can be extended to support the multiprocessor architecture; and the communication mapping may be completed by networking aspects via the AADL `bus` and `virtual bus` components. In another direction, the AADL2LNT transformation may be completed by defining new mapping levels such as the access mapping to consider the AADL shared resources.

Formal verification

The empirical evaluation performed in this thesis proves the effectiveness of the proposed solution to verify realistic large scale systems. However, results are

relatively limited in the case of highly-connected models, that involve communication and synchronization increasing significantly the generated state spaces. In this context, an important perspective concerns the use of the compositional state space construction based on the compositionality, one of the most promising approach to fight the state explosion problem. This technique can be used in our work since it is provided by the CADP toolbox in [68]. It seems useful for the verification of AADL highly-connected models, especially when considering the Behavior annex mapping (to manage the additional synchronization required for concert data exchanging between the threads). The compositional verification relies on the *divide-and-conquer* paradigm to breakdown the complexity of large systems. It can be applied at the LTS generation level (to be included in the SVL script) to separately generate different processes, then minimize them separately before combining them.

Implementation and documentation

The current Ocarina extension allows an important generation and we are continuously working on our implementations to fix bugs and complete the generation in the behavioral mapping for a total automation.

Currently, the proposed AADL2LNT transformation is detailed through our publications [129, 130] and the present manuscript. We plan also to provide a practical designer manual (tutorial) containing concise and clear descriptions about the transformation scope, objectives and outputs, to facilitate the use of the Ocarina extension.

Bibliography

[1] ISO/IEC. LOTOS a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization Information Processing Systems Open Systems Interconnection, Geneve. 1989.

[2] RTCA/DO-178B: Software Considerations in Airborne Systems and Equipment Certification. 1998.

[3] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437, International Organization for Standardization âĂŞ Information Technology, Geneva. 2001.

[4] ISO/IEC/IEEE 42010: Systems and software engineering–architecture description. Technical report, ISO/IEC/IEEE, 2011.

[5] RTCA/DO-178C: Software Considerations in Airborne Systems and Equipment Certification. 2011.

[6] RTCA/DO-333: formal methods supplement to DO-178C and DO-278A. 2011.

[7] UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1. 2011.

[8] OMG Unified Modeling Language (OMG UML) Version 2.5. 2015.

[9] AS5506/3: Architecture Analysis and Design Language (AADL) Annex D: Behavior Model Annex. 2017.

[10] AS5506C: SAE Architecture Analysis and Design Language (AADL) AADL V2.2. 2017.

[11] ISO/IEC 15408: The Common Criteria for Information Technology Security Evaluation (CC), version 3.1, revision 5. 2017.

[12] T. Abdoul, J. Champeau, P. Dhaussy, P. Y. Pillain, and J.-C. Roger. AADL execution semantics transformation for formal verification. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 263–268. IEEE, 2008.

[13] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[14] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa. An overview of formal methods tools and techniques. In *Rigorous Software Development*, pages 15–44. Springer, 2011.

[15] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[16] M. Amrani. *Towards the Formal Verification of Model Transformations: An Application to Kermeta.* PhD thesis, University of Luxembourg, 2013.

[17] S. Andova, M. G. van den Brand, L. J. Engelen, and T. Verhoeff. Mde basics with a dsl focus. In *Formal Methods for Model-Driven Engineering*, pages 21–57. Springer, 2012.

[18] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Real-time system scheduling. In *Predictably Dependable Computing Systems*, pages 41–52. Springer, 1995.

[19] K. Bae, P. C. Ölveczky, A. Al-Nayeem, and J. Meseguer. Synchronous AADL and its formal analysis in real-time Maude. In *International Conference on Formal Engineering Methods*, pages 651–667. Springer, 2011.

[20] K. Bae, P. C. Ölveczky, and J. Meseguer. Definition, semantics, and analysis of multirate synchronous AADL. In *International Symposium on Formal Methods*, pages 94–109. Springer, 2014.

[21] K. Bae, P. C. Ölveczky, J. Meseguer, and A. Al-Nayeem. The SynchAADL2Maude tool. In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–62. Springer, 2012.

[22] J. C. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[23] J. E. Barnes. Experiences in the industrial use of formal methods. *Electronic Communications of the EASST*, 46, 2011.

[24] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.

[25] L. M. Barroca and J. A. McDermid. Formal methods: Use and relevance for the development of safety-critical systems. *The Computer Journal*, 35(6):579–599, 1992.

[26] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal-a tool suite for automatic verification of real-time systems. In *International Hybrid Systems Workshop*, pages 232–243. Springer, 1995.

[27] J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. 1982.

[28] 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 *International Conference on Reliable Software Technologies*, pages 207–221. Springer, 2009.

[29] B. Berthomieu*, P.-O. Ribet, and F. Vernadat. The tool tina–construction of abstract state spaces for petri nets and time petri nets. *International journal of production research*, 42(14):2741–2756, 2004.

[30] L. Besnard, A. Bouakaz, T. Gautier, P. Le Guernic, Y. Ma, J.-P. Talpin, and H. Yu. Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony. *Science of Computer Programming*, 106:54–77, 2015.

[31] L. Besnard, T. Gautier, P. Le Guernic, C. Guy, J.-P. Talpin, B. Larson, and E. Borde. Formal semantics of behavior specifications in the architecture analysis and design language standard. In *Cyber-Physical System Design from an Architecture Analysis Viewpoint*, pages 53–79. Springer, 2017.

[32] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[33] 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*, pages 273–280. IEEE, 2001.

[34] 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*, 6(02):201–227, 1996.

[35] B. S. Blanchard. *System engineering management*. John Wiley & Sons, 2004.

[36] 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*, 106:30–53, 2015.

[37] E. Boiten. Modeling in event-b–system and software engineeringabrial jean-raymondcambridge university press, may 2010 isbn-10: 0521895561. *Journal of Functional Programming*, 22(2):217–219, 2012.

[38] 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 *International Conference on Computer Safety, Reliability, and Security*, pages 173–186. Springer, 2009.

[39] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended AADL models. *The Computer Journal*, 54(5):754–775, 2010.

[40] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended AADL models. *The Computer Journal*, 54(5):754–775, 2010.

[41] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, and R. Wimmer. A model checker for aadl. In *International Conference on Computer Aided Verification*, pages 562–565. Springer, 2010.

[42] J. R. Büchi. Weak second-order arithmetic and finite automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[43] A. Burns. The Ravenscar Profile. *ACM SIGAda Ada Letters*, 19(4):49–52, 1999.

[44] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *ACM SIGAda Ada Letters*, 24(2):1–74, 2004.

[45] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems*. Springer, 2005.

[46] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[47] S. Care. Pacemakers. http://www.sterlingcare.com/resources/resources/diseases-and-conditions-library/view/pacemakers/, 2012.

[48] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference manual of the LNT to lotos translator. 2018.

[49] 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*, pages 5–19. Springer, 2009.

[50] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real-time systems of periodic tasks with offsets. *Theoretical computer science*, 310(1-3):117–134, 2004.

[51] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[52] R. Cleaveland and S. T. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, 2002.

[53] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[54] J.-P. Courtiat, C. A. Santos, C. Lohr, and B. Outtaj. Experience with rt-lotos, a temporal extension of the lotos formal description technique. *Computer Communications*, 23(12):1104–1123, 2000.

[55] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[56] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *International Symposium on Formal Methods*, pages 253–271. Springer, 1999.

[57] J. Davies and S. Schneider. A brief history of timed csp. *Theoretical Computer Science*, 138(2):243–271, 1995.

[58] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):35, 2011.

[59] K. Evensen and K. Weiss. A comparison and evaluation of real-time software systems modeling languages. In *AIAA Infotech@ Aerospace 2010*, page 3504. 2010.

[60] P. H. Feiler and D. P. Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[61] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

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

[63] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[64] H. Garavel. Open/Cæsar: An open software architecture for verification, simulation, and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–84. Springer, 1998.

[65] H. Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–84. Springer, 1998.

[66] H. Garavel and R.-P. Hautbois. An experiment with the lotos formal description technique on the flight warning computer of airbus 330/340 aircrafts. In *Proc. of the first AMAST International Workshop on Real-Time Systems*. Citeseer, 1993.

[67] H. Garavel and F. Lang. Svl: a scripting language for compositional verification. In *Formal Techniques for Networked and Distributed Systems*, pages 377–392. Springer, 2002.

[68] H. Garavel, F. Lang, and R. Mateescu. Compositional verification of asynchronous concurrent systems using CADP. *Acta Informatica*, pages 1–56, 2015.

[69] 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*, 15(2):89–107, 2013.

[70] H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. In *ModelEd, TestEd, TrustEd*, pages 3–26. Springer, 2017.

[71] T. Gautier, C. Guy, A. Honorat, P. Le Guernic, J.-P. Talpin, and L. Besnard. Polychronous automata and their use for formal validation of AADL models. *Frontiers of Computer Science*, 2017.

[72] D. Geniet and J.-P. Dubernard. Scheduling hard sporadic tasks with regular languages and generating functions. *Theoretical computer science*, 313(1):119–132, 2004.

[73] M. J. Gordon and T. F. Melham. Introduction to hol a theorem proving environment for higher order logic. 1993.

[74] J. F. Groote. The syntax and semantics of timed μcrl. In *CWI, PO BOX 94079, 1090 GB*. Citeseer, 1997.

[75] J. F. Groote and A. Ponse. The syntax and semantics of μcrl. In *Algebra of Communicating Processes*, pages 26–62. Springer, 1995.

[76] S. Gui, L. Luo, Y. Li, and L. Wang. Formal schedulability analysis and simulation for AADL. In *Embedded Software and Systems, 2008. ICESS'08. International Conference on*, pages 429–435. IEEE, 2008.

[77] M. E. Hamdane, A. Chaoui, and M. Strecker. From AADL to Timed Automaton-A Verification Approach. *International Journal of Software Engineering & Its Applications*, 7(4), 2013.

[78] M. E. Hamdane, A. Chaoui, and M. Strecker. From AADL to Timed Automaton-A Verification Approach. *International Journal of Software Engineering & Its Applications*, 7(4), 2013.

[79] M. E.-K. Hamdane, A. Chaoui, and M. Strecker. Toolchain based on mde for the transformation of AADL models to timed automata models. *Journal of Software Engineering and Applications*, 6(03):147, 2013.

[80] I. Hamid and E. Najm. Real-time connectors for deterministic data-flow. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 173–182. IEEE, 2007.

[81] I. Hamid and E. Najm. Operational semantics of ada ravenscar. In *International Conference on Reliable Software Technologies*, pages 44–58. Springer, 2008.

[82] M. Hecht, A. Lam, and C. Vogl. A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex. In *ICECCS*, pages 361–366, 2011.

[83] U. Herzog. Formal methods for performance evaluation. In *School organized by the European Educational Forum*, pages 1–37. Springer, 2000.

[84] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[85] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics (NRL)*, 21(1):177–185, 1974.

[86] K. Hu, T. Zhang, Z. Yang, and W.-T. Tsai. Exploring AADL verification tool through model transformation. *Journal of Systems Architecture*, 61(3):141–156, 2015.

[87] 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)*, 7(4):42, 2008.

[88] M. Z. Iqbal, S. Ali, T. Yue, and L. Briand. Applying uml/marte on industrial projects: challenges, experiences, and guidelines. *Software & Systems Modeling*, 14(4):1367–1385, 2015.

[89] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 134–143. ACM, 2007.

[90] A. Johnsen, K. Lundqvist, P. Pettersson, and O. Jaradat. Automated Verification of AADL-Specifications Using UPPAAL. In *HASE*, pages 130–138, 2012.

[91] A. Johnsen, K. Lundqvist, P. Pettersson, M. Torelm, et al. AQAF: an Architecture Quality assurance framework for systems modeled in AADL. In *Quality of Software Architectures (QoSA), 2016 12th International ACM SIGSOFT Conference on*, pages 31–40. IEEE, 2016.

[92] F. Jouault and I. Kurtev. Transforming models with atl. In *International Conference on Model Driven Engineering Languages and Systems*, pages 128–138. Springer, 2005.

[93] A. K. Karna, Y. Chen, H. Yu, H. Zhong, and J. Zhao. The role of model checking in software engineering. *Frontiers of Computer Science*, pages 1–27, 2018.

[94] S. C. Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[95] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[96] J. C. Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550. IEEE, 2002.

[97] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[98] F. Kordon, J. Hugues, A. Canals, and A. Dohet. *Embedded systems: analysis and modeling with SysML, UML and AADL*. John Wiley & Sons, 2013.

[99] F. Kordon, J. Hugues, and X. Renault. From Model Driven Engineering to Verification Driven Engineering. In *SEUS*, pages 381–393, 2008.

[100] A. Kornecki and J. Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5(2):149–161, 2009.

[101] I. Kurtev. State of the art of qvt: A model transformation language standard. In *International Symposium on Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2007.

[102] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, and A. Tang. The road ahead for architectural languages. *IEEE Software*, 32(1):98–105, 2015.

[103] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[104] A. v. Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159. ACM, 2000.

[105] C. Larman and V. R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[106] B. R. Larson. Formal semantics for the pacemaker system specification. *ACM SIGAda Ada Letters*, 2014.

[107] 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 *Reliable Software Technologies–Ada-Europe 2009*, pages 237–250. Springer, 2009.

[108] L. Léonard and G. Leduc. A formal definition of time in lotos. *Formal Aspects of Computing*, 10(3):248–266, 1998.

[109] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[110] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, January 1973.

[111] 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*, 15(3):647–684, 2016.

[112] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, 1995.

[113] K. Lundqvist and L. Asplund. A ravenscar-compliant run-time kernel for safety-critical systems. *Real-Time Systems*, 24(1):29–54, 2003.

[114] 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*, pages 1–6. IEEE, 2011.

[115] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *European Software Engineering Conference*, pages 137–153. Springer, 1995.

[116] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What industry needs from architectural languages: A survey. *Software Engineering, IEEE Transactions on*, 39(6):869–891, 2013.

[117] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. Architectural languages today. http://www.di.univaq.it/malavolta/al/#home, 2017.

[118] C. Mascolo. Mobis: A specification language for mobile systems. In *International Conference on Coordination Languages and Models*, pages 37–52. Springer, 1999.

[119] R. Mateescu and H. Garavel. Xtl: a meta-language and tool for temporal logic model-checking. *STTTâĂŹ98*, pages 33–42, 1998.

[120] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003.

[121] R. Mateescu and D. Thivolle. A model checking language for concurrent value-passing systems. In *International Symposium on Formal Methods*, pages 148–164. Springer, 2008.

[122] D. D. McCracken and E. D. Reilly. Backus-naur form (bnf). 2003.

[123] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on software engineering*, 26(1):70–93, 2000.

[124] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[125] P. M. Merlin. *A study of the recoverability of computing systems*. PhD thesis, University of California, Irvine, 1974.

[126] Z. Mian, L. Bottaci, Y. Papadopoulos, and M. Biehl. System dependability modelling and analysis using AADL and hip-hops. *IFAC Proceedings Volumes*, 45(6):1647–1652, 2012.

[127] S. P. Miller and M. Srivas. Formal verification of the aamp5 microprocessor: A case study in the industrial use of formal methods. In *wift*, page 2. IEEE, 1995.

[128] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[129] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel. From AADL model to LNT specification. In *Reliable Software Technologies–Ada-Europe 2015*, pages 146–161. Springer, 2015.

[130] H. Mkaouar, B. Zalila, J. Hugues, and M. Jmaiel. An ocarina extension for AADL formal semantics generation. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1402–1409. ACM, 2018.

[131] A. K.-L. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[132] X. Nicollin and J. Sifakis. The algebra of timed processes, atp: Theory and application. *Information and Computation*, 114(1):131, 1994.

[133] I. Ober and N. Halbwachs. On the timed automata-based verification of ravenscar systems. In *International Conference on Reliable Software Technologies*, pages 30–43. Springer, 2008.

[134] 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*, pages 47–62. Springer, 2010.

[135] M. Ozkaya. The analysis of architectural languages for the needs of practitioners. *Software: Practice and Experience*, 48(5):985–1018, 2018.

[136] G. OâĂŹRegan. Z formal specification language. In *Concise Guide to Formal Methods*, pages 155–171. Springer, 2017.

[137] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52, 1992.

[138] K. Petersen, C. Wohlin, and D. Baca. The waterfall model in large-scale development. In *International Conference on Product-Focused Software Process Improvement*, pages 386–400. Springer, 2009.

[139] C. A. Petri. *Communication with automata*. PhD thesis, Institut fuer Instrumentelle Mathematik, 1962.

[140] G. D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[141] R. S. Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[142] A. N. Prior. Time and modality, 1957.

[143] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[144] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[145] X. Renault. *Mise en œuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systemes embarqués temps-réel répartis.* PhD thesis, Université Pierre et Marie Curie-Paris VI, 2009.

[146] X. Renault, F. Kordon, and J. Hugues. Adapting models to model checkers, a case study: Analysing AADL using Time or Colored Petri Nets. In *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*, pages 26–33. IEEE, 2009.

[147] X. Renault, F. Kordon, and J. Hugues. From AADL architectural models to Petri Nets: Checking model viability. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 313–320. IEEE, 2009.

[148] J.-F. Rolland, J.-P. Bodeveix, M. Filali, D. Chemouil, and D. Thomas. Modes in asynchronous systems. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 282–287. IEEE, 2008.

[149] A.-E. Rugina, K. Kanoun, and M. Kaâniche. The ADAPT Tool: From AADL Architectural Models to Stochastic Petri Nets through Model Transformation. *CoRR*, abs/0809.4108, 2008.

[150] N. Saeedloei and G. Gupta. Timed pi-calculus. In *International Symposium on Trustworthy Global Computing*, pages 119–135. Springer, 2013.

[151] D. Sangiorgi and D. Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[152] E. Seidewitz. What models mean. *IEEE software*, 20(5):26–32, 2003.

[153] B. Selic. A systematic approach to domain-specific language design using uml. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 2–9. IEEE, 2007.

[154] 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*, 28(2-3):101–155, 2004.

[155] N. Shankar, S. Owre, and J. M. Rushby. The pvs proof checker: A reference manual. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 3, 1993.

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

[157] O. Sokolsky and A. Chernoguzov. Performance analysis of AADL models using real-time calculus. In *Monterey Workshop*, pages 227–249. Springer, 2008.

[158] O. Sokolsky, I. Lee, and D. Clarke. Schedulability analysis of AADL models. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[159] O. Sokolsky, I. Lee, and D. Clarke. Process-algebraic interpretation of AADL models. In *International Conference on Reliable Software Technologies*, pages 222–236. Springer, 2009.

[160] I. Sommerville. *Software engineering*. New York: Addison-Wesley, 2010.

[161] I. D. Standard. Standard 00-55: The procurement of safety critical software in defence equipment. *UK Ministry of Defence*, 1991.

[162] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[163] M. Stigge and W. Yi. Graph-based models for real-time workload: a survey. *Real-time systems*, 51(5):602–636, 2015.

[164] W. M. Van der Aalst. Pi calculus versus petri nets: Let us eat âĂIJhumble pieâĂİ rather than further inflate the âĂIJpi hypeâĂİ. *BPTrends*, 3(5):1–11, 2005.

[165] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[166] V. Vyatkin. Software engineering in industrial automation: State-of-the-art review. *Industrial Informatics, IEEE Transactions on*, 9(3):1234–1249, 2013.

[167] J. M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990.

[168] C. Yang, Y. Dong, F. Zhang, E. Ahmad, and B. Gu. Formal semantics of AADL models with machine-readable CSP. In *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on*, pages 565–571. IEEE, 2012.

[169] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin. From AADL to timed abstract state machines: A verified model transformation. volume 93, pages 42–68. Elsevier, 2014.

[170] Z. Yang, K. Hu, D. Ma, and L. Pi. Towards a formal semantics for the AADL behavior annex. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1166–1171. European Design and Automation Association, 2009.

[171] H. Yu, Y. Ma, T. Gautier, L. Besnard, P. Le Guernic, and J.-P. Talpin. Polychronous modeling, analysis, verification and simulation for timed software architectures. *Journal of Systems Architecture*, 59(10):1157–1170, 2013.

[172] H. Yu, Y. Ma, T. Gautier, L. Besnard, J.-P. Talpin, P. Le Guernic, and Y. Sorel. Exploring system architectures in AADL via Polychrony and SynDEx. *Frontiers of Computer Science*, 7(5):627–649, 2013.

[173] F. Zhang, Y. Zhao, D. Ma, and W. Niu. Formal verification of behavioral AADL models by stateful timed CSP. *IEEE Access*, 5:27421–27438, 2017.

[174] Y. Zhang, Y. Dong, Y. Zhang, and W. Zhou. A study of the AADL mode based on timed automata. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pages 224–227. IEEE, 2011.

View publication stats