Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of compotation

Yue Ma

N° d'ordre : 4245 ANNÉE 2010

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Yue MA

préparée à l'unité de recherche 6074 IRISA
Institut de Recherche en Informatique et Systèmes Alèatoires
IFSIC

Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of computation

Thèse soutenue à Rennes le 29 Novembre 2010

devant le jury composé de :

Jean-Paul BODEVEIX
Professeur à l'Université Paul Sabatier/rapporteur

Frank SINGHOFF
Professeur à Université de Bretagne Occidentale / rapporteur

Christophe WOLINSKI
Professeur à l'Université de Rennes 1/examinateur

Abdoulaye GAMATIÉ
Chargé de recherche CNRS/examinateur

Thierry GAUTIER
Chargé de recherche INRIA/examinateur

Jean-Pierre TALPIN
Directeur de recherche INRIA/directeur de thèse

Acknowledgements

First of all, I am grateful to all the members of my dissertation committee. I wish to thank Prof. Christophe Wolinski for his acceptance to be the president of committee.

I would like to thank Jean-Paul Bodeveix, professor of Université Paul Sabatier, and Frank Singhoff, professor of Université de Bretagne Occidentale, for their acceptance to be reporters for this thesis.

I would like also to thank Abdoulaye Gamatié, CNRS researcher, to judge this work. I wish to thank Thierry Gautier, INRIA researcher, to judge this thesis, and help me constanly writing my articles and progressing my French during my three years.

This thesis would not have been possible without the guidance of my thesis advisor, Jean-Pierre Talpin. His continuous supervision, encouragement, and of course constructive criticism have been great help and support in my research work.

I also need to thank all the members of INRIA ESPRESSO team for sharing the good ambiance during my stay at INRIA. In addition, I appreciate the review of this thesis by Paul Le Guernic. I also want to express my additional gratitude to Loïc Besnard for sharing accommodation with me, and Huafeng Yu for sharing experiences when we worked together for the demo.

Special thanks to my family, I could not accomplish my study in France without your support and encouragement for these three years.

Contents

Introduction **1**

I Résumé en français 11

1 Résumé en français **13**
1.1 Introduction 13
1.2 Introduction au langage AADL et aux architectures avioniques 15
1.2.1 Abstractions du langage AADL 15
1.2.2 Annexe comportementale de AADL 16
1.2.3 Architecture avionique et ARINC 653 16
1.3 Langage Signal et modélisation ARINC en Signal 16
1.3.1 Le langage Signal 17
1.3.2 Modélisation de concepts ARINC en Signal 18
1.4 Travaux reliés : formalisations de AADL 19
1.5 Modélisation de composants AADL en processus Signal 20
1.5.1 Chaîne de transformation 20
1.5.2 Principes de transformation 21
1.5.3 Du temps logique abstrait vers un temps de simulation concret . . 21
1.5.4 Modélisation de thread 24
1.5.5 Modélisation des autres composants 27
1.6 Spécification de comportements AADL 28
1.6.1 Forme SSA 29
1.6.2 Systèmes de transition AADL 29
1.6.3 Interprétation des transitions / actions 30
1.7 Génération de modèles de simulation distribués 34
1.7.1 Placement 34
1.7.2 Ordonnanceur 36
1.7.3 Ajout des communications 36
1.8 Vérification et simulation 37
1.8.1 Vérification formelle 37
1.8.2 Simulation 38
1.9 Conclusion 39

II Conceptions of AADL and Signal **41**

2 Introduction to AADL and Avionic architectures **43**
2.1 AADL language abstractions 44
2.1.1 The SAE AADL standard 44
2.1.2 AADL meta-model and models 45
2.1.3 AADL open source tools 46
2.2 Summary of the core AADL components 48
2.2.1 Software components 50
2.2.2 Execution platform components 55
2.2.3 System component 58
2.3 System binding 59
2.4 Component interaction 60
2.4.1 Port 60
2.4.2 Port connection 61
2.5 Flows 63
2.6 AADL behavior annex 64
2.7 Avionics system architecture and ARINC653 65
2.7.1 Avionics system architecture overview 65
2.7.2 ARINC standard 66
2.7.3 AADL and ARINC 69
2.8 AADL components considered in this thesis 70
2.9 Conclusion 70

3 Signal Language and modeling ARINC in Signal **73**
3.1 Signal language 74
3.1.1 Signal, execution, process 74
3.1.2 Data types 75
3.1.3 Elementary processes 76
3.1.4 Process operators 77
3.1.5 Parallel semantic properties 78
3.1.6 Modularity features 79
3.1.7 Endochronous acyclic processes 80
3.1.8 Time domains and communications in Signal 80
3.1.9 Non-determinism modeling in Signal 82
3.1.10 Adequacy of Signal for AADL modeling 83
3.2 Modeling ARINC concepts in Signal 84
3.2.1 Partitions 84
3.2.2 Partition-level OS 85
3.2.3 Processes 86
3.2.4 APEX services 87
3.3 Conclusion 88

III Prototyping AADL models in a polychronous model of computation 89

4 Formalizing AADL 91
4.1 AADL background . . . 91
4.2 Related works . . . 92
4.2.1 Modeling AADL in MARTE . . . 93
4.2.2 Modeling AADL in SystemC . . . 94
4.2.3 Code generation from AADL to C . . . 94
4.2.4 Modeling AADL in Fiacre . . . 94
4.2.5 Modeling AADL in TASM . . . 96
4.2.6 Modeling AADL in ACSR . . . 98
4.2.7 Modeling ARINC653 systems using AADL . . . 98
4.2.8 Modeling AADL in BIP . . . 98
4.2.9 Modeling AADL in Lustre . . . 101
4.3 Summary and comparison . . . 103
4.4 Conclusion . . . 104

5 From AADL components to Signal processes 105
5.1 Transformation chain . . . 107
5.2 Transformation principles . . . 107
5.3 From abstract logical time to more concrete simulation time . . . 110
5.3.1 Modeling computation latencies . . . 111
5.3.2 Modeling propagation delays . . . 112
5.3.3 Towards modeling time-based scheduling . . . 113
5.4 Thread modeling . . . 114
5.4.1 Interpretation of a thread . . . 115
5.4.2 Implementation . . . 117
5.5 Processor modeling . . . 121
5.6 Bus modeling . . . 125
5.7 System modeling . . . 127
5.8 Other components modeling . . . 130
5.8.1 Process modeling . . . 130
5.8.2 Subprogram modeling . . . 132
5.8.3 Data modeling . . . 133
5.8.4 Device . . . 134
5.9 Port and port connection modeling . . . 135
5.9.1 Port modeling . . . 135
5.9.2 Port connection modeling . . . 136
5.10 Towards AADLv2.0 . . . 139
5.11 Conclusion . . . 140

6 AADL behavior specification 141
6.1 Static Single Assignment (SSA) . . . 142
6.2 AADL transition systems . . . 143
6.2.1 States . . . 144
6.2.2 Transitions . . . 144

6.2.3 Actions . . . 146
6.3 Interpretation of transitions/actions . . . 148
6.3.1 Step 1: actions to basic actions . . . 149
6.3.2 Step 2: basic actions to SSA form actions . . . 155
6.3.3 Step 3: SSA to Signal . . . 160
6.3.4 Global interpretation . . . 162
6.4 Case study . . . 162
6.5 Conclusion . . . 165

7 Distributed simulation model generation 167
7.1 Distributed code generation in Polychrony . . . 167
7.1.1 Mapping . . . 169
7.1.2 Scheduler . . . 171
7.1.3 Adding communications . . . 173
7.2 An example . . . 175
7.2.1 System description . . . 177
7.2.2 Modeling and Distributing the example in Signal . . . 177
7.3 Conclusion . . . 181

IV Validation 183

8 Validation of GALS systems 185
8.1 Formal verification . . . 186
8.1.1 Case study of a dual flight guidance system . . . 187
8.1.2 FGS Modeling in AADL . . . 188
8.1.3 Interpreting the model in Signal . . . 190
8.1.4 Checking safety properties with Sigali . . . 190
8.2 Simulation . . . 193
8.2.1 Case study of a door management system . . . 193
8.2.2 System Modeling in AADL . . . 194
8.2.3 Interpreting the model in Signal . . . 196
8.2.4 Other models and system integration . . . 197
8.2.5 Profiling . . . 199
8.2.6 VCD-based simulation . . . 200
8.3 Conclusion . . . 201

Conclusion 203

Bibliography 207

A SDSCS example 219
A.1 AADL specifications . . . 219
A.2 Signal specifications . . . 226

B FGS example **229**
B.1 AADL specifications 229
B.2 Signal specifications 231

Introduction

This thesis aims at developing a methodology to model and verify globally asynchronous locally synchronous (GALS) systems in an integrated modular avionics (IMA) design framework. This methodology consists of composing the synchronous models of individual processes according to the asynchronous model of an architecture. The resulting model can be simulated and verified using multi-clocked synchronous toolkits such as the Polychrony environment. Translations of heterogeneous models to polychronous models have been designed and implemented.

The modeling of real-time embedded systems

An embedded system [151, 48] is a computing system designed to perform dedicated functions often with real-time computing constraints. It is embedded as part of a complete device including electronic parts and mechanical parts. Nowadays, embedded systems can be found everywhere in our daily life and are often used to perform safety critical functions [105, 77] in domains such as avionics, automobile and telecommunications.

A real-time embedded system [150, 76] is one whose actions are subject to precise timing deadlines. A real-time system responds to periodic and sporadic input events (from, e.g., sensors) by timely calculating and performing output actions (on, e.g., actuators). Failing to respect deadlines may compromise correctness and have severe consequences [108]. One common example real-time software is that embedded within the ABS of modern cars. While the car is braking, the ABS periodically monitors speed (of the car, of the wheel) and may sporadically command to release the brake in order to prevent the wheel to lock while the car is not stopped. In other words, a real-time embedded system [96, 103, 150] is defined as a system whereby the correctness of the system depends not only on the logical result of computations, but also on the time at which the result is produced [150].

Examples of real-time embedded systems are aircraft engine control systems, nuclear monitoring systems and medical monitoring equipment, in which many embedded components (like the ABS) are operating and communicating in real-time. They are becoming more and more complex. This increasing complexity presents new challenges for system design: embedded systems can be constrained in time but also in size, power consumption or cost.

Model-based engineering enables the designers to deal with these concerns using the architecture description of the system as the main axis during the design phase.

Some of the classical design methodologies for real-time embedded systems are: structured analysis and design methods [160], object-oriented analysis and design meth-

ods (UML [30]) and formal system specification and design methods (SDL [32], Estelle [156, 79]).

Defining the architecture of the system before its implementation enables the analysis of the constraints imposed on the system from the beginning of the design cycle until the final implementation. Formal languages have been developed to support such architectural descriptions. They are known as architectural description languages (ADLs [129, 128, 131, 31, 4, 141]), modeling the system as a set of components and the interactions among them. One of the main ADLs currently used in avionics industry in system design for embedded systems is AADL (Architecture Analysis and Design Language) [1, 28, 41, 80].

AADL is developed as a new methodology for embedded system design, which was proposed around 2004. In AADL, the general purpose computing hardware, such as memories, processors, buses, etc., can all be modeled by software, which provides more flexibility. This modeling aspect of system design activity is becoming increasingly essential, since it allows prototyping and experiments without necessarily having a physical implementation of the system. This gives AADL a higher flexibility in design choices, lower cost, earlier decisions and fast to be adapted to new applications. Moreover, component-based approaches provide a way to significantly reduce overall development costs through modularity and re-usability. Thanks to this technology, a single AADL model can specify and analyze real-time embedded and high dependability systems, and map software onto computational hardware elements.

The AADL can model locally synchronous systems as well as asynchronous systems. The synchronous pattern consists of periodic threads which are logically simultaneous at every global real time. While for a globally asynchronous system, there are multiple reference times, for example, the threads executed on different clock processors, they represent different synchronization domains. This Globally Asynchronous Locally Synchronous (GALS) [71] model can be reflected in AADL by multiple synchronization domains and the asynchronous communications across synchronization domains.

From synchronous to GALS systems

The synchronous model

The synchronous assumption [50] was proposed in the late 80s for modeling, specifying, validating and implementing reactive and real-time system applications in an efficient and convenient way.

A synchronous model [52, 51, 93] follows the basic assumptions: first, the computations and internal communications are abstracted as instantaneous actions: they have a zero logical duration. Second, logical time is presented as a succession of events: it is handled according to a partial order over classes of simultaneous events; there is no explicit reference to a notion of physical time.

The synchronous model has had major successes due to several advantages: synchronous languages have formal and clear semantic definition, synchronous parallel composition reduces programming complexity and is useful for structuring programs, and many verification methods have been developed on the synchronous framework.

A synchronous system is viewed through the chronology and simultaneity of observed events during its execution. This is a main difference from classical approaches where the system execution is rather considered under its chronometric aspect (i.e., duration has a significant role). The mathematical foundations of the synchronous approach provide formal concepts that favor the trusted design of embedded real-time systems [88].

Typical synchronous languages are: Signal [115, 54, 90, 114], Lustre [92, 95] (data-flow synchronous languages), and Esterel [55, 162] (state based language). The three languages are built on a common mathematical framework that combines synchrony (i.e., time advances in lockstep with one or more clocks) with concurrency (i.e., functional concurrency) [52]. These synchronous languages benefit from the simplicity of the synchronous assumption.

The asynchronous model

An asynchronous system [145, 46, 91] is one to which a request is sent out and does not need to wait for a response. If a response is generated, notification from the system is received once it is complete. No global clock exists in the asynchronous paradigm.

An asynchronous system is represented by a program, which consists of a denumerable number of tasks. Under the control of a particular device, e.g., real-time operating system, these tasks run concurrently to achieve the system functions. The temporal logic of the system behavior is strongly influenced by the execution platform. The length of the logical execution of the task is unknown, which induces non-determinism [83].

In contrast to the synchronous model, the asynchronous model has the following characteristics [51]:

1. Reactions (programs progress via an infinite sequence of reactions) can not be observed any more. Since no global clock exists, global synchronization barriers which indicate the transition from one reaction to the next one are no more observable.

2. Composition occurs by means of interleaving flows shared between two processes.

This paradigm is much closer to distributed architectures than synchrony. It only requires communication channels to respect the condition that an ordered stream of sent data reaches its destination in the same order. In the domain of distributed systems, asynchronous languages (e.g., SDL [32], Ada [5]) are naturally and variously used. An asynchronous model may be used simply for the interfacing of a synchronous system to its environment and to other synchronous systems, or possibly for more complete applications. Examples of asynchronous systems are distributed computer networks and I/O systems for conventional computers.

The GALS model

The synchronous model turns out to be difficult to satisfy certain embedded systems' requirements, especially large distributed real-time systems. Providing a fully synchronized clock over multiple distributed nodes may make the model synchronization very

expensive and actually infeasible. Thus, a combination of synchronous and asynchronous design patterns is required.

Recent approaches introduce a growing amount of asynchrony, a mix of synchronous and asynchronous design patterns. This situation has been known as GALS (Globally Asynchronous Locally Synchronous [71, 134, 66]), which was proposed by Chapiro in 1984 [71]. Gathering the benefits of both the synchronous and asynchronous approaches, the GALS model is emerging as an architecture of choice for implementing complex specifications in both hardware and software. It is composed of several independent synchronous components which operate with their own local synchronous clocks, and connected through asynchronous communication schemes. In circuit design, it relates to the modeling of small synchronous blocks communicating asynchronously. In software design, it relates to finite automata that communicate with registers.

The main feature of these systems is the absence of a global timing reference and the use of several distinct local clocks, possibly running at different frequencies. Thus, unlike for a purely asynchronous design, the existing synchronous tools can be used for most of the development process, while the implementation can exploit the more efficient asynchronous communication schemes.

The idea of the GALS approach is to combine the advantages of synchronous and asynchronous design methodologies while avoiding their disadvantages: the clock distribution in a GALS circuit can be realized easier than in a synchronous circuit. A GALS model is easier to implement than a synchronous model, particularly when different components are distributed far away or that their computation speed is very high. Another advantage of the GALS implementation, specifically in the case of embedded systems, is the electric consumption. A distributed GALS system consumes less than its equivalent synchronous system, because each component can adjust its operation speed by reporting its workload, even pause when not seeking. In a synchronous system, the components can operate at different but constant speeds. Moreover, at each cycle, each component is active even if there is no data to be processed.

Existing problems and our solution

GALS designs have emerged in the recent years in response to the above mentioned challenges and have received major attention from the system level design community [134, 66]. However, developing separately synchronous software components and deploying them on the target architecture using classical design methods for asynchronous systems, it is difficult to validate the integrated system. The problem of validating the whole system is crucial: the execution of the software on the target architecture is generally asynchronous, but this phase of the design is the most error-phone. The validation can be performed by testing the implementation, however, this will result in later error detection. Further more, testing an asynchronous implementation is difficult.

In this thesis, we propose a methodology of modeling and validating of globally asynchronous composition of synchronous components in a multi-clock synchronous programming framework, Polychrony, especially in an IMA [39, 37] design architecture. A main goal of such an approach is to study properties of globally asynchronous systems using existing simulation and model-checking toolkits for the synchronous framework.

In order to support the virtual prototyping, simulation and formal validation of early, component-based, embedded architectures, we define a model of the AADL into the polychronous model of computation of the Signal programming language.

Our solution can be seen as a transformation of the design of asynchronously connected local synchronized components to a synchronous model. Since it may be non trivial to represent adequately asynchrony and non-determinism in a synchronous framework, we propose a method to use existing techniques and libraries of the Signal environment, consisting of a model of the APEX-ARINC-653 [37] real-time operating system services. It provides a suitable and adequate library to model embedded architectures in the specific case of Integrated Modular Avionics (IMA). This framework is the one considered in the TopCased [29] project.

Related works

Synchronous modeling of asynchronous systems is also studied in [94]. The authors define a generic semantic model for synchronous and asynchronous computation, after that, the attention is focused on implementing communication mechanisms.

There are also some tools to modeling non-synchronous systems using synchronous languages and developing system level design methodology:

- [124] relies on the MARTE [14] Time Model and the operational semantics of its companion language CCSL [142], to equip UML activities with the execution semantics of an AADL specification. It investigates how MARTE can be made to represent AADL periodic/aperiodic tasks communicating through event or data ports, in an approach to end-to-end flow latency analysis.

- AADL2Fiacre [56] deals with the transformation of AADL models into Fiacre [56] models to perform formal verification and simulation.

- AADL2BIP [138] models AADL data communication with BIP (Behavior Interaction Priority [7]). It focuses on deterministic data communication, and shows how BIP deals with the modeling of immediate and delayed data communications supporting undersampling and oversampling of AADL.

- [140] proposes a formal semantics for the AADL behavior annex using Timed Abstract State Machine (TASM [135]). A semantics of AADL execution model is given, and a prototype of behavior modeling and verification is proposed.

- AADL2SYNC [3] is an AADL to synchronous programs translator, which is extended in the framework of the European project ASSERT, resulting in the system-level tool box translating AADL to LUSTRE.

Polychrony

Polychrony is a framework based on Signal, a domain-specific, synchronous data-flow language dedicated to embedded and real-time system design [113]. While being declarative like Scade or Lustre, and not imperative like Esterel [72], its multi-clocked model

of computation (MoC) stands out by providing the capability to design systems where components own partially related activation clocks. This polychronous MoC is called *polychrony* [115].

The main characteristic of Polychrony is that it can be used to describe systems that contain components, which work at different or even independent clocks. Instead of requiring the user to define a global clock, Polychrony calculates clock trees, that result from the different clock dependencies expressed in the description.

Polychrony provides models and methods for a rapid, refinement-based, integration and a formal conformance-checking of GALS architecture. It contains tools for properties verification, and integrates the Signal compiler, which is able to generate sequential code or distributed code in some conditions, and it also includes a graphical user interface for Signal. It goes beyond the domain of purely synchronous circuits to embrace the context of architectures consisting of synchronous circuits and desynchronization protocols.

In the Polychrony workbench, time is represented by partially ordered synchronization and scheduling relations, to provide an additional ability to model high-level abstractions of systems paced by multiple clocks: locally synchronous and globally asynchronous systems. This gives the opportunity to seamlessly model heterogeneous and complex distributed embedded systems at a high level of abstraction, while reasoning within a simple and formally defined mathematical model.

In Polychrony, design can proceed in a compositional and refinement-based manner by first considering a weakly timed data-flow model of the system under consideration, and then providing expressive timing relation to gradually refine its synchronization and scheduling structure to finally check the correctness of the assembled components. Signal favors the progressive design of correct by construction systems by means of well-defined model transformations, that preserve the intended semantics of early requirement specifications to eventually provide a functionally correct deployment on the target architecture.

As mentioned earlier, the multi-clock polychronous model differs from other synchronous specification models by its capability to allow the design of systems, where each component holds its own activation clock as well as single-clocked systems in a uniform way. A great advantage is its convenience for component-based design approaches that allow modular development of increasingly complex modern systems.

Polychrony provides the ability to model and build GALS systems in a fully synchronous design framework, and deploy it on an asynchronous network preserving all properties of the system proven in the synchronous framework. Thanks to the polychronous approach, bounded FIFOs [86] are provided for communications between synchronous components, allowing to find a desynchronizing protocol to formally investigate the behavior of synchronous components in an asynchronous environment [133, 75].

Contribution

This thesis focuses on the modeling and validation of AADL systems in a multi-clock synchronous programming framework, Polychrony, especially in a IMA design architecture. The synchronous modeling and high-level validation, which are based on the synchronous language Signal, are the main contributions of this thesis.

We define a translation based on the semantics of AADL into the polychronous model

of computation of the Signal programming language. This solution can be seen as a transformation of the design of asynchronously connected local synchronized components to a synchronous model.

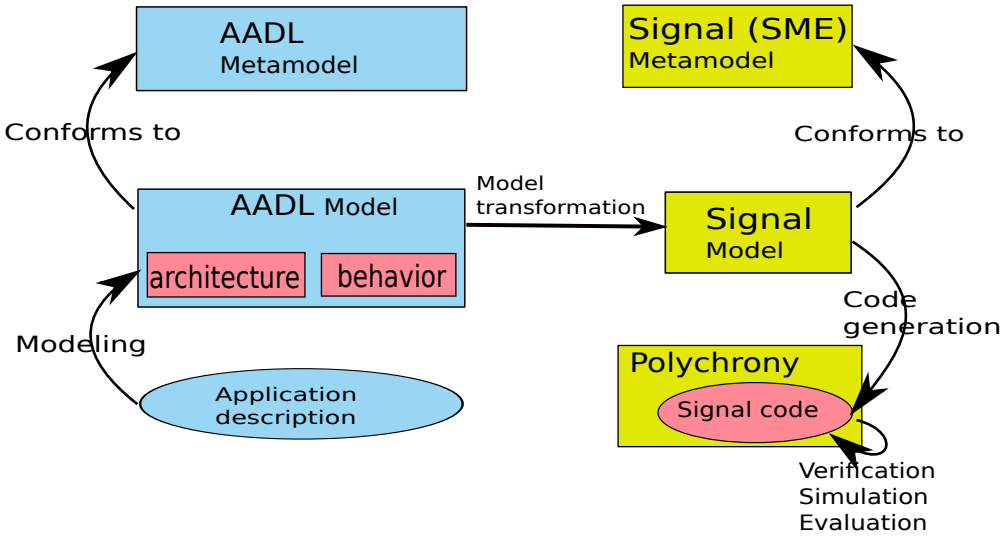


Figure 1: A global view of the approach

A global view of the transformation is illustrated in Figure 1. It starts from using the AADL language to model real-time applications, which may be represented from an initial textual description. Then by giving the semantics of AADL components in the polychronous model of computation, the AADL model is transformed into Signal model. Finally, Signal code is generated, and verification, simulation and other analyses are performed.

Since there are some difficulties in representing asynchrony and non-determinism in a synchronous framework, we propose to use existing techniques and libraries of the Signal environment [84, 85, 87], consisting of a model of the APEX-ARINC-653 [37] real-time operating system services. Consequently, the components can be translated into or projected onto different APEX-ARINC services.

AADL has been extended to describe complex behaviors without introducing external source code, by the Behavior Annex. To implement a complete system specification, we also translate the behaviors into synchronous equations. We formalize the semantics of the actions and transitions, and interpret them using SSA as an intermediate formalism [61, 154]. In this interpretation, we show not only how to translate the core imperative programming features into Signal equations, but also extend it to the mode automata that control the activation of such elementary transitions and actions.

Signal-Meta Under Eclipse (SME) [24] is the model-driven front-end of Polychrony framework. It provides the meta-model on which Signal model is based. The resulting Signal model generated from this transformation conforms to the SME meta-model. High-level validation is then carried out with the generated Signal code in order to check the correctness of the corresponding AADL specifications.

The implementation of the transformation has been carried out in the framework of OpenEmbeDD [17], which is an Eclipsed-based “Model Driven Engineering” platform dedicated to embedded and real-time systems.

All the preceding studies have been experimented with the Polychrony toolset. A case study of door management system is finally illustrated, with emphasis on the high-level modeling through AADL, and the simulation and formal validation through the tools associated with the Signal language.

Outline

In the present document, we propose a methodology to model and verify a GALS system in a polychronous framework. This method transforms the AADL model using the IMA architecture to Signal model. This dissertation has four parts:

Part I

Part I gives an extended abstract in French.

Chapter 1. The first chapter presents an extended abstract of this thesis in French. Each section presents an abstract of the corresponding chapter.

Part II

Part II presents the background. It has two chapters.

Chapter 2. This chapter introduces the AADL language. AADL is an SAE standard aimed at high level design and evaluation of the architecture of embedded systems. We introduce the components and behaviors of AADL. The IMA architecture and the ARINC standard, especially the ARINC653 specification which defines an APplication EXecutive (APEX) for space and time partitioning, are presented in this chapter.

Chapter 3. Chapter 3 is related to the Signal language. It first introduces the model of Signal. It starts with an introduction of polychronous model of computation, and then continues with a specification of the syntax and semantics of the Signal language. The modeling library of IMA ARINC applications in Signal is also presented.

Part III

Part III exhibits our contributions for modeling, prototyping and implementation, which includes four chapters.

Chapter 4. Chapter 4 reviews related works. It proposes an overview of existing AADL models and transformations. A brief comparison of these works is given in this chapter.

Chapter 5. Chapter 5 describes the modeling of AADL specifications into the polychronous model using APEX-ARINC services in the IMA framework. The general modeling approach and transformaton principles are first presented. Then the problems in AADL to Signal transformation are depicted, and solutions are given. The transformation of a subset of AADL components is described, including the main executable and schedulable component, the thread, the component responsible for scheduling, the processor, and communication components, bus, etc. Apart from the component modeling, two types of data port communication are also represented in Signal.

Chapter 6. In Chapter 6, we present the transformation of AADL behavior specification, mainly for the transitions and actions. This interpretation uses SSA as an intermediate formalism. It gives a thorough description of an inductive SSA transformation algorithm across transitions and actions that produces synchronous equations.

Chapter 7. Chapter 7 focuses on the distribution. The principles that generate distributed Signal from the AADL specification are presented. Then the scheduler and communication between the distributed programs are described. A case study is given to explain the distribution.

Part IV

Part IV contains one chapter about verification and simulation.

Chapter 8. Chapter 8 presents the verification and simulation of GALS systems in the case of our considered approach from AADL. We present the static resolution and dynamic model-checking techniques that come with the Polychrony environment to verify the functional requirements of the system. A case study of dual flight guidance system is presented, which gives formal verification. Then another case study of door management system is given, which illustrates the implementation of the proposed transformation of AADL and the simulation.

Conclusion. We conclude the thesis and discuss some perspectives of these works.

Part I

Résumé en français

Chapter 1

Résumé en français

1.1 Introduction

Les systèmes embarqués se rencontrent aujourd'hui partout, y compris dans la vie quotidienne, et sont souvent utilisés pour effectuer des tâches critiques [105, 77] dans des domaines tels que l'avionique, l'automobile et les télécommunications. Un système embarqué temps réel [150, 76] est un système dont les actions sont assujetties à des délais temporels. Un système temps réel répond à des événements d'entrée périodiques ou sporadiques (par exemple, à partir de capteurs) par le calcul et l'exécution d'actions permettant de produire des sorties (par exemple, sur des actionneurs). Un défaut de respect des délais peut compromettre l'exactitude des résultats et avoir des conséquences graves (temps réel strict) [108].

Le langage AADL (Architecture Analysis and Design Language) est proposé comme support à une nouvelle méthodologie pour la conception de systèmes embarqués. En AADL, la partie matérielle, tel que mémoires, processeurs, bus, etc, peut être modélisée sous forme logicielle, ce qui offre une plus grande souplesse car cela permet des prototypages et expérimentations sans nécessairement disposer d'une implantation physique du système. En outre, les approches à base de composants, comme c'est le cas en AADL, fournissent un moyen de réduire considérablement les coûts de développement via la modularité et la réutilisation de composants.

Le modèle synchrone a obtenu des succès majeurs dans les systèmes temps réel en raison d'un certain nombre d'avantages : les langages synchrones sont basés sur une définition sémantique précise et claire ; la composition parallèle synchrone réduit la complexité de programmation et est utile pour structurer les programmes ; et de nombreuses méthodes de vérification ont été développées dans le cadre synchrone.

Les approches les plus récentes tendent à introduire dans les modèles un nombre croissant d'éléments asynchrones, tout en essayant de préserver les bonnes propriétés des modèles synchrones. On obtient ainsi un mélange de modèles de conception synchrones et asynchrones, comme par exemple le modèle GALS (Globalement Asynchrone Localement Synchrone [71, 134, 66]). Rassemblant les avantages des deux approches, synchrone et asynchrone, le modèle GALS est en train de devenir une architecture de choix pour la mise en œuvre de spécifications complexes à la fois matérielles et logicielles.

Les approches GALS sont l'objet d'une attention particulière de la part de la communauté des concepteurs au niveau système [134, 66]. Le problème de la validation de

l'ensemble du système est essentiel : l'exécution du logiciel sur l'architecture cible est généralement asynchrone, mais cette phase de la conception est la plus sujette aux erreurs. La validation peut être effectuée au moyen de tests de la mise en œuvre, cependant, cela implique une détection tardive des erreurs. De plus, tester une implémentation asynchrone pose des problèmes de non déterminisme.

Dans cette thèse, nous proposons une méthodologie de modélisation et de validation de la composition globalement asynchrone de composants synchrones. Nous nous plaçons dans le cadre de la programmation synchrone multi-horloge, ou *polychrone*, en considérant plus particulièrement les architectures de type IMA [39, 37]. Un des principaux objectifs de l'approche est d'étudier les propriétés des systèmes globalement asynchrones en utilisant des outils de simulation et de vérification de modèle (*model-checking*) existant dans le cadre synchrone. Pour permettre le prototypage virtuel, la simulation et la validation formelle au plus tôt des architectures embarquées à base de composants, nous définissons une modélisation de AADL dans le modèle de calcul polychrone du langage de programmation Signal.

Notre solution peut être considérée comme une transformation d'une conception en composants localement synchronisés connectés de façon asynchrone, vers un modèle synchrone. Une vision globale de la transformation est illustrée sur la figure 1.1. On utilise en premier lieu le langage AADL pour modéliser une application temps réel, qui peut être représentée à partir d'une description textuelle initiale. Étant donné la sémantique des composants AADL dans le modèle de calcul polychrone, le modèle AADL est alors transformé en modèle Signal. À partir de ce modèle, un programme Signal est généré, et l'outil Polychrony peut être utilisé pour effectuer de la vérification, des simulations et d'autres analyses.

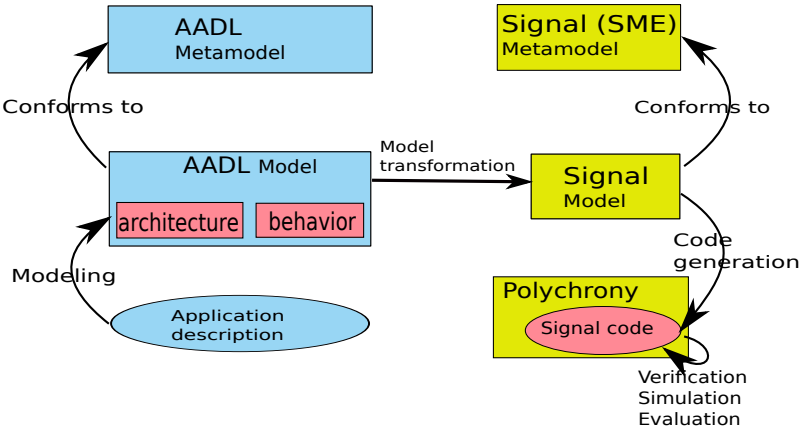


Figure 1.1: Vision globale de l'approche.

Sachant qu'il peut être non trivial de représenter de façon adéquate l'asynchrone et le non déterminisme dans un cadre synchrone, nous proposons une méthode qui utilise des techniques et des bibliothèques existant dans l'environnement de Signal, consistant notamment en un modèle des services du système d'exploitation temps réel APEX-ARINC-653 [37]. Nous disposons ainsi d'une bibliothèque permettant de modéliser des architectures embarquées dans le cadre IMA.

1.2 Introduction au langage AADL et aux architectures avioniques

1.2.1 Abstractions du langage AADL

Le langage AADL est une norme de la SAE, qui a été défini pour les systèmes embarqués critiques temps réel. Il doit rendre possible l'utilisation d'approches formelles diverses pour l'analyse de systèmes constitués de composants matériels et logiciels.

Catégories de composants Pour modéliser les systèmes embarqués complexes, AADL fournit trois catégories de composants : composants logiciels, composants de plate-forme d'exécution et composants "composites".

- Les données, les sous-programmes, les "*threads*"[1] et les processus représentent collectivement le logiciel d'une application ; ils sont appelés composants logiciels.
- Les composants de la plate-forme d'exécution comprennent les processeurs, les dispositifs (*device*), les mémoires et les bus. Ils supportent l'exécution des threads, le stockage des données et du code, et la communication entre les threads.
- Les systèmes sont appelés composants composites. Ils permettent d'organiser en structures hiérarchiques avec des interfaces bien définies les composants logiciels et les composants de la plate-forme d'exécution.

Types et implémentations de composants Les composants AADL sont définis par des déclarations de type et d'implémentations. Un type de composant représente l'interface fonctionnelle du composant et ses attributs observables de l'extérieur. Une implémentation décrit le contenu du composant, spécifie sa structure interne en termes de sous-composants, les connexions entre les éléments de ces sous-composants, les flots sur une séquence de sous-composants, les modes permettant de représenter les états de fonctionnement, et les propriétés.

Propriétés Une propriété spécifie des caractéristiques du composant qui s'appliquent à toutes les instances de ce composant, sauf si elles sont redéfinies dans des implémentations ou des extensions. Des valeurs peuvent être attribuées aux propriétés par des déclarations d'association de propriété.

Domaine temporel Trois horloges (t_d, t_s, t_f) sont associées à chaque thread t. L'horloge t_d est l'horloge à laquelle le thread est "dispatché". Si le port prédéfini **dispatch** est connecté, l'arrivée d'un événement sur ce port entraîne le dispatch du thread. L'horloge t_s est l'horloge à laquelle le thread démarre, et t_f est l'horloge à laquelle le thread se termine. Ces deux horloges peuvent être spécifiées respectivement par les propriétés **Input_Time** et **Output_Time**. De nombreuses autres caractéristiques temporelles, par exemple l'échéance, le temps d'exécution, le temps de transmission, etc., peuvent être spécifiées par les propriétés définies dans AS5506 [41].

[1] Nous gardons ici le terme anglais plutôt que "fil d'exécution" ou "tâche".

1.2.2 Annexe comportementale de AADL

L'annexe comportementale (*Behavior Annex*) de AADL [74] est une extension au noyau du standard qui fournit un moyen de spécifier le comportement fonctionnel local des composants. Elle permet une description précise des comportements, tels que les calculs, les communications au moyen des ports, etc. Une annexe comportementale peut être attachée à un thread ou à un sous-programme. Les threads et les sous-programmes démarrent à partir d'un état initial, et une transition vers un état final (resp., un état de retour) termine le thread (resp. le sous-programme). Les transitions peuvent être gardées par des conditions, et des actions peuvent y être attachées.

1.2.3 Architecture avionique et ARINC 653

Architecture IMA Une caractéristique forte des architectures IMA [39] est le fait que plusieurs applications avioniques peuvent être hébergées sur un seul système, partagé. Ces applications sont assurées d'une allocation statique sécurisée des ressources partagées, de sorte qu'aucune propagation de faute ne se produise d'un composant à un autre. Ce problème est adressé au moyen d'un *partitionnement* fonctionnel des applications selon la disponibilité des ressources en temps et en mémoire [37].

Une *partition* est composée de *processus* qui représentent les unités d'exécution. Chaque *processus* est caractérisé par des informations utiles à l'*OS de niveau partition*, responsable de l'exécution correcte des *processus* dans une *partition*. Des mécanismes appropriés sont prévus pour la communication et la synchronisation entre *processus* (par exemple, *buffers, événements, sémaphores*) et entre *partitions* (par exemple, *ports* et *canaux*).

La norme avionique ARINC définit les principes de base du partitionnement, ainsi qu'un ensemble de services, conformes à l'architecture IMA. L'interface APEX, définie dans la norme ARINC, comprend les services de communication et de synchronisation, et les services de gestion des *processus* et des *partitions*.

AADL et ARINC AADL peut être utilisé en particulier pour une modélisation conforme à ARINC. AADL et ARINC ont certaines caractéristiques similaires. La *partition* ARINC est proche du processus AADL. Le processus AADL représente un espace d'adressage protégé, un espace de partitionnement où une protection est assurée contre les accès d'autres composants à l'intérieur du processus. Ainsi, la *partition* ARINC et le processus AADL sont des unités du partitionnement. Les mécanismes de communication définis dans ARINC supportent les communications de messages en file et sans file. Cela est similaire aux connexions AADL. Une différence est que ARINC ne prend pas en compte, à ce niveau, la connexion matérielle par bus entre les différents composants.

1.3 Langage Signal et modélisation ARINC en Signal

L'approche synchrone est l'une des solutions possibles pour une conception sûre des systèmes embarqués. Le modèle multi-horloge ou polychrone se distingue des autres modèles synchrones par son cadre uniforme. Il permet aussi bien la conception de systèmes

dans lesquels chaque composant dispose de sa propre horloge d'activation, que de systèmes mono-horloge. Cette caractéristique rend la sémantique de Signal plus proche de la sémantique de AADL que ne l'est celle d'autres modèles purement synchrones ou asynchrones. Cela facilitera ainsi la validation du système.

1.3.1 Le langage Signal

Le langage flot de données synchrone Signal est dédié à la conception de systèmes embarqués dans des domaines d'applications critiques. Les caractéristiques propres au modèle relationnel qui sous-tend Signal sont d'une part de fournir la notion de polychronie — c'est-à-dire la capacité à décrire les circuits et les systèmes avec plusieurs horloges — et d'autre part de permettre le raffinement — autrement dit, la capacité à permettre la conception de système depuis les étapes initiales de spécification des exigences, jusqu'aux étapes finales de synthèse et de déploiement.

Signal est un langage relationnel qui s'appuie sur le modèle polychrone [58]. Signal manipule des suites non bornées de valeurs typées $(x_t)_{t\in\mathbb{N}}$, appelées *signaux*, notées $\mathbf{x}$ et indexées implicitement par les valeurs discrètes de leur horloge, notée $\hat{x}$. À un instant donné, un signal peut être présent ou absent. Deux signaux sont dits synchrones s'ils sont toujours présents (ou absents) aux mêmes instants.

En Signal, un processus (dénoté P ou Q) consiste en la composition synchrone (notée $P \mid Q$) d'équations sur signaux (notées $x := y\ f\ z$). Une équation $x := y\ f\ z$ définit le signal de sortie x par la relation sur ses signaux d'entrée y et z auxquels est appliqué l'opérateur f. Le processus P/x restreint la portée du signal x au processus P. La syntaxe abstraite d'un processus P en Signal est définie comme suit :

$$P, Q \ ::= \ x := y\ f\ z \mid P \mid Q \mid P/x$$

Domaines temporels et communications en Signal Nous discutons ici certains écarts et similitudes sémantiques entre AADL et Signal. Signal peut fournir un outillage à base d'opérateurs dérivés permettant de réduire les écarts.

- **Outillage d'horloges**. Les horloges peuvent être étroitement liées aux domaines temporels AADL et des opérateurs sont définis en Signal pour manipuler les horloges.

- **Horloges périodiques en Signal**. Des horloges périodiques peuvent être spécifiées en Signal à l'aide de relations affines sur les horloges. Dans Polychrony, le calcul d'horloges de Signal implémente des règles de synchronisabilité basées sur les propriétés des relations affines, et vis-à-vis desquelles les contraintes de synchronisation peuvent être examinées.

- **Communications**. Le principe de base de la communication en Signal est la diffusion, ce qui signifie qu'un signal donné est transmis comme plusieurs signaux identiques. Il est également possible en Signal d'avoir plusieurs signaux ou expressions associés à un signal donné, au moyen de définitions partielles.

- **Outillage pour le report des communications**. Le report des communications peut être implémenté en Signal à l'aide de cellules mémoire et de FIFO.

- **Outillage pour briser l'atomicité**. Pour un programme Signal, l'horloge la plus rapide d'un processus n'est pas toujours une horloge d'entrée : des instants nouveaux peuvent être insérés entre les instants existants. Le sur-échantillonnage permet la spécification de contraintes entre les entrées et les sorties de telle sorte qu'il ne puisse pas se produire de valeur d'entrée supplémentaire tant que les contraintes en question ne sont pas respectées par les calculs (intermédiaires) de la sortie [115].

1.3.2 Modélisation de concepts ARINC en Signal

Les applications avioniques qui s'appuient sur la norme avionique ARINC 653, basée sur l'architecture IMA, peuvent être spécifiées dans le modèle de Signal. Une bibliothèque de services APEX ARINC est fournie en Signal. L'environnement de conception Polychrony comprend ainsi une bibliothèque Signal de composants correspondant aux services d'exécutif temps réel définis dans ARINC [37]. Cette bibliothèque, conçue par Abdoulaye Gamatié [83], s'appuie sur quelques blocs de base [84, 85], qui permettent de modéliser les *partitions* : il s'agit des services APEX-ARINC-653, d'un modèle de RTOS, et d'entités d'exécution.

Services de l'APEX Les services de l'APEX modélisés en Signal comprennent les services de communication et de synchronisation utilisés par les *processus* (par exemple, *SEND_BUFFER*, *WAIT_EVENT*, *READ_BLOCKBOARD*), les services de gestion des *processus* (par exemple, *START*, *RESUME*), les services de gestion des *partitions* (par exemple, *SET_PARTITION_MODE*), et des services de gestion du temps (par exemple, *PERIODIC_WAIT*).

OS de niveau partition Le rôle de l'*OS de niveau partition* est d'assurer l'exécution correcte des *processus* dans une *partition*. Chaque fois que la *partition* s'exécute, l'*OS de niveau partition* sélectionne un *processus* actif dans la *partition*.

***Processus* ARINC** La définition d'un modèle de *processus* ARINC en Signal prend en compte d'une part sa partie calcul et d'autre part sa partie contrôle. Le sous-composant CONTROL spécifie la partie contrôle du *processus*. Il s'agit d'un système de transition qui indique quelles instructions doivent être exécutées lorsque le modèle du *processus* est actif. Le sous-composant COMPUTE décrit les actions effectuées par le *processus*. Il est composé de *blocs* d'actions. Ces *blocs* représentent des pièces élémentaires de code à exécuter sans interruption. Les calculs associés à un *bloc* sont supposés *se terminer dans un laps de temps borné*.

Partitions Après la phase d'initialisation, la *partition* est activée (par exemple, lors de la réception d'un signal *Active_partition_ID*). L'*OS de niveau partition* sélectionne un *processus* actif dans la *partition*. Ensuite, la sous-partie CONTROL de chaque *processus* vérifie si le *processus* concerné peut s'exécuter. L'exécution du modèle de la *partition* suit ce schéma de base jusqu'à ce que l'*OS de niveau module* sélectionne une nouvelle *partition* à exécuter.

1.4 Travaux reliés : formalisations de AADL

Le langage AADL fournit un bon support pour la description et l'analyse de systèmes embarqués complexes. Afin de valider des propriétés formelles sur un modèle AADL, d'effectuer des analyses d'ordonnançabilité ou de performance, d'effectuer de la vérification, un cadre formel qui puisse fournir des diagnostics sur le système doit être défini et utilisé. Un tel objectif ne peut être atteint que si l'on peut transformer le modèle AADL dans un autre modèle, dont les outils associés offrent ces fonctionnalités. Nous donnons ici un bref apercu de quelques transformations de AADL.

- L'utilisation de MARTE pour modéliser AADL a concerné principalement les deux protocoles de communication, immédiat et retardé, dans l'optique d'une analyse de la latence sur les flots "*end-to-end*". Ce travail s'efforce de construire un simulateur générique pour AADL, mais l'objectif d'un langage analysable formellement reste une perspective [119].

- La modélisation de AADL en Fiacre se concentre sur un objectif de vérification de modèle [56]. Par rapport à BIP, Fiacre a des constructions moins puissantes mais il a de bonnes propriétés au niveau de la compositionnalité et du temps réel.

- La traduction de AADL vers BIP permet la simulation de modèles AADL, ainsi que l'application de techniques de vérification. L'absence de localité dans BIP rend difficile le raisonnement compositionnel [138].

- La modélisation de AADL en TASM permet de présenter la sémantique temporelle de l'annexe comportementale de AADL en utilisant TASM. TASM dispose de mécanismes de consommation de ressources plus abstraits, mais prend difficilement en compte certains modèles d'ordonnancement [140].

- La traduction référencée de AADL vers SystemC vise la simulation et l'analyse de performances. Cependant, les spécifications de comportement n'ont pas été prises en considération [157].

- L'objet de la génération de code C depuis AADL développée dans Ocarina est de cibler des applications distribuées à haute intégrité. Avec l'outil PolyORB-HI-C, des blocs Simulink peuvent être utilisés comme sous-programmes AADL [**?**]. Une transformation de blocs Simulink (utilisés en tant que threads AADL) en Signal est également implémentée dans notre travail coopératif effectué dans le cadre du projet CESAR [35]. Nous utilisons AADL pour modéliser l'architecture et Simulink pour modéliser les parties fonctionnelles.

- L'objectif de la modélisation de ARINC653 à l'aide de AADL est de proposer un processus approprié de développement de type MDE permettant de saisir et de prendre en compte les exigences architecturales en utilisant AADL et son annexe ARINC653 [73]. Cette approche modélise des architectures ARINC653 en AADL, alors que notre travail consiste en une démarche inverse : nous modélisons un système AADL en Signal dans un cadre ARINC653.

- L'objectif principal de AADL2SYNC est d'effectuer une simulation et une validation qui prennent en compte à la fois l'architecture du système et les aspects fonctionnels. Ce travail construit un simulateur exprimé dans un langage purement synchrone : Lustre. Un protocole de communication quasi-synchrone est utilisé pour émuler l'asynchronisme et remédier ainsi aux limitations dues au caractère purement synchrone ; néanmoins sa capacité d'expression reste limitée [3, 99].

Par rapport à ces travaux, notre approche a des objectifs multiples. Nous modélisons un système AADL en Signal afin d'effectuer de la vérification formelle (en utilisant Sigali), de la simulation (en utilisant VCD), et de la génération de code C/Java, puisque le modèle polychrone du langage Signal offre en effet un support formel pour l'analyse, la vérification, la simulation et la génération de code, mises en œuvre dans la plate-forme Polychrony. En outre, le modèle polychrone fournit des modèles et des méthodes pour l'intégration rapide, basée sur le raffinement, et la vérification formelle de conformité des architectures GALS [152].

1.5 Modélisation de composants AADL en processus Signal

Cette section se concentre sur la modélisation synchrone des composants AADL dans une architecture IMA, de sorte qu'un modèle AADL puisse être traduit en un modèle exécutable du langage flot de données polychrone Signal. Cette modélisation est une contribution qui doit aider à combler le fossé existant entre modèles asynchrones et synchrones.

1.5.1 Chaîne de transformation

Notre transformation de modèles AADL en spécifications synchrones est séparée en deux étapes : d'une part, la transformation elle-même, des modèles AADL vers des modèles synchrones, puis la génération de code synchrone à partir des modèles synchrones obtenus à la première étape.

Nous formalisons la transformation de AADL en isolant les catégories syntaxiques de base qui caractérisent ses capacités expressives : systèmes, processus, threads, sous-programmes, données, dispositifs, processeurs, bus et connexions.

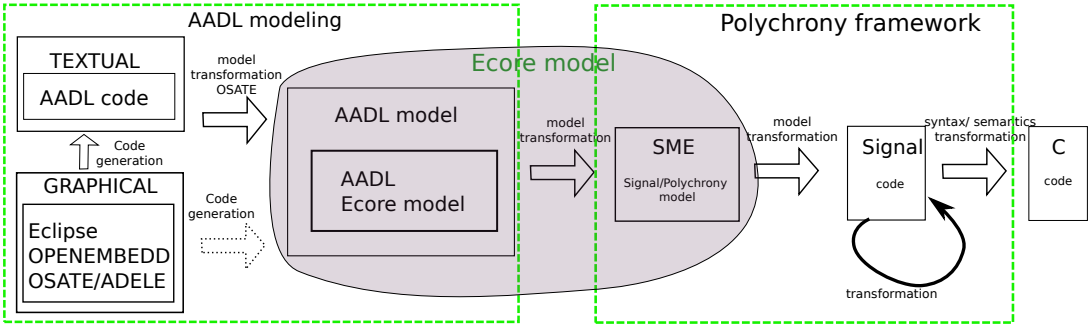


Figure 1.2: Vue globale de la chaîne de transformation AADL-SME/Signal.

La figure 1.2 illustre la chaîne de transformation semi-automatique d'un modèle AADL vers un modèle Signal, jusqu'à la génération de code exécutable, en C ou Java par exemple. Dans la transformation, les modèles SME [24], qui sont conformes au méta-modèle de Signal, sont considérés comme des modèles intermédiaires.

Le modèle ecore de AADL est traduit en un modèle SME/Signal en utilisant ATL (Atlas Transformation Language), dans lequel sont définies les règles de transformation. Le modèle SME peut ensuite être transformé en programme Signal. Le programme Signal est alors compilé et un code exécutable C (ou Java/C++) peut être généré.

1.5.2 Principes de transformation

Cette transformation de modèle est basée sur l'étude des similarités entre AADL et les services APEX-ARINC. La transformation d'un modèle AADL vers Signal repose sur l'architecture IMA [121]. Les principes de base de la transformation de base sont présentés dans la table 5.1.

AADL	**Signal**
thread	*processus* ARINC
processus	*partition* ARINC
port	FIFO bornée
connexion de ports de données	processus Signal (selon le type de la connexion)
processeur	*OS de niveau partition* en ARINC
type de données	type de données Signal
système	processus Signal constitué de sous-processus
bus	processus de communication Signal
dispositif	processus Signal

Table 1.1: Principes de base de la transformation.

Un exemple permet de donner une première idée de la transformation. Le système SDSCS (dont la description détaillée se trouve dans l'annexe A), représenté sur la figure 1.3, a un processus *doors_process* qui traite les messages provenant des dispositifs *Door1*, *Door2*, *LGS*, *DPS* et *OCU*. Le processus *doors_process* se compose de trois threads : *door_handler1*, *door_handler2* et *doors_mix*, qui calculent et génèrent les sorties.

Le modèle Signal correspondant (dans l'architecture ARINC) est montré (partiellement) sur la figure 1.4. Le processus *doors_process* est traduit en une *partition* ARINC, *partition_doors_process*. Les trois *processus* qui la composent, *process_door_handler1*, *process_door_handler2* et *process_doors_mix*, correspondent aux trois threads de la figure 1.3. L'ordonnanceur est modélisé par l'*OS de niveau partition* (*partition_level_OS*). Les dispositifs sont implémentés comme des processus Signal extérieurs à la *partition*, à laquelle ils fournissent leur interface externe.

1.5.3 Du temps logique abstrait vers un temps de simulation concret

Le paradigme synchrone fournit une représentation idéalisée du parallélisme. Tandis que AADL prend en compte les durées de calcul et les délais de communication, permettant

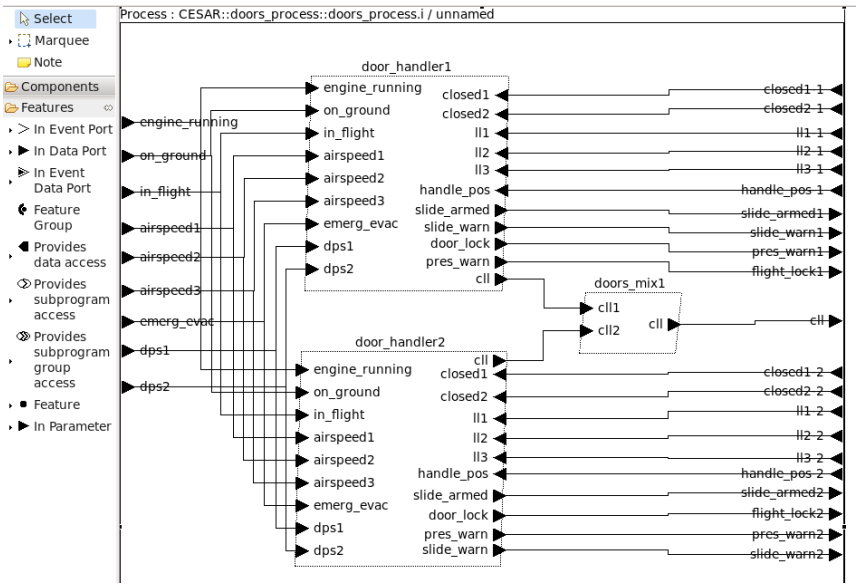


Figure 1.3: Les threads du processus *doors_process*.

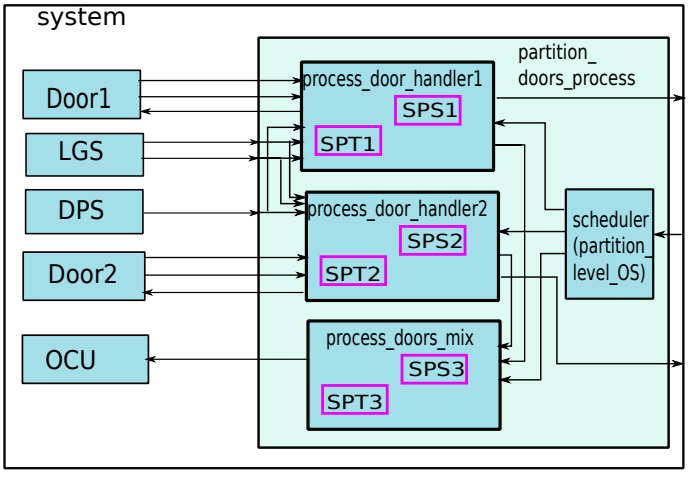
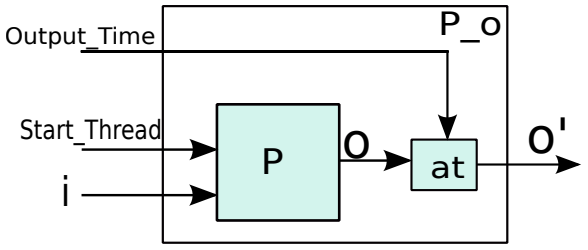


Figure 1.4: Modèle Signal du système SDSCS simplifié.

ainsi de produire des données d'un même instant logique à des instants différents dans la mise en œuvre. Ces instants sont définis précisément dans les propriétés des ports et des threads. Pour résoudre ce problème, nous conservons la vision idéale de calculs et communications instantanés, et déportons le calcul des latences et des délais de communication vers des processus spécifiques de "mémorisation", qui introduisent les retards et synchronisations requis. En conséquence, certaines propriétés entraînent la définition de signaux de synchronisation explicites. L'utilisation du cadre polychrone, adapté à la modélisation d'un temps logique abstrait, doit faire face aux problèmes décrits ci-après.

Modélisation des latences de calcul. Une caractéristique principale des programmes polychrones est l'exécution logique instantanée, par rapport au temps logique. Les composants dans le modèle polychrone ne consomment pas de temps logique : les sorties sont générées immédiatement lorsque les entrées sont reçues. Alors qu'en AADL, un thread peut exécuter une fonction ou un calcul pendant un intervalle de temps spécifié, défini par les propriétés temporelles. Par conséquent, la modélisation de AADL dans le cadre polychrone nécessite une forme d'adaptateur pour interfacer le temps logique abstrait et le temps de simulation concret.




```
P_o = defer_Output (P)

process P_o =
(? i; event Start_Thread,Output_Time;
 ! o'; )
(| o := P(i,Start_Thread)
 | o' := at(o,Output_Time)
 |)
where
   o;
end;
```

Figure 1.5: Modélisation d'une tâche consommant du temps.

À chaque sortie *o* d'un processus *P*, nous associons un processus *P_o* dont la sortie est la valeur de *o* retardée jusqu'à ce que son *Output_Time* représenté par le signal événement d'entrée *Output_Time* se produise (figure 1.5). En raison de l'ordonnancement, un processus qui est logiquement synchrone d'un signal "*dispatch*" peut être effectivement démarré plus tard. Ainsi, à chaque processus *P*, nous associons un signal événement d'entrée "*Start_Thread*" et l'exécution de *P* est synchronisée avec *Start_Thread*.

Modélisation des délais de propagation. Lors de l'exécution de programmes synchrones, chaque événement ou signal significatif est précisément daté par rapport aux autres signaux, et par rapport à la séquence des pas de calcul. Alors que pour un modèle AADL, l'instant de disponibilité des entrées peut être déterminé par différentes valeurs de propriétés.

Le langage Signal fournit des moyens d'exprimer l'activation : ce sont les horloges des signaux. L'idée principale pour modéliser le temps non précisément connu de AADL dans un cadre synchrone est d'utiliser des entrées supplémentaires, appelées conditions d'activation, pour modéliser les délais de propagation.

Une condition d'activation (pour l'instant de démarrage) peut être utilisée pour exprimer l'activation d'un thread, qu'il soit périodique ou apériodique. La modélisation

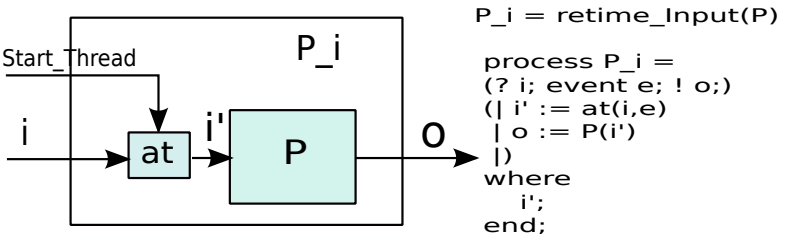


Figure 1.6: Condition d'activation.

représentée sur la figure 1.6 considère un programme synchrone P (qui est la transformation d'un thread), une entrée d'activation *Start_Thread*, et définit un nouveau programme P_i dont l'activation est conditionnée.

Vers la modélisation d'ordonnancement basé sur le temps. Un autre problème se pose à partir des mécanismes d'activation et de retard tels que décrits dans les paragraphes précédents : comment contrôler ces conditions d'activation et de retard, et d'où doivent-elles être générées ?

Pour réoudre ce problème, nous supposons qu'est associé à chaque thread P un environnement temporel SPS, qui calcule les horloges de démarrage et de terminaison du thread, ainsi que d'autres signaux de contrôle, lorsque le thread est activé par l'ordonnanceur. Quand que le thread est dispatché, les instants de démarrage et de terminaison peuvent être calculés en fonction des propriétés temporelles spécifiées.

1.5.4 Modélisation de thread

Les threads sont les principaux composants AADL exécutables et ordonnançables. Un thread Th représente une unité concurrente ordonnançable d'exécution séquentielle définie par un code source. Pour caractériser sa capacité expressive, un thread Th encapsule une fonctionnalité qui peut consister en des ports $P \in \mathscr{F}$, des connexions $C = P \times P$, des propriétś $R \in \mathscr{R}$, des spécifications de comportement T/S et des sous-programmes Su qui peuvent être invoqués par le thread.

$$Th = < P,\ C,\ R,\ Su,\ T/S >$$

La sémantique d'exécution d'un thread AADL est la suivante :

1. Lire et geler les entrées. Le contenu des données entrantes est gelé pour la durée d'exécution du thread. Par défaut, l'entrée est gelée au moment du dispatch. Si la propriété *Input_Time* est spécifiée, ce moment est déterminé par la valeur de la propriété. Toute entrée arrivant après ce gel devient disponible à l'instant d'entrée suivant.

2. Exécuter et calculer. Quand l'activité du thread entre dans l'état de calcul, l'exécution de la séquence de code source correspondant au point d'entrée du thread est gérée par un ordonnanceur.

3. Actualiser et rendre disponibles les sorties. Par défaut, une sortie est transférée vers d'autres composants à l'instant de terminaison (au moment de l'échéance en cas

de connexion de port de données retardée), ou comme spécifié par la valeur de la propriété *Output_Time*.

Étapes d'interprétation

1. Un thread AADL *Th* est d'abord traduit en un processus Signal SP, qui correspond à un *processus* ARINC, du point de vue de la structure fonctionnelle. SP a les mêmes flots d'entrée/sortie que *Th*, plus un "tick" additionnel, qui est un tick interne provenant du processeur (l'ordonnanceur). Ce tick sera utilisé dans la transformation des comportements et des actions/calculs qu'ils contiennent. Les actions effectives, représentées par la spécification du comportement, sont décrites dans la section consacrée à la spécification du comportement.

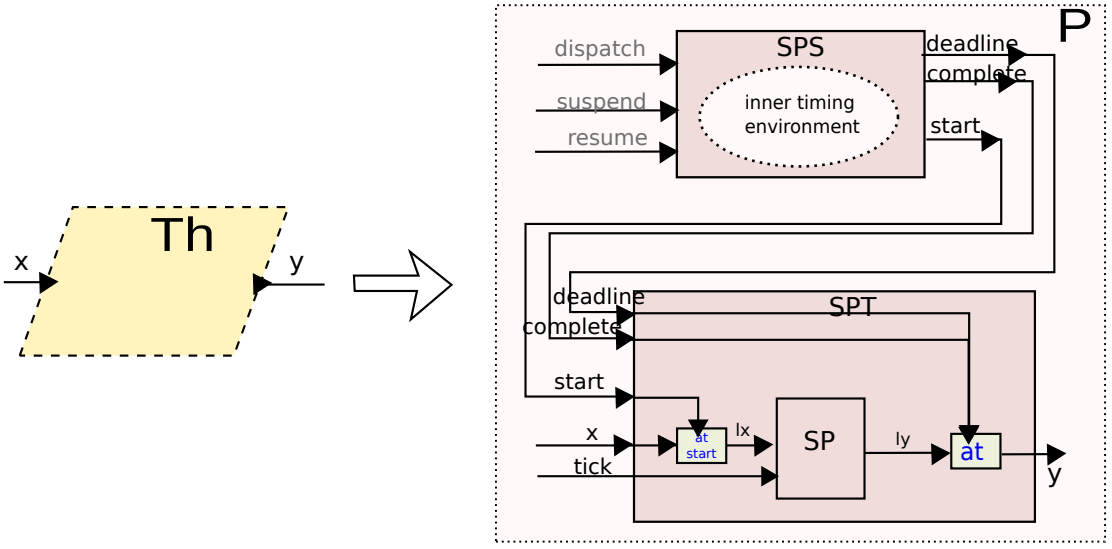


Figure 1.7: Traduction d'un thread AADL en un thread Signal.

2. En raison de la différence de sémantique temporelle entre AADL et Signal, les propriétés temporelles d'un thread AADL sont traduites par un autre processus Signal, SPT (figure 1.7), qui joue le rôle d'interface sémantique temporelle entre AADL et Signal.

   Les principales fonctions de SPT en regard de SP sont : 1) mémoriser les signaux d'entrée, et les conserver jusqu'à ce que le thread soit activé par l'événement *start*, 2) activer le *processus* Signal fonctionnel SP au moment du *start*, 3) mémoriser les sorties du thread jusqu'à la date de terminaison (lorsque l'événement *s_complete* est arrivé). La mémorisation des signaux et l'activation du thread servent ici de pont entre la sémantique des threads en AADL et le modèle synchrone.

3. L'exécution d'un thread est caractérisée par certains aspects temps réel. Un thread est ordonnancé suivant des propriétés temporelles. En raison de cette sémantique de contrôle temps réel, un nouveau *processus*, SPS (figure 1.7), est ajouté, à l'intérieur duquel les signaux de contrôle temporel sont automatiquement calculés lorsqu'il est

activé. Quand il reçoit les signaux gérant l'ordonnancement (par exemple, *dispatch*) depuis l'ordonnanceur de threads, il commence à calculer ses propres signaux temporels pour l'activation et la complétion du *processus* SPT.

Règles d'abstraction Nous donnons une description abstraite des règles de transformation d'un thread $Th = < P,\ C,\ R,\ Su,\ T/S >$ en un processus Signal (en tant que *processus* APEX ARINC). La notation $\mathcal{I}$ représente l'interprétation.

1. Un composant thread est traduit par un *processus* ARINC, paramétré par $\mathcal{I}(P)$, $\mathcal{I}(C)$, $\mathcal{I}(R)$, $\mathcal{I}(Su)$, $\mathcal{I}(T/S)$. La notation $Program^{parameters}$ est utilisée pour dénoter que les paramètres *parameters* serviront à la définition du *Program*. Le rôle de ces paramètres est expliqué plus en détail ci-dessous. La fonctionnalité du thread est traduite en deux sous-processus, SPS et SPT, à l'intérieur du *processus*.

$$\mathcal{I}(Th) = (SPS^{\mathcal{I}(R)} \mid SPT^{\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)})$$

2. Le sous-processus SPS est paramétré par les propriétés R. L'interface de SPS inclut les entrées *tick, dispatch, suspend, resume* et les sorties *start, complete, deadline*.

$$\begin{aligned} SPS = (&? \textit{ event tick},\ dispatch,\ suspend,\ resume; \\ &! \textit{ event start},\ complete,\ deadline;) \end{aligned}$$

Les propriétés $r \in R$, par exemple, *Input_Time*, *Output_Time*, sont interprétées dans SPS. Les valeurs de ces propriétés sont utilisées pour compter les ticks logiques pour les instants de démarrage et de terminaison du thread.

3. Le sous-processus $SPT^{\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)}$ est paramétré par $\mathcal{I}(P)$, $\mathcal{I}(C)$, $\mathcal{I}(Su)$, $\mathcal{I}(T/S)$. Les ports P sont traduits comme des entrées/sorties, et les connexions, sous-programmes et comportements sont traduits dans le sous-processus SPT. Le détail de ces paramètres est expliqué ci-dessous.

4. Les ports d'entrée/sortie P du thread sont traduits en entrées/sorties de Signal. L'interface du *processus* ARINC inclut ces signaux et les entrées événements *tick, dispatch, suspend, resume* reçues de l'ordonnanceur. La valeur *p.direction* désigne la direction d'un port p.

$$\begin{aligned} inputs &= \{\text{tick, dispatch, suspend, resume}\} \cup \mathcal{I}(p) \quad \forall p \in P,\ p.direction = \textbf{in} \\ outputs &= \{\mathcal{I}(p)\} \quad \forall p \in P,\ p.direction = \textbf{out} \end{aligned}$$

5. Les transitions/actions T/S sont interprétées dans un processus synchrone SP en suivant les étapes présentées dans la section suivante. Ce processus synchrone SP est encapsulé dans le sous-processus SPT. La valeur *c.type* désigne le *type* de la connexion c, qui peut être soit **immediate** ou **delayed**.

$$SPT = (IM \mid SP^{\mathcal{I}(Su),\mathcal{I}(C)} \mid OM)$$

où

$$SP^{\mathcal{I}(Su),\mathcal{I}(C)} = \mathcal{I}(T/S)$$

$$IM = (ii_{p_{k_1}} := at(\mathcal{I}(p_{k_1}),\ start) \mid ii_{p_{k_2}} := at(\mathcal{I}(p_{k_2}),\ start) \mid \dots)$$

pour tous $p_{k_j} \in P,\ t.q.\ p_{k_j}.direction =$ **in**

$$OM = (OM_{imm} \mid OM_{delayed})$$

$$OM_{imm} = (oo_{p_{k_1}} := at(\mathcal{I}(p_{k_1}),\ complete) \mid oo_{p_{k_2}} := at(\mathcal{I}(p_{k_2}),\ complete) \mid \dots)$$

pour tous $p_{k_j} \in P,\ t.q.\ \exists c \in C,\ c = (p_1, p_{k_j}),\ c.type =$ **immediate**, $p_{k_j}.direction =$ **out**

$$OM_{delayed} = (oo_{p_{k_1}} := at(\mathcal{I}(p_{k_1}),\ deadline) \mid oo_{p_{k_2}} := at(\mathcal{I}(p_{k_2}),\ deadline) \mid \dots)$$

pour tous $p_{k_j} \in P,\ t.q.\ \exists c \in C,\ c = (p_1, p_{k_j}),\ c.type =$ **delayed**, $p_{k_j}.direction =$ **out**

6. Chaque sous-programme $su \in Su$ est interprété comme un processus Signal PP_{su} :

$$\mathcal{I}(su) = PP_{su}(?\ In1, \dots Inm; !\ Out1, \dots Outn;)\ PP_{su}\_BODY$$

Il est invoqué comme une instance dans le sous-processus SP :

$$(|\ o1 := Out1 \mid \dots \mid on := Outn \mid PP_{su}\_BODY| \ In1 := i1 \mid \dots \mid Inm := im)$$

Une connexion dans un thread $c = (p_1, p_2) \in C$ relie les ports (paramètres) des processus correspondant aux sous-programmes invoqués. Une telle connexion est interprétée comme une affectation reliant les entrées/sorties des modèles de sous-programmes :

$$\mathcal{I}(p_2) := \mathcal{I}(p_1)$$

1.5.5 Modélisation des autres composants

Modélisation de processeur Un composant processeur est une abstraction du matériel et des logiciels chargés de l'exécution et de l'ordonnancement des threads. Le processeur est transformé en un ordonnanceur, qui correspond à l'*OS de niveau partition* de la *partition*.

Modélisation de bus Un composant bus représente le matériel et les protocoles de communication associés qui permettent les interactions entre les autres composants de la plate-

forme d'exécution. Pour modéliser un bus en Signal, nous considérons deux fonctions de "bufferisation" de données (deux *buffers*) qui communiquent par une fonction *connect*. Chaque buffer reçoit les données et les stocke jusqu'à ce que le bus (resp. le lecteur) les récupère. La fonction *connect* transfère les données vers le buffer du lecteur.

Modélisation de système Le système est le composant de plus haut niveau d'un modèle AADL, il représente un composite de logiciels applicatifs qui interagissent, d'une plate-forme d'exécution, et de composants système. Un système est transformé en un processus Signal, qui comporte une composition de *modules* ARINC et un ordonnanceur permettant d'activer les *modules*. Un *module* est un processus Signal qui se compose des sous-processus interprétés à partir des processus AADL (liés à un même processeur).

Modélisation de processus Un processus AADL est traduit en une *partition* ARINC (représentée comme un processus Signal). Les threads contenus dans le processus seront traduits par des *processus* ARINC à l'intérieur de la *partition*, afin qu'ils puissent accéder aux ressources restreintes à la *partition*. Un ordonnanceur interne est nécessaire pour ordonnancer ces threads. Les processus liés à un même processeur sont interprétés dans un *module* ARINC.

Modélisation de sous-programme Les sous-programmes sont des éléments pouvant être invoqués, qui fournissent une fonction de serveur aux composants qui les appellent. Un sous-programme est traduit en un processus Signal paramétré par les ports (paramètres) eux-mêmes traduits.

Modélisation de données Les types de données sont similaires structurellement aux déclarations de types de Signal. Chaque type de données simple est transformé en une déclaration de type de Signal. Le composant "donnée composée" est interprété comme un type structure.

Modélisation de connexion de port Deux types de connexions de ports de données sont modélisés : les connexions immédiates et les connexions retardées. Sur la base de leur sémantique temporelle spécifique, l'implémentation assure une communication déterministe entre les threads.

1.6 Spécification de comportements AADL

L'annexe comportementale décrit les comportements sous la forme d'un système de transition [42]. Elle peut être attachée à l'implémentation d'un composant, principalement pour les threads et les sous-programmes. Les comportements décrits dans cette annexe sont basés sur des variables d'état dont l'évolution est spécifiée par les transitions. L'annexe comportementale fournit une extension du mécanisme de dispatch du modèle d'exécution.

Pour effectuer la vérification de modèle (*model checking*) d'un système décrit en AADL, il est nécessaire de disposer des comportements des composants. En vue d'une

validation et d'une vérification au plus tôt du système, nous proposons une approche permettant d'interpréter automatiquement les comportements AADL dans un modèle de calcul approprié. Dans cette méthode, les notations de l'annexe comportementale sont intégrées dans le modèle de calcul polychrone, à des fins de vérification formelle et de génération de code.

Dans l'annexe comportementale AADL, les affectations multiples d'une même variable dans une même transition sont légales, alors qu'en Signal, il n'est pas autorisé de définir plusieurs fois un signal dans un même instant. Par conséquent, une réécriture des affectations est nécessaire dans l'interprétation. Dans la forme SSA (*static single assignment*) [70, 161], chaque variable n'apparaît qu'une seule fois en partie gauche des affectations : elle n'est définie qu'une seule fois. Notre interprétation utilise SSA comme formalisme intermédiaire.

Les transitions et les actions sont transformées en un ensemble d'équations synchrones. Cette transformation de modèle repose sur un algorithme inductif de transformation vers SSA appliqué sur les transitions et les actions [122]. Pour ce qui concerne les exigences de vérification, cet algorithme minimise le nombre d'états des automates, et permet donc un *model checking* avec de bonnes performances.

1.6.1 Forme SSA

Un programme est dit en forme SSA [70, 64, 63, 69] lorsque chaque variable dans le programme n'apparaît qu'une seule fois en partie gauche des affectations. Une seule nouvelle valeur au plus d'une variable x peut être définie dans un instant donné. La forme SSA d'un programme remplace les affectations d'une variable x du programme par des affectations à de nouvelles versions x_1, $x_2 \ldots$ de x, en indiçant de manière unique chaque affectation.

Un opérateur, noté *PHI*, est nécessaire pour choisir la valeur en fonction du flot de contrôle du programme, lorsqu'une une variable peut être affectée dans les deux branches d'une instruction conditionnelle ou dans le corps d'une boucle. Comme SSA est une représentation intermédiaire, l'introduction de nœuds *PHI* n'entraînera pas nécessairement de surcoût d'exécution. La compilation de Signal permettra d'optimiser le code d'exécution final.

Cette approche introduit une méthode effective pour transformer les spécifications de comportement consistant en des transitions et des actions en un ensemble d'équations synchrones. Notre transformation minimise les variables d'état nécessaires et la synchronisation des composants.

1.6.2 Systèmes de transition AADL

Des spécifications de comportement peuvent être attachées aux implémentations de composants AADL à l'aide d'une annexe. L'annexe comportementale de AADL décrit un système de transition.

Transitions La section des transitions déclare un ensemble de transitions T. Lorsque la spécification de comportement d'un thread est déclenchée pour la première fois, le thread démarre son exécution depuis un état **initial**, spécifié comme s **: initial state**. L'exécution

se termine dans un état **complete** (**return** pour un sous-programme), spécifié comme t **: complete state** (ou t **: return state**). Depuis un état s, si la garde g est vraie, une transition est activée, la partie action S est effectuée, puis un nouvel état, t, est atteint ; cela s'écrit : $s - [g] \rightarrow t\,\{S\}$. La notation ";" en AADL dénote la composition parallèle de transitions.

$$\mathscr{T} \ni T \ ::= \ s - [g] \rightarrow t\,\{S\} \quad | \, T_1; T_2 \qquad \begin{array}{l} s, t : \text{ états} \\ g : \text{ garde} \\ S : \text{ actions} \end{array}$$

Actions Une action $S \in \mathscr{S}$ est, récursivement, une séquence d'actions, une boucle sur des actions, une instruction conditionnelle, une action temporelle (**computation()**, **delay()**), ou une action de base. Une action de base $B \in \mathscr{B}$ est, récursivement, une séquence d'actions de base, une action d'affectation $w := f(y, z)$, une action de réception de message ($p?(x)$), ou une action d'émission de message ($p!(x)$).

$$\begin{array}{rl} \mathscr{B} \ni B ::= & w := f(y,z) \quad | \, p?(x) \quad | \, p!(x) \quad | \, B_1; B_2 \\ \mathscr{S} \ni S ::= & \textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2 \\ & | \, \textbf{for } (x \textbf{ in } X)\ S \\ & | \, \textbf{delay}(min,\ max) \\ & | \, \textbf{computation}(min,\ max) \\ & | \, B \\ & | \, S_1; S_2 \end{array}$$

1.6.3 Interprétation des transitions / actions

Afin de distinguer entre les transitions de phases d'interprétation différentes, nous introduisons quelques notations. Nous utilisons $S \in \mathscr{S}$ pour représenter une action générale de la transition d'origine $T \in \mathscr{T}$, et $U \in \mathscr{U}$ pour représenter les transitions intermédiaires, dites transitions de base, dont les actions sont des actions de base $B \in \mathscr{B}$.

$$\begin{array}{rcl} \mathscr{U} \ni U & ::= & s - [g] \rightarrow t\,\{B\} \quad | \quad U_1; U_2 \\ \mathscr{T} \ni T & ::= & s - [g] \rightarrow t\,\{S\} \quad | \quad T_1; T_2 \end{array}$$

La notation $def(A)$ est introduite pour désigner la partie gauche w d'une action d'affectation A ($w := f(y, z)$), et $use(A)$ pour désigner la partie droite $f(y, z)$.

Une action en forme SSA $A \in \mathscr{A}$ est, récursivement, une séquence d'actions en forme SSA, ou une affectation $w := f(y, z)$, où w est défini au plus une fois dans A. En d'autres

termes, pour tout $\mathscr{S} \ni A = A_1; A_2$, si $w \in def(A_1)$, alors $w \notin def(A_2)$, et de même pour l'inverse. La transition associée est appelée transition en forme SSA $W \in \mathscr{W}$.

$$\begin{aligned} \mathscr{A} \ni A &::= w := f(y,z) \quad | \quad A_1; A_2 \\ \mathscr{W} \ni W &::= s - [g] \rightarrow t \ \{A\} \quad | \quad W_1; W_2 \end{aligned}$$

Comme les actions sont attachées aux transitions, nous devons prendre en considération les états de départ et d'arrivée dans l'interprétation des actions.

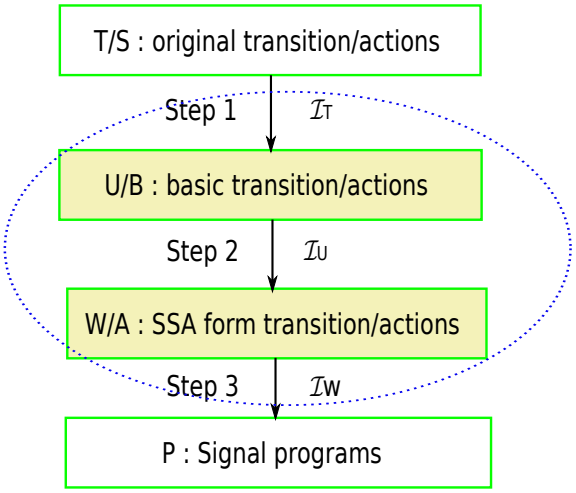


Figure 1.8: Chaîne de transformation de l'annexe comportementale.

Nous utilisons $\mathcal{I}(T)$ pour dénoter l'interprétation d'une transition T vers un processus Signal. La transformation des transitions/actions AADL vers $\mathcal{I}(T)$ en Signal est adressée en trois étapes (comme illustré sur la figure 1.8) :

$$\mathcal{I}(T): \quad T \quad \overset{\mathcal{I}_T}{\mapsto} \quad U \quad \overset{\mathcal{I}_U}{\mapsto} \quad W \quad \overset{\mathcal{I}_W}{\mapsto} \quad P$$

Étape 1 : actions vers actions de base. $T \overset{\mathcal{I}_T}{\mapsto} U$. Chaque transition $T \in \mathscr{T}$ avec une séquence d'actions générales $S \in \mathscr{S}$ est décomposée en ensembles de transitions de base $U \in \mathscr{U}$, dans lesquelles toutes les actions sont des actions de base $B \in \mathscr{B}$.

Nous utilisons $\mathcal{I}_T(T)$ pour représenter cette interprétation, d'une transition générale $T \in \mathscr{T}$ vers des transitions de base $U \in \mathscr{U}$.

$$\mathcal{I}_T(T) : T \mapsto U \quad \text{où} \quad \begin{aligned} T &\in \mathscr{T} \\ U &\in \mathscr{U} \end{aligned}$$

Pour la décomposition d'une transition d'origine T en transitions de base U, il suffit de décomposer les actions non basiques (par exemple, les instructions conditionnelles et

les boucles) vers des actions de base, avec des transitions intermédiaires. En premier lieu, lors du traitement des séquences, les actions non basiques doivent être séparées dans les actions d'origine ; en second lieu, la décomposition des actions non basiques en actions de base est effectuée.

- **Traitement des séquences.** Nous utilisons une fonction récursive intermédiaire $\mathcal{I}_{TS}$ pour rendre explicite la séparation des actions d'une transition T comme la composition séquentielle d'un premier bloc d'actions de base déjà traitées, noté B, et d'un deuxième bloc, noté S, où les actions ne sont pas encore transformées. La règle de base de la transformation récursive est que, depuis les premières actions d'origine, si une action est une action de base, alors elle est déplacée dans B, ceci jusqu'à occurrence d'une action non basique. B est séparé, et une transition intermédiaire est introduite pour les actions B, depuis l'état source s vers un état intermédiaire s_1. Pour les actions restantes S, les actions non basiques seront d'abord décomposées en utilisant les règles présentées dans la section sur la décomposition, puis pour le reste, la séparation se poursuit selon les règles dcrites ici.

- **Décomposition.** La décomposition d'une action composite S ($S \in \mathscr{S}$ et $S \notin \mathscr{B}$) d'une transition T — action conditionnelle ou boucle ou action temporelle (**computation()** ou **delay()**) — retourne une transition intermédiaire $U \in \mathscr{U}$ (qui peut être une composition de transitions) où les actions de U sont des actions de base.

Étape 2 : actions de base vers actions en forme SSA. $U \overset{\mathcal{I}_U}{\mapsto} W$. Pour chaque transition intermédiaire $U \in \mathscr{U}$, les actions de base $B \in \mathscr{B}$ sont représentées en forme SSA $A \in \mathscr{A}$ et de nouvelles transitions sont introduites si nécessaire. Chaque définition d'une variable d'origine w est remplacée par une nouvelle version (puisqu'il ne peut pas y avoir de définitions multiples dans la forme SSA), de sorte que les actions en forme SSA peuvent être exécutées dans un même instant.

Comme les actions de base sont constituées récursivement d'une séquence de trois types d'actions, l'affectation, la réception de message et l'émission de message, dans cette étape, seuls ces types d'actions (resp. de transitions) doivent être interprétés. Dans cette étape, l'interprétation est effectuée en deux phases : d'une part, l'interprétation des actions de base en forme SSA ; d'autre part, le traitement des séquences de transitions de base vers la forme SSA.

La function $\mathcal{I}_U$ est utilisée pour représenter l'interprétation d'une transition intermédiaire $U \in \mathscr{U}$, d'action $B \in \mathscr{B}$, qui sera représentée en transitions de forme SSA $W \in \mathscr{W}$, d'actions $A \in \mathscr{A}$. Dans cette étape, une transition intermédiaire $U = s - [c] \rightarrow t \ \{B\}$ (de l'état source s vers l'état destination t, gardée par la condition c, et d'action de base B) sera interprétée en transitions de forme SSA $W \in \mathscr{W}$, dans laquelle les actions A sont toutes en forme SSA.

$$\mathcal{I}_U : U \rightarrow W \quad \text{où} \quad \begin{array}{l} U \in \mathscr{U} \\ W \in \mathscr{W} \end{array}$$

Étape 3 : SSA vers Signal. $W \overset{\mathcal{I}_W}{\mapsto} P$. Les actions en forme SSA $A \in \mathscr{A}$ et les transitions sont traduites en équations Signal.

- **Interprétation des actions de forme SSA.** L'interprétation $\mathcal{I}_A(A)^g = P$ d'une action en forme SSA A prend en paramètre la garde g qui conduit à l'exécution de cette action, et retourne une équation Signal P. La seule action en forme SSA est l'affectation, ou la séquence d'affectations. Par conséquent, l'interprétation dans cette étape ne considère que l'action d'affectation et la séquence.

  Pour une affectation $x := f(y,z)$, les signaux utilisés (en partie droite de l'affectation) seront échantillonnés par la garde g : **when** g. Une définition partielle ::= est utilisée ici car il peut y avoir d'autres définitions de x avec d'autres gardes. Un opérateur **cell** est introduit sur les opérandes pour les rendre disponibles à leur horloge commune minimale.

$$\mathcal{I}_A(x := f(y,z))^g \;=\; (x ::= f(y \textbf{ cell } h,\ z \textbf{ cell } h) \textbf{ when } g \mid h := y \hat{+} z)/h$$

  Pour une séquence d'affectations, l'interprétation peut être effectuée en parallèle. La notation "|" est utilisée pour dénoter la composition parallèle en Signal.

$$\mathcal{I}_A(A_1; A_2)^g \;=\; \mathcal{I}_A(A_1)^g \mid \mathcal{I}_A(A_1)^g$$

- **Interprétation des transitions de forme SSA.** La notation $\mathcal{I}_W(W)^{st,nst}$ est utilisée pour représenter l'interprétation de la transition en forme SSA $W \in \mathscr{W}$ vers un processus Signal. Elle est paramétrée par les variables st, nst représentant respectivement l'état courant et l'état suivant du système de transition.

$$\mathcal{I}_W(W_1; W_2)^{st,nst} = \mathcal{I}_W(W_1)^{st,nst} \mid \mathcal{I}_W(W_2)^{st,nst}$$

$$\mathcal{I}_W(W)^{st,nst} \;=\; (P \mid nst ::= t \textbf{ when } g) \quad \text{où} \quad \begin{array}{l} W = s - [c] \rightarrow t\ \{A\} \\ g = \textbf{when } (st = s) \textbf{ when } c \\ P = \mathcal{I}_A(A)^g \end{array}$$

  L'équation $nst ::= t$ **when** g exprime que, lorsque la garde g est vraie, ce qui signifie lorsque l'état courant st est s et lorsque la condition c est satisfaite, l'état suivant nst de la transition sera t. Lorsqu'une transition se termine, l'état de sortie devient l'état courant.

Interprétation globale. Au final, la transformation globale $\mathcal{I}$ d'un système de transition $T = (T_1; \ldots; T_n)^{s_0,\ tick}$, en suivant l'interprétation en trois étapes, doit tenir compte de l'état initial s_0 du système de transition et de l'horloge, notée *tick*, représentant les instants auxquels le thread considéré est actif — cette horloge est définie par l'ordonnanceur. Le signal d'entrée *tick*, qui est un *signal de contrôle* de type `event`, connecte le système

d'exploitation (l'ordonnanceur) au thread pour l'informer qu'il est sélectionné pour être effectivement exécuté.

$$\mathcal{I}(T_1;T_2)^{s_0,\ tick} = \mathcal{I}(T_1)^{s_0,\ tick} \mid \mathcal{I}(T_2)^{s_0,\ tick}$$
$$\mathcal{I}(T)^{s_0,\ tick} = (P \mid st := nst\ \$\ \textbf{init}\ s_0 \mid st\ \hat{}= tick)\ /st, nst \quad \text{où} \quad P = \mathcal{I}_W(W)^{st,nst}$$
$$W = \mathcal{I}_U(U)$$
$$U = \mathcal{I}_T(T)$$

Dans cette transformation, nous montrons non seulement comment traduire les fonctionnalités de base de la programmation impérative en équations, mais nous considérons aussi les automates de modes qui contrôlent l'activation de ces transitions et actions élémentaires.

1.7 Génération de modèles de simulation distribués

Un système distribué [44, 116] peut être beaucoup plus étendu et puissant que des combinaisons de systèmes autonomes. Polychrony fournit des méthodes générales de compilation, en particulier pour la génération de code distribué [47, 58].

Pour une transformation complète, nous considérons deux spécifications. La première est le programme Signal qui comprend tous les composants de calcul et représente les flots de données, programme traduit à partir des composants AADL. La seconde est la spécification de l'architecture matérielle : dans la spécification d'architecture AADL, la façon dont le système doit être distribué est clairement définie ; par exemple, quels processus doivent être exécutés et ordonnancés par quel processeur, et comment le processeur ordonnancera périodiquement ou sporadiquement les processus.

La figure 1.9 illustre les étapes permettant de distribuer semi-automatiquement un programme Signal. Tout d'abord, les composants spécifiés sont placés (*mapping*) sur l'architecture cible, un ordonnanceur est ajouté pour chaque processeur, et les synchronisations d'horloges entre les composants partitionnés sont synthétisées. Le mécanisme des pragmas en Signal (pragma "RunOn") est utilisé pour représenter les informations de placement. Ensuite, des modèles de communication sont ajoutés entre ces composants. MPI, qui est un protocole du communication indépendant du langage, utilisé pour programmer des ordinateurs parallèles, est utilisé ici pour les communications entre les programmes distribués. L'étape finale consiste en la génération des codes exécutables distribués [123].

1.7.1 Placement

Comme un processus AADL est traduit en une *partition* ARINC, et qu'il est lié à un processeur comme spécifié dans les *propriétés*, on peut affecter chaque *partition* à un processeur donné. Chaque *partition* est rythmée par sa propre horloge. Le pragma Signal "RunOn" [58] est utilisé dans l'environnement Polychrony pour les informations de partitionnement : quand un partitionnement basé sur l'utilisation du pragma "RunOn"est ap-

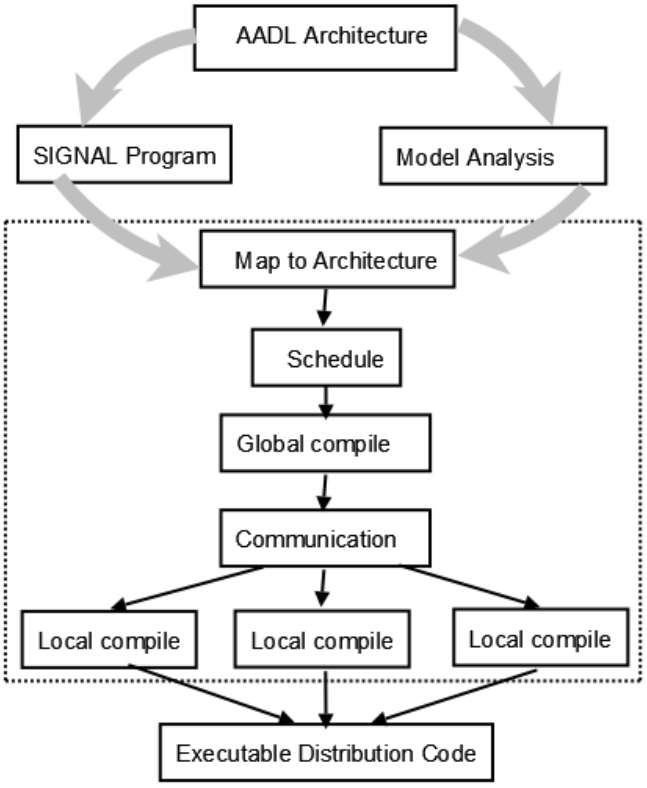


Figure 1.9: Génération de code distribué.

pliqué à un système, l'application globale est partitionnée selon les différentes valeurs du pragma de façon à obtenir des sous-modèles de processus. L'arbre des horloges (la racine de l'arbre représente l'horloge la plus fréquente) et l'interface de ces sous-modèles peuvent être complétés de manière à ce qu'ils représentent des processus *endochrones* [153] (un processus endochrone est insensible aux délais de propagation internes et externes).

Les principes du placement sont les suivants :

1. Description flot de données du logiciel, en utilisant le langage Signal (programme *P*).

2. Description de l'architecture cible. Celle-ci est spécifiée dans le modèle AADL, en particulier dans les *propriétés*.

3. Placement du programme logiciel sur l'architecture cible. Le programme *P* est ainsi récrit sous la forme `(|P1|...|Pn|)`, où n est le nombre de "processeurs". Cette étape n'est qu'une restructuration syntaxique du programme d'origine. Un processus Signal *Pi* est annoté avec un pragma "RunOn i". Ce pragma sera utilisé pour le partitionnement.

1.7.2 Ordonnanceur

L'ordonnanceur sélectionne les tâches à exécuter en fonction de la politique d'ordonnancement. Dans le modèle APEX-ARINC, l'*OS de niveau module* est l'ordonnanceur qui va ordonnancer les *partitions* à l'intérieur d'un même module. L'ordonnancement des *partitions* est strictement déterministe (du point de vue du temps) dans la norme ARINC 653. Chaque *partition* est ordonnancée selon sa fenêtre temporelle de partition. L'ordonnancement est fixe pour une configuration particulière de partitions sur le processeur. À l'intérieur de la *partition*, un *OS de niveau partition* est utilisé pour l'ordonnancement des *processus* (traduits des threads AADL).

Les principes d'ordonnancement sont les suivants :

1. Ordonnancement des *partitions*. Nos règles de transformation suivent la norme AADL, selon laquelle chaque processeur peut être attribué à plusieurs processus, ce qui signifie qu'un processeur peut exécuter plusieurs *partitions*. Comme l'ordonnanceur des *partitions* s'exécutant sur un processeur est déterministe, il est statique et fixe selon les tables de configuration. Nous créons à cette fin un ordonnanceur statique simple (cf. *Module_Level_OS* de APEX-ARINC).

2. Compilation globale. Après avoir ajouté l'ordonnanceur dans le programme, celui-ci peut être compilé globalement afin de permettre la séparation ultérieure des différentes parties. Cette phase rend explicites toutes les synchronisations de l'application, puis synthétise, si elle existe, l'horloge maîtresse du programme, et détecte les contraintes de synchronisation.

   Le compilateur utilise le *calcul d'horloges* pour analyser statiquement chaque équation du programme, pour identifier la structure des horloges des variables, et pour ordonnancer l'ensemble des calculs. Si la composition des équations contient des contraintes d'horloges, cela signifie qu'il existe des contraintes de synchronisation.

   Le programme distribué est généré automatiquement dans l'environnement Polychrony, en appliquant la compilation en mode *distribution*.

   Par rapport au programme d'origine, une réécriture a été effectuée pour les parties séparées. Toutes les propriétés de calcul sont préservées, et un ordonnanceur a été ajouté pour chacun des processeurs.

1.7.3 Ajout des communications

Un programme distribué consiste en un ensemble de processus interagissant avec l'environnement et communiquant entre eux. Les raisons de ces communications sont d'envoyer des données ou des signaux à d'autres processus, de se synchroniser avec d'autres processus, ou de demander une action à un autre processus.

Il existe deux solutions de base pour les communications : les variables partagées et l'envoi de messages. Les variables partagées ne permettent pas la synchronisation entre processus parallèles, à moins qu'un mécanisme complexe soit défini au-dessus. En outre, elles rendent la vérification formelle plus difficile. L'autre solution est la communication par envoi de messages. Les messages permettant de communiquer dans les systèmes distribués peuvent être de type synchrone ou asynchrone. Dans notre implémentation,

nous utilisons MPI (Message Passing Interface) [159]. Il s'agit d'une interface de passage de messages, ainsi que de protocoles et de spécifications sémantiques pour la façon dont ses fonctions doivent se comporter dans une mise en œuvre.

MPI préserve l'ordre et l'intégrité des messages (par exemple, par des accusés de réception et des numéros de séquence). Cela permet d'assurer que les valeurs ne sont pas mélangées dans les séquences, pourvu que les actions *send* soient exécutées sur un emplacement dans le même ordre que les actions *recv* correspondantes sur l'autre emplacement. À partir de la spécification de distribution, une balise unique est attribuée à chaque variable commune du programme.

Pour exécuter les parties distribuées, il est nécessaire de configurer les machines physiques pour leur exécution. La table 7.3 donne un exemple détaillé de configuration : le fichier binaire exécutable généré qui se trouve dans DIRECTORY0 (resp. DIRECTORY1) sera exécuté sur node0 (resp. node1). Les deux machines (noeuds) sont spécifiées par les adresses hôtes dans le fichier "lamhosts".

lamhosts: 10.0.1.3 %the first machine% 10.0.1.5 %the second machine%
c0 -wd DIRECTORY0 10.0.1.3 % node0 runs on 10.0.1.3% c1 -wd DIRECTORY1 10.0.1.5 % node1 runs on 10.0.1.5%

Table 1.2: Exemple de configuration.

Finalement, les parties peuvent être compilées localement et exécutées respectivement sur les différentes machines. Les programmes sont exécutés sur les processeurs distribués.

1.8 Vérification et simulation

Dans cette section, nous décrivons une chaîne d'outils de vérification formelle et de simulation pour AADL. Nous donnons une vue de haut niveau des outils impliqués et illustrons les transformations successives requises par notre processus de vérification. Deux études de cas sont présentées pour illustrer la validation d'applications AADL, en particulier par vérification formelle et par simulation.

1.8.1 Vérification formelle

Ces dernières années, AADL a été de plus en plus utilisé pour des systèmes embarqués critiques dans le domaine avionique à des fins de démonstration. L'objectif principal n'est pas seulement lié à l'exploration d'architecture et l'analyse. La vérification formelle des aspects fonctionnels devient aussi une démarche de validation très intéressante. La validation de modèles AADL par des méthodes formelles a été étudiée dans [132, 67, 164, 139], où des sémantiques formelles sont associées aux spécifications AADL, de telle sorte qu'une vérification formelle puisse être effectuée.

Notre approche s'inscrit dans la vérification de modèle dans le cadre de Polychrony. Les spécifications AADL sont traduites en Signal, puis l'outil de vérification de modèle associé à Polychrony, appelé Sigali [126], est utilisé pour vérifier des propriétés de sûreté.

Sigali utilise des *systèmes dynamiques polynomiaux* [125] sur $\mathbb{Z}/_{3\mathbb{Z}}$ = {-1, 0, 1}) comme modèle formel. Les programmes Signal peuvent être transformés en équations polynomiales, qui sont manipulées par Sigali. Les trois états d'un signal booléen peuvent être codés comme : présent $\wedge$ vrai $\Leftrightarrow$ 1; présent $\wedge$ faux $\Leftrightarrow$ -1; absent $\Leftrightarrow$ 0. Pour les signaux non booléens, seules leurs horloges sont encodées, c'est-à-dire que leurs valeurs sont ignorées.

La première étude de cas permet d'illustrer la vérification de modèle de modèles AADL via Polychrony. AADL est utilisé pour modéliser un système de guidage de vol simplifié (FGS). Les modèles AADL sont alors traduits, via Signal, en une représentation symbolique, des systèmes dynamiques polynomiaux, manipulés par le vérificateur de modèle Sigali. Avec Sigali, des propriétés de sûreté exprimées en CTL sont vérifiées sur le système.

1.8.2 Simulation

Dans cette section, nous décrivons le processus de conception pour la co-modélisation et la co-simulation d'une étude de cas du projet CESAR dans le cadre de Polychrony. Ce processus de conception est basé sur une approche de co-conception au niveau système. Polychrony est utilisé comme plate-forme commune de développement. La simulation est réalisée à des fins de vérification et d'analyse temporelle au niveau système.

Le "profilage" de logiciel peut être considéré comme une sorte d'analyse dynamique de programme au moyen des informations recueillies lorsque le programme s'exécute. Une telle analyse est en général effectuée à des fins d'amélioration des performances. Le profilage est aussi adopté dans Polychrony pour l'évaluation de performance des programmes Signal [106] [82]. Le processus de profilage comprend la spécification de propriétés temporelles, un homomorphisme temporel, et une co-simulation.

Dans le cadre de Polychrony, le profilage se réfère à l'analyse temporelle par l'association d'informations de *date* et de *durée* à des programmes Signal. À chaque signal, *x*, du programme, est associé un signal de date, *date_x*, pour indiquer ses instants de disponibilité. Ce signal de date peut être spécifié avec une horloge métrique, une horloge logique ou des cycles d'horloge. Dans le premier cas, les signaux de date sont des nombres réels positifs, et dans les autres cas, ce sont des entiers positifs. À chaque opération dans le programme Signal est associée une information de *durée*, qui peut avoir les mêmes types de donnée que les signaux de dates. La *durée* peut aussi être représentée par un couple de nombres correspondant au meilleur et au pire cas. En outre, ces informations sont fonction de plusieurs paramètres tels que le type d'opération, les types de données, et les architectures de mise en œuvre.

La seconde étude de cas est un système simplifié de contrôle des portes et toboggans (SDSCS) en avionique. En plus de la simulation AADL, cette étude de cas illustre également une approche de co-conception de système de haut niveau et hétérogène, basée sur les modèles, pour une architecture de système globalement asynchrone et localement synchrone (GALS). Dans cette étude de cas, l'architecture matérielle distribuée, ainsi que les aspects asynchrones, sont modélisés en AADL, tandis que le comportement fonctionnel, exprimé dans le modèle à flots de données synchronisés, est modélisé en Simulink. SME/Polychrony est adopté comme formalisme commun permettant de faire la liaison entre deux modèles hétérogènes. La modélisation AADL et les résultats de transforma-

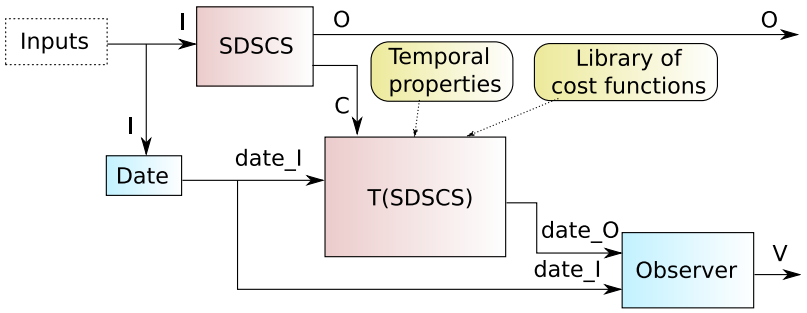


Figure 1.10: Co-simulation d'un programme Signal et de son comportement temporel.

tion en Signal sont présentés. Une co-simulation rapide (phase initiale) avec profilage par prise en compte des contraintes de temps (figure 1.10), ainsi qu'une démonstration de simulation par des afficheurs VCD (figure 1.11) ont été réalisées dans le cadre de Polychrony.

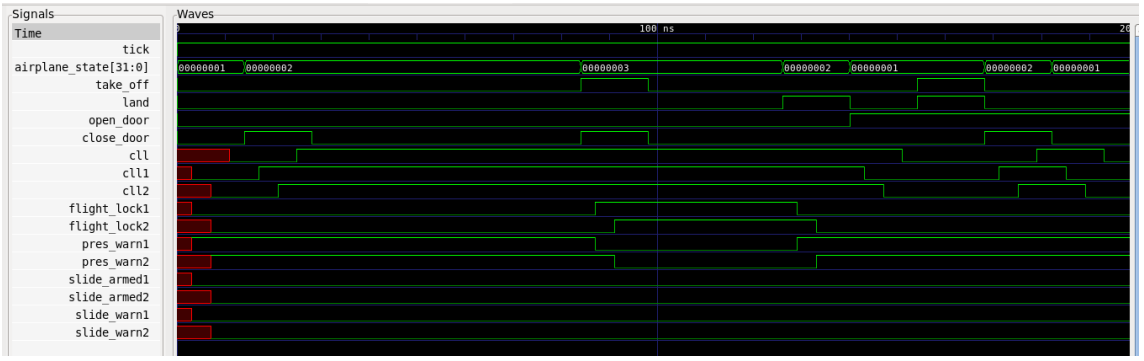


Figure 1.11: Simulation avec afficheur VCD : GTKWave

1.9 Conclusion

La principale motivation des travaux présentés dans cette thèse est de combler le fossé entre les modèles de AADL et Polychrony. Nous proposons pour cela une méthodologie pour la modélisation de niveau système et la validation de systèmes embarqués spécifiés en AADL via le modèle de calcul polychrone. Cette méthodologie comprend la modélisation de niveau système via AADL, des transformations automatiques à partir du modèle AADL de haut niveau vers le modèle polychrone, la distribution de code, la vérification formelle, la simulation du modèle polychrone obtenu, et le raffinement du modèle en fonction des résultats de validation.

Contributions Notre conception au niveau système prend en compte à la fois l'architecture du système, plus spécialement pour une architecture décrite selon le modèle "avionique modulaire intégrée" (IMA), et les aspects fonctionnels, par exemple, avec des composants logiciels implémentés dans le langage de programmation synchrone Signal. À partir de spécifications AADL de haut niveau, la distribution automatique de code

pour la simulation a été expérimentée via le modèle polychrone et les pragmas de distribution de Signal. En outre, l'annexe comportementale de AADL, qui est une extension pour la spécification des comportements effectifs, est également prise en compte dans la traduction de AADL vers Signal. Elle repose sur l'utilisation de SSA (*Static Single Assignment*) comme formalisme intermédiaire. La simulation, la validation et la vérification de haut niveau sont ensuite effectuées au moyen de technologies et outils associés, comme l'utilisation de Sigali, le profilage, ou les démonstrateurs de changements de valeurs, qui permettent de vérifier formellement et d'analyser le modèle résultant. Des techniques et des bibliothèques existantes de Polychrony, qui consistent en un modèle des services temps réel de APEX-ARINC, ont notamment été utilisées.

Perspectives Nous prévoyons d'étendre cette modélisation afin de couvrir un périmètre plus large de AADL, ainsi que des aspects temporels supplémentaires qui sont liés à la conception GALS et permettent des analyses temporelles plus sophistiquées. Une autre extension consistera à fournir une bibliothèque de modèles Signal adaptés à la modélisation de composants standards. Cette bibliothèque permettra de réduire les tâches répétitives de modélisation. Des connexions à d'autres outils de simulation pour ce qui concerne l'analyse temporelle sont aussi prévues, comme par exemple SynDEx [25] ou RT-Builder [22]. Pour la modélisation et la vérification de systèmes GALS, il serait intéressant de pouvoir insérer des buffers de désynchronisation dans une démarche de raffinement, et de vérifier la correction du raffinement vis-à-vis de la préservation de flot et des contraintes temps réel. Au-delà des connexions point à point, différents types d'interconnexions entre les composants synchrones permettraient la vérification de systèmes GALS plus complexes. Enfin, des extensions de notre modèle d'ordonnancement constituent aussi une perspective à ce travail.

Part II

Conceptions of AADL and Signal

Chapter 2

Introduction to AADL and Avionic architectures

Contents

2.1 AADL language abstractions 44
2.1.1 The SAE AADL standard 44
2.1.2 AADL meta-model and models 45
2.1.3 AADL open source tools 46
2.2 Summary of the core AADL components 48
2.2.1 Software components 50
2.2.2 Execution platform components 55
2.2.3 System component 58
2.3 System binding 59
2.4 Component interaction 60
2.4.1 Port 60
2.4.2 Port connection 61
2.5 Flows 63
2.6 AADL behavior annex 64
2.7 Avionics system architecture and ARINC653 65
2.7.1 Avionics system architecture overview 65
2.7.2 ARINC standard 66
2.7.3 AADL and ARINC 69
2.8 AADL components considered in this thesis 70
2.9 Conclusion 70

This chapter is devoted to the introduction of the Architecture Analysis and Design Language (AADL).

Developed by the Society of Automotive Engineers (SAE), AADL was approved and published as SAE standard (AS5506) in November 2004. The purpose of the SAE AADL

standard is to provide a standard and sufficiently precise way of modeling the architecture of an embedded real-time system, such as an avionics system or automotive control system, to permit analysis of its properties, and to support the predictable integration of its implementation.

As an architecture description language, AADL supports modeling of the embedded software runtime architecture, the execution hardware, and the interface to the physical environment of embedded systems. A system can be automatically integrated from AADL models, when fully specified and when source code is provided for the software components.

The AADL standard consists of a suite of documents: the core AADL language standard and a series of extensible annexes. In this chapter, we focus on the core AADL standard (SAE AS5506) and the Behavior Annex.

This chapter gives a brief introduction of AADL language. First, an overview of AADL standard is given in Section 2.1. The meta-model, model representations and available toolsets are introduced. Then section 2.2 presents the core components considered in this thesis. The components are separated into three distinct sets of categories: software, execution platform and system components. For a complete system specification, the system binding (section 2.3), component communication (section 2.4) are taken into consideration. The Behavior Annex, which is an extension of the core standard to offer a way to specify the local functional behavior of the components, is presented in Section 2.5. The behaviors described in this annex are seen as specifications of the actual behaviors. They are based on state variables, whose evolution is specified by the transitions. ARINC specification is dedicated to provide a general interface with the set of basic services to access the operating-system and other system-specific resources. The second version of AADL standard includes an ARINC653 annex (not complete) involving a definition of ARINC653 architectural elements in AADL. A brief introduction of IMA architecture and ARINC653 standard is given in Section 2.6.

2.1 AADL language abstractions

2.1.1 The SAE AADL standard

The SAE AADL is a standard that has been developed for embedded real-time safety critical systems, which will support the use of various formal approaches to analyze the impact of the composition of systems from hardware and software.

In 2001, MetaH [158, 107] has been taken as the basis of a standardization effort, aiming to define an Avionics Architecture Description Language standard. The first AADL standard was released in 2004, named AS5506. AADL version 2 was published in January 2009. This version introduces virtual processor and virtual bus components, in support of partitions, virtual channels and protocol stacks. Other improvements include abstract components, classifier parameterization, and component multiplicity to better support architecture patterns for system families and reference architectures as well as scaling to large-scale systems [40]. The main basic components remain the same. Our work only take these components in consideration.

An AADL specification describes the software, hardware, and system part of an embedded real-time system. It defines a system by describing jointly software and hardware

components and their interactions. The language can also be used to describe the dynamic behavior of the runtime architecture by providing support to model operational modes and mode transitions.

While the core language provides a number of modeling concepts with precise semantics including the mapping to execution platforms and the specification of execution time behavior, it is not possible to foresee all possible architecture analysis. The language is designed to be extensible to accommodate analysis of the runtime architectures that the core language does not completely support. The objective is to be able to perform various analysis (schedulability, sizing analysis, safety analysis) very early in the development process. Such extensions will be defined as part of a new Annex appended to the standard. The SAE AADL standard consists of a suite of documents: the core language standard (AS5506) and a series of annexes, such as Error Model Annex, Behavior Annex, etc., listed in Figure 2.1.

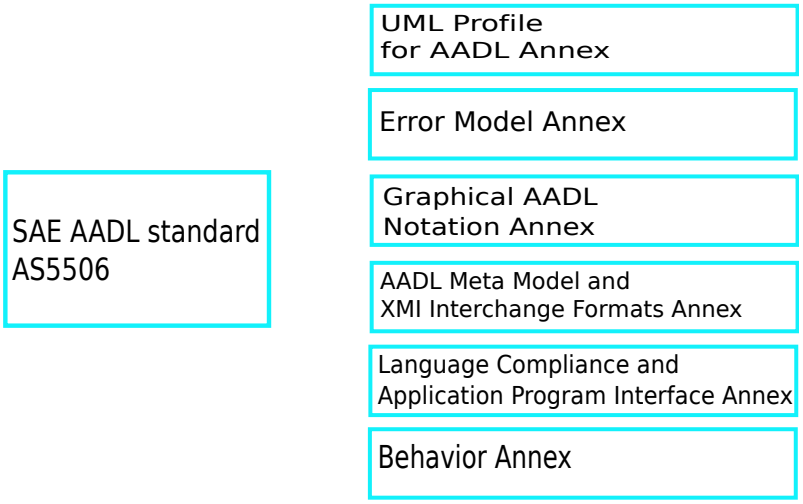


Figure 2.1: AADL core standard and annexes

The Behavior Annex enables the expression of high level composition concepts. The behaviors described in this annex are seen as specifications of the actual behaviors. They are based on state variables whose evolution is specified by the transitions expressed in the behavioral annex.

2.1.2 AADL meta-model and models

The AADL is a textual and graphical language used to model and analyze the software and hardware architecture of embedded systems. It is used to describe the structure of component-based systems as an assembly of software components mapped onto an execution platform, and specially targets real-time embedded systems.

The AADL meta-model defines the structure of AADL models. This meta-model is represented as a set of class diagrams with additional EMF-specific properties, that support the automatic generation of methods for manipulation of AADL object models.

AADL defines seven EMF Ecore meta-model packages: core, component, feature, connection, flow, instance and property.

An object representation of AADL specification corresponds to a semantically decorated abstract syntax tree. This object representation of AADL model can be persistently stored as XML format. An AADL model can also be represented as textual code model or graphical model.

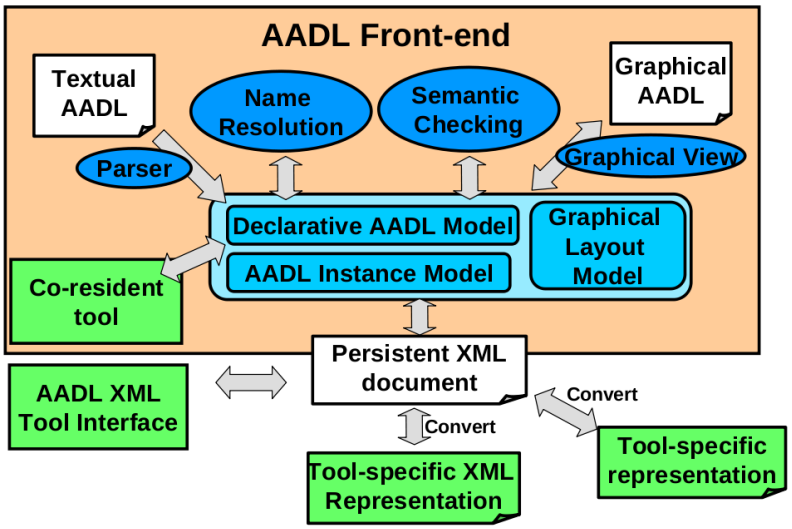


Figure 2.2: AADL model representations

Figure 2.2 shows the three model representations and their roles with respect to the textual and graphical AADL representation:

- A declarative AADL model: AADL specifications in the form of textual representations. Checking the syntax and semantics of the AADL text is done by parsing it and reporting errors.
- An AADL instance model: an arborescent hierarchic representation of the system, its components and subcomponents, whose root is a system implementation. It defines an abstract syntax tree. It can be converted to XML representation.
- A graphical model: graphical display of AADL models. The diagram representation gives a more intuitive view of the model.

2.1.3 AADL open source tools

OSATE [18] (Open Source AADL Tool Environment) was developed in the Eclipse framework, providing the standardized XML definitions for use by Eclipse AADL plug-in analysis or interfacing with external toolsets. The current OSATE1.5 is based on the first AADL standard version. It permits creating AADL textual specifications (.aadl files, shown in Figure 2.3) and XML-related object models (.aaxl files, shown in Figure 2.4).

Figure 2.3 is a textual AADL file example. The textual specification is a readable collection of textual declarations that comply with the AADL standard. The editor includes a

```
1 system sys1
2 end sys1;
3
4 system implementation sys1.impl
5 subcomponents
6   control: process Control;
7   compute: process Compute.impl;
8 connections
9   con1: data port compute.data_out -> control.data_in;
10  con2: data port control.c_out -> compute.c_in;
11 end sys1.impl;
12
13 process Control
14 features
15   data_in: in data port;
16   c_out: out data port;
17 end Control;
18
19 process Compute
20 features
21   data_out: out data port;
22   c_in: in data port;
23 end Compute;
24
25 process implementation Compute.impl
26 subcomponents
27  t: thread compute_thread;
28 end Compute.impl;
29
30 thread compute_thread
31 end compute_thread;
```

Figure 2.3: AADL textual model representation

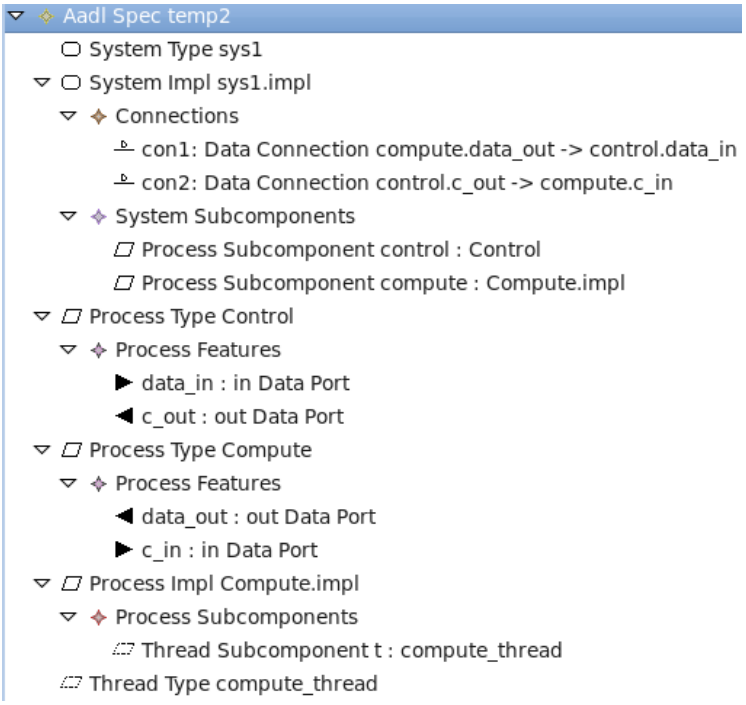


Figure 2.4: AADL object model representation

parser to verify the syntax and semantic aspects of the specification. The keywords, such as **system implementation**, **subcomponents**, **features** and **data port**, are reserved.

Figure 2.4 represents an XML-related object model (.aaxl file), which can be generated by converting from the (.aadl) text file. The AADL object editor allows viewing the system instance hierarchy through the declarative AADL model. Subcomponents can be recursively expanded into the corresponding component implementation referenced by the subcomponent.

An AADL graphical editor is provided by the ADELE Eclipse plug-in environment, see figure 2.5. The top-level diagram represents a library of component declarations. Each component implementation has its own instance diagram. Here we present a general system to give a first impression. A flow starts from *Device1* and sinks in *Device2* through two process instances: *ProcessA* and *ProcessP*. The two processes are hosted on two different processors, (the allocation information is specified in the properties). The devices communicate via a bus *Bus1*. We will present the components in detail in the following subsections.

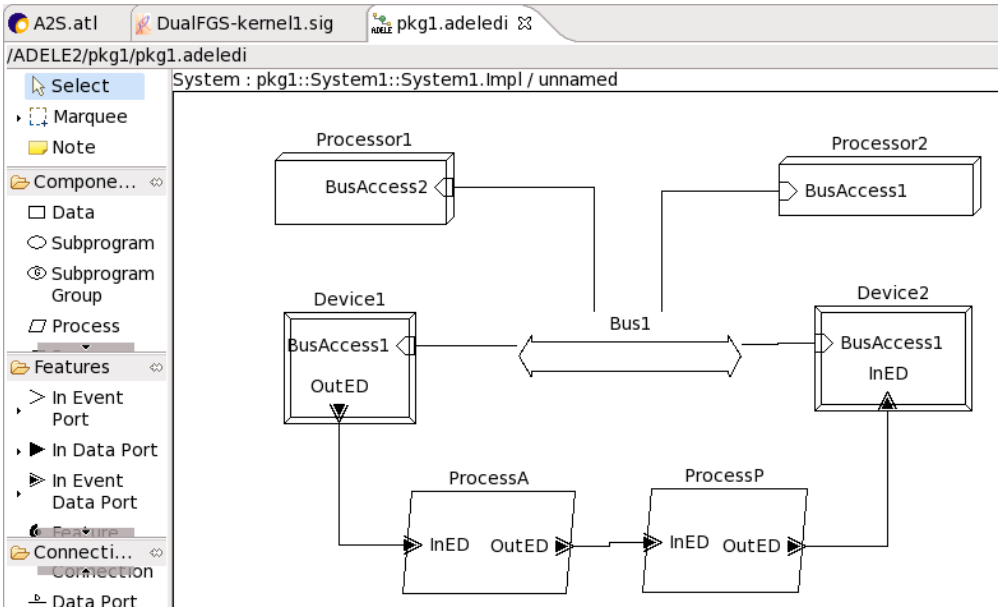


Figure 2.5: AADL graphical model representation

We also list some other AADL open source toolsets:

- TOPCASED [29]: develops a meta-modeling framework.
- OCARINA [15]: develops an AADL graphics toolset.
- CHEDDAR [8]: advanced scheduling analysis toolset.
- GME [104]: AADL architecture specification and safety analysis framework.

2.2 Summary of the core AADL components

AADL supports the modeling of application software components, execution platform components and the binding of software onto execution platform. To model complex

embedded systems, AADL provides three distinct sets of component categories: software, execution and composite component.

- Data, subprograms, threads, and processes collectively represent application software, they are called software components.
- Execution platform components support the execution of threads, the storage of data and code, and the communication between threads.
- Systems are called composite components. They permit software and execution platform components to be organized into hierarchical structures with well-defined interfaces.

Type and implementation AADL components are defined through type and implementation declarations.

A component type represents the functional interface of the component and externally observable attributes.

The implementation describes the contents of the component, specifies an internal structure in terms of subcomponents, connections between the features of those subcomponents, flows across a sequence of subcomponents, modes to represent operational states and properties. Following is a partial code from a real example (Appendix A), showing a type and implementation declaration of a periodic thread *door_handler*.

```
thread door_handler
  features
    in_flight: in data port;
    cll: out data port;
end door_handler;

thread implementation door_handler.impl
  properties
    Dispatch_Protocol => Periodic;
    Period => 50 ms;
end door_handler.impl;
```

AADL allows multiple implementations to be declared with the same functional interface, that is to say, each type may be associated with zero, one or more implementations.

Components may be hierarchically decomposed into collections or assemblies of interacting subcomponents. A subcomponent declares a component that is contained in another component. A component implementation can contain arrays of subcomponents. A subcomponent names a component type and component implementation to specify an interface and implementation for the subcomponent. Thus, component types and implementations act as component classifiers. The hierarchy of a system instance is based upon the set of subcomponents of the top-level system implementation. It is completed by iteratively traversing the tree of the component classifiers specified starting at the top-level system implementation subcomponents.

Features A feature specifies the interaction points with other components, including the inputs and accesses required by the component, and all the outputs and items the component provides. Features are specified in the component type. In the above example, two ports *in_flight* and *cll* are declared as **in** and **out data port** respectively in the **features** clause.

Properties A property specifies properties of the component that apply to all instances of this component, unless overwritten in implementations or extensions. Properties can be assigned values through property association declarations. They can be defined either in component type or implementation. If they are specified in the component type, then the implementations of the type will inherit all the properties. In the above thread implementation *door_handler.impl* example, two properties are defined: **Dispatch_Protocol** and **Period**, and associated values are given: *Periodic* and *50ms*.

2.2.1 Software components

AADL has the following categories of software components: process, thread, thread group, subprogram and data.

2.2.1.1 Process

The process abstraction represents a protected virtual address space, a space partitioning unit where protection is provided from other components accessing anything inside the process.

The virtual address space contains the program formed by the source text associated with the process and its subcomponents. Three subcomponent categories are allowed: thread, thread group and data. Table 2.1 summarizes the permitted type declaration and implementation declaration elements of a process. A process can only be a subcomponent of a system component. A complete implementation of a process must contain at least one explicitly declared thread or thread group subcomponent.

Category	Type	Implementation
process	Features : • server subprogram • provides/requires data access Flow specifications: yes Properties: yes Modes: yes	Subcomponents : • data • thread (group) Connections: yes Modes: yes Properties: yes

Table 2.1: Summary of Permitted Process Declarations

Figure 2.6 presents an example of process called *pcs*. The process implementation *pcs.impl* contains two subcomponents: *thsnd* and *thrcv*, and three types of connections (*event* port, *event data* port and *data* port connections).

```
process pcs
  features
    req: in event port;
    resp: out data port;
end pcs;

process implementation pcs.impl
  subcomponents
    thsnd: thread thsnd.impl;
    thrcv: thread thrcv.impl;
  connections
    event port req -> thsnd.dr1;
    event data port thsnd.ds1 -> thrcv.dr2;
    data port thrcv.ds2 -> resp;
end pcs.impl;
```

```
thread thsnd
  features
    dr1: in event port;
    ds1: out event data port;
end thsnd;

thread implementation thsnd.impl
end thsnd.impl;

thread thrcv
  features
    dr2: in event data port;
    ds2: out data port;
end thrcv;

thread implementation thrcv.impl
end thrcv.impl;
```

Figure 2.6: Example of AADL thread and process

2.2.1.2 Thread

A thread represents a sequential flow of control that executes instructions within a binary image produced from source text. A thread always executes within the virtual address space of a process, and it models a schedulable unit that transits between various scheduling states. Multiple threads model concurrent executions of tasks. A scheduler manages the execution of threads. A generic semantics for a thread is defined based on automata.

An example can be seen in Figure 2.6 (right part). Two threads are given in this example: *thsnd* and *thrcv*. The thread type gives the interaction interface: for example, *dr1* is declared as **in event port**, *ds1* as **out event data port**. The detailed implementation is not shown.

Table 2.2 summarizes the legal type and implementation declarations for a thread.

Category	Type	Implementation
thread	Features : • port (group) • provides/ requires data access • provides/ requires subprogram access • provides/ requires subprogram group access Flow specifications: yes Properties: yes Modes: yes	Subcomponents : • data • subprogram (group) Subprogram calls: yes Flows: yes Connections: yes Modes: yes Properties: yes

Table 2.2: Summary of Permitted Thread Subclause Declarations

Timing domain Three events (t_d, t_s, t_f) are associated with each thread t. The event t_d indicates when the thread is dispatched, t_s represents a signal at which the thread starts, and t_f is the one at which the thread finishes.

Thread dispatching The **Dispatch_Protocol** property of a thread determines the characteristics of dispatch requests to the thread. The supported dispatch protocols are: peri-

odic, aperiodic, sporadic and background.

1. Periodic: repeated dispatches occurring at a specified time interval. A dispatch request is issued at the beginning (t_d) of time intervals specified by **Period** property value. The *period* value is the thread period.

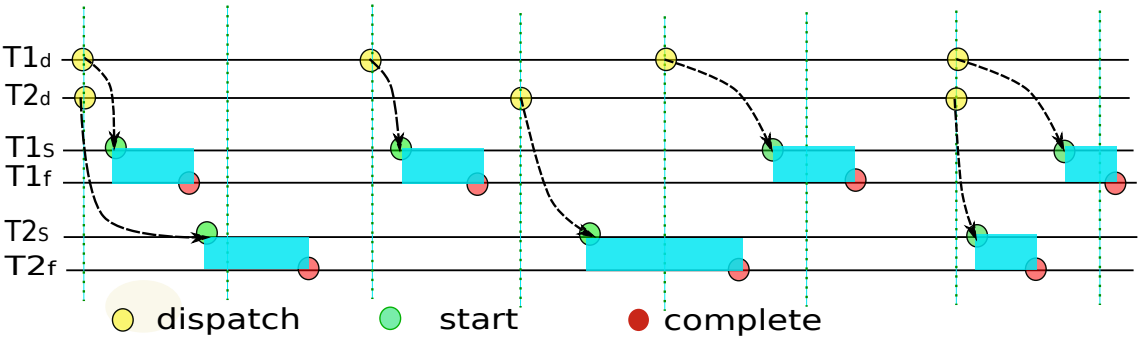


Figure 2.7: Periodic thread clock relations.

Figure 2.7 represents the execution of two synchronized periodic threads (T_1 and T_2) in the absence of preemption or other interruptions (events, abort, call subprograms, get resource, exit (mode), etc.). The two threads do not have the same period, but share simultaneous dispatch (every three T_1 or every two T_2, the two threads are dispatched at the same time).

2. Aperiodic: event-triggered dispatch of threads. A dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a *provides subprogram access* feature of the thread. There is no constraint on the inter-arrival time of events, event data or remote subprogram calls.

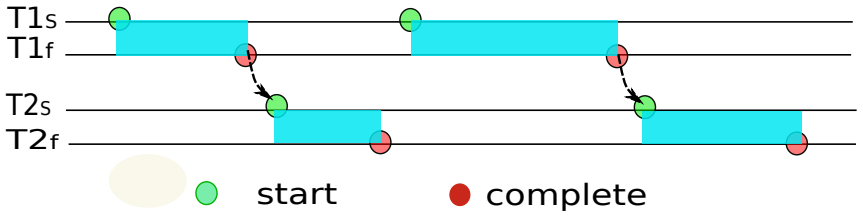


Figure 2.8: Aperiodic threads clock relations.

An aperiodic thread is event-triggered. Figure 2.8 illustrates the alternation relations. In this example, the termination of T_1 asynchronously triggers the start of T_2.

3. Sporadic: event-driven dispatch of threads with a minimum dispatch separation. A dispatch request is the result of an event or event data arriving at an event or event data port of the thread, or a remote subprogram call arriving at a *provides subprogram access* feature of the thread. The time interval between successive dispatch requests will never be less than the associated **Period** property value.

4. Background: a dispatch initiated once with execution until completion. A background thread is scheduled to execute when all other threads' timing requirements are met. If more than one background thread is dispatched, the processor's scheduling protocol determines how such background threads are scheduled.

Thread states and actions Figure 2.9 shows the state transitions of thread execution under normal operation.

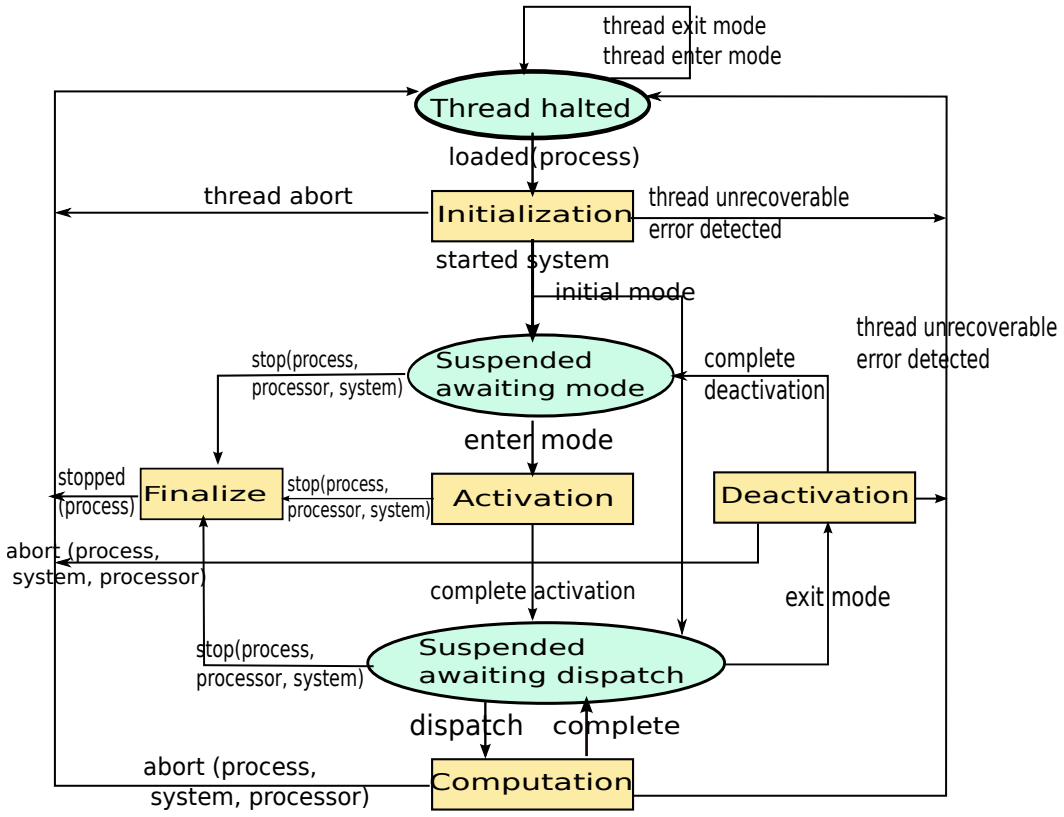


Figure 2.9: Thread states and actions

All threads start from the initial state: **thread halted**. When the loading of the virtual address space as declared by the enclosing process completes, a thread is initialized by performing an initialization code sequence in the source text. Once initialization is completed, the thread enters the **suspended awaiting mode** state, if the thread is not in the initial mode, otherwise, it enters the **suspended awaiting dispatch** state. When in the **suspended awaiting dispatch** state, a thread is awaiting a dispatch request for performing the execution.

Thread scheduling and execution A thread is provided with a priority given in associated property field, which is determined by the chosen scheduling policy. When a system is started, all threads are in a **halted** state. They are initialized and enter in the **awaiting** state, waiting for activation. The scheduler manages the activation of the threads. It chooses a thread with highest priority from the **ready** threads. The selected thread starts running. After its completion, it returns to the awaiting state.

When a thread is in **computation** state (see the bottom of Figure 2.9, here we give a more detailed description of the **computation** state), the execution of the thread's entry-point source text code sequence is managed by a scheduler. The scheduler coordinates all thread executions on one processor as well as concurrent access to shared resources. While executing an entrypoint, a thread can be in one of the five substates: **ready**, **running**, **awaiting resource**, **awaiting return**, and **awaiting resume** (in Figure 2.10).

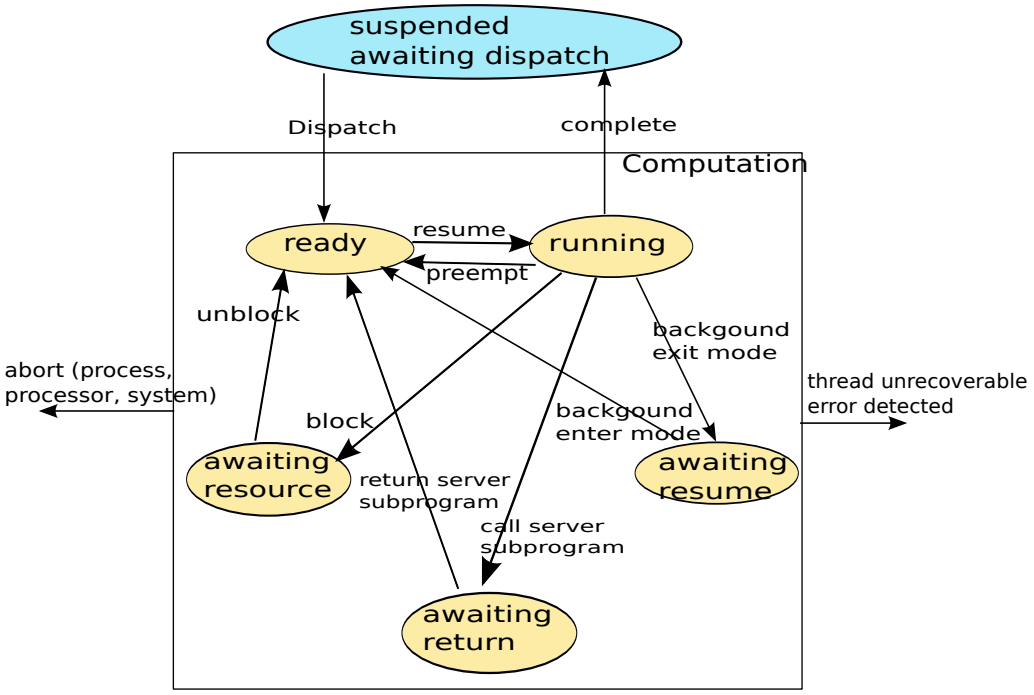


Figure 2.10: Thread Scheduling States

A thread initially enters the **ready** state. The thread in the **ready** state that has the maximum priority starts or continues its execution. It ensures that only one thread is in the **running** state on a particular processor. If no thread is in the **ready** state, the processor is idle until a thread enters the **ready** state.

A thread will remain in the **running** state until it completes execution of the dispatch, or until a thread entering the **ready** state preempts it if the specified scheduling protocol prescribes preemption, or until it blocks on a shared resource, or until an error occurs. In the case of completion, the thread goes back to the **suspended awaiting dispatch** state (described in Figure 2.9), ready to waiting for another dispatch request. In the case of preemption, the thread returns to the **ready** state. In the case of resource blocking, the thread transits to the **awaiting resource** state.

2.2.1.3 Thread group

A thread group represents an organizational component to logically group threads contained in processes. The type of a thread group component specifies the features and required subcomponent access through which threads contained in a thread group interact with components outside the thread group. Thread group implementations represent the contained threads and their connectivity.

2.2.1.4 Data

Data represent static data and data types within a system. Figure 2.11 contains textual (left) and corresponding graphical (right) declaration of a substructure of a data. The substructure of the data implementation *Person.impl* consists of four data subcomponents: *name*, *surname*, *age* and *gender*. They are typed by AADL basic type *string*, *int* and an enumerated type *Gender*.

```
data Person
end Person;

data implementation Person.impl
  subcomponents
    name: data string;
    surname: data string;
    age: data int;
    gender: data Gender;
end Person.impl;
```

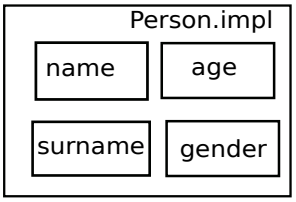


Figure 2.11: Sample data component text and graphical representations

2.2.1.5 Subprogram

A subprogram component represents sequentially executed source text that is called with parameter. Subprograms can be called from threads and from other subprograms. These calls are sequential calls local to the virtual address space of the thread. Ordering of subprogram calls is by default determined by the order of the subprogram call declarations.

2.2.2 Execution platform components

This subsection describes a subset of execution platform components: processor, device and bus. Execution platform components can represent high-level abstractions of physical and computing components. A detailed AADL model of their implementation can be represented by system implementations that are associated with the execution platform component.The AADL describes interfaces and properties of execution platform components including processor, memory, communication channels, and devices interfacing with the external environment. Detailed designs for such hardware components may be specified by associating source text written in a hardware description language such as VHDL.

2.2.2.1 Processor

A processor is an abstraction of hardware and associated software that is responsible for scheduling and executing threads. It may have port, port group and bus access features, and its implementation may have memory as its subcomponents. Figure 2.12 shows both textual and graphical representations of a processor, with a memory subcomponent contained.

```
processor Intel_Linux
features
  afdx_access: requires bus access AFDX;
end Intel_Linux;

processor implementation Intel_Linux.impl
subcomponents
  HSRAM: memory RAM;
end Intel_Linux.impl;

memory RAM
end RAM;
```

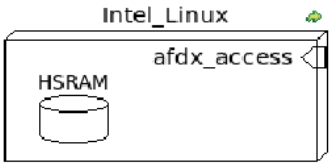


Figure 2.12: Sample processor component text and graphical representations

2.2.2.2 Device

Device abstractions represent entities that interface with the external environment of an application system. A device may represent a physical entity or its software equivalent. Devices are logically connected to application software components and "physically" connected to processors.

A device can be viewed to be a primary part of either the application system or the execution platform. Examples of devices are sensors and actuators that interface with the external physical world.

```
device RDC
features
afdx_access: requires bus access AFDX;
rate: out data port;
end RDC;

processor CPIOM
  features
    afdx_access: requires bus access AFDX;
end CPIOM;

bus AFDX
end AFDX;
```

Figure 2.13: A sample device specification

A device interacts with both execution platform components and application software components. It has physical connections to processors via bus access connections. This models software executing on a processor accessing the physical device. The logical connections are represented by connection declarations between device ports and application software component ports.

Figure 2.13 shows an excerpt from an AADL specification that describes a **device** *RDC* interacting through a **bus** *AFDX* with a **processor** *CPIOM*. The requirement for **bus access** is specified in the type declaration for *RDC*. Similarly, the need for **bus access** is declared within the **processor** type declaration for *CPIOM*. Notice that the **out data port** declared on the sensor **device** provides the rate data.

2.2.2.3 Bus

A bus represents hardware and associated communication protocols that enable interactions among other execution platform components. This communication is specified using **access** and **binding** declaration to a bus.

In the example of Figure 2.13, there are a **bus** type declaration (*AFDX*), a **processor** type declaration (*CPIOM*) and a **device** type declaration (*RDC*). Both the *CPIOM* **processor** and the *RDC* **device** declare a **requires bus access** of **bus** *AFDX*. They are connected by the **bus** *AFDX*.

We give a simple implementation of the **bus** *AFDX*. A property *Transmission_Time* is given. A bus implementation must not contain a connection, flow or subprogram call subclause. It can only have properties or mode declarations.

```
bus AFDX
end AFDX;

bus implementation AFDX.impl
  properties
    Transmission_Time => 1 ms .. 2 ms;
end AFDX.impl;
```

2.2.2.4 Memory

Memory represents storage components for data and executable code. Memory component consists of randomly accessible physical storage (i.e., RAM, ROM) or complex permanent storage (e.g., disks, reflective memory). Since they have a physical runtime presence, memory components have properties such as word size and word count.

```
memory RAM
  features
    bus1: requires bus access X_1553;
  properties
    Word_Size: Size => 8 bits;
end RAM;

bus X_1553
end X_1553;
```

In the example of Figure 2.12, there is a memory type declaraion: *RAM*. A more detailed declaration of the type *RAM* is given here with a **feature** *bus1* that establishes that all instances of *RAM* require access to the **bus** *X_1553*, and a **property** *Word_Size* that specifies the size of the smallest independently readable and writable unit of storage in the memory.

Since memory is related to the physical storage, we do not translate it in Signal, but leave it to be used in the simulation phase.

2.2.2.5 Virtual processor

A virtual processor represents a virtual machine or hierarchical scheduler. Threads can be bound to virtual processors, and a virtual processor must be bound to or must be a subcomponent of a processor.

2.2.2.6 Virtual bus

A virtual bus represents virtual channel or communication protocol that performs transmission within processors or across buses. Virtual buses can be contained in or bound to processors and buses. Virtual buses within a processor support connections between components on the same processor. Virtual buses on buses support connections between components across processors.

Virtual processor and virtual bus are two new components introduced in AADLv2.0. These two components are not taken into account in our work.

2.2.3 System component

A system represents an assembly of software, execution platform, and system components. Systems may require access to data and bus components declared outside the system, and may provide access to data and bus components declared within it. Systems may be hierarchically nested.

A system instance represents the runtime architecture of an operational physical system. It consists of application software components and execution platform components. Component type and component implementation declarations must be instantiated to create a complete system instance. A system instance that represents the containment hierarchy of the physical system is created by instantiating a top-level system implementation and then recursively instantiating the subcomponents and their subcomponents.

```
system integrated_control
end integrated_control;

system implementation integrated_control.integrated_control_system
subcomponents
control_process: process controller.speed_control;
set_point_data: data set_points;
navigation_system: system core_system.navigation;
real_time_processor: processor rt_fast.rt_processor;
hs_memory: memory rt_memory.high_speed;
high_speed_bus: bus network_bus.HSbus;
end integrated_control.integrated_control_system;
```

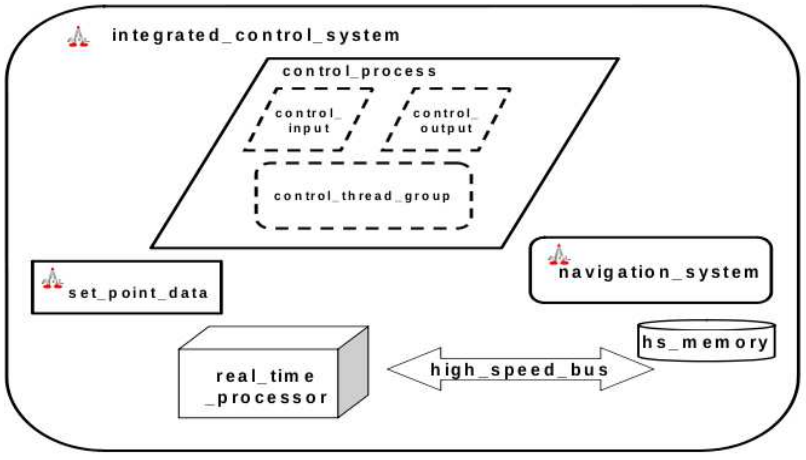


Figure 2.14: A sample system specification: textual and graphical

The composition of a system implementation is declared through subcomponents. Figure 2.14 provides both textual and graphical representations of a system *integrated_control*. The details of the type declarations are not included. The explicit subcomponent declarations are shown in the **system implementation**. It consists of a **process** *control_process*, a **data** *set_point_data*, a sub **system** *navigation_system*, a **processor** *real_time_processor*, a **memory** *hs_memory* and a **bus** *high_speed_bus*. The supporting subcomponent type or implementation declarations are not shown. In the graphical view of the **system implementation**, the subcomponents of *integrated_control_system* of the type *integrated_control* are shown.

2.3 System binding

For a complete system specification, the application component instances must be bound to appropriate execution platform components. The decisions and methods required to combine the components of a system to produce a physical system implementation are collectively called **binding**. For example, threads must be bound to processing elements, and processes must be bound to memory. Similarly, interprocessor connections must be bound to buses. These bindings are defined through **property** associations.

Binding properties are declared in the system implementation that contains in its containment hierarchy both the components to be bound and the execution platform components that are the target of the binding.

There are three categories of **binding properties** that provide support for: *allowed bindings*, *actual bindings* and *identified available memory and processor resources*.

```
system sys1
end sys1;

system implementation sys1.impl
subcomponents
  pa: process p1.impl;
  cpiom1: processor CPIOM;
  cpiom2: processor CPIOM;
properties
  Allowed_Processor_Binding =>
           (reference cpiom1, reference cpiom2) applies to pa.ta;
  Allowed_Processor_Binding => reference cpiom1 applies to pa.tb;
  Actual_Processor_Binding => reference cpiom2 applies to pa.ta;
end sys1.impl;

process p1
end p1;

process implementation p1.impl
subcomponents
  ta: thread t1;
  tb: thread t1;
end p1.impl;

thread t1
end t1;
```

Figure 2.15: Textual example of system binding

A set of properties are predeclared in the property set *AADL_Properties*, which is part of AADL specification. In this section, we only list three properties that will be used

in our transformation. The others can be referenced to in Appendix A of SAE AS5506 standard.

- The *Allowed_Processor_Binding* property specifies the set of processors that are available for binding.
- A thread is bound to the processor specified by the *Actual_Processor_Binding* property. The process of binding threads to processors determines the value of this property. If there is more than one processor listed, a scheduler will dynamically assign the thread to one at a time.
- Connections are bound to the bus, processor or device specified by the *Actual_Connection_Binding* property.

Figure 2.15 shows a textual example of system binding. The *Allowed_Processor_Binding* property declares that **processors** *cpiom1, cpiom2* are available for **thread** *pa.ta*, and **processor** *cpiom1* is available for **thread** *pa.tb*. **Thread** *ta* of **Process** *pa* is actually bound to **processor** *cpiom2* specified by the *Actual_Processor_Binding* property.

2.4 Component interaction

The interactions among components represent the connections established between interface elements at runtime. Connections establish one of the following interactions: port connections which are explicit relationship declared between ports or between port groups, component access connections that access to a common data or bus component, subprogram calls, and parameter connections. In this thesis, we mainly take into account the port connections among components.

2.4.1 Port

A port represents a communication interface for the directional exchange of **data**, **event** or both **event data** between components. Three types of ports are defined in AADL standard:

- Data port: interface without queuing.
- Event port: interface for the communication of event that may be queued.
- Event data port: interface for message transmission with queuing.

The port properties: *Queue_Size*, *Queue_Processing_Protocol*, and *Overflow_Handling_Protocol* specify queue characteristics of the event or event data ports. If an event arrives and the number of queued events (and any associated data) is more than the specified queue size, the *Overflow_Handling_Protocol* property determines the action. If the property value is *Error*, an error is emitted. If the property value is set to be *DropNewest* or *DropOldest*, the newly arrived or oldest event in the queue is dropped.

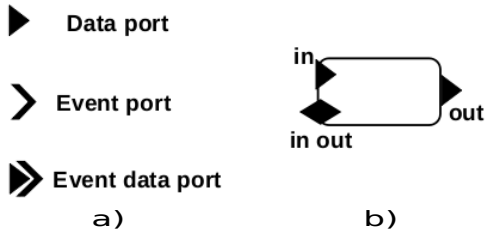


Figure 2.16: Port graphical representation

The graphical representations of data port, event port and event data port are summarized in Figure 2.16 (a).

Ports are declared as features in component type declarations. They are directional, and can be specified as **in**, **out** or **inout** (Figure 2.16 (b)).

2.4.2 Port connection

Port connections are explicit relationships declared between ports or between port groups that enable the directional exchange of data and events among components. Connection declarations between ports are shown in Figure 2.17. For example, the connection *data_con* is between the **out data port** *data_out* of the **process** *compute* (written as *compute.data_out*) and the **in data port** *data_in* of the **process** *control* (written as *control.data_in*). The connection *control_con* connects the **out data port** *control_out* of the **process** *control* to the **in data port** *control_in* of the **process** *compute*.

A port can be the source of more than one port connection, but a port cannot be the destination of more than one port connection, which means that a source port can connect to several destination ports, but the destination port has only one source port.

The type of connection between thread data ports establishes specific timing semantics for data that are transferred between originating and terminating threads.

AADL provides two types of data port communication mechanisms between threads: *immediate* and *delayed* connections. These two connections can only be applied to periodic threads.

Immediate connection For immediate port connections (**data port** $x \rightarrow y$), data transmission is initiated when the source thread completes and enters the suspended state.

Immediate connections execution orders are illustrated in Figure 2.18. The sending and receiving threads (T_1 and T_2) must share a common (multiple simultaneous) dispatch. The value provided to the receiving thread T_2 is the value produced by T_1 at its last recent execution. In Figure 2.18(a), the sending thread T_1 is dispatched at a higher frequency than the receiving thread T_2. The execution of T_2 is delayed until T_1 has completed. In such an under-sampling case, the third value produced by T_1 is ignored, since in an immediate connection, the sender always communicates with the receiver in the same dispatch frame. In Figure 2.18(b), the receiving thread T_2 is dispatched at a higher frequency than the sending thread T_1. In this case, the values available for the first two sequential executions of T_2 are the same: the value produced by the first execution of T_1.

```
51 system sys1
52 end sys1;
53
54 system implementation sys1.impl
55 subcomponents
56   control: process Control;
57   compute: process Compute;
58 connections
59   data_con: data port compute.data_out -> control.data_in;
60   control_con: data port control.control_out -> compute.control_in;
61 end sys1.impl;
62
63 process Control
64 features
65   data_in: in data port;
66   control_out: out data port;
67 end Control;
68
69 process Compute
70 features
71   data_out: out data port;
72   control_in: in data port;
73 end Compute;
```

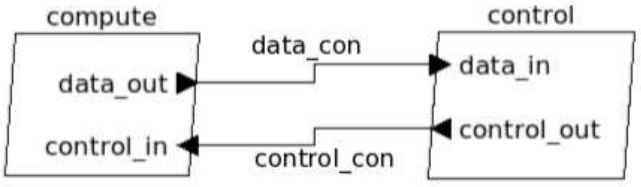


Figure 2.17: Sample declaration of port connection

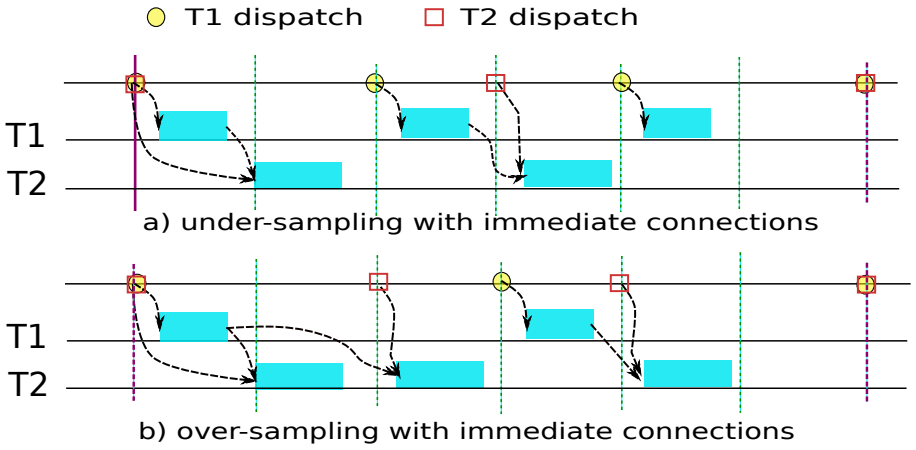


Figure 2.18: Immediate connection execution order

Delayed connection For delayed port connections (**data port** $x \twoheadrightarrow y$), the value from the sending thread is transmitted at its deadline, and is available to the receiving thread at its next dispatch.

In case of delayed connection, the value that the receiving thread receives is the output at the most recent deadline of the sending thread. Figure 2.19 shows the delayed connection execution order in case of under-sampling and over-sampling. In Figure 2.19(a), an under-sampling case is shown: the data provided to the receiving thread T_2 is the value

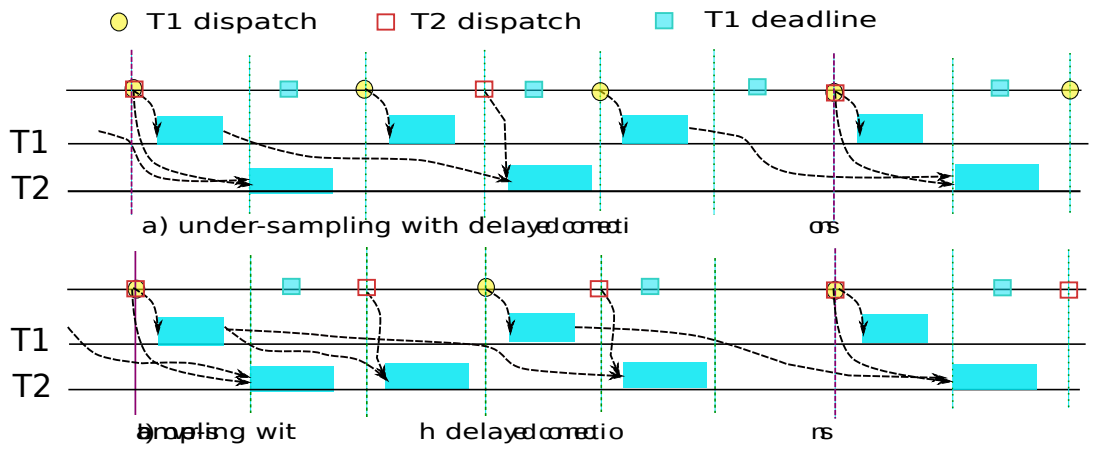


Figure 2.19: Delayed connection execution order

produced by T_1 that is available at its deadline before the dispatch of T_2. In Figure 2.19(b), the receiving thread T_2 is dispatched at a higher frequency than the sending thread T_1. In this case, the values available for the second and third sequential executions of T_2 are the same: the value produced by the first execution of T_1, because the value produced by the second execution of T_1 is not available for the third dispatch of T_2.

2.5 Flows

Flows describe externally observable flow of information in terms of application logic through a component. Flows describe actual flow sequences through components and sets of components across one or more connections. Flows specification enables the detailed description and analysis of an abstract information path through a system. The specification of an end-to-end flow involves the declaration of the elements of the flow (sources, sinks and paths), and explicit implementation declarations that describe the details of a complete path through the system. A flow can involve port connections in its specification declaration.

Flows are directional. Within a component implementation, flow declarations define the details of:

- Flow path through a component. A flow path through a component consists of alternation sequences of paths through and connections among subcomponents within the component.

  Figure 2.20 is a graphical representation of the flow path. It shows the **flows implementation** declarations through the component *cruise_control*. The **flow implementation** originates at the *brake_event* **event data port**, and sinks in the **data port** *throttle_setting*. The **flow** involves the connections *C1*, *C3* and *C5* within the component implementation, as well as the paths through the subcomponents of that implementation. Partial code specification of this flow path is shown:

```
flows
  brake_flow: flow path brake_event -> C1 ->
```

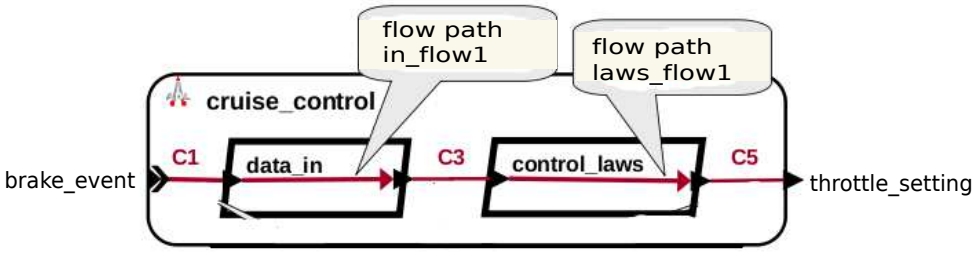


Figure 2.20: Flow path through a component

```
data_in.in_flow1 -> C3 -> control_laws.laws_flow1
-> C5 -> throttle_setting;
```

- End-to-end flows within a component. An end-to-end flow within a component involves the declaration of a path from a flow source to a flow sink within the component.

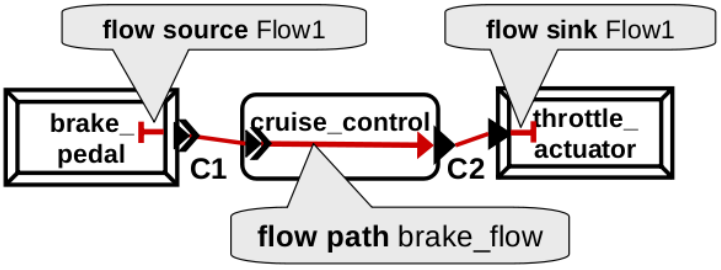


Figure 2.21: An end-to-end flow within a component

Figure 2.21 illustrates this type of declaration: an end-to-end flow is defined between the source *Flow1* in the device component *brake_pedal*, and the sink *Flow1* in the device component *throttle_actuator*. Partial code specification of this end-to-end flow is illustrated:

```
flows
  brake_flow: end to end flow brake_pedal.Flow1 -> C1 ->
  cruise_control.brake_flow -> C2 -> throttle_actuator.Flow1;
```

2.6 AADL behavior annex

The AADL *Behavior Annex* [74] is an extension of the core of the standard to offer a way to specify the local functional behavior of the components. It supports describing precisely the behaviors, such as port communication, computation, timing, etc.

A Behavior Annex can be attached to a thread or a subprogram: threads or subprograms start from an initial state, and a transition to a complete (resp. return) state ends a thread (resp. subprogram). Transitions may be guarded by conditions, and actions may be attached.

```
thread implementation door_handler.imp
annex behavior_specification {**
  states
   s0: initial complete state;
  transitions
   s0 -[on (dps > 3) and handle]-> s0 {
         warn_diff_pres:=true;
         door_info:=locked;
         if (on_ground) door_locked:=false;
         else door_locked:=true; end if;
        };
   s0 -[on not (dps > 3 and handle)]-> s0 {
         warn_diff_pres:=false;
         door_info:=locked;
         if (in_flight) door_locked:= true;
         else door_locked:=false; end if;
        };
  **};
end door_handler.imp;
```

Figure 2.22: Behavior Annex example

The example presented in Figure 2.22 is a behavior specification of a *door_handler* thread from a SDSCS (Simplified Doors and Slides Control System) [109]. It comprises two **transitions** with an **initial** state *s0* which is also a **complete** state. The thread will execute the transitions depending on the guarded conditions. Take the first transition for instance, when the guard condition (**on** (dps > 3) **and** handle) is satisfied, it is triggered, the attached actions (two assignments and a condition action) are executed. When the execution finishes, it will enter back to the state *s0*.

In this section, we only give a brief introduction of the AADL behavior annex. The detailed semantics of transitions and actions will be described in Chapter 6.

2.7 Avionics system architecture and ARINC653

The second version of AADL standard is defining the ARINC653 architectural elements in AADL architecture, included in ARINC653 annex (but not completed yet). In this section, we will give an overview of the ARINC653 standard. First, a brief introduction of avionics system architecture is presented.

2.7.1 Avionics system architecture overview

In the field of avionics, increasing complexity and high criticality of embedded real-time systems are a number of challenges relating to their development. These challenges include the correct system design, the development effort, the system reliability, and the time of placing the "product" on the market. Thus there is a great need to design methodologies that take into account the mentioned challenges. Such methodologies must include formal specifications, to make possible verification and analysis. On the other hand, the automatic generation of code (possibly distributed) should also be possible.

In this section, we will introduce some concepts of avionics systems. Two architectures in avionic systems are presented: the *Federated* architecture and the *Integrated Modular Avionics* architecture.

2.7.1.1 Federated architecture

Traditionally, avionics functions are implemented such that each function has its dedicated fault-tolerant computer system. This architecture is called *federated* [143]. A great advantage of such an architecture is fault containment. However, there is a potential risk of massive usage of computing resources, since each function may require its dedicated computer. As a result, maintenance costs may increase rapidly.

2.7.1.2 Integrated Modular Avionics architecture

Advances in the computer technology encouraged the avionics industry to deal with major obstacles in federated architecture, with an integrated suite of control modules that share the computing resources. The new approach, which is called *Integrated Modular Avionics* (IMA) [39], introduces methods that can achieve high levels of reusability and cost effectiveness.

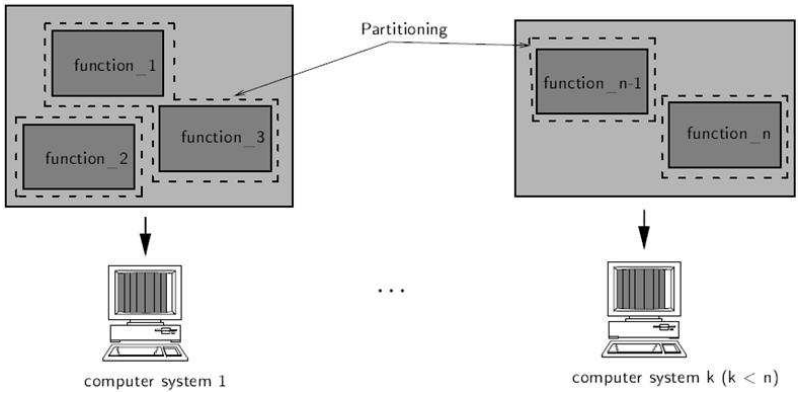


Figure 2.23: IMA: different functions can share a fault-tolerant computer

A main feature in a IMA architecture is that several avionics applications can be hosted on a single, shared computer system (Figure 2.23). They are guaranteed a safe static allocation of shared resources, so that no fault propagation occurs from one component to another one. This is addressed through a functional *partitioning* of the applications with respect to available time and memory resources [37].

The avionics standard ARINC defines the basic principles of partitioning and a set of services, following the IMA architecture.

2.7.2 ARINC standard

In this section, we will present the notions defined in document 653 by the ARINC standard, which follows the IMA architecture.

A *partition* is a logical allocation unit resulting from a functional decomposition of the system. IMA platforms consist of a number of *modules* grouped in cabinets throughout the aircraft. A *module* can contain several *partitions* that possibly belong to functions of different criticality levels. Any memory access outside of a *partition*'s defined area is prohibited. The *processor* is allocated to each *partition* for a fixed time window within a

major time frame. *Partitions* are composed of *processes* which represent the basic execution units. *Processes* in a partition share the resources allocated to the partition, and can be executed concurrently with other processes of the same partition [87].

The ARINC653 specification [37] is dedicated to provide a general interface with the set of basic services to access the operating system and other system-specific resources. This interface, called APEX (APplication-EXecutive), presented in Figure 2.24, provides the execution control of the partitions, including the communication between them. It is constituted of a set of services that are called APEX services.

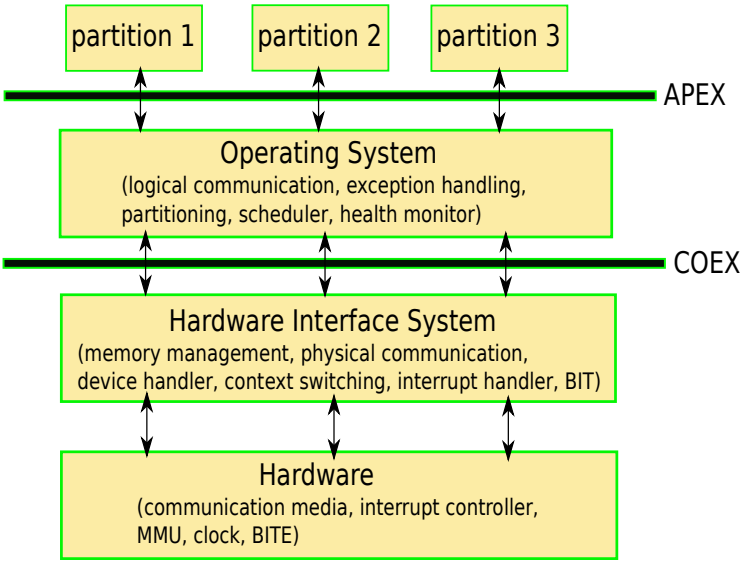


Figure 2.24: Software architecture for IMA system

The IMA software architecture consists of hardware interface system, operating system and application software, as shown in Figure 2.24. Combined with the hardware, the hardware interface system forms a core module to support operating system and application execution via the COEX (COre-EXecutive) interface. The operating system executive implements a set of services defined in the APEX interface, such that application software can use the system resources and control the scheduling and communication.

2.7.2.1 Partition

Central to the IMA philosophy is the concept of partitioning, where the functions resident on a core module are partitioned with respect to space and time. A partition represents such a unit of the application designed to satisfy these constraints. It is characterized by some unique attributes, such as configuration and context. These information are used to manage the partitions within a module.

Partition management The core module OS is responsible for enforcing partitioning and managing the individual partitions within that core module.

A processor is allocated to each partition for a fixed time window within a major time frame maintained by the *module-level OS*. In other words, through a predefined configu-

ration table of partition windows, the APEX imposes a cyclic scheduling to ensure that each partition receives a fixed amount of processing time.

The order of partition activation is defined at configuration time using configuration tables. This provides a deterministic scheduling methodology, where the partitions are furnished with a predetermined amount of time to access processor resources. Timing partitioning therefore ensures each partition uninterrupted access to common resources during their assigned time periods.

Each partition has predetermined areas of memory allocated to it. These unique memory spaces are identified based upon the requirements of the individual partitions, and vary in size and access rights. Memory partitioning is ensured by prohibiting memory accesses outside of a partition's defined memory area.

Partition communication A partition cannot be distributed over multiple processors, neither in the same module nor in different modules. Suitable mechanisms and devices are provided for communication between partitions.

The basic mechanisms for linking partitions are logical *ports* and *channels*. A channel defines a logical link between one source and one or more destination partitions. Partitions have access to channels via defined access points, called ports. Each individual channel may be configured to operate in a specific mode: sampling mode or queuing mode. No queuing is performed in the former mode: a message remains in the source port, until it is transmitted via the channel, or it is overwritten. In the queuing mode, ports are allowed to store messages from a source partition in queues, until they are received by the destination partition. No messages will be lost in this mode. The queuing discipline for messages is bounded First-In First-Out (FIFO).

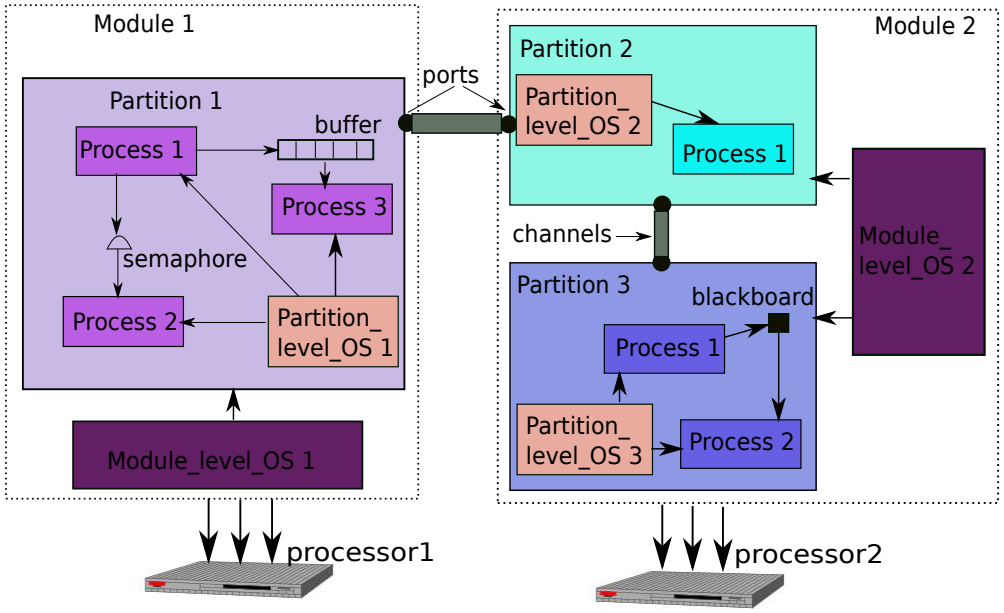


Figure 2.25: An example of IMA application partitioning

Figure 2.25 describes an application executed on two processors. It consists of three partitions: *partition 1*, *partition 2* and *partition 3*. The first partition executes on the first

processor, the other two share the second processor. Each partition comprises processes. We will present the process and its characteristics in the next subsection.

2.7.2.2 Process

Within each application partition, the basic execution unit is a *process*. Processes in a partition share the resources allocated to the partition, and can be executed concurrently with other processes of the same partition. When a partition is activated, its owned processes run concurrently to perform the functions associated with the partition.

Process management Each process is uniquely characterized by information useful to the *partition-level OS*, which is responsible for the correct execution of processes within a partition.

The scheduling algorithm within each partition is priority preemptive. At any process rescheduling event, the scheduler always selects the highest priority process in the ready state to receive processor resources. That process will control processor resources until another process reschedule event occurs.

Process communication Communication mechanisms between processes are *buffer* and *blackboard*, which can support the communication of a single message type between multiple source and destination processes. The communication is indirect, in that the participating processes address the buffer or blackboard rather than the opposing processes directly, thus providing a level of process independence.

- The bounded *buffer* allows to send and receive messages following a FIFO order. Each new instance of a message may carry uniquely different data and is not allowed to overwrite previous ones during the transfer. No message should be lost.
- The *blackboard* is used to display and read message. No message queuing is allowed. Any message written to a blackboard remains there, until it is either cleared, or overwritten by a new instance of the message. This allows processes to display messages at any time, and processes to access the latest message at any time.

Inter-process synchronization is achieved by *semaphore* and *event*. The *event* permits the application to notify some processes in the partition of the occurrence of a condition.

2.7.3 AADL and ARINC

ARINC653 is an avionics standard following the IMA architecture. AADL can be used for ARINC653 modeling, e.g., processes as *partitions*. In this section, we list some similarities between the two.

A main feature of ARINC653 is the *partitioning*: several avionics applications (*processes*) can be hosted on a single *partition*. They can share the resources in this *partition*, but have no right to access the resources in other *partitions*. The feature is similar to AADL process. The AADL process represents a protected address space, a space partitioning where protection is provided from other components accessing anything inside

the process. Hence, both the ARINC *partition* and AADL process have a "*partitioning*" application.

The communication mechanisms defined in ARINC support both queuing and no-queuing message communication. This is similar to the AADL connection. The difference is that ARINC does not take into account the hardware bus connection between different components.

The ARINC scheduling mechanism can also be used as a targeted operating system of AADL. The difficulty is that the scheduling in ARINC653 is divided in two levels: the *module_level_OS* and *partition_level_OS*. The former one is a static scheduling: the *modules* are scheduled based on a predefined configuration table. However, the *partition_level_OS* is still suitable for modeling the scheduling of the AADL threads in the same process.

This utilization needs improvements, which is addressed in AADLv2 as an annex for common guidelines for ARINC653 modeling, and extensible to other partitioned architectures.

2.8 AADL components considered in this thesis

In this thesis, we take into account the architecture and functionality aspects of an AADL system. The architecture aspects considered in this thesis mainly consist of the basic software components, the hardware for executing and communications, the composite system component, and some related properties. The Behavior Annex is considered to specify the local functional behaviors of the components.

As an architecture description language, AADL can describe the connected components that form an architecture. A subset of components is considered and implemented in this work: the thread (especially for periodic thread), which is the main executable and schedulable component; the processor, which is responsible for executing and scheduling (a specific static scheduler is implemented as an instance); the bus, that represents the communication execution platform; the process, which is the protected address space; the subprogram, that is the callable server function provider; the data type; the device, which is the external environment inerface; the data/event/event data port, that represents the communication interface; immediate and delayed data port connections; top-level system component; and properties related to thread temporal specifications and process allocation information. Each component instance is considered with its associated type or implementation. The transformation of these components will be presented in Chapter 5.

The functionality of a component specifies its functional behavior when it is in execution. The behaviors in the Behavior Annex are specified in order to perform model checking and advanced code generation. In reason of early system validation and verification, the interpretation of Behavior Annex is considered in this thesis as one of the contributions in Chapter 6.

2.9 Conclusion

This chapter presented the language AADL, intended for modeling embedded real-time system architecture. This language is used to describe the structure of such systems as an

assembly of software components that are mapped to an execution platform. It supports the modeling of both software and hardware.

The components are presented in three categories: software, execution platform and system. The system binding, component interactions, thread scheduling and execution are also presented for a complete system specification. These notations will be specified in detail and transformed into Signal in Chapter 5. We also give a short introduction of an extensible annex: Behavior Annex. The formal syntax and semantics of Behavior Annex will be defined in Chapter 6. A brief introduction of the Integrated Modular Avionics (IMA) architecture, and the associated standard ARINC653, is presented then, which is used as a basic framework in our transformation.

Chapter 3

The Signal Language and modeling ARINC in Signal

Contents

3.1 Signal language 74
3.1.1 Signal, execution, process 74
3.1.2 Data types 75
3.1.3 Elementary processes 76
3.1.4 Process operators 77
3.1.5 Parallel semantic properties 78
3.1.6 Modularity features 79
3.1.7 Endochronous acyclic processes 80
3.1.8 Time domains and communications in Signal 80
3.1.9 Non-determinism modeling in Signal 82
3.1.10 Adequacy of Signal for AADL modeling 83
3.2 Modeling ARINC concepts in Signal 84
3.2.1 Partitions 84
3.2.2 Partition-level OS 85
3.2.3 Processes 86
3.2.4 APEX services 87
3.3 Conclusion 88

In this chapter, the synchronous language Signal and its polychronous model of computation will be presented, with a description of its syntax and semantics. Since the Signal language is used to describe a real world avionic application based on the recent Integrated Modular Avionics concept (IMA), we will also present the modeling of the IMA ARINC components in Signal.

The synchronous approach is one of the possible solutions for a safe design of embedded systems. Its mathematical basis provides formal concepts that favor the trusted

design of embedded real-time systems. The multi-clock or polychronous model stands out from other synchronous models, by its uniform framework. It allows the design of systems, where each component holds its own activation clock, as well as single-clocked systems in a uniform way. This feature makes the Signal semantics closer to AADL semantics than other pure synchronous or asynchronous models. This will facilitate the system validation.

The data-flow synchronous Signal language is dedicated to the design of embedded systems for critical application domains. The unique features of the relational model behind Signal are to provide the notion of polychrony: the capability to describe circuits and systems with several clocks, and to support refinement—the ability to assist and support system design from the early stages of requirement specification, to the later stages of synthesis and deployment.

The IMA architecture is a recently proposed architecture in avionics applications. In an IMA system, several functions can be grouped into one core module, and are allowed to share the same computer resources. They are guaranteed a safe allocation of shared resources, so that no fault propagation occurs from one component to another. Avionics applications which rely on the avionics standard ARINC653, based on the IMA architecture, can be specified in the Signal model. A library of APEX ARINC services is provided in Signal.

The remainder of this section is organized as follows: section 3.1 first introduces the Signal language, and section 3.2 concentrates on the modeling of ARINC concepts in Signal. The modeling of a library of components based on the avionic APEX-ARINC standard is presented. Finally, conclusions are given.

3.1 Signal language

Signal [110, 111, 112, 49, 113, 89, 60] is a declarative language expressed within the polychronous model of computation. Signal relies on a handful of primitive constructs, which can be combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. In the following, we present the main features of the Signal language and its associated concepts. We give a sketch of the primitive constructs and a few derived constructs often used. For each of them, the corresponding syntax and definition are mentioned. Since the semantics of Signal is not the main topic of this thesis, we give simplified definitions of operators. For further detail, interested readers can refer to [115, 58].

3.1.1 Signal, execution, process

A *pure signal* is a total function $T \rightarrow D$, where T, its *time domain*, is a chain in a partial order (for instance an increasing sequence of integers) and D is some data type; we name *pure flow* of such a pure signal s, the sequence of its values in D. In this context, the lack of value is usually represented by *nil* or *null*; we represent it by the symbol # (stating for "no event").

For all chains TT, $T \subset TT$, a pure signal $s : T \rightarrow D$ can be extended to a *synchronized signal* $ss : TT \rightarrow D_{\#}$ (where $D_{\#} = D \cup \{\#\}$) such that for all t in T, $ss(t) = s(t)$ and

$ss(t) = \#$ when t is not in T; where we name *synchronized flow of a synchronized signal* ss the sequence of its values in $D_{\#}$; conversely we name *pure signal of the synchronized flow* ss, the unique pure signal s from which it is extended and *pure flow of* ss the pure flow of s. The *pure time domain* of a synchronized signal is the time domain of its pure signal. Given a pure signal s (respectively, a synchronized signal ss), s_t (respectively, ss_t) denotes the t^{th} value of its pure flow (respectively, its synchronized flow).

An *execution* is the assignment of a tuple of synchronized signals defined on the same time domain (as synchronized signals), to a tuple of variables. Let TT be the time domain of an execution, a *clock* in this execution is a "characteristic function" $clk : TT \rightarrow \{\#, true\}$; notice that it is a synchronized signal. The clock of a signal x in an execution is the (unique) clock that has the same *pure time domain* as x; it is denoted by $\hat{x}$.

A *process* is a set of executions defined by a system of equations over signals that specifies relations between signal values and clocks. A *program* is a process.

Two signals are said to be *synchronous in an execution* iff they have the same clock (or equivalenty the same pure time domain in this execution). They are said to be *synchronous in a process* (or simply *synchronous*), iff they are *synchronous* in all executions of this process.

Consider a given operator which has, for example, two input signals and one output signal all being synchronous. They are *logically related* in the following sense: for any t, the t^{th} token on the first input is evaluated with the t^{th} token on the second input, to produce the t^{th} token on the output. This is precisely the notion of *simultaneity*. However, for two occurrences of a given signal, we can say that one is before the other (*chronology*). Then, for the synchronous approach, an *event* is associated with a set of logically instantaneous ordered calculations and communications.

3.1.2 Data types

A flow is a sequence of values that belong to the same data type. Standard data types such as Boolean, integer... (or more specific ones such as *event*–see below) are provided in the Signal language. One can also find more sophisticated data types such as *sliding window* on a signal, *bundles* (a structure the fields of which are signals that are not necessarily synchronous), used to represent union types or signal multiplexing.

1. The `event` type: to be able to compute on (or to check properties of) clocks, Signal provides a particular type of signals called `event`. An `event` signal is *true* if and only if it is present (otherwise, it is #).

2. Signal declaration: `tox x` declares a signal `x` whose common element type is `tox`. Such a declaration is a process that contains all executions that assign to `x` a signal the image of which is in the domain denoted by `tox`.

In the remainder of this chapter, when the type of a signal does not matter or when it is clear from the context, one may omit to mention it.

3.1.3 Elementary processes

An elementary process is defined by an equation that associates with a signal variable an expression built on operators over signals; the arguments of operators can be expressions and variables.

- **Stepwise extensions.** Let `f` be a symbol denoting a n-ary function $[\![f]\!]$ on values (e.g., Boolean, arithmetic or array operation). Then, the Signal expression

  ```
  y := f(x1,...,xn)
  ```

  defines the process equal to the set of executions that satisfy:

  $$\begin{cases} - \text{ the signals } y, x1, ..., xn \text{ are synchronous,} \\ - \text{ their pure flows have same length } l \text{ and satisfy } \forall t \leq l, y_t = [\![f]\!](x1_t, \ldots, xn_t) \end{cases}$$

  If f is a function, its stepwise extension is a *pure flow* function. Infix notation is used for usual operators.

- **Delay.** This operator defines the signal whose t^{th} element is the $(t-1)^{th}$ element of its (pure flow) input, at any instant but the first one, where it takes an initialization value. Then, the Signal expression

  ```
  y := x $ 1 init c
  ```

  defines the process equal to the set of executions that satisfy:

  $$\begin{cases} -y, x \text{ are synchronous,} \\ - \text{ pure flows have same length } l \text{ and satisfy } \forall t \leq l, \begin{cases} (t > 1) \Rightarrow y_t = x_{t-1} \\ (t = 1) \Rightarrow y_t = c \end{cases} \end{cases}$$

  The delay operator is thus a *pure flow* function. For short, it can be denoted `y:=x$` and the initialization can be implicit.

  Derived operator

  - *Constant*: `x := v`; when `x` is present its value is the constant value `v`; `x := v` is a derived equation equivalent to `x := x $ 1 init v`.

    Note that this equation does not have input: it is a pure flow function with arity 0.

- **Sampling.** This operator has one data input and one Boolean "control" input. When one of the inputs is absent, the output is also absent; at any logical instant where both input signals are defined, the output is present (and equal to the current data input value) if and only if the control input holds the value *true*. Then, the Signal expression

  ```
  y := x when b
  ```

defines the process equal to the set of executions that satisfy:

$$\begin{cases} -y, x, b \text{ are extended to the same infinite domain } T, \text{ respectively as } yy, xx, bb, \\ - \text{ synchronized flows are infinite and satisfy } \forall t \in T, \begin{cases} (bb_t = true) \Rightarrow yy_t = xx_t \\ (bb_t \neq true) \Rightarrow yy_t = \# \end{cases} \end{cases}$$

The `when` operator is thus a *synchronized flow* function.

- **Deterministic merging.** The unique output provided by this operator is defined (i.e., with a value different from #) at any logical instant where at least one of its two inputs is defined (and non-defined otherwise); a priority makes it deterministic. Then, the Signal expression

```
z := x default y
```

defines the process equal to the set of executions that satisfy:

$$\begin{cases} - \text{ the time domain } T \text{ of } z \text{ is the union of the time domains of } x \text{ and } y, \\ -z, x, y \text{ are extended to the same infinite domain } TT \supseteq T, \text{ resp. as } zz, xx, yy, \\ - \text{ synchronized flows satisfy } \forall t \in TT, \begin{cases} (xx_t \neq \#) \Rightarrow zz_t = xx_t \\ (xx_t = \#) \Rightarrow zz_t = yy_t \end{cases} \end{cases}$$

The `default` operator is thus a *synchronized flow* function.

3.1.4 Process operators

A process is defined by composing elementary processes.

- **Restriction.** This operator allows one to consider as local signals a subset of the signals defined in a given process. If `x` is a signal with type `tox` defined in a process `P`,

```
P where tox x   or  P where x
```

defines a new process `Q` where communication ways (for composition) are those of `P`, except `x`. Let A the variables of `P` and B the variables of `Q`: we say that `P` is *restricted* to B, and executions of `P` are restricted in `Q` to variables of B. More precisely, the executions in `Q` are the executions in `P` from which `x` signal is removed (the projection of these executions on remaining variables). This has several consequences:

  - if `P` has a single output signal named `x`, then `P where x` is a pure synchronization process. The generated code (if any) is mostly a synchronization code used to ensure signal occurrence consumptions.
  - if `P` has a single signal named `x`, `P where x` denotes the neutral process: it cannot influence any other process. Hence no code is generated for it.

Derived equations

  - `(| P where x, y |)` $=_{\Delta}$ `(| (| P where x |) where y |)`
    For short, it can be denoted `P/x,y`.

- **Parallel composition.** Resynchronizations (by freely inserting #'s) have to take place when composing processes with common signals. However, this is only a formal manipulation. If `P` and `Q` denote two processes, the *composition* of `P` and `Q`, written `(| P | Q |)` defines a new process in which common names refer to common synchronized signals. Then, `P` and `Q` communicate (synchronously) through their common signals. More precisely, let `XPP` (resp., `XQQ`) be the variables of `P` (resp., `Q`); the executions in `(| P | Q |)` are the executions whose projections on `XPP` are executions of `P`, and projections on `XQQ` are executions of `Q`. In other words, `(| P | Q |)` defines the set of behaviors that satisfies both `P` and `Q` constraints (equations).

  Polychrony example : `(| x := a | y := b|)` defines a process that has two independent clocks. This process is a Kahn [102] process (i.e., is a flow function); it can be executed as two independent threads, on the same processor (provided that the scheduler is fair) or on distinct processors; it can also be executed as a single reactive process, scanning its input and then executing none, one or two assignments depending on the input configuration.

3.1.5 Parallel semantic properties

Process expression in normal form. The following properties of parallel composition are intensively used to compile processes:

- associativity: `(| P | Q |) | R` $\equiv$ `P | (| Q | R |)`
- commutativity: `P | Q` $\equiv$ `Q | P`
- idempotence: `P | P` $\equiv$ `P` is satisfied by processes that do not produce side effects (for instance due to call to system functions). This property allows to replicate processes.
- externalization of restrictions: if `x` is not a signal of `P`, `P | (| Q where x |)` $\equiv$ `(| P | Q |) where x`

Hence, a process expression can be normalized, modulo required variable substitution, as the composition of elementary processes, included in terminal restrictions.

Process abstraction. A process *Pa* is, by definition, a process abstraction of a process *P* if $P|Pa = P$. This means that every execution of *P* restricted to variables of *Pa* is an execution of *Pa* and thus all safety properties satisfied by executions of *Pa* are also satisfied by executions of *P*. This is the key concept to verify properties of AADL descriptions.

3.1.6 Modularity features

Process model Given a process (a set of equations) P_body, a *process model* MP associates an interface with P_body, such that P_body can be expanded using this interface. A process model is a Signal term

```
process MP ( ? t_I1 I1; ...; t_Im Im;
             ! t_O1 O1; ...; t_On On;)
   P_body
```

that specifies its typed input signals after "?", and its typed output signals after "!". Assuming that there is no name conflict (such a conflict is solved by trivial renaming), the instantiation of MP is defined by:

```
(| (Y1, ..., Yn) := MP(E1, ..., Em) |)
```

$=_{\Delta}$

```
(| I1:=E1 |...| Im:=Em | P_body | Y1:=O1 |...| Yn:=On |)
     where t_I1 I1; ...; t_Im Im; t_O1 O1; ...; t_On On;
```

Example Here we use primitive operators to define a rudimentary increasing counter, that can be reset to 0. The process **Count** accepts an input **reset** signal and delivers the integer output signal **val**. The process allows input signal **reset** and output **val** to have independent clocks.

The local variable **counter** is initialized to 0 and stores the previous value of the signal **val** (equation **counter := val$ 1 init 0**). When an input **reset** occurs, the signal **val** is reset to 0 (expression (**0 when reset**)). Otherwise, the signal **val** takes an increment of the variable **counter** (expression (**counter+1**)).

process Count =
(? event reset; ! integer val;)
(| counter := val $1 **init** 0
| val := (0 **when** reset) **default** (counter + 1)
|) **where** integer counter;
end;

Figure 3.1 presents a trace of the **Count** process execution. The activity of **Count** is governed by the clock of its output **val** which differs from that of its input **reset**. If the signal **val** is solicited by the environment, then either **reset** is absent and **Count** increments **val**, or **reset** is present and **Count** sets **val** to 0. **Count** is a polychronous process.

count event	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14
reset		true					true			true				
val	1	0	1	2	3	4	0	1	2	3	0	0	1	2
counter	0	1	0	1	2	3	4	0	1	2	3	0	0	1

Figure 3.1: Execution trace of process **count**

3.1.7 Endochronous acyclic processes

When considering embedded systems specified in Signal, the purpose of code generation is to synthesize an executable program that is able to deterministically compute the value of all signals defined in a process. Because these signals are timed by symbolic synchronization relation, one first needs to define a function from these relations to compute the clock of each signal. We say that a process is *endochronous* when there is a unique (deterministic) way to compute the clocks of its signals. Note that, for simulation purpose, one may wish to generate code for non deterministic processes (for example, partially defined ones) or even for processes that may contain deadlocks. Endochrony is a crucial property for processes to be executable: an endochronous process is a function over pure flows. It means that the pure flows resulting from its execution on an asynchronous architecture do not depend on propagation delays or operator latencies. It results from this property that a network of endochronous processes is a Kahn Process Network (KPN) and thus is a function over pure flows [60]. But it is not necessarily a function over synchronized flows: synchronizations related to the number of #'s are lost because #'s are ignored.

3.1.8 Time domains and communications in Signal

In this section, we introduce the reader to AADL-Signal semantic gaps and similarities. Signal provides some derived operators, operations and tools that can bridge the gaps.

- **Signal clock tools.** Clocks can be strongly related to AADL time domains and operators are defined to handle clocks.

  - *Clock of a signal*: `^x` returns the *clock* of `x`; it is defined by `(^x)` $=_{\Delta}$ `(x = x)`, where = denotes the stepwise extension of usual equality operator.
  - *Clock selection*: `when b` returns the clock that represents the (implicit) set of instants at which the signal `b` is *true*; in semantics, this clock is denoted by $[b]$. `(when b)` $=_{\Delta}$ `(b when b)` is a pure flow function.
  - *Clock union (or clock max)*: `x1^+x2` (denoted by $\hat{+}$ as a semantic operator) returns an event signal that is present iff `x1` or `x2` is present. `(x1^+x2)`$=_{\Delta}$`((^x1) default (^x2))`
  - *Clock difference*: `x1^-x2` returns an event signal that is present iff `x1` is present and `x2` is absent. `(x1^-x2)`$=_{\Delta}$`(when ((not ^x2) default ^x1))`
  - *Null clock*: the signal `when(b when (not b))` is never present: it is called *null clock* and is denoted by `^0` in the Signal syntax, $\hat{0}$ as a semantic constant.

- *Clock product*: `x1^*x2` (denoted by $\hat{*}$ as a semantic operator) returns the clock that represents the intersection of pure time domains of the signals `x1` and `x2`. When their clock product is $\hat{0}$, `x1` and `x2` are said to be *exclusive*. `(x1^*x2)` $=_{\Delta}$ `(^x1) when (^x2))`
- *Synchronization*: `x1^=x2` specifies that `x1` and `x2` are synchronous. `(| x1 ^= x2 |)` $=_{\Delta}$ `(| h := (^x1 = ^x2) |) where h`
- *Clock inclusion*: `x1 ^< x2` specifies that time domain of `x1` is included in time domain of `x2`. `(| x1 ^< x2 |)` $=_{\Delta}$ `(| x1 ^= (x1^*x2) |)`

- **Signal periodic clocks.** Periodic clocks can be specified in Signal using affine clock relations. A calculus of affine clock relations has been defined [147, 163, 148], in which symbolic signal clocks are associated with affine functions: two periodic signals x, y are said in (n, ϕ, d)-affine relation iff their respective clocks $\hat{x}$, $\hat{y}$ can be expressed as functions $\hat{x} = \{n \cdot t + \phi_1 \mid t \in \hat{z}\}$ and $\hat{y} = \{d \cdot t + \phi_2 \mid t \in \hat{z}\}$ of a common reference of discrete time $\hat{z}$ (n, d, ϕ_1, ϕ_2 are integers, $n, d > 0$ and $\phi = \phi_2 - \phi_1$).

  Affine clock relations yield an expressive calculus for the specification and the analysis of time-triggered systems. A particular case of affine relation is the case of $(1, \phi, d)$-affine relation, with $\phi \geq 0$: $\hat{y}$ is a subsampling of positive phase ϕ and strictly positive period d on $\hat{x}$. Most of the decidable and algorithmically affordable analysis concerns $(1, \phi, d)$-affine relations.

  In Polychrony, the Signal clock calculus implements synchronisability rules based on properties of affine relations, against which synchronization constraints can be assessed.

- **Communications.** General communication principle in Signal is broadcast, which means that, one signal is communicated as several ones. It is also possible in Signal to have several signals or expressions associated with one signal by partial definitions.

  - *Partial signal definition*: `y ::= x` is a partial definition for the signal `y` which is equal to `x`, when `x` is defined; when `x` is not defined its value is free. `(|y ::= x|)` $=_{\Delta}$ `(|y := x default y|)`

    This process is generally non deterministic. Nevertheless, it is very useful to define components such as transitions in automata, or modes in real-time processes, that contribute to the definition of the same signal.

    The clock calculus can compute sufficient conditions to guarantee that the overall definition is consistent (different components cannot give different values at the same instant) and total (a value is given at all instants of the time domain of `y`).

- **Tools for deferring communications.** Deferring communications can be implemented in Signal using memory cell and FIFO.

  - `y := x cell c init x0` behaves as a synchronized memory cell: `y` is present with the most recent value of `x` when `x` is present or

`c` is present and *true*. It is defined by the following program: `(| y := x default (y $1 init x0) | y ^= x ^+ when c |)`

- `y := var x init x0` behaves as a standard memory cell: when `y` is present, its value is the most recent value of `x` (including the current instant); the clock of `x` and the clock of `y` are mostly independent (the single constraint is that their time domains belong to a common chain). It is defined by the following program: `y := (x cell ^y init x0) when ^y`
- *Bounded FIFO* [86] is intended to be used for message exchanges between several entities. A basic FIFO queue works as follows: on a write request, the incoming message is inserted in the queue regardless of its size limit. If the queue was full, the oldest message is lost. The other messages are shifted forward, and the incoming one is put at the end of the queue. On a read request, there is an outgoing message: if the queue was empty, an arbitrary message called *default message* is returned; otherwise the outgoing message is the message that has been read last. The corresponding Signal interface of the basic FIFO is given below [86]:

```
process basic_FIFO =
   { type message_type; integer fifo_size, message_type default_mess;}
   ( ? message_type mess_in; event access_clock;
     ! message_type mess_out; integer nbmess; boolean OK_write, OK_read;)
```

A safe queue is also modeled in [86]. Read and write control are added for the purpose of ensuring a safe access to the basic queue. A safe queue can serve in the description of some communication protocol such as the LTTA (Loosely Time-Triggered Architectures) protocol [53].

- **Tools for breaking atomicity.** For a Signal program, the fastest clock of a process is not always an input clock, new instants can be inserted between existing ones. Oversampling enables the specification of constraints between inputs and outputs in such a way that no further input may occur as long as the given constraints are not met by the (intermediate) calculations of the output [115]. The abstract notion of absence between two occurrences of useful values along a signal is flexible enough to enable reasoning on the successive refinements and transformations of the description of a system.

3.1.9 Non-determinism modeling in Signal

Determinism is a key property for critical program execution. However, specification level, property description, abstractions need non-determinism resulting from partial descriptions. In the Polychrony framework, one can give the non-determinism modeling and refine it to determinism by adding the constraints. There existes several ways to illustrate it, in verification, since the Signal model does not need to be executable, so it may be a non-deterministic specification.

For example, a process

```
process oracle(? ! x,y;)
(| x^= when z
```

```
  | y^= when not z
  |)
 where boolean z; end;
```

which could be used by Sigali [126] for verifier whether x and y are execlusive.

For the simulation which needs to be an executable specfication, we can parameterize such a system with a clock "oracle" of a non-deterministic choice. This could be useful in testing, with the automatically generated sequences of tests.

```
process oracle (? boolean z; ! x,y;)
(| x^= when z
 | y^= when not z
 |)
```

Here, "z" is then provided by the test sequences, or simply by building a distributed simulator from sepetated compiled Signal modules (not by the verification of these isochronous properties). Therefore in principle, we can effectively model the non-determinism in Signal.

3.1.10 Adequacy of Signal for AADL modeling

In the polychronous language Signal, time is represented by partially ordered synchronization and scheduling relations, to provide an additional ability to model high-level abstractions of systems paced by multiple clocks: globally asynchronous systems. The multi-clock model stands out from other synchronous specification models and adapts well for AADL thanks to several capabilities:

- **Multi-clock.** The multi-clock feature allows Signal to model systems with several clocks, where each component holds its own activation clock, as well as single-clocked systems, in a uniform way. This feature well suits for the GALS architecture.

- **Refinement.** Polychrony supports formal system design refinement, from the early stages of requirements specification, to the late stages of synthesis and deployment. In the polychronous model of computation, one can design a system with partially ordered clocks, and then refine it to obtain master-clocked components, integrated within a multi-clocked architecture.

- **Modularity.** A great advantage of its modularity feature is its convenience for component-based design approach, that allows modular development of complex systems. This feature makes it possible to address the design of each component separately.

- **Parallelism model.** Synchronous parallel composition reduces programming complexity. Parallel composition relates the activation clocks of the different components, allowing to build complex systems.

3.2 Modeling ARINC concepts in Signal

The Polychrony design environment includes a library in Signal containing real-time executive services defined by ARINC [37], which was designed by Abdoulaye Gamatié [83]. It relies on a few basic blocks [84, 85], which allow to model partitions: APEX-ARINC-653 services, an RTOS model and executive entities. In the following, each block and the way its corresponding Signal model is obtained is shown.

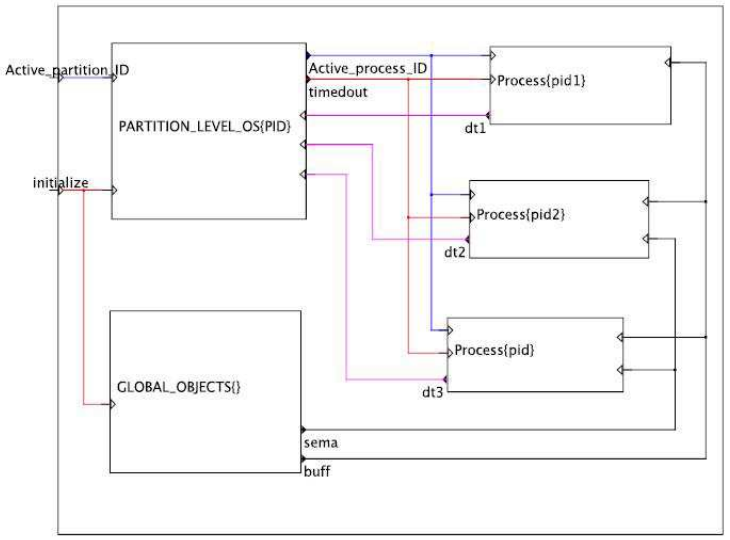


Figure 3.2: An example of partition model composed of three processes

3.2.1 Partitions

In Chapter 2, we have already presented the notion of *partition*. A partition is composed of processes which represent the basic execution units. Processes in a partition share the resources allocated to the partition, but cannot access the resources in other partitions. In Figure 2.25, *Partition1* is composed of three processes.

Figure 3.2 roughly shows a global view of a partition composed of three processes in Signal. In this model, the component `GLOBAL_OBJECTS` appears for structuring. In particular, it provides the processes with communication and synchronization mechanisms (e.g. `buff`, `sema`).

Figure 3.3 gives a textual example of a partition, named *ON_FLIGHT*. It contains three processes: *POSITION_INDICATOR()*, *FUEL_INDICATOR()* and *PARAMETER_REFRESHER()*. Each process has a unique identifier specified in the parameter, for example, process *POSITION_INDICATOR* is identified 1. All the processes are created at the initialization phase of the partition. After the initialization phase, the partition gets activated (i.e. when receiving *active_partition_ID* – this signal is produced by the

```
process ON_FLIGHT =
  ( ? PartitionID_type active_partition_ID;
      event initialize;
    ! ProcessID_type active_process_ID;
  )
(| (| (end_processing1,active_block1) := POSITION_INDICATOR{1,6}(active_process_ID,timedout,
                          board cell (^active_process_ID),buff1 cell (^active_process_ID),
                          buff2 cell (^active_process_ID),evt cell (^active_process_ID))
    | (end_processing2,active_block2) := FUEL_INDICATOR{2,6}(active_process_ID,timedout,
                        buff1 cell (^active_process_ID),sema cell (^active_process_ID),
                        s_port cell (^active_process_ID),evt cell (^active_process_ID),
                        global_params cell (^active_process_ID))
    | (end_processing3,active_block3) := PARAMETER_REFRESHER{3,6}(active_process_ID,timedout,
                          board cell (^active_process_ID),buff2 cell (^active_process_ID),
                          sema cell (^active_process_ID),evt cell (^active_process_ID),
                          global_params cell (^active_process_ID))
    | (| global_params := 3.14 when initialize
       | (board,ret_board) := CREATE_BLACKBOARD{}("board" when initialize,50)
       | (buff2,ret_buff2) := CREATE_BUFFER{}("buffer2" when initialize,3,50,#FIFO)
       | (buff1,ret_buff1) := CREATE_BUFFER{}("buffer1" when initialize,3,50,#FIFO)
       | (evt,ret_evt) := CREATE_EVENT{}("event" when initialize)
       | (sema,ret_sema) := CREATE_SEMAPHORE{}("semaphore" when initialize,1,2,#FIFO)
       | (s_port,ret_sport) := CREATE_SAMPLING_PORT{}("s_port" when initialize,4,#SOURCE,5.0)
       |)
    | (| (active_process_ID,timedout) := PARTITION_LEVEL_OS{1}
                                        (active_partition_ID,initialize,end_processing)
       | end_processing := end_processing1 ^+ end_processing2 ^+ end_processing3
       |)
    |)
 |)
where
...
end;
```

Figure 3.3: A Signal textual example of partition model

module-level OS, which is in charge of the partitions management). Whenever the partition executes, the *PARTITION_LEVEL_OS* selects an active process within the partition. This is represented by its output signal *active_process_ID*, which is sent to each process. The communication mechanisms *buffers* and *blackboards* are created when the partition is initialized, e.g. *CREATE_BLACKBOARD{}*.

3.2.2 Partition-level OS

The role of the partition-level OS is to ensure the correct concurrent execution of processes within the partition (each process must have exclusive control on the processor). A sample model of the partition-level OS is depicted in Figure 3.4.

Figure 3.4: Interface of the partition-level OS

In Figure 3.4, the input `Active_partition_ID` represents the identifier of the running partition selected by the *module-level OS*, and it denotes an execution order when it identifies the current partition. The presence of the input signal `initialize` corresponds

to the initialization phase of the partition: creation of all the mechanisms and processes contained in the partition. Whenever the partition executes, the `partition_level_OS` selects an active process within the partition. The process is identified by the value carried by the output signal `Active_process_ID`, which is sent to each process. The signal `dt` denotes duration information corresponding to process execution. It is used to update time counter values. The signal `timedout` produced by the partition-level OS carries information about the current status of the time counters used within the partition. For instance, a time counter is used for a wait when a process gets interrupted on a service request with time-out. As the partition-level OS is responsible for the management of time counters, it notifies each interrupted ARINC process of the partition with the expiration of its associated time counter. This is reflected by the signal `timedout`.

3.2.3 Processes

The definition of an ARINC process model basically takes into account its computation and control parts. In Figure 3.5, two sub-components are clearly distinguished within the model: *CONTROL* and *COMPUTE*.

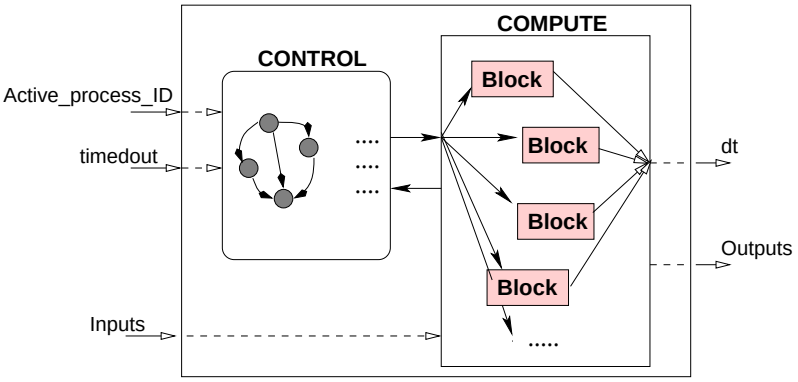


Figure 3.5: ARINC process model.

Any process is seen as a reactive component, which reacts whenever an execution order (denoted by the input `Active_process_ID`) is received. The input `timedout` notifies processes of time-out expiration, while the output `dt` is emitted by the process after completion. In addition, there are other inputs (resp. outputs) needed for (resp. produced by) the process computations. The CONTROL and COMPUTE sub-components cooperate to achieve the correct execution of the process model.

The CONTROL sub-component specifies the control part of the process. Basically, it is a transition system that indicates which statements should be executed when the process model reacts. Whenever the input `Active_process_ID` identifies the ARINC process, this process "executes". Depending on the current state of the transition system representing the execution flow of the process, a *block* of actions in the COMPUTE sub-component is selected to be executed *instantaneously* (with respect to logical time of the synchronous modeling).

The COMPUTE sub-component describes the actions computed by the process. It is composed of *blocks* of actions. They represent elementary pieces of code to be executed

without interruption. The statements associated with a *block* are assumed to *complete within a bounded amount of time*. Since a block is supposed to be non-interrupted, it is imposed that it contains either one single system call, or one or more data computation functions. In this way, a *block* can be executed *instantaneously*.

3.2.4 APEX services

The APEX services modeled in Signal include communication and synchronization services used by processes (e.g. *SEND_BUFFER*, *WAIT_EVENT*, *READ_BLACKBOARD*), process management services (e.g. *START*, *RESUME*), partition management services (e.g. *SET_PARTITION_MODE*), and time management services (e.g. *PERIODIC_WAIT*).

In the following, the modeling of the *READ_BLACKBOARD* service is presented. This service is used to read a message in a blackboard. The input parameters are the blackboard *identifier*, and a *time-out* duration that limits the waiting time if the blackboard is empty. The outputs are a *message* and a *return code* for the diagnostics of the service request.

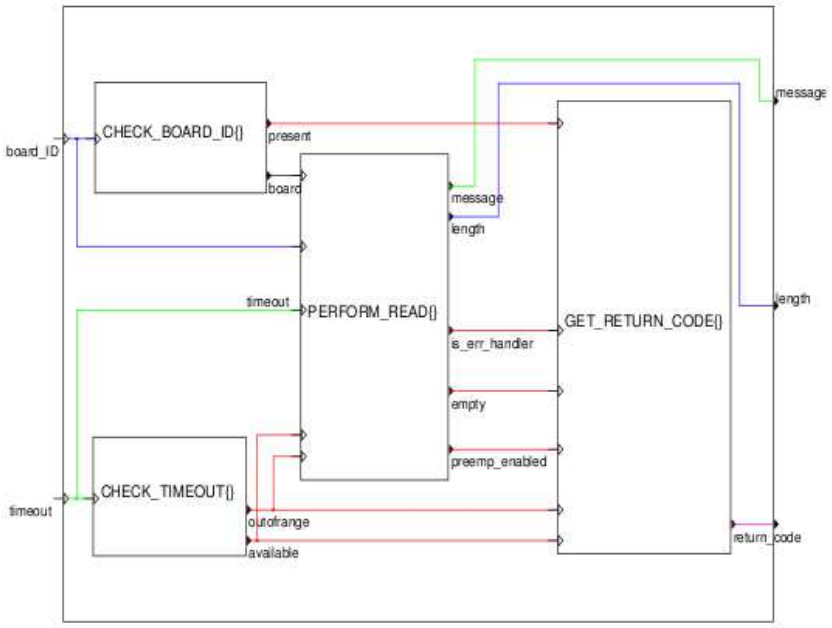


Figure 3.6: Signal model of READ_BLACKBOARD

Figure 3.6 presents a graphical description of the service. There are four main sub-processes, which are represented by inner boxes in the figure. The sub-processes *CHECK_BOARD_ID* and *CHECK_TIMEOUT* verify the validity of input parameters *board_ID* and *timeout*. If these inputs are valid, *PERFORM_READ* tries to read the specified blackboard. Afterward, it has to send the latest message displayed on the blackboard. *PERFORM_READ* also transmits all the necessary information to *GET_RETURN_CODE*, which defines the final diagnostic message of the service request.

```
process READ_BLACKBOARD =
  { ProcessID_type process_ID; }
  ( ? Comm_ComponentID_type board_ID;
```

```
      SystemTime_type timeout;
    ! MessageArea_type message;
      MessageSize_type length;
      ReturnCode_type return_code; )
(| board_ID ^= timeout ^= present ^= outofrange ^= available
            ^= C_return_code                                    (s1)
 | board ^= empty ^= when present                                (s2)
 | message ^= length ^= when (not empty)                        (s3)
 | is_err_handler ^= when empty when available                  (s4)
 | preemp_enabled ^= when (not is_err_handler)                  (s5)
 | C_return_code := (when ((not present) or outofrange)) default
     (when empty when (not available)) default
     (when ((not preemp_enabled) default is_err_handler)) default
     (when (not empty)) default false                           (s6)
 | return_code ^= when C_return_code                            (s7)
 |)
```

Here we list the interface and partial textual implementation of the *READ_BLACKBOARD*. The equations `(s1)`, `(s2)`, `(s3)`, `(s4)` and `(s5)` express some synchronization. The signal *C_return_code* denotes the presence of a return code, as described in equation `(s6)`. The detailed communication and computation could be implemented in external C code.

3.3 Conclusion

We introduced the language Signal and its polychronous model of computation in this chapter. Signal is a polychronous data-flow language. The Signal formal model provides the capability to describe systems with several clocks as relational specifications. Using Signal allows to specify an application, to design an architecture, to refine detailed components to RTOS or hardware description. The syntax and semantics of Signal have been presented.

We also focus on the definition of Signal models of components required for the description of avionics applications. The Signal framework has solid mathematical foundations enabling formal methods for specification, verification, analysis and transformations. The modularity and abstraction of the Signal language allow for scalability. The description of a large application is achieved by specifying first, either completely or partially (by using abstractions), sub-parts of the application. After that, the resulting programs can be composed in order to obtain new components. These components can be also composed and so on, until the application description is complete. A crucial issue about the design of safety critical systems, such as avionics, is the correctness of these systems. In Polychrony, the functional properties of a system can be checked using tools like the compiler or the model checker Sigali. Therefore, the design of avionic applications can be addressed and well modeled based on model refinement within the development environment Polychrony, associated with the synchronous language Signal.

A library of APEX-ARINC services, providing Signal models of RTOS functionalities, is provided. It specifes a Signal description of avionic applications based on IMA. The synchronous modeling of AADL components, which is based on the IMA architecture, will be presented in Chapter 5 as a major contribution of this thesis.

Part III

Prototyping AADL models in a polychronous model of computation

Chapter 4

Formalizing AADL

Contents

4.1 AADL background 91

4.2 Related works 92

4.2.1 Modeling AADL in MARTE 93

4.2.2 Modeling AADL in SystemC 94

4.2.3 Code generation from AADL to C 94

4.2.4 Modeling AADL in Fiacre 94

4.2.5 Modeling AADL in TASM 96

4.2.6 Modeling AADL in ACSR 98

4.2.7 Modeling ARINC653 systems using AADL 98

4.2.8 Modeling AADL in BIP 98

4.2.9 Modeling AADL in Lustre 101

4.3 Summary and comparison 103

4.4 Conclusion 104

This chapter is devoted to the introduction of some related AADL model formalization and transformations. Section 4.1 gives an overview of the AADL background and current users. In order to support some analysis or verification or other purposes, the AADL model is formalized by many other languages. Section 4.2 presents the different formalization of AADL. In section 4.3, a brief comparison of these formalization is presented. Conclusions are given in section 4.4.

4.1 AADL background

The AADL language offers advantages of actually being an international standard, while precisely defining the semantics of a predefined set of components, including real-time concerns. It targets critical developments and offers appropriate modeling techniques to cover system and software engineering activities.

Traditionally, embedded systems specification, design, development and validation of systems has been lacking in a precise capture of the system architecture and its analysis early in and throughout the development process. System integration becomes high risk, and system evolution becomes expensive and results in rapidly outdated components.

By contrast, improved embedded systems engineering practice would be architecture-based and model-driven. Well-defined software system architecture provides a framework to which system components are designed and integrated. System models that precisely capture this architecture provide the basis for predictable embedded system engineering through repeated analysis early in and throughout the development life cycle. The SAE AADL is such an industry standard notation that was explicitly designed to support such model-based embedded system engineering.

In [2], it is reported that some companies and research centers are using AADL, or are contributing to AADL:

- Honeywell [12]. As the originator of MetaH, experiences with the MetaH language and toolset have led to the development of the AADL standard.
- SEI [23]. The Software Engineering Institute (SEI) has provided technical leadership in the development of AADL. A hands-on course on Model-based Engineering with AADL and the open source AADL tool environment has been developed.
- Rockwell Collins [21]. Rockwell Collins has done a pilot study on modeling and analysis of an avionics systems architecture using AADL.
- ESA [11]. The European Space Agency (ESA) in cooperation with Axlog has identified AADL as a key technology for their future.
- ElliDiss [10]. Ellidiss has enhanced its STOOD tool environment to support AADL.
- EADS [9]. The European Aeronautic Defense and Space company (EADS Germany) has investigated the use of AADL for modeling the reference architecture for the Allied Standard Avionics Architecture Council (ASAAC) program.
- Airbus [6]. Airbus France has investigated the use of AADL and UML2.0 under the COTRE initiative. The TopCased [29] project, started by Airbus, sets out to provide new development tools of AADL, i.e., OSATE and behavioral annex editor. Our work is founded by Airbus.

4.2 Related works

The AADL language provides an efficient support to take into account complex embedded systems. In order to validate formal properties on AADL models, or schedulability, or performance analysis, or verification of an AADL model, a formal framework which provides diagnostics on the system must be used. This goal can be achieved if we can transform the AADL model into some other model, whose associated toolsets offer these capabilities. Here we give a brief view of some related AADL transformations. We try to classify these works according to their main purpose, from the point of view of modeling, simulation, verification and other purpose, though this classification is not a strict one.

4.2.1 Modeling AADL in MARTE

For the purpose of modeling, MARTE [14] (Modeling and Analysis of Real Time and Embedded systems) is used to model AADL, especially for the communications. MARTE is a UML profile recently adopted by OMG (Object Management Group). Being a UML profile, it benefits from the large set of rapidly improving UML graphical editors and has strong connections with SysML, offering an opportunity to have a complete and integrated design flow, from system-level specification and requirement analysis to implementation, code generation, schedulability and performance analysis.

Using MARTE as a foundation profile to model AADL specific concepts would benefit to both communities. It would ease interoperability of AADL with UML models, and would allow some selected MARTE models to be analyzed by AADL tools.

[119] builds the semantic construction using MARTE Time Model [45], to represent faithfully AADL periodic/aperiodic tasks communicating through event or data ports, in an approach to end-to-end flow latency analysis. The idea is to define once a model library for AADL with MARTE.

The authors choose to represent the AADL flow using a UML activity diagram. Figure 4.1 gives an example of end-to-end flow with UML and MARTE.

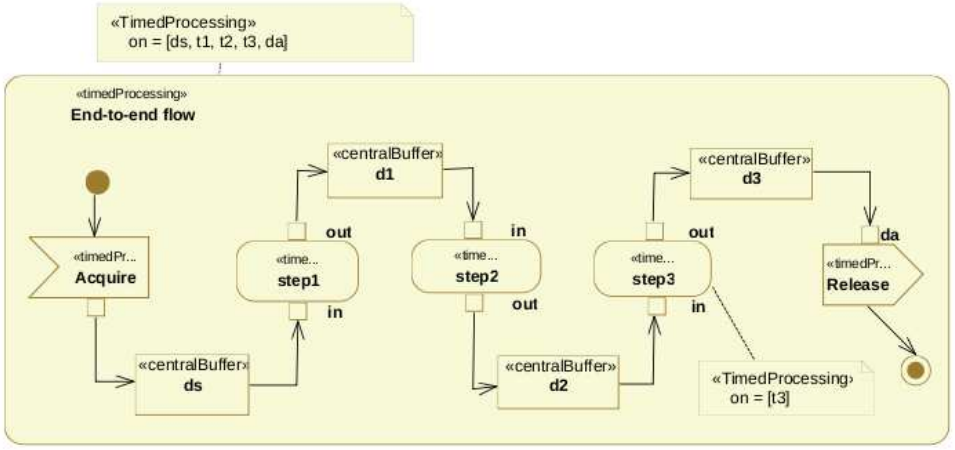


Figure 4.1: End-to-end flow with UML and MARTE

This UML diagram is *untimed*, to make it behave the exact same way as an AADL model, they propose to use the MARTE Time Profile. The MARTE Time subprofile, inspired from the theory of tags systems [117], provides a set of general mechanisms to define the model of computation and communication.

Two kinds of communications are modeled in MARTE:

- **Data-driven communications.** The execution of a given task is triggered by the availability of the data produced by the preceding task. The CCSL clock relation *alternatesWith* can be used to model such data-driven communications:

  step1.finish **alternatesWith** step2.start

- **Sampled communications.** Pure data are only sampled and used as such whenever the task is activated.

[124] relies on the MARTE Time Model and the operational semantics of its companion language CCSL, to equip UML activities with the execution semantics of an AADL specification. This is part of a much broader effort to build a generic simulator for UML models with semantics explicitly defined within the model.

4.2.2 Modeling AADL in SystemC

SystemC is a set of C++ classes and macros which provide an event-driven simulation kernel in C++. SystemC is applied to system-level modeling, architectual exploration, performance modeling, software development, functional verification, and high-level synthesis.

For the purpose of simulation, AADS [157] is developed as an AADL SystemC simulation tool. It supports the performance analysis of AADL specifications throughout the refinement process from the initial system architecture till the complete application and execution platform are developed. It parses the AADL model, so the functionality is translated to an equivalent POSIX model and the architecture is represented in XML.

The generation of the SystemC model from the AADL specification is not straightforward. Nevertheless, the SystemC model generated by AADS is able to capture the fundamental dynamic properties of the initial system specification.

4.2.3 Code generation from AADL to C

PolyORB-HI-C [20] is an AADL runtime used by C code generator. It is developed by Ocarina [15] for distributed high-integrity applications based on the Ada Ravenscar [120] profile. Two target languages are supported: Ada2005 and C. It is used for system primitives and resource management. This piece of software is a link between Ocarina generated code and the underlying runtime system.

[65] presents an experimentation: code generation from a sub-part of AADL model to C code (compliant with a specific standard (OSEK/VDX)) using MDA tools. In the experimentation, the authors focus on the AADL thread component. Two transformations generate either C language code (compliant with OSEK/VDX) and corresponding OIL configuration code.

Within only one mode, without fault handling and without considering subprogram calls, the suspended, ready and running states of AADL threads and OSEK/VDX tasks follow the same semantics. Thus, generated OSEK/VDX tasks may respect the AADL thread execution semantics for periodic, sporadic or aperiodic threads.

In the OSEK/VDX standard, all threads execute on the same processor, share a virtual address space. Thus, if many threads may be considered, only one process is used. The generated C code corresponds to the implementation of dynamic thread semantics.

4.2.4 Modeling AADL in Fiacre

Fiacre [57] is a formal intermediate model to represent both the behavioral and timing aspects of systems in embedded and distributed systems for formal verification and simulation purposes. It is designed both as the target language of model transformation engines

from various models, such as SDL, UML, AADL, and also as the source language of compilers into targeted verification toolboxes, such as CADP and Tina. In order to support verification of AADL models, modeling AADL in Fiacre is proposed.

The Fiacre language has two main notions:

- **process.** A **process** describes the behavior of sequential components. A **process** is defined by a set of control states and parameters, each associated with a set of complex transitions.
- **component.** A **component** describes the composition of **processes**, possibly in a hierarchical manner.

The component *root* presented in the following is a simple token ring example. The model consists of a **component** *root* that chooses non-deterministically the **process** *Node* that initially owns the token (**process** *start*).

```
process Start [ start0 : none , start1 : none , start2 : none ] is
        states s0 , s1
        from s0 select start0 [ ] start1 [ ] start2 end ; to s1

process Node [ prev : none , succ : none , start : in none ] is
        states idle , wait , cs , st_1
        from idle select start ; to st_1 [ ] prev ; to st_1 end
        from st_1 succ ; select to idle [ ] to wait end
        from wait prev ; to cs
        from cs   succ ; to idle

component root is
        port s0 : none , s1 : none , s2 : none ,
             p0 : none , p1 : none , p2 : none ,
        par * in
            Start [ s0 , s1 , s2 ]
        || Node [ p0 , p1 , s0 ]
        || Node [ p1 , p2 , s1 ]
        || Node [ p2 , p0 , s2 ]
        end
root
```

AADL2Fiacre [56] deals with the transformation of AADL models into Fiacre models to perform formal verification and simulation. The transformation relies on AADL properties and on the behavioral annex of AADL. The authors follow a model-driven approach. They developed a meta-model of the Fiacre language. Hence the transformation from AADL to Fiacre is obtained through model transformation. This translation is based on the formal semantics of the AADL execution model.

The transformation of AADL code into Fiacre includes the following main features:

- **Transformation of an AADL architecture.** To consider the system as a hierarchy of components, the hardware components are ignored. In order to offer to threads a simple access to their ports, a new component, called a **glue**, is introduced at each level of the hierarchy, which desynchronizes sender and receiver ports. The AADL threads do not communicate directly: the **glue** process manages communications and scheduling protocols (Figure 4.2).

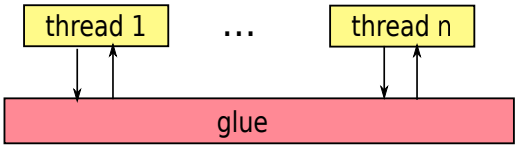


Figure 4.2: Threads and the glue

- **Transformation of thread behaviors.** A thread is translated into a Fiacre process taking as parameters ports linked to the **glue**. Annex specific constructs associated to the AADL programming interface are translated to communications through control channels managed by the **glue**. Thread dispatch is synchronized with data transfers within the **glue**.

- **Modeling communication semantics.** Two communication protocols are considered: immediate and delayed. The use of an immediate communication avoids declaring an explicit offset for the reader process. The delayed mechanism ensures a deterministic communication between threads of different periods.

Tina (TIme Petri Net Analyzer) provides a software environment to edit and analyze Petri Nets and Time Petri Nets. For verification purposes, Fiacre programs are compiled into suitable input formalisms for the Tina and CADP toolboxes. The Tina back-end translates Fiacre programs into Time Transition System and the specification requirements into State-Event LTL formulas. It is used to verify the model in some expected properties.

4.2.5 Modeling AADL in TASM

The Timed Abstract State Machine (TASM) [136] is based on the theory of Abstract State Machine (ASM) [62]. It extends ASM to enable expression of timing, resource, communication, composition, parallelism. A basic TASM specification contains an abstract state machine and an environment. The environment contains environment variables and the universe of types that variables can have. The machine consists of three parts: *monitored variables*, *controlled variables* and *mutually exclusive rules* with the form of "*if condition then action*".

In brief, the execution model of TASM is a loop: read input variables, wait for the duration of the execution, write output variables, and wait for synchronization.

Technically, a TASM specification is made up of *machines*. A specification must hold at least one *main machine* with its set of rules. Beside main machines, it is also possible to define *sub machines* and *function machines*.

[165] proposes a formal semantics for the AADL behavior annex using TASM. A semantics of AADL execution model is given, and a prototype of behavior modeling and verification is proposed.

Only the synchronous execution model is considered in [165]. It gives the relation between execution model and behavior annex (Figure 4.3).

A periodic thread is specified in TASM by two main machines: one to manage period, and the other one to represent the execution of thread with deadline, WCET, resource and execution rules.

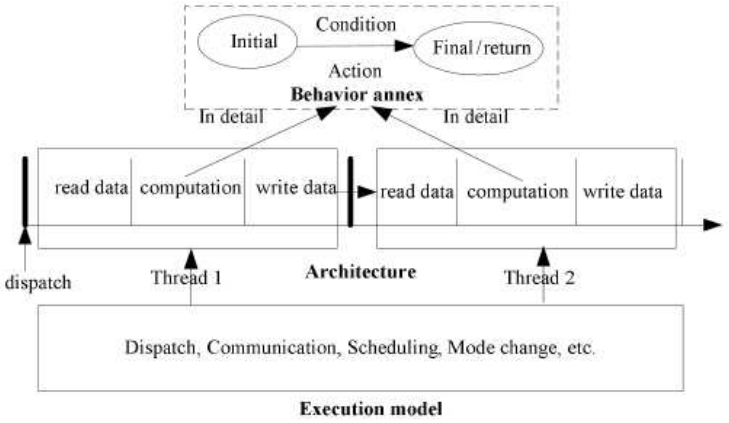


Figure 4.3: Behavior annex and execution model

```
MAIN MACHINE: Period
Rule: period
{ t := period;
  if Nextdispatch = false then Nextdispatch := true;}

MAIN MACHINE: Execution
Rule: execution
{ t :=[0,deadline];
  processor := WCET*100/deadline;
  if Complete = false then Complete := true;}
```

TASM explicit time model with powerful logical time constraints allows specifying precisely the scheduling aspects of threads, parallel and communication mechanisms can be used to describe the synchronization. The immediate and delayed communication semantics are defined in TASM. The delayed communication modeling is detailed for the case of *synchronous*, *oversampling* and *undersampling*.

To define the formal semantics of behavior annex, as the hierarchical composite mechanism in TASM, *state* is mapped to controlled/ monitored variables, *guard* is specified using function machines with boolean output parameter, *action* is specified using function or sub machines, and the whole transition is expressed by rules of a main machine.

A subset of actions are modeled, including: send/receive events, remote subprogram call, client-server protocol and time behavior. For example, the time behaviors (*computation(min, max)* and *delay(min, max)*) are modeled by the *main machines*. A control/ monitored variable is defined: *CurrentState:={Current, Next}*.

```
MAIN MACHINE: Computation
Rule: computation{min, max}
{ t := [min, max]
  processor := WCET*100/deadline;
  if CurrentState = Current then CurrentState := Next;}

MAIN MACHINE: Delay
Rule: delay{min, max}
{ t := [min, max]
  if CurrentState = Current then CurrentState := Next;}
```

The purpose of modeling AADL in TASM is to perform formal verification. This verification is achieved with UPPAAL by mapping each main machine to a timed automaton. Timing correctness is defined as a reachable state of the system, being reachable within an acceptably bounded amount of time.

4.2.6 Modeling AADL in ACSR

Algebra of Communicating Shared Resources (ACSR) [118] is a real-time process algebra that makes the notion of resource explicit in system models. Restrictions on simultaneous access to shared resources are introduced into the operational semantics of ACSR, which allow to perform analysis of scheduling properties of the system model.

[149] presents a semantics-preserving translation of AADL models into the real-time process algebra ACSR, to perform schedulability analysis of AADL models. The authors describe an approach to provide formal analysis of timing properties, including schedulability analysis of AADL models. In order to analyze a model, they automatically translate it into the real-time process algebra ACSR, and use the ACSR-based tool VERSA to explore the state space of the model, looking for violations of timing requirements.

The basis for the translation is given by the semantic definition for thread components in the AADL standard. Schedulability analysis of the translated ACSR model is enabled by the ability of ACSR to describe resource requirements and use priorities to control access to shared resources. Furthermore, this analysis tool offers a set of failing scenarios in the case when the system is non-schedulable or violations of timing requirements are discovered.

4.2.7 Modeling ARINC653 systems using AADL

[73] presents an approach for the modeling, verification, implementation and simulation of ARINC653 systems using AADL. It details a modeling approach exploiting the new features of AADLv2 for the design of ARINC653 architectures.

The approach uses Cheddar [8] scheduling analysis tool to check scheduling aspects with regards to runtime requirements. Then an AADL-to-ARINC653 code generator (Ocarina [15]) and a dedicated AADL/ARINC653 runtime (POK [19]) are used to generate the system automatically.

In this work, an ARINC653 *module* is represented in AADL by a processor. An ARINC653 *partition* modeling is achieved with AADL virtual processor and process. An ARINC653 *process* is mapped to AADL thread. The intra-partition and inter-partition communication services are also represented in AADL, listed in Table 4.1.

Advantages of this approach are the use of verification mechanisms to automatically detect potential problems before implementation efforts, and provide an ARINC653 compliant runtime to operate the produced system.

4.2.8 Modeling AADL in BIP

BIP [7] (Behavior Interaction Priority) is a language for the description and composition of components as well as associated tools for analyzing models and generating code on

ARINC653	AADL
module	processor
partition	virtual processor and process
process	thread
buffer	event data port
blackboard	data port
event	event port
queuing port	event data port
sampling port	data port

Table 4.1: Modeling ARINC653 architecture in AADL

a dedicated platform. It is used to model heterogeneous real-time components. The BIP component model is the superposition of three layers:

- **Lower layer**. The lower layer describes the behaviors of a component as a set of transitions. An *atomic* component consists of a set of *ports* used for the synchronization with other components, a set of *transitions* and a set of local variables. Transitions describe the behavior of the component, and they are represented as a labeled relation between *control states*. Figure 4.4 shows an atomic component with two control states S_i and S_j, ports *in* and *out*, and corresponding transitions guarded by guard g_i and g_j. A compound component allows defining new components from existing sub-components by creating their instances, specifying the connectors between them and the priorities.

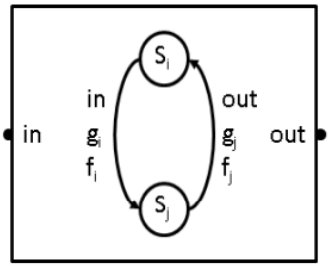


Figure 4.4: BIP atomic component

- **Intermediate layer**. The intermediate layer includes connectors describing the interactions between transitions of the layer underneath.
- **Upper layer**. The upper layer consists of a set of priority rules used to describe scheduling policies for interactions.

[68] studies a general methodology and an associated tool for translating AADL and behavior annex specification into the BIP language. The software, hardware and system components are translated from AADL to BIP. The behavior annex and connections are also translated. For example, a thread is modeled in BIP by an atomic component, and a subprogram is translated into an atomic or compound BIP component. This translation

allows simulation of AADL models, as well as application of verification techniques. Two model checking techniques for verification have been applied, once the BIP model is generated: deadlock detection by using the tool Aldebaran, and verification of thread deadlines and component synchronization by using BIP observers.

[138] models AADL data communication with BIP. It focuses on deterministic data communication, and shows how BIP deals with the modeling of immediate and delayed data communications supporting undersampling and oversampling of AADL.

Since the two types of communication rely on the notion of periodic thread, the authors first propose a model of AADL periodic thread in BIP. Compared to [68], they ignore initialization states as well as the support for the behavior annex. An AADL periodic thread is modeled in BIP by a compound component.

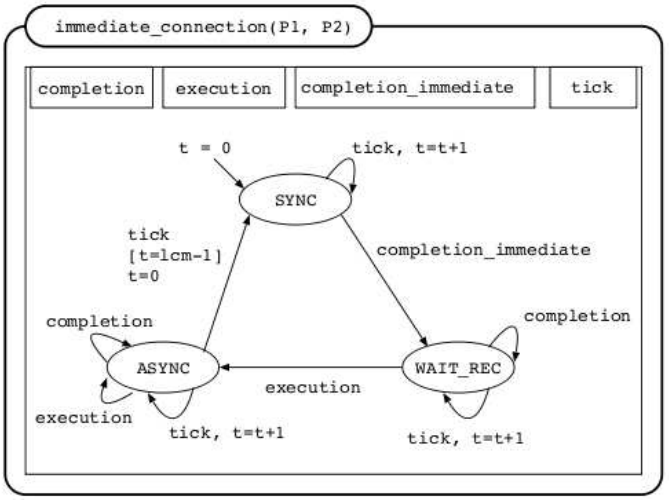


Figure 4.5: BIP model of an immediate connection

Figure 4.5 describes the automation associated with an immediate connection. When time is aligned to the LCM (Least common multiple) of the periods of the two interacting threads, the data transfer is allowed through the synchronization on the port *completion_immediate*. In this case, the execution of the receiving thread is delayed until the completion of the sender thread: *execution* is not allowed in the *SYNC* state. Otherwise (*ASYNC* state), completion of the first thread and execution of the second thread are not synchronized and no data transfer is performed.

The specification of a delayed communication in BIP is specified by the automaton shown in Figure 4.6.The component declares two variables: the variable *next* is transmitted by the sender thread at completion. It is copied to the variable *current* at sender deadline. This variable is transmitted to the receiver thread at its dispatch. Thus, two variables are needed to manage delayed communication. So in case of a communication between synchronous processes, the reader will get the new input at the start of the next period. It is therefore necessary to keep the last data to send it at the end of period. Data are always sent with nearly a period of delay.

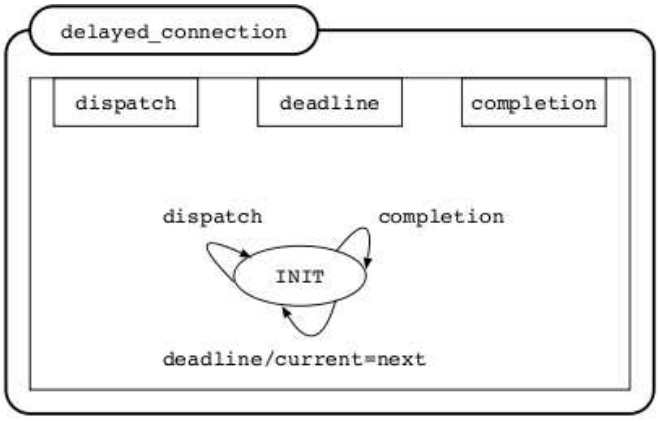


Figure 4.6: BIP model of a delayed connection

4.2.9 Modeling AADL in Lustre

AADL2SYNC [3] is an AADL to synchronous programs translator, which is extended in the framework of the European project ASSERT, resulting in the system-level tool box translating AADL to Lustre.

The main object is to perform simulation and validation that take into account both the system architecture and the functional aspects. The authors proposed in [99] to build automatically a simulator of the architecture, expressed in a synchronous language. The non-synchronous composition of synchronous processes is expressed by sporadic activation and oracle-driven non-determinism.

An activation condition is added to periodic and sporadic components (processes and threads). The following gives an example of a periodic clock generator. For a component C of period p, the scheduler defines a boolean variable *C_activate_clk* that is true every p cycles of the outer processor. The returned activation condition is a boolean that is set to true when the node needs to be activated.

```
node clock_of_period(period: int; qs_tick: bool)
returns (activate_clock: bool);
let
  pcpt = period -> pre cpt;
  cpt = if activate_clock then period else
        if qs_tick then (pcpt - 1) else pcpt;
  activate_clock = true -> (pcpt = 1) and qs_tick;
tel
```

The general construction is that: the processes are all sporadically activated according to activation conditions emitted by a global scheduler. The scheduler is non-deterministic: it receives one boolean oracle for each condition it has to elaborate.

Although this approach also translates AADL to a synchronous language, it considers a purely synchronous model of computation (that of Lustre), in which clocks need to be totally ordered (by contrast to the relational, multi-clocked MoC considered in Signal). This limitation requires the emulation of asynchrony by using a specific protocol of **quasi-synchronous** communication. Its expressive capability is limited compared to simply abstracting asynchrony using partially-ordered clock relations.

A **quasi-synchronous** clock generator shown below is given in [100]. Node *qs* is the main node of the quasi-synchronous scheduler for processors with the same clock rate. The idea is that, starting from a n-array of boolean values containing candidate values for the quasi-synchronous clocks (i.e., chosen randomly), to check that no clock drift excess occurs, and to force the culprit clocks to be false when necessary to avoid this drift excess. The node *compute_advance* computes the relative advance of *ck*1 with regard to *ck*2. The node *ba_fill* creates an array of size *n* filled with *x*. Node *ba_none* checks if all elements of the array are false.

```
-- Compute the relative advance of clk1 w.r.t. clk2
node compute_advance(clk1, clk2 : bool) returns (c : int);
var pc : int;
let
    pc = 0 -> pre c;
    c = if clk2 then 0 -- reset the advance of clk1
        else if clk1 then pc + 1 -- clk1 got ahead of one tic
        else pc;
tel

-- Create an array of size n filled with x
node ba_fill(const n: int; x : bool) returns (t: bool^n);
let
    t = x^n;
tel

-- Check if all elements of the array are false
node ba_none(const n : int; I : bool^n) returns ( ok : bool);
var
    Nor : bool^n;
let
    Nor = [not I[0]] | (Nor[0..n-2] and not I[1..n-1]);
    ok = Nor[n-1];
tel

node qs(const n:int; const d:int; alea:bool^n) returns (select:bool^n);
var
    advance_max_is_reached, problems : bool^n^n;
    filter : bool^n;
    advance, padvance : int^n^n;
let
    advance_max_is_reached = padvance >= d^n^n;
    -- there is a problem if the max advance is reached and no tic occurs
    problems = advance_max_is_reached and not alea^n;
    -- force the tic for clocks that would be more than d tics late
    filter = ba_none(n^n, problems);
    select = alea and filter;
    advance = compute_advance(ba_fill(n^n, select), select^n);
    padvance = 0^n^n -> pre advance;
tel
```

Synchronous modeling of asynchronous systems is also studied in [94]. The authors define a generic semantic model for synchronous and asynchronous computation, after that, the attention is focused on implementing communication mechanisms.

[97] deals with expressing the behavior of complex scheduling policies managing shared resources. A synchronous specification for different shared resource scheduling protocols is provided: no lock, blocking, the well-known basic priority inheritance protocol (BIP) and the priority ceiling protocol (PCP). The authors also show how various properties related to determinism, schedulability, or the absence of inter-locking can be automatically model-checked on given architecture models. This results in an automated translation of AADL models into a purely boolean synchronous (Lustre) scheduler, that can be directly model checked, possibly with the actual software.

4.3 Summary and comparison

In this section, we will make a short comparison of these related works, mainly from the objective and some characteristics.

Using MARTE to model AADL mainly concerns the two communication protocols: immediate and delayed, for a purpose of end-to-end flow latency analysis. One characteristic of this work is: to define once, and for all, a model library for AADL with MARTE [119]. The end-user is not expected to enter into the detail about this library. Compared to our work, this work makes efforts to build a generic simulator specifically for AADL, but targeting a formal analyzable language remains a perspective.

Modeling AADL in Fiacre focuses on the objective of model verification, since Fiacre is a front end of Tina verification toolset. Compared to BIP, Fiacre has less powerful constructs but has good compositional and real-time properties. It offers basic concepts coming from timed transition system which permit the expression of real-time features quite easily [138].

The translation from AADL to BIP allows simulation of AADL models, as well as application of verification techniques, i.e., state exploration or component-based deadlock detection. This work takes into account threads, processes and processors as well as Behavior Annex, but do not include the AADL communication protocols.

Modeling AADL in TASM allows presenting timing semantics of the AADL behavioral annex using TASM, which is based on abstract state machines extended with resource consumption annotations. This work focuses on formal verification and analysis of AADL models with regards to mode change. TASM offers more abstract resource consumption mechanisms, but hardly supports some scheduling patterns [140]. Resource management allows the specification of a fair preemptive scheduler. However, specifying other scheduling policies has to be evaluated [137].

The translation from AADL to SystemC aims at simulation and performance analysis. An AADL SystemC simulation tool is developed. However, the behavior specifications have not been taken into consideration.

The objective of code generation from AADL to C code developed by Ocarina is for distributed high-integrity application. The generated code can be run on both POSIX or RTEMS platforms. With the PolyORB-HI-C tool, the Simulink blocks can be used as AADL subprograms. In that case, the generated code from AADL automatically calls the generated code from Simulink. A transformation of Simulink blocks (used as AADL threads) in Signal is also implemented in our cooperation work in project CESAR [35]. We use AADL to model the architecture and Simulink to model the functional applica-

tions. A case study is implemented for the simulation and analysis.

Compared to our work, the translation of AADL models into ACSR focuses on formal analysis of timing properties, including schedulability analysis, of AADL models. A set of failing scenarios in case the system is non-schedulable are offered.

The goal of modeling ARINC653 using AADL is to propose an appropriate MDE-based development process to capture architecture requirements using AADL and its ARINC653 annex. It models ARINC653 architectures in AADL, while our work is the inverse: we model an AADL system in Signal in a framework of ARINC653. There are many similarities between these two works, i.e., the ARINC653 partition is mapped to an AADL process in this work, while the AADL process is translated to an ARINC653 partition in our transformation, and the ARINC653 process is mapped to an AADL thread in this work, while we translate the AADL thread to an ARINC653 process presented in Signal, etc. Of course, many differences exist, for example, an ARINC653 module is represented by an AADL processor in [73], but in our transformation, we interpret an AADL processor as a scheduler for scheduling the processes (threads) that are bound to it. The virtual processor has not been considered in our work.

The main objective of AADL2SYNC is to perform simulation and validation that take into account both the system architecture and the functional aspects. This work builds a simulator expressed in a synchronous language: Lustre. Various asynchronous aspects of AADL, e.g., task execution time and clock drifts, are taken into account. Compared to Signal, Lustre is a purely synchronous model of computation. On one side, a quasi-synchronous communication protocol is used to emulate asynchrony to resolve this limitation, still its expressive capability is limited. On the other side, an automated translation of a purely synchronous scheduler is defined in [98], that can be directly model-checked.

Compared to these works, our work has multi-purpose objectives. We model the AADL system in Signal in order to perform formal verification (using Sigali), simulation (using VCD), and C/Java code generation, since the polychronous model of the Signal language offers formal support for analysis, verification, simulation and code generation in the Polychrony platform. Furthermore, the polychronous model provides models and methods for a rapid, refinement-based, integration and a formal conformance-checking of GALS architectures [152].

4.4 Conclusion

We introduced some background of AADL in this chapter. AADL is a well-defined software/hardware co-design language which supports model-based embedded system engineering. The formalization of AADL model in other languages is required, in order to provide formal property validation, schedulability analysis, resource sharing, and verification. We list some of the AADL model transformations to give an idea of the related works. A comparison of these works is given in the final section.

Chapter 5

From AADL components to Signal processes

Contents

5.1 Transformation chain 107

5.2 Transformation principles 107

5.3 From abstract logical time to more concrete simulation time 110

5.3.1 Modeling computation latencies 111

5.3.2 Modeling propagation delays 112

5.3.3 Towards modeling time-based scheduling 113

5.4 Thread modeling 114

5.4.1 Interpretation of a thread 115

5.4.2 Implementation 117

5.5 Processor modeling 121

5.6 Bus modeling 125

5.7 System modeling 127

5.8 Other components modeling 130

5.8.1 Process modeling 130

5.8.2 Subprogram modeling 132

5.8.3 Data modeling 133

5.8.4 Device 134

5.9 Port and port connection modeling 135

5.9.1 Port modeling 135

5.9.2 Port connection modeling 136

5.10 Towards AADLv2.0 139

5.11 Conclusion 140

Embedded systems are an integral part of safety critical systems. On one side, architectural design and early analysis of embedded systems are two of the major challenges for designers using modeling languages (for example, AADL). On the other side, synchronous languages (such as Signal) have been used successfully for the design of real-time critical applications, to significantly ease the modeling and validation of software components. The synchronous parallel composition helps in structuring the model, without introducing non-determinism. Their associated toolkits provide formal transformation, automatic code generation and verification.

AADL provides a standardized textual and graphical notation for describing software and hardware system architectures and their functional interfaces. The language is aimed at high level design and evaluation of the architecture of embedded systems. Such systems are described as an assembly of software components that are mapped to an execution platform in AADL.

Although AADL allows one a fast design entry and software/hardware co-design, system validation and verification is a critical challenge. In order to support the virtual prototyping, simulation and formal validation of early, component-based, embedded architectures, we propose to use formal methods to ensure the quality of system design. Our major objective is to perform simulation and validation that take into account both the system architecture and the functional aspects. We consider the case where software components are implemented in the synchronous programming language Signal. We propose a model of the AADL into the polychronous model of computation of the Signal programming language. This modeling is based on the IMA architecture. Since a thread is the main executable component and it is the only component that can be scheduled in AADL, our work mainly focuses on the thread component (especially the periodic thread) and its associated components modeling, e.g., the execution model, communications, behaviors, etc.

This chapter focuses on the synchronous modeling of AADL components in the IMA architecture, so that an AADL model can be translated into an executable model of synchronous dataflow language. This modeling is a contribution to bridge the gap between asynchronous and synchronous languages.

This translation can be seen as a basis to establish a formal semantics of AADL execution model and to perform verification of AADL models. For this purpose, we present the abstract transformation from asynchronous model to a synchronous model. The general modeling approach and transformation principles are first presented in section 5.1 and 5.2. Then in section 5.3, we depict the problems in this asynchronous-synchronous transformation, and also give the possible solutions in the Signal programming language. We show how an AADL architecture can be automatically translated into a synchronous model in Signal.

Our approach is based on the IMA modeling concepts. We use the Signal library of APEX ARINC services to represent the AADL architectures and components, and also the communications between the components. We give an implementation of a subset of the AADL components. Since thread is the main executable and schedulable component, thread model is first presented in section 5.4. Processor is responsible for scheduling threads, hence processor modeling is presented just after the thread modeling in section 5.5. Bus is the communication among execution platform components, so the mod-

eling of bus is then presented in section 5.6. System modeling is presented in section 5.7. The modeling of other components is briefly presented in section 5.8. Port connections enable directional exchange of data among components, their modeling is presented in section 5.9. Finally, the concluding remarks are given.

5.1 Transformation chain

The transformation of AADL models into synchronous specifications is separated in two steps: firstly, a transformation of AADL models into synchronous models; then, generation of synchronous code from synchronous models obtained from the first step.

We formalize the transformation of AADL by isolating the core syntactic categories that characterize its expressive capability: systems, processes, threads, subprograms, data, devices, processors, buses and the connections.

Figure 5.1 illustrates the automatic model transformation chain from AADL model to Signal model, so that the final executable code, such as C or Java, is generated. In the transformation, SME models, which conform to the Signal meta-model, are considered as intermediate models.

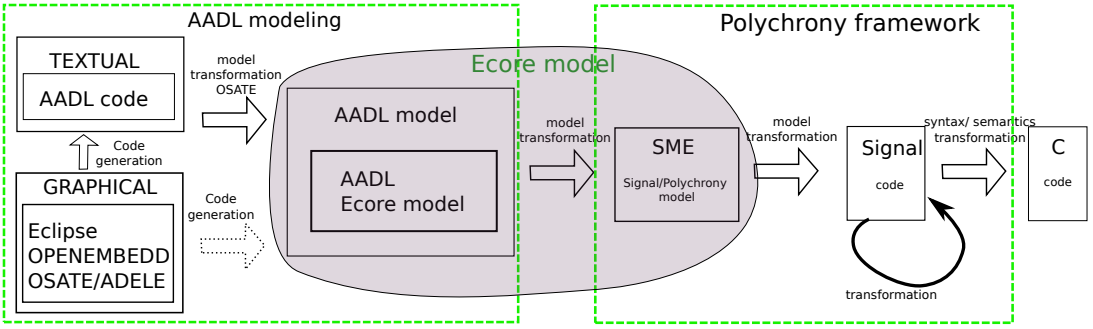


Figure 5.1: A global view of AADL-SME transformation chain

The AADL model can be either presented as textual or graphical specifications. Technically, using the AADL graphical editor ADELE, the graphical *xx.adeleli* model can generate *xx.aadl* text code. The *xx.aaxl* ecore model, which conforms to the AADL meta-model, can be generated automatically from the *xx.aadl* in OSATE toolkit.

The AADL ecore model is translated to SME Signal model using ATL (Atlas Transformation Language) defining the transformation rules. The details of the transformation will be presented in the next subsections. The SME model can be transformed to Signal code. After compilation, the executable C (or Java/C++) code is generated.

5.2 Transformation principles

This model transformation is based on the studying of the similarity relationship between AADL and APEX-ARINC services. The transformation from the AADL model to Signal is based on the IMA Architecture [121].

ARINC653 (Avionics Application Standard Software Interface) is a standard that specifies an API for software of avionics, following the IMA architecture. It defines an

APplication EXecutive (APEX) for space and time partitioning. An ARINC *partition* is a logical allocation unit resulting from a functional decomposition of the system.

As mentioned in Chapter 2, a main feature of ARINC653 is the *partitioning*: several executive units (*processes*) can be logically hosted on a single *partition*. They are only allowed to share the resources in this *partition*. This feature is similar to AADL process: an AADL process represents a protected address space, a space partitioning where protection is provided from other components accessing anything inside the process. Hence, an AADL process can be modeled by the ARINC *partition*.

Similarly, an AADL thread is the main executable unit executed in a process, while an ARINC *process* is the executive unit in the *partition*. Therefore, this relation makes it possible to model the AADL thread by the ARINC *process*.

The processor is responsible for the scheduling of threads in AADL, and it may have memory subcomponents. It could correspond to an "infrastructure" containing a scheduler and local resources. For a simple implementation, we consider it as the scheduler in ARINC. The difficulty is that the scheduling in ARINC653 is divided in two levels: the *module_level_OS* and *partition_level_OS* (refer to Section 3.2.2). The former one is a static scheduling: the *modules* are scheduled based on a predefined configuration table. However, the *partition_level_OS* is still suitable or at least partially suitable (with some improvements) for modeling the scheduling of the AADL threads in the same process. The scheduler in a *partition_level_OS* can either be static or dynamic, depending on the chosen scheduling protocol. In general, the ARINC scheduling policy is priority preemptive. The priorities could be fixed, e.g., Rate_Monotonic, or dynamic, e.g., Earliest_Deadline_First. In Section 5.5, a general purpose scheduler structure referring to *partition_level_OS* is modeled in Signal, the detailed scheduling policy is left for further development (e.g., using external C code implementation). Schedulability analysis, comparison, and other analyses, such as WCET/BCET, can be performed based on different schedulers. A specific static scheduler is given in Appendix A as a simple implementation.

Other mechanisms defined in ARINC, e.g., the communication mechanism (queuing and no-queuing), can be used as some implementation of AADL connection.

AADL	**Signal**
thread	ARINC *process*
process	ARINC *partition*
port	bounded FIFO
data port connection	Signal process (according to the connection type)
processor	ARINC *partition_level_OS*
data type	Signal data type
system	Signal process consisting of sub-processes
bus	Signal communication process
device	Signal process

Table 5.1: Basic transformation principles

Basic transformation principles are first given in Table 5.1. For example, a thread is an executable unit with behavior specifications described in either target language or behavior annex. The detailed execution behavior will be presented in Chapter 6. In this

chapter, we only consider the thread execution temporal semantics. A thread is associated with some temporal properties: *period, dispatch, start, complete* and *deadline*. A thread P can be defined by these properties, noted as SPS, and its internal behavior transitions/actions T/S: $P =< SPS, T/S >$. The interpretation will be the composition of temporal properties SPS and actual execution transitions/actions T/S. The thread is dispatched at *dispatch* time. The executions are activated at *start* time, and the outputs are available to send out at *complete* or *deadline* time. These properties are implemented in the SPS timing environment. The transitions/actions T/S are implemented in a process SPT. The signals that enter or depart SPT are based on these temporal signals (generated by SPS) as specified in the AADL properties. The interpretation is described in Section 5.4.

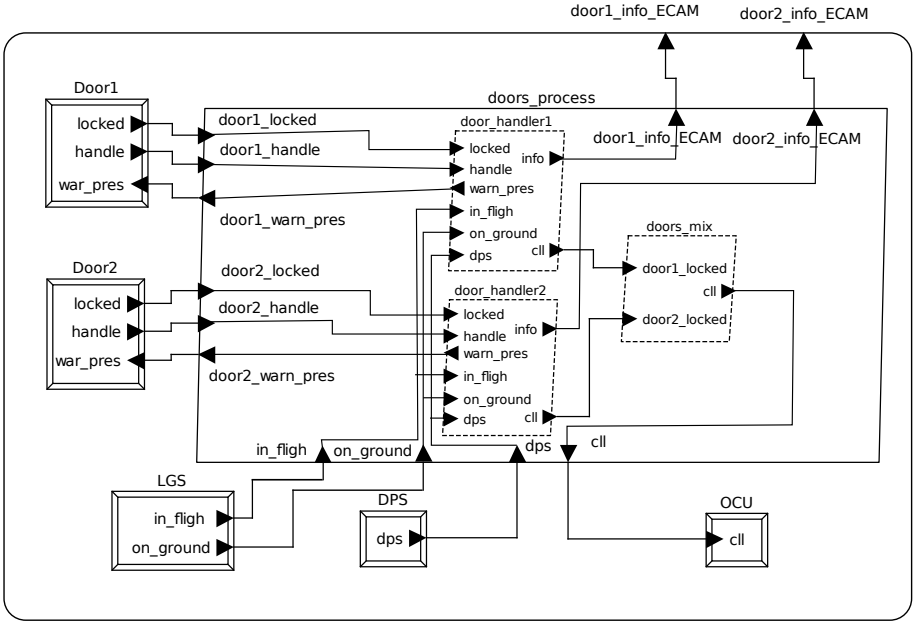


Figure 5.2: A simple illustration of SDSCS system

The details of each component modeling will be given in the following sections. In this section, we will show an example to just give a first idea of the transformation. The SDSCS system (the detailed description can be found in Chapter 8 and Appendix A) in Figure 5.2 has one process *doors_process* processing (or sending) the messages from (or to) the devices: *Door1*, *Door2*, *LGS*, *DPS* and *OCU*. The *doors_process* consists of three threads: *door_handler1*, *door_handler2* and *doors_mix*, computing and generating the outputs.

The corresponding Signal (within ARINC architecture) model is partially shown in Figure 5.3. The process *doors_process* is translated to an ARINC *partition partition_doors_process*. The three compositional *processes*, *process_door_handler1*, *process_door_handler2* and *process_doors_mix*, are corresponding to the three threads in Figure 5.2. The *scheduler* is modeled by the *partition_level_OS*. The devices are implemented as Signal processes outside the *partition* providing external interface.

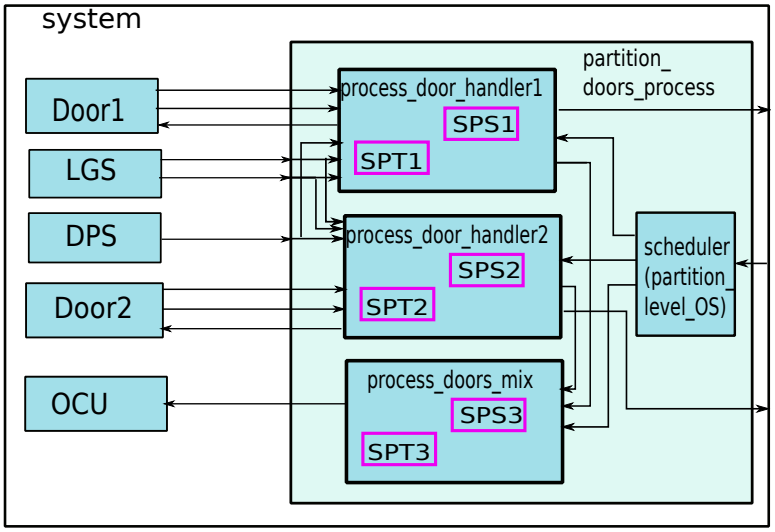


Figure 5.3: Signal model of the simple SDSCS system

5.3 From abstract logical time to more concrete simulation time

A high-level view of the transformation from AADL to synchronous model is given in this section. Synchronous paradigm provides an idealized representation of parallelism. While AADL takes into account computing latencies and communication delays, allowing to produce data of the same logical instants at different implementation instants. Those instants are precisely defined in the port and thread properties. To tackle this problem, we keep the ideal view of instantaneous computations and communications moving computing latencies and communication delays to specific "memory" processes, that introduce delays and well suited synchronizations. As a consequence, various properties result in explicit synchronization signals. Using the polychronous framework to model abstract logical time has to resolve the problems described in the following subsections.

First we briefly recall three simple Signal operations that will be used later.

- **Delay**. This operation (the '$' operator of Signal) gives access to past values of a signal. The delayed signal of s has the same clock as s. This delayed signal, $s\$1$ **init** s_0, takes the previous value of s.

- **Sampling**. The sampling operation (s **when** b) returns the value of s when b is true. The clock of s **when** b is less frequent than the clock of s and the clock of b.

- **Cell**. The cell operation is introduced for repeating some signal s on the instants of another boolean signal b. The result contains all values of s, and also the value of previous s when b is true. The clock of s **cell** b is the union of clock of s and of when b (Figure 5.4).

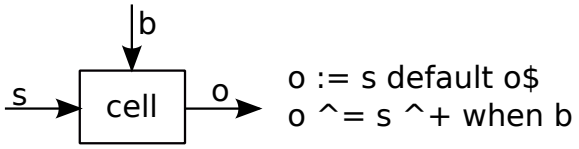


Figure 5.4: Cell example.

5.3.1 Modeling computation latencies

Problem A main feature of polychronous programs is the logical instantaneous execution, with respect to logical time. Components in polychronous model take no logical time: the outputs are immediately generated when the inputs are received. While in AADL, a thread may perform a function or a computation for a specified time interval, defined by the timing properties. Therefore, modeling AADL in the polychronous framework requires some adapter for interfacing abstract logical time and concrete simulation time.

With each output *o* of a process *P*, we associate a process *P_o* the output of which is the value of *o* delayed until its *Output_Time* represented by the input event signal *Output_Time* occurs. Due to scheduling, a process that is logically synchronous to a dispatch signal "*dispatch*", can be actually started later. Thus, for each process *P*, we associate an input event signal "*Start_Thread*" and execution of *P* is synchronized to *Start_Thread*.

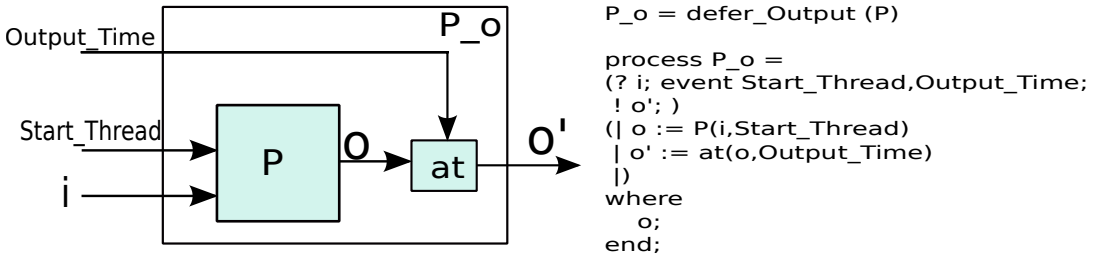


Figure 5.5: Modeling a time consuming task.

Solution We propose to translate *P_o* as a logical instantaneous process *P* and to delay *P*'s outputs until they are (AADL) available. This function is noted as $P\_o = defer\_Output(P)$. In Figure 5.5, the modeling principle is that: the task (e.g., a thread) itself is modeled by a Signal process *P* (which represents the synchronous program with activation condition), activated by an activation condition *Start_Thread*, which will be specified in the next subsection. The output of *P* is immediately available when it is computed. An additional delay component (the **at** operation) is introduced to hold the value. The output will be delayed and available until the next occurrence of the output enabling clock *Output_Time*.

A new operation is defined to express this delay action: **at**. Figure 5.6 shows its graphical representation. The process **at** is defined as a composition of a **cell** and **sampling** operations (the complete definition of this function is listed on the right of Figure 5.6):

$$o := i \text{ cell } e \text{ when } e$$

This **at** operation repeats the input signal *i* on the instants of activation signal *e*. The clock of the result output *o* is synchronous with the activation signal *e*. The value of the output takes the most recent value of *i*.

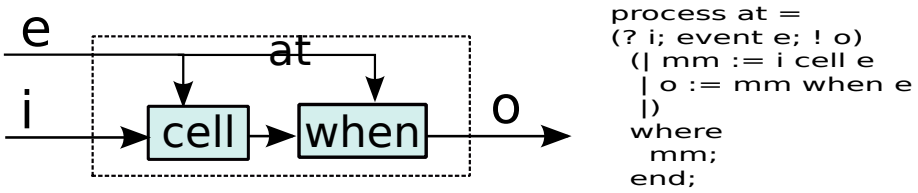


Figure 5.6: The “at” operation

In AADL, the output of a thread is available and transferred to other components at completion time by default, or at deadline time in the case of delayed port communication. Therefore, the clock of *Output_Time* is based on the type of the communication, which will be explained in later sections.

5.3.2 Modeling propagation delays

Problem During the execution of synchronous programs, each significant event or signal is precisely dated with respect to other signals, and with respect to the sequence of steps. While for AADL model, the input available time can be determined by different property values (a more detailed description is presented in Section 5.4).

Solution The Signal language provides features for expressing activation (the clocks of signals). The main idea to model AADL unprecisely known time within a synchronous framework is to use additional inputs, called activation condition, to model the propagation delays.

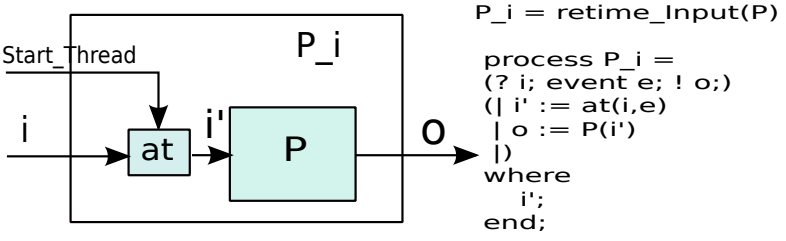


Figure 5.7: Activation condition.

An activation condition (for start time) is used to express the activation of a thread, either periodic or aperiodic. In Figure 5.7, the modeling takes a synchronous program *P* (which is the transformation of a thread), an activation event input *Start_Thread*, and produces a new conditional activation program *P_i*. This function is noted as *P_i = retime_Input*(*P*). The activation program *P_i* is represented as follows: it introduces an event input *Start_Thread*, the input *i* is resynchronized with *Start_Thread* before entering synchronous *P*. So that, the synchronous program *P* is activated by the activation condition *Start_Thread*.

5.3.3 Towards modeling time-based scheduling

Problem Based on the activation and delay mechanism as described in the previous two subsections, another problem comes out: how to control these activation and delay conditions, and from where they are generated.

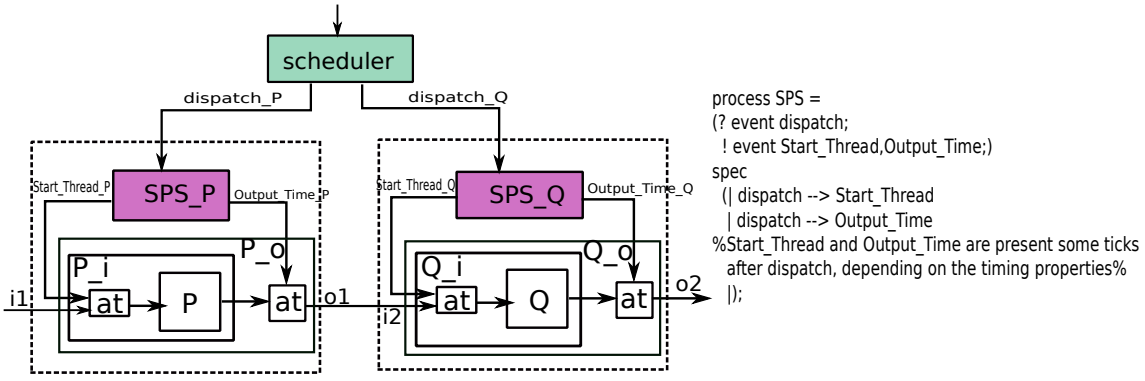


Figure 5.8: Modeling asynchronous composition.

Solution To solve this problem, we suppose that each thread P is associated with a timing environment SPS (Figure 5.8), which computes the start and completion clocks and other timing control signals of the thread, once it is activated by the scheduler. As we have mentioned in Chapter 2, each thread has some associated timing properties, such as *Deadline*, *Execution_Time* and so on. Once the thread is dispatched, the start and completion time can be calculated according to the specified timing properties.

Figure 5.8 shows a composition of two threads P and Q presented in Signal. Each component is activated by its own activation event *Start_Thread*. The two activation events *Start_Thread_P* and *Start_Thread_Q* are unrelated if there is no further control, which in this case, will result in non-deterministic execution. To avoid this situation, a timing environment process SPS is added for each thread and activated by the scheduler. Once SPS is triggered by the signal *dispatch* from the scheduler, it generates activation clock *Start_Thread* to activate the component, and computes the output available clock *Output_Time* that satisfies the specified clock properties.

Thus the input and output enabling signals *Start_Thread* and *Output_Time* are provided by the timing environment process SPS: this program calculates the input and output available time, according to the timing specification, once the thread is scheduled. The inner synchronous process P is started when the activation signal *Start_Thread* is present, and the outputs are available for sending when the signal *Output_Time* is present. The right part of Figure 5.8 shows an abstract specification of the process SPS. The signals *Start_Thread* and *Output_Time* are present some ticks after the dispatch of the thread.

The scheduler will decide when and which component can be activated. At each instant, only one component may be running. The scheduler calculates on the actual scheduling policy, and grantees the generated timing environment activations never occur at the same instant.

Having resolved these problems, now we can model the AADL threads and other components easily. The component modeling will be arranged in the following order: first, the main executable and schedulable component, which is the thread; then, the processor, which is responsible for scheduling the threads; then, the communication execution component, the bus, and the system; finally, the other components.

5.4 Thread modeling

Threads are the main executable and schedulable AADL components. A thread Th represents a concurrent schedulable unit of sequential execution through source code. To characterize its expressive capability, a thread Th encapsulates a functionality that may consist of ports, connections, properties, behavior specifications and subprograms that can be called by the thread.

$$Th = < P,\ C,\ R,\ Su,\ T/S >$$

- $P \in \mathscr{F}$ is the feature that consists of the ports, which declares the interface of the thread.
- $C = P \times P$ is the port connection, which is an explicit relation between ports that enables the directional exchange of data or event among its ports and other components.
- $R \in \mathscr{R}$ is the properties, which provide runtime aspects. For example, property *Period* declares the period for a periodic thread, and *Compute_Deadline* specifies the maximum amount of time allowed for the execution of a thread's computation sequence.
- Su is the subprograms called by the thread.
- T/S is the transition system defined with actions to provide the functional behavior.

The execution semantics of an AADL thread is as follows:

1. Read and freeze inputs. The content of incoming data is frozen for the duration of the thread execution. By default, the input is frozen at the time of dispatch. If the *Input_Time* property is specified, it is determined by the property value. The value can be *Dispatch_Time* (the default value, $t = 0$), *Start_Time* (the time is within a specified time range, $Start\_Time_{low} \leq t \leq Start\_Time_{high}$), *Completion_Time* (input is frozen at a specified amount of execution time relative to execution completion, $c_{complete} + Completion\_Time_{low} \leq t \leq c_{complete} + Completion\_Time_{high}$) and *None* (input is not frozen). Any input arriving after this frozen time becomes available at the next input time.
2. Execute and compute. When a thread action enters in computation state, the execution of the thread's entrypoint source text code sequence is managed by a scheduler. The actions can be external source text code, or represented by AADL behavior annex (which will be presented in next chapter).

3. Refresh and make available the actual output. The output is transferred to other components at complete time by default (at deadline time in case of delayed data port connection) or as specified by the value of the *Output_Time* property. Thus the value produced by the thread is available only for the threads that start after the completion of thread.

Figure 5.9 shows the default execution timing of an AADL thread.

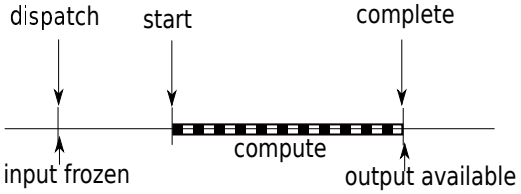


Figure 5.9: Default execution timing of a thread

5.4.1 Interpretation of a thread

Before starting the interpretation, let us recall the *process* concept of APEX ARINC. In APEX ARINC, *processes* represent the executive unit for an application. A *process* is a programming unit contained within a partition which executes concurrently with other *processes* of the same partition. It comprises the executable program, and attributes such as priority, period and deadline. These processes share common resources and execute concurrently within a *partition*.

An AADL thread Th is translated to an ARINC *process* P (a Signal process). The timing aspects of the thread will be translated as **process attributes**. The notation $\mathcal{I}$ is used to represent the translation.

$\mathcal{I}(Th) = P$ where Th is a thread, P is a Signal process representing an ARINC *process*

The detailed transformation from a thread Th to an ARINC *process* (represented as a Signal process) P can be obtained as follows, in these successive steps:

Interpretation steps

1. An AADL thread Th is firstly translated to a Signal process SP, which corresponds to an ARINC *process*, from the point of view of functional structure. SP has the same input/output flows as Th, plus an additional tick, which is an internal tick coming from the processor (the scheduler). This tick will be used later in the transformation of the behaviors or inside actions/computations. Each input (resp. output) port corresponds to an input (resp. output) signal. The actual inside actions, represented by the behavior specification, will be described in the next chapter.

   In this step, we only consider the interface of the components. The functional implementation will be presented in Chapter 6. SP is a synchronous interpretation of Th. It does not correspond to the real case of the actual AADL thread. The timing specifications will be taken into account in the next step.

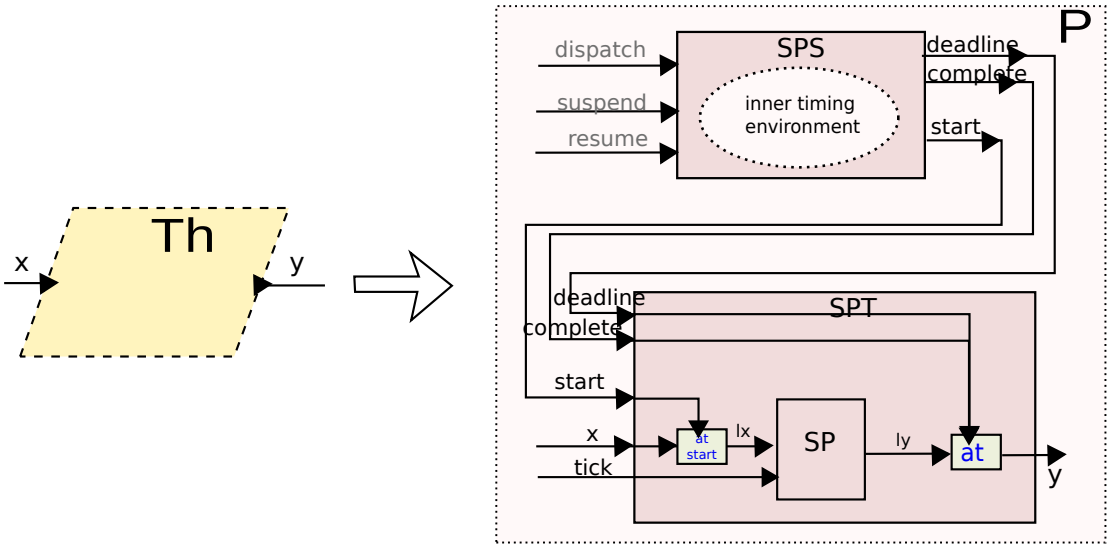


Figure 5.10: Translation of AADL thread with Signal thread.

2. According to the AADL specification, the input signals of a thread are transmitted from the sending threads at the complete time of these threads. Since the completion of each thread may take place at different instants, the input signals are not guaranteed to be synchronized. Due to the different timing semantics between AADL and Signal, timing properties of the AADL thread are translated by another Signal process, SPT (Figure 5.10), which plays the role of timing semantics interface between AADL and Signal.

   Three main functions of SPT with regard to SP are:

   (a) Memorize the input signals, which are received and frozen at the dispatch time, and preserve them until the thread is activated by the *start* event.

   (b) Activate the functional Signal *process* SP at the time of *start*. So that the input signals of SP are all synchronized at the clock of *start*.

   (c) Memorize the outputs of the thread until the time of completion (when the event *s_complete* is arrived).

   The memorization of signals and activation of thread are adopted here to bridge the AADL thread semantics and the synchronous model.

3. The execution of a thread is characterized by some real-time aspects. The timing properties specified in the thread, including period, dispatch time, start time, etc., are useful for the thread scheduling and execution. The thread will be scheduled following these properties. Due to this real-time control semantics, a new *process* SPS (Figure 5.10) is added, inside which the timing control signals are automatically computed once it is activated. When it receives the scheduling signals (e.g., *dispatch*) from the real thread scheduler, it starts to compute its own timing signals for activating and completing the *process* SPT.

SPS records all the temporal properties of the thread, e.g., *Period=10ms*, *Compute_Dealine=8ms*, and calculates the timing signals (*start, deadline*...). When it receives a scheduling signal (e.g., *dispatch*), a *deadline* signal is generated 8ms after *dispatch* (this is represented by 8 logical ticks in the example).

5.4.2 Implementation

As shown in Figure 5.10, an AADL thread `Th` is implemented as a Signal process, including an internal timing environment `SPS`, and a synchronization *process* `SP` which is included in a container process `SPT`:

$$\mathcal{I}(Th) \;=\; (|\, SPS \mid SPT \,|)$$

- `SPS` is a timing environment for the thread. It interfaces the scheduler and the thread. It receives the scheduler control signals, and computes the time of when starting execution, when completing, when ending deadline. When the thread is scheduled, `SPS` is triggered by the dispatch signal, it will generate the execution timing signals for `SPT`.
- `SPT` is an interface for synchronizing and activating the inputs and outputs of synchronous interpretation *process* `SP`. Its interface contains the declaration of the IO flows of `SP`, and some additional input events, e.g. `start`, `complete`.

$$SPT \;=\; (|\, IM \mid SP \mid OM \,|) \text{ where } Ldecl \text{ end}$$

- `IM` is the composition of the memorization of each input signal `x` by `lx:=at(x, start)`, where `at` is the process we have already introduced in previous section, defined as `lx:=x cell start when start`.
- `OM` is the composition of the "retiming" of every output signal `ly` of *process* SP by `y:=at(ly, complete)` (by default) or by `y:=at(ly, deadline)`. So that the synchronous outputs are delayed until the complete (or deadline) time.
- `SP` is the synchronous interpretation of thread `Th`. It is guaranteed that `SP` is activated at the time of start by requiring each input appearing at start time before entering `SP`.
- `Ldecl` is the declaration of the signals `lx,...; ly,...` that have been introduced.

Abstract rules An abstract description of the rules for the transformation of thread $Th = < P, C, R, Su, T/S >$ into Signal process (as APEX ARINC *process*) is given. The notation $\mathcal{I}$ represents the interpretation.

1. The thread component is translated as an ARINC *process*, parameterized by $\mathcal{I}(P)$, $\mathcal{I}(C)$, $\mathcal{I}(R)$, $\mathcal{I}(Su)$, $\mathcal{I}(T/S)$. The notation $Program^{parameters}$ is used to note that the *parameters* will serve in the definition of the *Program*. The role of these *parameters* is further explained below. The functionality of the thread is translated into two subprocesses, SPS and SPT, within the *process*.

$$\mathcal{I}(Th) \;=\; (SPS^{\mathcal{I}(R)} \mid SPT^{\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)})$$

2. The subprocess SPS is parameterized with the properties R. The interface of SPS includes the inputs *tick, dispatch, suspend, resume* and outputs *start, complete, deadline*.

$$\begin{aligned} SPS \;=\; &(?\ event\ tick,\ dispatch,\ suspend,\ resume;\\ &!\ event\ start,\ complete,\ deadline;) \end{aligned}$$

The properties $r \in R$, e.g., *Input_Time*, *Output_Time*, are interpreted in SPS. The values of these properties (*Input_Time_value, Compute_Execution_Time_value, Output_Time_value*) are used to count the logical ticks of the start and completion time of the thread.

$$\begin{aligned} count \;&:=\; 0\ \textbf{when}\ dispatch\\ &\quad\ \textbf{default}\ count\$ + 1\ \textbf{when}\ tick\\ &\quad\ \textbf{default}\ count\$\\ Start \;&:=\; true\ \textbf{when}\ (count \;=\; Input\_Time\_value)\\ Complete \;&:=\; true\ \textbf{when}\ (count \;=\; Compute\_Execution\_Time\_value)\\ Deadline \;&:=\; true\ \textbf{when}\ (count \;=\; Output\_Time\_value) \end{aligned}$$

3. The subprocess $SPT^{\mathcal{I}(P),\ \mathcal{I}(C),\ \mathcal{I}(Su),\ \mathcal{I}(T/S)}$ is parameterized with $\mathcal{I}(P)$, $\mathcal{I}(C)$, $\mathcal{I}(Su)$, $\mathcal{I}(T/S)$. The ports P are translated as the inputs/outputs, and the connections, subprograms and behaviors are translated in the subprocess SPT. The detail of these parameters are explained below.

4. The in/out ports P of the thread are translated to Signal inputs/outputs (the interpretation is presented in section 5.9.1). The interface of the ARINC *process* includes these signals and the event inputs *tick, dispatch, suspend, resume* received from the scheduler. The value *p.direction* denotes the direction of a port p.

$$\begin{aligned} inputs \;&=\; \{\text{tick, dispatch, suspend, resume}\} \cup \mathcal{I}(p) \quad \forall p \in P,\ p.direction = \textbf{in}\\ outputs \;&=\; \{\mathcal{I}(p)\} \quad \forall p \in P,\ p.direction = \textbf{out} \end{aligned}$$

5. The transitions/actions T/S are interpreted into a synchronous process SP following the steps presented in Chapter 6. This synchronous process SP is encapsulated

in the subprocess SPT. The value $c.type$ denotes the connection *type* of connection c, which can be either **immediate** or **delayed**.

$$
\begin{aligned}
&SPT \;=\; (IM \mid SP^{\mathcal{I}(Su),\mathcal{I}(C)} \mid OM)\\
&\text{where}\\
&\quad SP^{\mathcal{I}(Su),\mathcal{I}(C)} \;=\; \mathcal{I}(T/S)\\
&\quad IM \;=\; (ii_{p_{k_1}} := at(\mathcal{I}(p_{k_1}),\ start) \mid ii_{p_{k_2}} := at(\mathcal{I}(p_{k_2}),\ start) \mid \ldots)\\
&\qquad \text{for all } p_{k_j} \in P,\ s.t.\ p_{k_j}.direction = \textbf{in}\\
&\quad OM \;=\; (OM_{imm} \mid OM_{delayed})\\
&\quad OM_{imm} = (oo_{p_{k_1}} := at(\mathcal{I}(p_{k_1}),\ complete) \mid oo_{p_{k_2}} := at(\mathcal{I}(p_{k_2}),\ complete) \mid \ldots)\\
&\qquad \text{for all } p_{k_j} \in P,\ s.t.\ \exists c \in C,\ c = (p_1, p_{k_j}),\ c.type = \textbf{immediate},\ p_{k_j}.direction = \textbf{out}\\
&\quad OM_{delayed} = (oo_{p_{k_1}} := at(\mathcal{I}(p_{k_1}),\ deadline) \mid oo_{p_{k_2}} := at(\mathcal{I}(p_{k_2}),\ deadline) \mid \ldots)\\
&\qquad \text{for all } p_{k_j} \in P,\ s.t.\ \exists c \in C,\ c = (p_1, p_{k_j}),\ c.type = \textbf{delayed},\ p_{k_j}.direction = \textbf{out}
\end{aligned}
$$

6. Each subprogram $su \in Su$ is interpreted as some Signal process PP_{su} as presented in section 5.8.2:

$$\mathcal{I}(su) \;=\; PP_{su}(?\ In1, \ldots Inm; !\ Out1, \ldots Outn;)\ PP_{su}\_BODY$$

It is invoked as an instance in the subprocess SP:

$$(|\ o1 := Out1 \mid \ \ldots \mid on := Outn \mid PP_{su}\_BODY| \ In1 := i1 \mid \ \ldots \mid Inm := im)$$

A connection in a thread $c = (p_1, p_2) \in C$ connects the ports (parameters) among the invoked subprogram processes. Such connection is interpreted as an assignment connecting the input/output instances of the subprogram models:

$$\mathcal{I}(p_2) := \mathcal{I}(p_1)$$

Example The thread *doors_mix* shown here is a periodic thread. Only the port features are listed.

```
thread doors_mix
  features
    cll1: in data port behavior::boolean;
    cll2: in data port behavior::boolean;
    cll: out data port behavior::boolean;
end doors_mix;

thread implementation doors_mix.impl
  properties
```

```
    Dispatch_Protocol => Periodic;
    Period => 50 ms;
end doors_mix.impl;
```

The corresponding Signal code is shown in Figure 5.11. The thread is firstly translated to a Signal process *SP_doors_mix*, in which the synchronization is not considered. The two inputs *cll1* and *cll2* are guaranteed to be synchronized in a synchronization container *SPT_doors_mix*.

```
process SP_doors_mix =
      ( ? boolean cll1, cll2;
          event tick;
        ! boolean cll;)
      (|... |);

 process SPT_doors_mix =
      ( ? event start, complete, deadline, tick;
          boolean cll1, cll2;
        ! boolean cll; )
    (| lcll1 := at{}(cll1,start)
     | lcll2 := at{}(cll2,start)
     | lcll := SP_doors_mix{}(lcll1, lcll2, tick)
     | cll := at{}(lcll, complete)
     |)
    where
    boolean lcll1, lcll2, lcll;
    end;

process SPS_doors_mix =
    (? event dispatch, suspend, resume, tick;
     ! event start, complete, deadline;)
     (| count1 := 0 when dispatch when tick
                default count1$+1 when tick
      | start := true when (count1 = 3)
      | ...
      |)
    where
    integer count1; end;

process T_doors_mix =
     ( ? event dispatch, suspend, resume, tick;
         boolean cll1, cll2;
       ! boolean cll;
     )
   (| (start, complete, deadline) := SPS_doors_mix (dispatch, suspend, resume, tick)
    | cll := SPT_doors_mix{}(start, complete, deadline, tick)
    |);
```

Figure 5.11: Signal code of the thread

The two inputs *cll1* and *cll2* are delayed until *start* event occurs, see `lcll1 := at(cll1,start)`. Therefore, the input signals entering *SP_doors_mix* are synchronous. The output generated by *SP_doors_mix* is also delayed until *complete*, see `cll:= at(lcll, complete)`.

Since these timing events are not received directly from the scheduler, the process *SPS_doors_mix* is used to generate these events, once it receives the scheduling signals, *dispatch, suspend, resume*. The whole transformation is integrated in the process *T_doors_mix*.

5.5 Processor modeling

Processor component is an abstraction of hardware and software responsible for executing and scheduling threads. The property **Scheduling_Procotol** defines the way the processor will be shared between the threads of the application. The possible scheduling protocols could be: Rate_Monotonic, Earliest_Deadline_First, Deadline_Monotonic or other user defined schedulers.

In ARINC services, *processes* run concurrently and execute functions associated with the *partition* in which they are contained. The *partition_level_OS* selects an active *process* within the *partition* whenever it executes, that is to say, at any time, there is only one *process* that is activated. The scheduling policy for *processes* is priority preemptive.

The processor can be abstracted as the scheduler of the AADL processes/threads which are bounded to the processor, corresponding to the *partition_level_OS* in Signal, which is the control for the concurrent execution of *processes* within the *partition*. For the Rate_Monotonic scheduler we consider in the implementation, we can set the ARINC *processes* priorities according to the thread period, then the priority preemptive scheduler can be used.

As mentioned before, the ARINC scheduling policy is priority preemptive. The priority can be fixed or dynamic, depending on the scheduling policy. Each time the scheduler is active, the active process is the one in ready state which has the highest priority. To describe the scheduling and the choice of the active process (Figure 5.12), two additional services are considered for the scheduling: PROCESS_SCHEDULINGREQUEST and PROCESS_GETACTIVE.

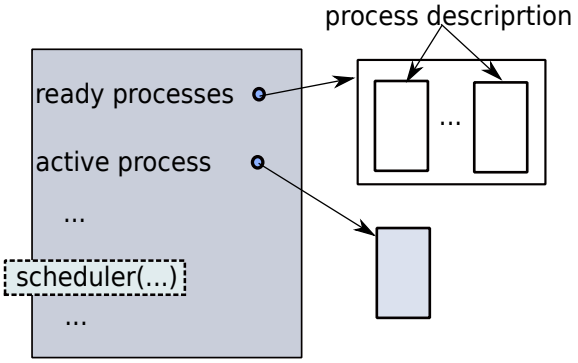


Figure 5.12: A process scheduler module

- The PROCESS_SCHEDULINGREQUEST service is invoked when the scheduler starts to choose an active process, or a process requests for rescheduling. It has an input event signal *request* and an output boolean signal *diagnostic*. The receiving of *request* signal will trigger a rescheduling. The output *diagnostic* is produced in the same instant. It is set to be true when the rescheduling is successful, otherwise it takes the value false. The interface of this service is given here.

```
process PROCESS_SCHEDULINGREQUEST =
  ( ? event request;  ! boolean diagnostic; )
  spec (| (| request --> diagnostic |)
```

```
        | (| request ^= diagnostic |)
        | )
    pragmas
      C_Code "&o1 = GLOBAL_PROCESS_MANAGER -> ProcessScheduling Request()"
    end pragmas;
```

- The PROCESS_GETACTIVE service is invoked after each rescheduling request for getting the information of the new active process. The Signal code is shown below. The input *request* denotes a receiving request. The output *process_ID* returns the active process identifier.

```
process PROCESS_GETACTIVE =
  ( ? event request;
    ! ProcessID_type process_ID;
      ProcessStatus_type status;)
  (| (pid, pstatus, ok) := prag_PROCESS_GETACTIVE{}()
   | process_ID := pid when ok
   | status := pstatus when ok
   | ok ^= request
   |)
  where
    ProcessID_type pid;
    ProcessStatus_type pstatus;
    boolean ok;
    process prag_PROCESS_GETACTIVE =
      ( ! ProcessID_type process_ID;
          ProcessStatus_type status;
          boolean ok;)
      spec (| process_ID ^= status ^= when ok |)
    pragmas
      C_Code "&o1 = GLOBAL_PROCESS_MANAGER -> ProcessGetActive(&o2, &o3)"
    end pragmas;
```

Interpretation We give an abstract interpretation of the processor in this part. A processor *Pr* is interpreted into a Signal process, and the ARINC service *PROCESS_CREATION* is invoked for each one of the threads which is bound to this processor (property **actual_processor_binding**). ARINC system services, *PROCESS_START*, *PROCESS_SCHEDULINGREQUEST*, etc., are used for the scheduling.

1. The *Processor* is transformed into a scheduler, which corresponds to the *partition_level_OS* of the *partition*.

$$\mathcal{I}(Pr) = (CREATION \mid START \mid SCHEDULING \mid SUSPEND)$$

2. CREATION. The ARINC *processes* (transformed from the *threads*) are recorded and created in the scheduler, with invocations to **PROCESS_CREATION**.

   The timing properties are set as the attributes of the thread. For example, the period of the *process* is set according to the property **Period**.

$$\mathcal{I}(R_{period}) = Period := period\_value$$

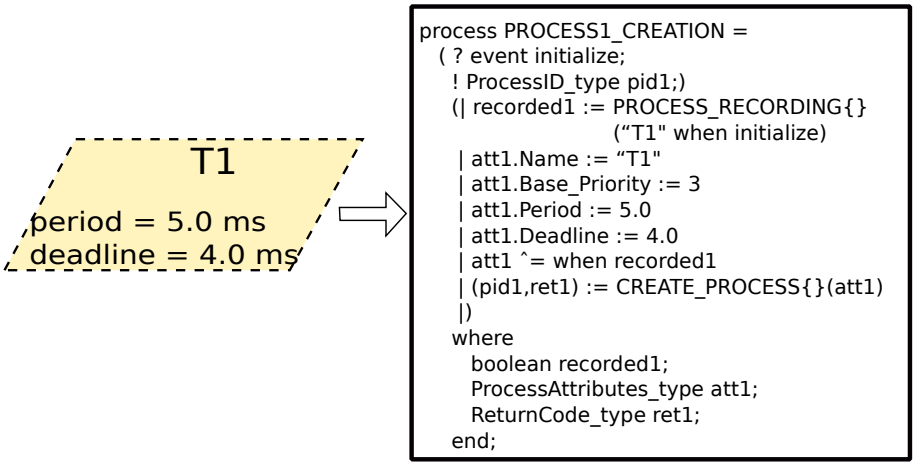


Figure 5.13: A process creation example

The period attribute is set to -1.0 when it is aperiodic.

$$Period := -1.0 \quad \text{if } Value(Dispatch\_Protocol) = aperiodic$$

The *process* priority is based on the scheduling policy. For example, in case of Rate_Monotonic, the *process* with the lowest period is set to the highest priority. In Figure 5.13, as an example, the attributes are set for a periodic thread named T1: the period and deadline attributes are set as the property of the thread, the priority is based on the comparison of the thread periods.

3. START. The *START()* process is invoked from the ARINC library to start the process to be active.

4. SCHEDULING. When the *partition* is activated, the *process* scheduling starts, a priority preemptive scheduling policy is provided. The *PROCESS_SCHEDULINGREQUEST()* process in invoked.

5. SUSPEND. The *process* will suspend when it finishes. The *SUSPEND_SELF()* process in invoked.

Abstract rules An abstract description of the rules for the transformation of processor *Pr* into Signal process is given.

1. A processor component *Pr* is translated to a Signal process *PP*, parameterized by the properties *R* that are defined in the bounded AADL processes and an activation signal *e*.

$$\mathcal{I}(Pr) = (CREATION^{\mathcal{I}(R)} \mid START \mid SCHEDULING^{e} \mid SUSPEND)^{\mathcal{I}(R),\, e}$$

2. The properties *R* used here are specified in the bounded processes or threads. The function *BindingProcess*(*Pr*) is used to get the processes that are bound to processor *Pr*. The property **Actual_Processor_Binding** (noted $R_{Actual\_Processor\_Binding}$,

specified in system component) has two parameters, *reference* and *appliesTo*: the referenced processes are applied to the specified processor.

$$BindingProcess(Pr) = \left\{ ps \in Ps \,\middle|\, \begin{array}{c} \exists r = R_{Actual\_Processor\_Binding} \in R, \\ r.appliesTo = Pr \ \wedge \ ps = r.reference \end{array} \right\}$$

From these processes ps, the period of each thread subcomponent can be obtained. The **Period** property is noted as R_{Period}, and its value is noted as *period_value*. Each **Period** property is interpreted into a period attribute of the corresponding ARINC *process* in the CREATION subprocess.

$$\mathcal{I}(R_{Period}) \ = \ Period := period\_value$$

3. The START, SCHEDULING and SUSPEND subprocesses invoke the processes from ARINC libraries: *START(), PROCESS_SCHEDULINGREQUEST()*, and *SUSPEND_SELF()*. When the activation signal e is received, the SCHEDULING starts to schedule the processes.

Example A Scheduler with two *processes* is given in Figure 5.14. The two *processes* are created and started. Then a scheduling request is performed, and one process is selected as active process.

```
1 process Scheduler = { integer Partition_ID; }
2   ( ? PartitionID_type active_partition_ID;
3       event initialize, end_processing;
4     ! ProcessID_type active_process_ID;
5       [2]boolean timedout;)
6   (| pid1 := PROCESS1_CREATION(initialize) %create the PROCESSes%
7    | return_code1 := START{}(pid1) %start the PROCESS to be active%
8    | pid2 := PROCESS2_CREATION(initialize)
9    | return_code2 := START{}(pid2)
10   | partition_is_running := when (active_partition_ID = Partition_ID)
11   | success := PROCESS_SCHEDULINGREQUEST{} (when partition_is_running)
12   %On receiving the input active signal, a priority preemptive scheduling is tried to be performed%
13   | (active_process_ID,status) := PROCESS_GETACTIVE{}(when success)
14   %invoked after each rescheduling request to get the current active PROCESS%
15   | timedout := UPDATE_COUNTERS{}() %manage the time counters associated with PROCESSes%
16   | timedout ^= when partition_is_running
17   | return_code3 := SUSPEND_SELF{}(7.0 when end_processing)
18   |)
19   where ...;
```

Figure 5.14: A scheduler example in Signal

First, the scheduler creates the two *processes* at *initialize* time, see lines 6 and 8. The *PROCESS1_CREATION()* and *PROCESS2_CREATION()* can be referenced on the example of Figure 5.13. The attributes are based on properties of the threads. Once they are created, they will be started, see process *START()* in lines 7 and 9. In this *START* process, the process will be checked whether it is valid or not, and its state will be set to *READY*.

The partition is activated when the received *active_partition_ID* equals to its *Partition_ID* itself, shown in line 10. After that, the scheduling starts when the partition is running. The processes *PROCESS_SCHEDULINGREQUEST* and *PROCESS_GETACTIVE* will be invoked. The *UPDATE_COUNTERS* in line 15 manages the time counters associated with the processes. When all the processes end processing, the scheduler will suspend itself, see line 17, by *SUSPEND_SELF*.

5.6 Bus modeling

A bus component represents hardware and associated communication protocols that enable interactions among other execution platform components (i.e., memory, processor and device). For example, a connection between two threads, each executing on a separate processor, is done through a bus between those processors. This communication is specified in AADL using **access** and **binding** declarations to a bus.

- **Bus access**. The execution platform components can declare a need for **access** to a **bus** through a **requires bus** reference. For example,

```
processor Intel_Linux
features
f1: requires bus access bus1;
end Intel_Linux;

bus bus1
end bus1;
```

- **Connection binding**. Connections between software components that are bound to different processors transmit their information across buses. The property **Actual_Connection_Binding** can be used to specify the actual bus applying to the port connections. For example, the connection *conn1* is bound to bus *bus1*.

Actual_Connection_Binding => **reference** *bus*1 **applies to** *conn*1

The **connection binding** of a port connection *conn1* is translated as the connection transfer through the bus process, see the top part of Figure 5.15. The output of *process1* is passed to *bus* transfer. The output from the *bus* is then sent directly to *process2*. The internal processing of the bus can be implemented as simple data transfer, or more elaborate queued or delayed transfer.

A bus connecting two components, for example, two threads which are bound to two different processors, is not granted to be synchronous. If the bus connects the input directly to the output, without any delay, there will be some problem. In AADL model, the input of the receiver and the output of the sender may not appear at the same instant. In that case, a delay is needed for the adapter. In our transformation, as described before, the output and input of a thread are both buffered before becoming available.

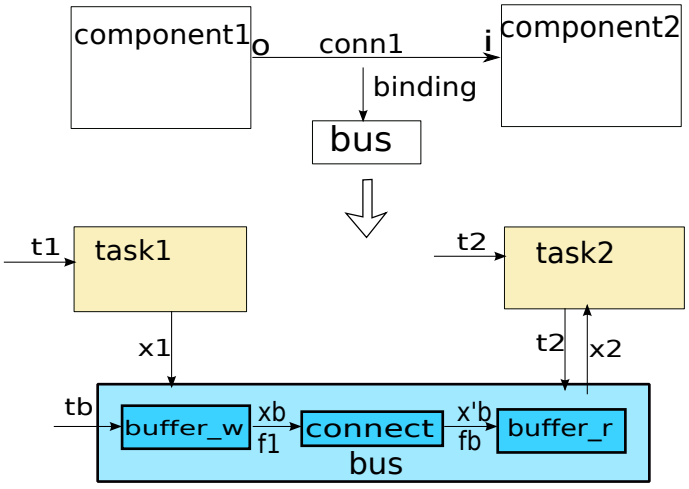


Figure 5.15: Translation of AADL connection binding bus to Signal Process

Interpretation We consider a simple case study, as presented at the bottom of Figure 5.15. The clock t_1 is the clock of the writer *task1* generating or releasing the output x_1. At any time when t_1 is present, the writer's output x_1 contains the last value that is available. At time t_b, the bus fetches the last value of x_1, and transfers it to store in the input buffer of the reader *task2*, denoted by x_b. At t_2, the reader loads the input buffer x_b into the variable x_2.

To model such a bus in Signal, we consider two data-buffering functions (two *buffers*) that communicate by a *connect* function, as shown in Figure 5.15. In Signal, we model an interface of these functions that exposes their control.

The *buffer_w* accepts an input x_1, which is generated by the writer, and buffers it until the bus tick t_b arrives, and defines the boolean flag f_1 that is carried along with it over the bus. The output x_b gets the last value of x_1 at each tick of t_b (the **at()** function has already been defined in Section 5.3.1).

$$(x_b, f_1) = \textbf{buffer\_w}(x_1,\ t_b) \stackrel{def}{=} \left(\begin{array}{l} x_b\hat{} = f_1\hat{} = \textbf{when}\ t_b \\ |\ x_b := \textbf{at}(x_1,\ \textbf{when}\ t_b) \\ |\ f_1 := \textbf{not}\ (f_1\$1\ \textbf{init true}) \end{array} \right)$$

The *connect* forwards the input x_b and f_1 to the reader's buffer. In general, it can be a FIFO [86] or some specification of buffer. In this first implementation, for simplicity, this connection is defined as the output equal to the input, $x'_b := x_b$, and the flag f_b equal to f_1.

$$(x'_b, f_b) = \textbf{connect}(x_b, f_1) \stackrel{def}{=} (x'_b := x_b \mid f_b := f_1)$$

The *buffer_r* receives the input x'_b if the value of x'_b changes, detected by the process **filter**, and stores it until the reader fetches it at t_2. The process **filter** receives a boolean input signal f_b, and produces an output signal f' every time the value of the input changes.

$$f' = \textbf{filter}(f_b) \stackrel{def}{=} (f' := \textbf{true when } (f_b \neq zf) \mid zf := f_b\$1 \textbf{ init true})/zf$$

$$x_2 = \textbf{buffer\_r}(x'_b,\ f_b,\ t_2) \stackrel{def}{=} (x_2\hat{} = \textbf{when } t_2 \mid x_2 := \textbf{at}(x'_b \textbf{ when filter}(f_b),\ \textbf{when } t_2))$$

The process *bus* is defined by its three components **buffer_w**, **connect** and **buffer_r**.

$$x_2 = \textbf{bus}(x_1, t_1, t_b, t_2) \stackrel{def}{=} (x_2 := \textbf{buffer\_r}(\textbf{connect}(\textbf{buffer\_w}(x_1,\ t_b)),\ t_2))$$

5.7 System modeling

The system is the top-level component of the AADL model, it represents a composite of interacting application software, execution platform, or system components. System can be organized into a hierarchy that can represent complex systems of systems as well as the integrated software and hardware of a dedicated application system.

A system $S =< P, C, R, W, Z >$ specifies the runtime architecture of an operational physical system.

$$\begin{aligned} \text{(system) } S &::= < P, C, R, W, Z > \\ \text{(connection) } C &::= P \times P \\ \text{(flow) } W &::= C \times C \\ \text{(subcomponents) } Z &::= S \mid Ps \mid Pr \mid M \mid B \mid Dv \mid Da \mid Z \parallel Z \end{aligned}$$

- $P \in \mathscr{F}$ is the feature that consists of the ports, which declares the interface of the system.
- $C = P \times P$ is the port connection, which is an explicit relation between ports that enables the directional exchange of data or event among components, or a data/bus access connection, which enables multiple components access to a common data/bus component.
- $R \in \mathscr{R}$ is the property which provides descriptive information. For example, property **Actual_processor_binding** declares the processor binding property along with the AADL components and functionality to which it applies.
- $W = C \times C$ specifies the flows that enable the detailed description and analysis of an abstract information path through a system. The flow path declares the flow through a component from one feature (port) to another.
- $Z \in \mathscr{Z}$ is the subcomponents of the system, which may consist of system, process Ps, processor Pr, memory M, bus B, device Dv and data Da.

A system may have one or several processors, and each processor can execute one or several processes, whereas an ARINC *partition* cannot be distributed over multiple

processors. The processor is allocated to each *partition* for a fixed time window within a time frame maintained by the core module level OS. So here we translate the system to a top-level Signal process, and separate it into several *modules* according to its owned processors.

Interpretation An abstract description of the interpretation is presented.

1. The system $S =< P, C, R, W, Z >$ is transformed into a Signal process, including a composition of ARINC *modules* $M_1 \mid M_2 \mid \ldots \mid M_n$, and a scheduler MS for activating the *modules*. A *module* M is a Signal process which consists of the subprocesses that are interpreted from the AADL processes (bound to a same processor). The corresponding Signal process is parameterized by a global tick gt, which is a main clock of the system, and the inputs/outputs that are translated from the ports $\mathcal{I}(P)$ (refer to Section 5.9.1).

$$\mathcal{I}(S) = \left(\begin{array}{l} M_1^{\mathcal{I}(C),\ \mathcal{I}(R),\ \mathcal{I}(W),\ \mathcal{I}(Z)} \\ \mid M_2^{\mathcal{I}(C),\ \mathcal{I}(R),\ \mathcal{I}(W),\ \mathcal{I}(Z)} \\ \mid \ldots \\ \mid M_n^{\mathcal{I}(C),\ \mathcal{I}(R),\ \mathcal{I}(W),\ \mathcal{I}(Z)} \\ \mid MS^{\,gt} \end{array} \right)^{\mathcal{I}(P),\ gt}$$

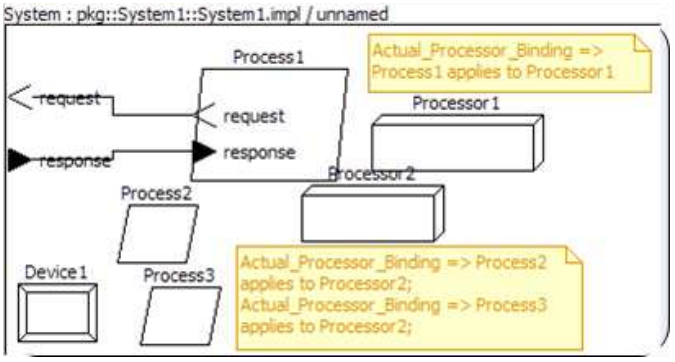
2. The processes bound to the same processor $pr \in Pr$, are translated to a *module* M_i, which receives control from its active signal e_i (generated by scheduler MS). The processes binding information can be obtained using the function *BindingProcess(Pr)* (refer to Section 5.5). The translation of the processor ($\mathcal{I}(Pr)$) is a scheduler PS, which is responsible for scheduling the processes. The interpretation of the processor ($\mathcal{I}(Pr)$) and process ($\mathcal{I}(Ps)$) can refer to Section 5.5 and Section 5.8.1 respectively. The interpretation of connection $\mathcal{I}(C)$ is similar to that presented in Section 5.4.2. A flow can be separated in a sequence of connections, hence its interpretation $\mathcal{I}(W)$ can refer to the interpretation of the connection.

$$M_i = (PS \mid P')^{\mathcal{I}(C),\ \mathcal{I}(R),\ \mathcal{I}(W),\ \mathcal{I}(Z),\ e_i}$$
$$PS^{\,e_i} = \mathcal{I}(pr)$$
$$P' = (\mathcal{I}(ps_1) \mid \ldots \mid \mathcal{I}(ps_m) \mid \mathcal{I}(C) \mid \mathcal{I}(W))^{\mathcal{I}(R)}$$

where $Ps_i = \{ps_1, \ldots, ps_m\}$ in the set of processes defined in Z such that

$$Ps_i = BindingProcess(pr) = \left\{ ps \in Ps \;\middle|\; \begin{array}{c} \exists r = R_{Actual\_Processor\_Binding} \in R, \\ r.appliesTo = pr \wedge ps = r.reference \end{array} \right\}$$

3. MS is a scheduler responsible for scheduling or activating the *modules*. The enable




```
system system1
  features
    request: out event port;
    response: in data port;
end system1;

system implementation system1.impl
  subcomponents
    Process1: process process1;
    Process2: process process2;
    Process3: process process2;
    Processor1: processor cpu;
    Processor2: processor cpu;
    Device1: device sensor;
  connections
    c1: event port Process1.request -> request;
    c2: data port response -> Process1.response;
  properties
    Actual_Processor_Binding => Process1 applies to Processor1;
    Actual_Processor_Binding => Process2 applies to Processor2;
    Actual_Processor_Binding => Process3 applies to Processor2;
end system1.impl;

process process1
  features
    request: out event port;
    response: in data port;
end process1;

process process2
end process2;

processor cpu
end cpu;

device sensor
end sensor;
```

Figure 5.16: An AADL system example

clock e_i of a *module* is related to the global tick gt. We use a process f to note the relation, which may depend on the scheduling policy.

$$MS = (e_1, \ldots, e_n) := f(gt)$$

In our translation, each processor corresponds to only one *module*, which means that no *modules* share the same processor. Hence for scheduling the *modules*, in this simple implementation, an activation clock e_i for the *module* is generated when the global tick gt is received.

$$MS = (e_1 := gt \mid \ldots \mid e_n := gt)$$

4. The subcomponents *Z* are transformed to Signal processes referring to their respective transformation rules.

5. The connection *C* is transformed into a Signal process, referring to section 5.9.

Example The system in Figure 5.16 shows both graphical and textual code of a client system. It contains two processors: *processor1* and *processor2*. The three processes are distributed to different processors: *process1* is bound to *processor1*, and the other two processes are bound to *processor2*, see the properties defined, for example, *Actual_Processor_Binding => Process1 applies to Processor1*.

The corresponding Signal model contains two *modules* (Figure 5.17). They are scheduled by a *module_level_OS*, which defines a static scheduling timing window for each *module*. The device subcomponent is translated as a Signal process in the model.

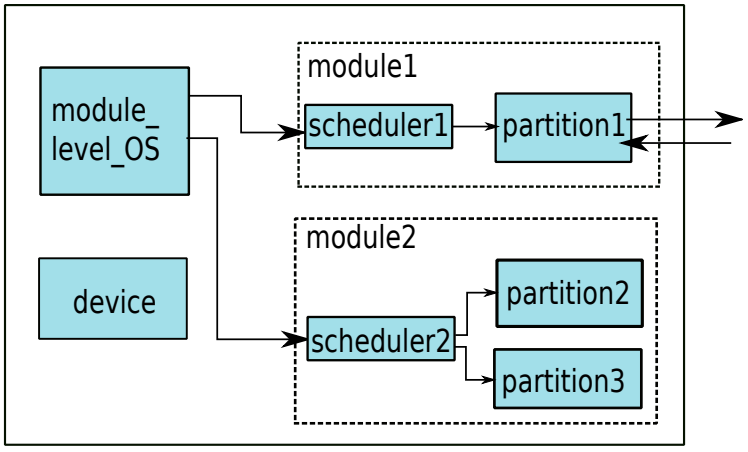


Figure 5.17: Signal process of the AADL system example

Each *module* is composed of a scheduler (translated from the processor) and the *partitions* on this processor. For instance, *module1* contains *scheduler1* (translated from *Processor1*) and *partition1* includes the subcomponent threads (translated as ARINC *process1*, *process2* and *process3*) of process1.

5.8 Other components modeling

5.8.1 Process modeling

The AADL process component represents a protected address space, a space partitioning where protection is provided from other components accessing anything inside the process. A process *x* encapsulates a functionality *A* that may consist of threads or thread group subcomponent, flows, connections, features and properties.

(process) **process** *x A where A, B* ::= **connection** *x A* | **subcomponent** *x A* | *A||B*

Interpretation of a process The unit of IMA partitioning is called a **partition**. **Partition** is a logical allocation unit resulting from a functional decomposition of the system. A **partition** is composed of **processes** that represent the executive units. A **partition** is basically the same as a program in a single application environment.

An AADL process A is translated to an ARINC **partition** P (represented as a Signal process). The threads contained in the process will be translated as ARINC *processes* within the *partition*, so that they can access the resource restricted in the **partition**. An inner scheduler is needed for scheduling these threads, which has already been presented in the section 5.5. The processes bound to the same processor are interpreted into an ARINC **module** M.

A function `BoundProcessor(A)` is introduced to return the processor to which the process *A* is bound. For example, for two processes *process1, process2* bound to the same processor, this can be denoted as:

$$BoundProcessor(\text{process1}) = BoundProcessor(\text{process2})$$

Abstract rules An abstract description of the rules for the interpretation of a process $Ps =< P, Th, C, R >$ into Signal process (as APEX ARINC *partition*) is given. A process Ps may have threads Th as its subcomponents, and it may contain ports P, connections C and some properties R.

1. A process Ps is interpreted into a Signal process consisting of a set of subprocesses translated from the threads and a scheduler. This process is parameterized by the interpretation of the ports $\mathcal{I}(P)$.

$$\mathcal{I}(Ps) \;=\; (PPs^{\mathcal{I}(Th),\ \mathcal{I}(C)} \mid Scheduler^{\mathcal{I}(R)})^{\mathcal{I}(P)}$$

2. The interpretation of the threads $\mathcal{I}(Th)$ and connections $\mathcal{I}(C)$ can refer to Section 5.4 and Section 5.9.2.

$$PPs \;=\; (\mathcal{I}(th_1) \mid \ldots \mid \mathcal{I}(th_n) \mid \mathcal{I}(c_1) \mid \ldots \mid \mathcal{I}(c_m)) \quad \text{where } th_i \in Th,\ c_i \in C$$

3. The interpretation of the scheduler ($Scheduler^{\mathcal{I}(R)}$) can refer to Section 5.5.

Example A process example is given here (partial code from the *doors_process* process in the SDSCS system, refer to Appendix A). There are three threads in this process as subcomponents.

```
process doors_process
  features
    in_flight: in data port behavior::boolean;
    cll: out data port behavior::boolean;
end doors_process;

process implementation doors_process.imp
  subcomponents
```

```
    door_handler1: thread door_handler.imp;
    door_handler2: thread door_handler.imp;
    door_mix: thread doors_mix.imp;
  connections
    conn1: data port in_flight -> door_handler1.in_flight;
    conn2: data port in_flight -> door_handler2.in_flight;
    conn3: data port door_mix.cll -> cll;
end doors_process.imp;
```

The corresponding Signal process is composed of three subprocesses interpreted from the threads, and one subprocess for scheduling, see Figure 5.18. The detailed code can refer to Appendix A.

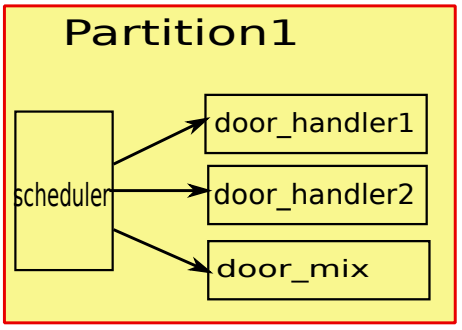


Figure 5.18: An AADL process in Signal

5.8.2 Subprogram modeling

Subprograms are callable components providing server functions to components that call them. A subprogram Su defines a functional unit of compilation and of execution that can be called from many contexts in the system under design.

Subprogram components represent elementary pieces of code that processes inputs to produce outputs. A subprogram encapsulates an expressive functionality that may consist of behavior specifications, ports (parameters), subprogram calls and connections among these called subprograms.

$$Su = < T/S,\ P,\ SuCall,\ C >$$

Interpretation A subprogram is translated to a Signal *process* parameterized by the translated ports (parameters) $\mathcal{I}(P)$.

$$\mathcal{I}(Su) = (\ \mathcal{I}(T/S) \mid \mathcal{I}(SuCall) \mid \mathcal{I}(C)\ )^{\mathcal{I}(P)}$$

The interpretation of the ports $\mathcal{I}(P)$ and the interpretation of the behavior specification of transitions/actions $\mathcal{I}(P)$ can refer to Section 5.9.1 and Chapter 6 respectively.

The subprogram calls *SuCall* are interpreted as instantiations of the called subprograms. For example, a subprogram $Su1$ is interpreted to a Signal process:

$$\mathcal{I}(Su1) = PP_{Su1}(?\ In1, \ldots Inm; !\ Out1, \ldots Outn;)\ PP_{Su1}\_BODY1$$

The instance of called subprogram $Su1$ is interpreted as a Signal process invocation.

$$Su1Call \;=\; (|\, o1 := Out1 \mid \ldots \mid on := Outn \mid PP_{Su1}\_BODY1 \mid In1 := i1 \mid \ldots \mid Inm := im)$$
$$\text{where } Su1Call \in SuCall$$

The connections $C = P \times P$ connect the ports in the called subprograms, and they are interpreted similarly to the connections (between the subprograms) presented in Section 5.4.2.

$$\mathcal{I}(c) \;=\; \mathcal{I}(p_2) := \mathcal{I}(p_1) \qquad \text{where } c = (p_1, p_2) \in C$$

5.8.3 Data modeling

Data represent static data and data types within a system in AADL. A data, noted **data** *x A*, defines an application data type. It could be used in the component properties.

$$\begin{aligned}(data)\quad \textbf{data}\ x\ A\ \textit{where}\ A ::= \ & \textbf{aadlinteger}\ A \mid \textbf{aadlfloat}\ A \\ & \mid \textbf{aadlboolean}\ A \mid \textbf{aadlstring}\ A \mid A_1 \parallel A_2\end{aligned}$$

Data types are structurally similar to Signal type declarations. Each simple Data type is transformed into a Signal type declaration.

$$\begin{aligned}\mathcal{I}(\textbf{aadlinteger}\ A) &= \langle \textbf{integer}\ A \rangle \\ \mathcal{I}(\textbf{aadlfloat}\ A) &= \langle \textbf{dreal}\ A \rangle \\ \mathcal{I}(\textbf{aadlboolean}\ A) &= \langle \textbf{boolean}\ A \rangle \\ \mathcal{I}(\textbf{aadlstring}\ A) &= \langle \textbf{string}\ A \rangle\end{aligned}$$

The compound data component is interpreted as a struct type.

$$\mathcal{I}(A \parallel B) \;=\; \langle \textbf{struct}\ (\mathcal{I}(A)\ ;\mathcal{I}(B)) \rangle$$

Data access A thread is considered to be in a critical region when it is accessing a shared data component. When a thread enters a critical region, a *Get_Resource* operation is performed on the shared data component. Upon exit from a critical region, a *Release_Resource* operation is performed.

Example Figure 5.19 is a data type example. On the left, the AADL data implementation includes three subcomponents: *name*, *text* and *size*. Each subcomponent is a basic data type (*aadlstring*, *aadlinteger*). On the right part of the figure is the corresponding

Signal data type structure. It is translated as a compound data structure, consisting of three attributes typed as *string* or *integer*.

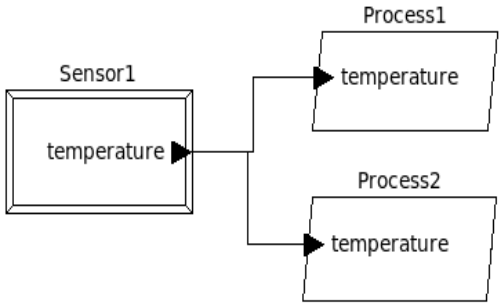
```
data message
end message;

data implementation message.impl
  subcomponents
  name: data aadlstring;
  text: data aadlstring;
  size: data aadlinteger;
end message.impl;
```

```
type message =
struct ( string name;
         string text;
         integer size; );
```

Figure 5.19: Data example and its Signal representation

5.8.4 Device

A device **device** *x A* interfaces the AADL model with its environment. A device often has complex behaviors, in this thesis, we only take its interfaces into consideration.




```
system sys
end sys;

system implementation sys.impl
  subcomponents
    sensor1: device sensor;
    process1: process pcs;
    process2: process pcs;
  connections
    c1: data port sensor1.temperature -> process1.temperature;
    c2: data port sensor1.temperature -> process2.temperature;
end sys.impl;

device sensor
  features
    temperature: out data port;
end sensor;

process pcs
  features
    temperature: in data port;
end pcs;
```

Figure 5.20: AADL device example

Devices are not translated as the other components, they are modeled outside the *partition*, the behavior implementation can be provided in some host language, such as C or JAVA.

Interpretation Each *device* is transformed into a Signal *process*, outside the *partitions*, providing external environment and complex behaviors. The input/ output ports are translated as the interfaces to the partition.

Example Figure 5.20 is a device example which has one output data port *temperature*. It is connected to two processes: *process1* and *process2*.

It is transformed into a Signal *process* with an output **temperature**, and this output is connected to the *partititon* as a source, shown in Figure 5.21. The interface of the device process is shown in the right.

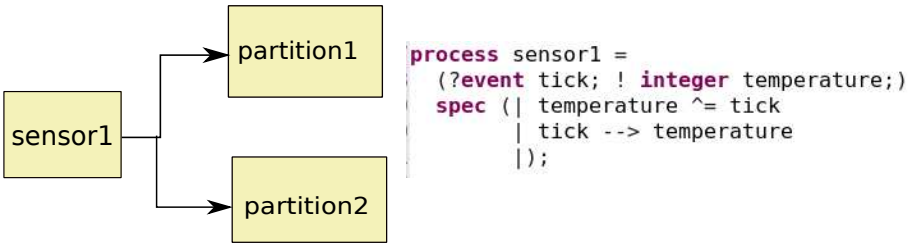


Figure 5.21: Signal representation of a device

5.9 Port and port connection modeling

Since ports are declared as features in all the component type declarations, and the port connections are declared between the ports of different components, the port and port connection modeling are presented in this separate section.

5.9.1 Port modeling

Port is a communication interface for the directional exchange of data, events, or both between components. Data ports allow communication without queuing. Event and event data ports represent buffered communications. They are declared as **features** in component type declarations. Ports are directional (**in** or **out**). An **in** port represents input needed by the receiver, and an **out** port represents output provided by the sender.

(*port*) **port** *x* ::= **in data port** *x X* | **in event port** *x X* | **in event data port** *x X*
| **out data port** *x X* | **out event port** *x X* | **out event data port** *x X*

Interpretation The event and event data ports represent interfaces for the communication of bounded queuing. They can be interpreted to bounded FIFO [86] in Signal. This is facilitated by offering a sliding window [58] operator, which allows for the specification of bounded FIFO protocols. It is not implemented in the current state, and a unique no-queuing signal is used instead for simple. The in/out port is interpreted to an input/output of the corresponding Signal process. The type is interpreted from the data type.

Example Figure 5.22 gives a simple port specification in a thread: *request* is an **in event port**, and *response* is an **out data port**, which is typed of message data. The corresponding Signal code is shown on the right. These two ports are translated as input/ output of the process.

```
thread t1
  features
    request: in event port;
    response: out data port message;
end t1;
```

```
process t1 =
  (? event request; ...;
   ! message response; ...;)
   (|...|);
```

Figure 5.22: Port example

5.9.2 Port connection modeling

Connections are explicit relationships declared between ports that enable the directional exchange of data or events among components. Interface elements are declared within the **features** section of a component type declaration. Paths of interaction between interface elements are declared explicitly within component implementations.

A connection can connect an event port x to an event port y, written **event** $x \rightarrow y$, or a data port x to a data port y, written **data** $x \rightarrow y$, or an event data port x to an event data port y, written **event data** $x \rightarrow y$. A connection from data port x to data port y, can either be an immediate connection, written **data** $x \rightarrow y$, or a delayed connection, written **data** $x \twoheadrightarrow y$. For a delayed data port connection, the value from the sending thread is transmitted at its deadline and is availabe to the receiving thread at its next dispatch. In this case, the data available to a receiving thread is that value produced at the most recent deadline of the sending thread.

The event and event data port connections can be implemented as bounded FIFO communication in Signal [86]. In the next subsections, we will focus on the data port connection.

5.9.2.1 Data port connection

Each periodic thread is scheduled periodically. A periodic thread is associated with some timing properties:

- The *Dispatch_Protocol* property is set to be *Periodic*.
- The *period* P_i is a fixed interval between two adjacent release of thread T_i.
- The *deadline* D_i is defined by *Deadline* property.
- The *Compute_Execution_Time* defines the execution time between best execution time B_i and worst case execution time W_i.

Once the periodic thread is dispatched, it calculates the actual execution time, which takes time between B_i and W_i. The thread has to be ended before D_i. A small example of periodic thread with timing properties is shown below:

```
thread T1
end T1;

thread implementation T1.impl
properties
  Dispatch_Protocol => Periodic;
  Period => 10 ms;
  Compute_Execution_Time => 3 ms .. 5ms;
  Deadline => 10 ms;
end T1.impl;
```

The type of connection between thread data ports requires specific timing semantics.

Figure 5.23 illustrates the instants where execution and communications take place. In general, a message is fetched at the dispatch time. In case of an immediate connection, (the sending and receiving threads must share a common simultaneous dispatch), a message is received at the start time. The output data is available at completion for an immediate data connection, or at deadline for a delayed data connection.

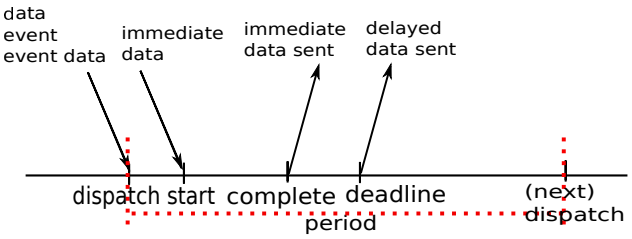


Figure 5.23: Thread communication time in AADL.

These connections have an influence on the following two aspects: first on the time at which the inputs and outputs are available and second, on the scheduling of the threads.

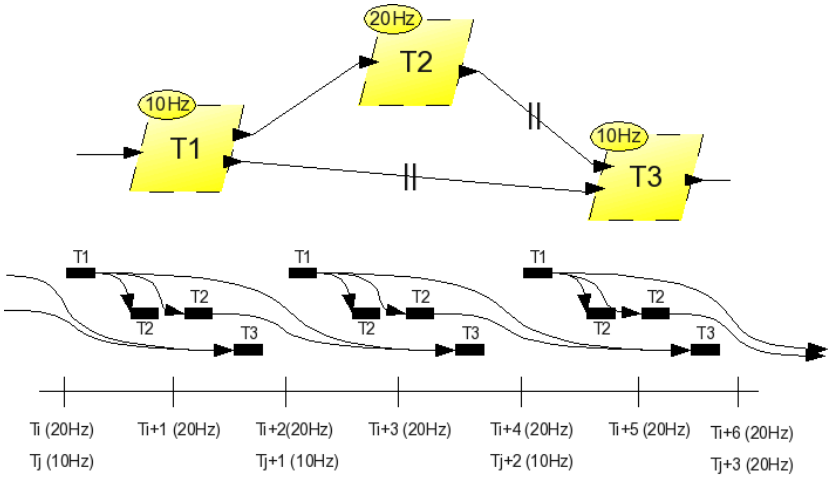


Figure 5.24: An example of communicating threads with different frequencies

The two data port communications have three types of thread configuration: same period sharing the same dispatch, oversampling and under-sampling. Figure 5.24 illustrates data consumption and scheduling of threads in consideration of two types of communication and threads with different frequencies. $T2$ is twice quicker than $T1$ and $T3$. The

connection from $T1$ to $T2$ is immediate, so the values available for two sequential executions of $T2$ are the same: the value produced within the 10Hz execution frame of $T1$. The connection from $T1$ to $T3$ is delayed, and the two threads share a common period. $T3$ receives the value produced by $T1$ in the previous period. The connection from $T2$ to $T3$ is also delayed, but $T2$ has a higher frequency. The data provided to an execution of $T3$ is the value produced by $T2$ that is available at the previous dispatch of the thread.

5.9.2.2 Data port connection modeling

For a communication of data port connection, the system satisfies the following form:

- The port connection is of type data port connection.
- The sender and receiver threads are both periodic, they have a *Dispatch_Protocol=>Periodic*.
- Periodic threads are guaranteed to be dispatched at each period.

The communication specification ensures that the time of transfer between the threads is well-defined. While the application code operates on the port variables, buffers are required to guarantee the availability of the variables. In this subsection, we propose a specification in Signal of the two data port communication types.

Modeling immediate data port connection For an immediate data port connection, both communicating periodic threads must be synchronized and dispatched simultaneously. Figure 5.25 illustrates immediate communication.

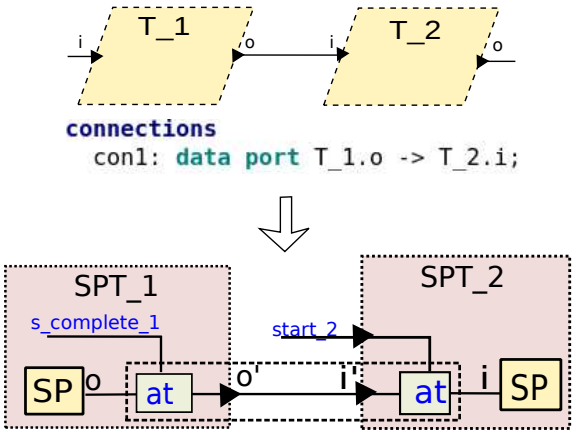


Figure 5.25: Translation of AADL immediate data connection with Signal processes

An immediate communication is implemented as a direct data dependency between two processes. The output is memorized until the complete time of the sending thread, and then transmitted to the receiving thread. It is memorized until the start time of the receiving thread.

An immediate communication is implemented as:

```
process immediate =
(? event s_complete_1, start_2;  o;  ! i;)
(| o_1:=at(o, s_complete_1)
 | i:=at(o_1, start_2)
 | )
  where o_1; end;
```

The use of an immediate connection avoids declaring an explicit offset for the communicating threads. The scheduler will dispatch the sender first, and ensure the receiver starts after the completion of the sender.

Modeling delayed data port connection Figure 5.26 illustrates this type of data connection. The output is buffered until the deadline of the sender thread, and then sent out to the receiver. The receiver will memorize the input until next dispatch.

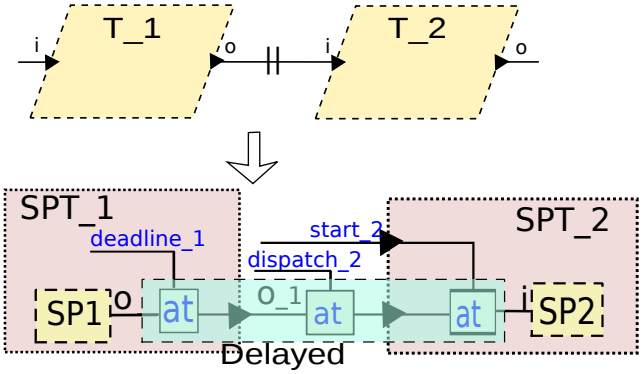


Figure 5.26: Translation of AADL delayed data connection with Signal processes

A delayed communication is implemented as

```
process delayed =
(? event s_deadline_1, dispatch_2, start_2;  o;  ! i;)
(| o_1:=at(o, s_deadline_1)
 | o_2:=at(o_1, dispatch_2)
 | i:=at(o_2, start_2)
 | )
where o_1, o_2; end;
```

This protocol ensures a deterministic communication between threads. Data availability depends on the deadline of the sender, which is specified and fixed by the thread property.

5.10 Towards AADLv2.0

AADLv2.0 [38] was published in January 2009. This version makes some improvements with respect to AADLv1.0 mainly for subprogram support, and introduces new components, i.e., virtual processor and virtual bus, and refines flows (allows flows through access features) and bus access (provided bus access for processor, memory and device),

etc. [38, 40]. The main basic components (i.e., threads, processes, bus, connections) remain the same or almost the same.

Since AADL thread is the main executable and schedulable component, it is taken as one of the most concerned components in our consideration. In AADLv2.0, the refinements for threads are mainly focused on the port access input time and output time. The difference is that the port send/receive time can be specified explicitly. New properties, *Input_Time* and *Output_Time*, are defined to specify the time values. For these new properties, the implementation can be improved, e.g., the activation of the inputs (resp. outputs) depends on the *Input_Time* (resp. *Output_Time*) if it is specified, otherwise the start time (resp. complete time).

The data port communication in AADLv2.0 introduces a declaration of sampling connection besides the immediate and delayed connection. In a sampling semantic connection, the recipient samples the output of the sender at dispatch time or as specified by the *Input_Time* property of the recipient port. This will require the improvement of data port connection implementation. For this new type of data port connection, a sampling process will be needed for the receiving thread. The output of the sender will be sampled by the signal (dispatch or *Input_Time*) depending on the specification.

AADLv2.0 provides **bus access** for processor, memory and device. The communications among these execution platforms can be explicitly specified through the bus. This feature can easily be implemented as bus connection in Signal similar to the bus connection binding.

The new introduced virtual processor and virtual bus are similar to process and bus. Virtual processors must be bound to or be a subcomponent of processors. It represents a logical resource that is capable of scheduling and executing threads. A virtual bus represents logical bus abstraction such as a virtual channel or communication protocol. These two components may have similar and refined implementations as processor and bus.

5.11 Conclusion

This chapter presents the AADL components transformation and their representations in Signal. The translation principles are given firstly. The transformation is based on the IMA architecture. Then we explain how we express AADL architectures and components in a synchronous framework, by allowing activation conditions, and modeling time consuming by means of delay outputs. We give a transformation description of a subset of AADL components, from the main execution component thread, the responsible scheduler processor, to communication execution platform bus and whole system and the other components. Apart from the component models, two types of data communication are also proposed for the Signal representation. This model transformation only considers the system architecture, further functional behaviors and implementation will be detailed in next chapter.

Chapter 6

AADL behavior specification

Contents

6.1 Static Single Assignment (SSA) 142

6.2 AADL transition systems 143

6.2.1 States 144

6.2.2 Transitions 144

6.2.3 Actions 146

6.3 Interpretation of transitions/actions 148

6.3.1 Step 1: actions to basic actions 149

6.3.2 Step 2: basic actions to SSA form actions 155

6.3.3 Step 3: SSA to Signal 160

6.3.4 Global interpretation 162

6.4 Case study 162

6.5 Conclusion 165

This chapter focuses on the transformation of AADL behavior specification [42]. Architecture and functionality are two main aspects of a system in AADL. The functionality of a component specifies the functional behaviors when it is in execution. In the previous chapter, we have presented the architecture and components of AADL and the transformation to Signal. The semantics and transformation of AADL behavior will be presented in this chapter.

The behavioral aspects can be described either with target languages, such as Simulink, VHDL, etc., or with AADL Behavior Annex. The AADL Behavior Annex, proposed in 2006, is an extension of the core AADL standard to specify the local functional behaviors of the components, without expressing them with external target language. The detailed behaviors are specified in this annex in order to perform model checking and advanced code generation [81]. It extends AADL with precise behavioral and timing analysis.

Behavior Annex lies in the computing state, which is an extension of the dispatch mechanism of execution model. The behaviors described in this annex are based on state

variables whose evolution is specified by the transitions.The behavior specifications describe a transition system [42]. It is attached to component implementations, mainly for threads and subprograms.

To perform model checking of a system described in AADL, the components behaviors are required. In reason of early system validation and verification, as mentioned in Chapter 5, an approach to automatically interpret the AADL behaviors into a suitable model of computation is proposed. In this method, notations of the Behavior Annex are embedded in the polychronous model of computation, for the purpose of formal verification and code generation.

In AADL Behavior Annex, multiple assignment of the same variable in the same transition is legal, while in Signal, it is not allowed to multi-define a signal in the same instant. Therefore, a rewriting of the assignments is required in the interpretation. In SSA (static single assignment) [70, 161] form, each variable appears only once on the left hand side of an assignment: it is assigned only once. Therefore, our interpretation uses SSA as an intermediate formalism.

The transitions and actions are transformed into a set of synchronous equations. This model transformation relies on an inductive SSA transformation algorithm across transitions/actions [122]. With regard to verification requirements, it minimizes the number of states across automata, hence provides better model checking performances.

This transformation is addressed in three steps: first, from the original transitions/actions to basic form transitions/actions (actions only consisting of assignments or sending/receiving actions); second, from the basic form transitions/actions to SSA form transitions/actions; Finally, from SSA form transitions/actions to Signal equations. The first two steps are used to get its SSA form representation, since the translation from SSA to Signal is easy to implement, and similar experiments have already been performed for obtaining Signal code through SSA from C/C++ programs [61].

In this chapter, a brief description of SSA is first given in section 6.1. The syntax and informal semantics of the Behavior Annex are described in section 6.2. Then, in section 6.3, we present general rules to interprete the AADL Behavior Annex into Signal, which results in a model transformation chain from Behavior Annex to synchronous models. A case study is presented in section 6.4. Conclusions are given finally.

In this transformation, we show not only how to translate the core imperative programming features into equations, but we also consider the mode automata that control the activation of such elementary transitions and actions.

6.1 Static Single Assignment (SSA)

A program is said to be in static single assignment (SSA) [70, 64, 63, 69] form whenever each variable in the program appears only once on the left hand side of an assignment. Only one new value of a variable x should at most be defined within an instant. The SSA form of a program replaces assignments of a program variable x with assignments to new versions x_1, x_2, ... of x, uniquely indexing each assignment.

```
void ones(int x, int *y)    | bb_0: ix_1=x;
{ int ix, count;            |       count_1=0;
  ix=x;                     |       goto L1;
```

```
  count=0;                  | L_0:  count_2=count_3+ix;
  while (ix) {              |       ix_2=ix_3-1;
    count=count+ix;         | L_1:  count_3=PHI<count_1, count_2>;
    ix=ix-1;}               |       ix_3=PHI<ix_1, ix_2>;
  *y=count ;                |       if (ix_3!=0) goto L_0;
}                           | L_2:  *y=count_3;
                            |      return;
```

We depict the structure of the SSA form with a typical C program (see above). Its SSA form (right) consists of four blocks. The block labeled *bb_0* initializes the local state variables *ix* and *count*. Then it passes control to the block *L_1*. *L_1* evaluates the termination condition of the loop. If the termination condition is satisfied, control goes to block *L_2* where the result is copied to the output *y*. Otherwise, control goes back to *L_0*, which contains the actual loop contents that decreases *ix*.

In this SSA form, each variable is rewritten with a new version, for example, *ix* is rewritten as *ix_1, ix_2* and *ix_3*. For each assignment, the use of *ix* (on the right side of the equation) refers to the last version of *ix*.

In block *L_1*, `ix_3=PHI<ix_1,ix_2>`, a *PHI* node is added to choose the value of *ix*. It merges all the different versions of the variables coming from the predecessors of the current block. It has the form $x_i = PHI < \ldots, x_j, \ldots, x_k, \cdots >$, where operands of *PHI* are new names of *x* associated with the predecessors of the current block. The *PHI* operator is needed to choose the value depending on the program control-flow, where a variable can be assigned in both branches of a conditional statement or in the body of a loop.

As SSA is an intermediate representation, introducing the *PHI*-node will not necessarily consume execution cost. The compilation from Signal will optimize the final execution code.

This approach introduces an effective method for transforming behavior specifications consisting of transitions and actions into a set of synchronous equations. Our transformation minimizes the needed state variables and component synchronization.

6.2 AADL transition systems

Behavior specifications can be attached to AADL component implementations using an annex. The AADL Behavior Annex describes a transition system. The overall structure of the behavior annex is a non-deterministic hierarchical automaton [81] using a Stateflow [27]-like mode automaton syntax: the annex mainly declares states and transitions with guards and an action part. The actions are related to the transitions not to the states. The skeleton of the annex behavior specification is first given. This annex consists of these optional sections: state variables, states and transitions.

```
annex behavior_specification {**
  state variables
    x: Tox;
  states
    s: initial state;
    t: return state;
  transitions
```

```
    s-[g]->t {S};
**};
```

We will formalize a semantics of AADL Behavior Annex by isolating the core syntactic categories that characterize its expressive capability: states, transitions and actions.

6.2.1 States

A state variable x declares typed identifier. State variable $x : Tox$ declares a state variable x whose type is Tox. Type Tox is a data classifier of the AADL model. A state variable $x1$ with a type of *Behavior::integer* is declared in the example.

```
state variables
  x1: Behavior::integer;
```

States s declare automaton states which can be qualified as **initial**, **complete** or **return**: s : (**initial** | **complete** | **return**) **state**. A state $s0$ qualified as **initial** and **return** is declared in the example.

```
states
  s0: initial return state;
```

6.2.2 Transitions

A transition T defines a system transition from a source state s to a destination state t. The transition can be guarded with events or boolean conditions g, and actions S can be attached.

Guard The guard g contains an optional event or event data receipt e and an optional boolean condition.

$$\begin{aligned} g &::= \textbf{on}\ exp \quad | \quad \textbf{when}\ exp \quad | \quad e \\ e &::= p?[(x)] \end{aligned}$$

- An event e can be a receipt from an event port ($p?$), or from an event data port ($p?(x)$) where x is a state variable or a data subcomponent.
- The condition may depend on the received data, and it can be split in two parts: the **on** part expresses conditions over the current state, and the **when** part expresses a condition over the data to be read.

  An expression exp may be a port identifier *port_id*, or a *constant*, or an arithmetic or boolean operation over expressions.

exp	::=	exp **op** exp	*binary operation*
		| **not** exp	*unary operation*
		| port_id	*port identifier*
		| constant	*constant value*
op	::=	**and** | **or**	*boolean operation*
		| < | <= | > | >= | = | ! =	*boolean operation*
		| + | − | ∗ | /	*arithmetic operation*

Transition The transitions section declares a set of finite transitions T. When the behavior specification of a thread is first triggered, it starts execution from an **initial** state, specified as s **: initial state**, and ends with a **complete** (**return** for subprogram) state, specified as t **: complete state** (or t **: return state**). From state s, if the guard g is true, a transition is enabled, the action part S is performed, and then enters state t, written $s - [g] \rightarrow t\,\{S\}$. The notation ";" in AADL notes the parallel composition of transitions.

$$\mathscr{T} \ni T \;::=\; s - [g] \rightarrow t\,\{S\} \;\mid T_1; T_2 \qquad \begin{array}{l} s, t : \textit{state} \\ g : \textit{guard} \\ S : \textit{actions} \end{array}$$

Figure 6.1 introduces the behavior specification at a syntactic level. It comprises two transitions with an initial state *s0*, which is also a complete state, written **s0: initial complete state**. The thread will execute the transitions depending on the guarded conditions.

```
thread implementation door_handler.imp
annex behavior_specification {**
  states
   s0: initial complete state;
  transitions
   s0 -[on (dps > 3) and handle]-> s0 {
         warn_diff_pres:=true;
         door_info:=locked;
         if (on_ground) door_locked:=false;
         else door_locked:=true; end if;
        };
   s0 -[on not (dps > 3 and handle)]-> s0 {
         warn_diff_pres:=false;
         door_info:=locked;
         if (in_flight) door_locked:= true;
         else door_locked:=false; end if;
        };
  **};
end door_handler.imp;
```

Figure 6.1: Behavior Annex example

- When dispatched, the execution of the component starts on an **initial** state. When the thread *door_handler.impl* in Figure 6.1 is dispatched and started to compute, it transits from state *s0*.

- From the current state, if the guard is satisfied, the transition is enabled. For the example in Figure 6.1, if the boolean condition *(dps>3)* **and** *handle* is true, the first transition will be triggered, otherwise, the second one will be triggered. Meanwhile, the attached actions are executed during this transition. After that, it enters back to the destination state *s0*.

- A thread completes its execution after reaching a **complete** state. It will resume from that state at next dispatch.

6.2.3 Actions

An action $S \in \mathscr{S}$ is recursively a sequence of actions, a loop on actions, an if statement, a timing action (**computation()**, **delay()**) or basic actions. A basic action $B \in \mathscr{B}$ is recursively a sequence of basic actions, assignment action $w := f(y, z)$, or message reception action ($p?(x)$), or message emission action ($p!(x)$).

$$
\begin{aligned}
\mathscr{B} \ni B ::= {} & w := f(y,z) \quad | \; p?(x) \quad | \; p!(x) \quad | \; B_1; B_2 \\
\mathscr{S} \ni S ::= {} & \textbf{if } x \textbf{ then } S_1 \textbf{ else } S_2 \\
& | \; \textbf{for } (x \textbf{ in } X)\ S \\
& | \; \textbf{delay}(min,\ max) \\
& | \; \textbf{computation}(min,\ max) \\
& | \; B \\
& | \; S_1; S_2
\end{aligned}
$$

- An assignment $w := f(y, z)$ defines the value of w from the function f of the current values of y and z. w, y, z can be a port identifier, or a state variable, or an unsigned numeric or constant. The function f can either be a boolean function (**not, and, or, >, >=, <, <=, !=**) or an arithmetic function (**+, -, *, /**).

- The message sending or receiving actions express the communication of messages. A statement $p!(x)$ writes data x to the event data port p and calls the send service. Each time instant, the same port p can only call the send service once. The event is immediately sent to the destination with the stored data. Message sending on output ports is always non blocking. If present, the data value is written into the output port, and a send call is implicitly performed. A statement $p?(x)$ dequeues a data from event port p in the variable x.

- The conditional **if** x **then** S_1 **else** S_2 executes S_1 if the current value of x is true, otherwise executes S_2.

```
annex behavior_specification {**
  state variables
    X: DataX;
  states
    s0: initial state;
    s1: state;
    t: complete state;
  transitions
    l1: s0 -[]-> s1 { for (i in 1 .. 2) { r := r + i;} };
    l2: s1 -[]-> t { for (x in X) { ... } };
```

```
data DataX
  features
    X1: subprogram X1;
    X2: subprogram X2;
end DataX;

subprogram X1
  features
    result: out parameter DataX;
end X1;

subprogram X2
  features
    result: out parameter DataX;
end X2;
```

Figure 6.2: A loop example, iteration over finite integer ranges and over enumerated type

- The finite loop **for** (*x* **in** *X*) *S* allows iterations over finite integer ranges or over unparameterized types, which are specified by a data classifier (for example, a subcomponent data of enumerated type). Figure 6.2 (left) gives an example of loop. In the first transition, labeled *l1*, the loop iterates over integer range 1..2. In the second transition, labeled *l2*, the loop iterates over an enumerated data type *DataX*. The constants of this type *DataX* are declared as subprograms having one output parameter of this type, shown in Figure 6.2 (right).

- The timing actions **delay**(*min*, *max*) and **computation**(*min*, *max*) specify non-deterministic waiting time and computation time intervals: **delay**(*min*, *max*) expresses a suspension for a possibly non-deterministic period of time between *min* and *max*; **computation**(*min*, *max*) expresses the use of the CPU for a possibly non-deterministic period of time between *min* and *max*. The difference of the two actions is whether using CPU or not. For a **delay** action, the thread or subprogram will be suspend and preempted.

  The timing actions are different from the timing properties specified in the component. For example, the *start* property of a thread specifies the moment when the thread can start its execution, and *deadline* property specifies the moment when the thread must stop its execution. When a **computation**(2ms, 3ms) action is triggered, its execution will last for 2.5ms, for example. It can be seen that, during these 2.5ms, the thread does some computation which consumes CPU but is not specified.

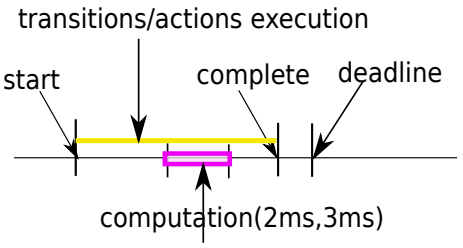


Figure 6.3: A **computation** example in the execution of a thread

Figure 6.3 shows a thread having a **computation** action during the execution. The

transitions/actions are performed in the time interval from *start* time to *complete* time, labeled with yellow line. A **computation** action (labeled with red rectangle) is executed somewhere when it is triggered, which takes a time duration between 2ms and 3ms (2.5ms for example). It is an action, but with specified duration time.

- Sequences of actions $S_1; S_2$ are executed in order.

The action is related to the transition, and not to the states: if a transition is enabled, the action part is performed, and then enters the transition destination state. The execution of an action starts at an occurrence of its active clock (when the transition to which it is attached is enabled), and shall end before the next occurrence of that event.

6.3 Interpretation of transitions/actions

Since the Behavior Annex supports precise behavior specification without requiring external source code, it offers a way to specify the functional behaviors locally. Therefore, the modeling of AADL in Signal also includes the interpretation of Behavior Annex.

In order to distinguish between the transitions of different interpretation stages, we first introduce some notations. We use $S \in \mathscr{S}$ to represent a general action in the original transition $T \in \mathscr{T}$, and $U \in \mathscr{U}$ to represent the intermediate transitions, called basic transitions, that are attached with the basic actions $B \in \mathscr{B}$.

$$\begin{aligned} \mathscr{U} \ni U &::= s - [g] \rightarrow t \, \{B\} \quad | \quad U_1; U_2 \\ \mathscr{T} \ni T &::= s - [g] \rightarrow t \, \{S\} \quad | \quad T_1; T_2 \end{aligned}$$

A notation $def(A)$ is introduced to refer the left hand side w of an assignment action A ($w := f(y, z)$), and $use(A)$ to refer the right hand side $f(y, z)$.

A SSA form action $A \in \mathscr{A}$ is recursively a sequence of SSA form actions, or an assignment of $w := f(y, z)$, in which w is defined at most once in A. In other words, for any $\mathscr{S} \ni A = A_1; A_2$, if $w \in def(A_1)$, then $w \notin def(A_2)$, the same for the reverse. Its associated transition is called SSA form transition $W \in \mathscr{W}$.

$$\begin{aligned} \mathscr{A} \ni A &::= w := f(y, z) \quad | \quad A_1; A_2 \\ \mathscr{W} \ni W &::= s - [g] \rightarrow t \, \{A\} \quad | \quad W_1; W_2 \end{aligned}$$

Transformation steps Because actions are attached to the transitions, we must take into consideration the states which they depart from and enter in, when interpreting the actions.

We use $\mathcal{I}(T)$ to note the interpretation of a transition T to Signal process. The transformation of the AADL transitions/actions to Signal $\mathcal{I}(T)$ is addressed in three steps (as illustrated in Figure 6.4):

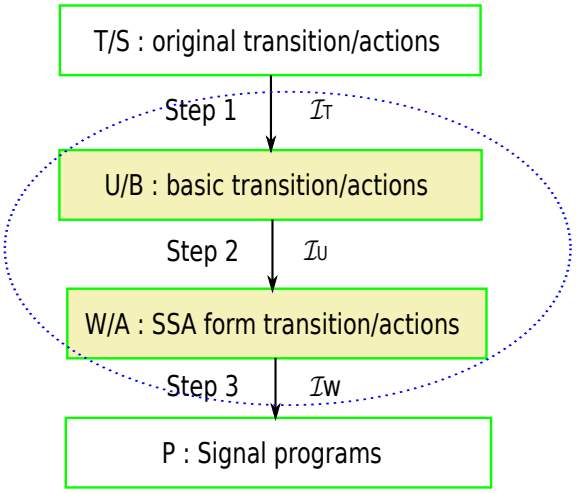


Figure 6.4: Behavior annex transformation chain

$$\mathcal{I}(T): \quad T \overset{\mathcal{I}_T}{\mapsto} U \overset{\mathcal{I}_U}{\mapsto} W \overset{\mathcal{I}_W}{\mapsto} P$$

- Step 1: $T \overset{\mathcal{I}_T}{\mapsto} U$. Each transition $T \in \mathscr{T}$ with attached sequence of general actions $S \in \mathscr{S}$, is decomposed into sets of basic transitions $U \in \mathscr{U}$, in which all the actions are basic actions $B \in \mathscr{B}$.

- Step 2: $U \overset{\mathcal{I}_U}{\mapsto} W$. For each intermediate transition $U \in \mathscr{U}$, depict the basic actions $B \in \mathscr{B}$ in SSA form $A \in \mathscr{A}$ and introduce new transitions if necessary. Each definition of an original variable w is replaced by a new version, no multiple definitions exist in SSA form, so that the SSA form actions can be executed in the same instant.

- Step 3: $W \overset{\mathcal{I}_W}{\mapsto} P$. Translate the SSA form actions $A \in \mathscr{A}$ and transitions to Signal equations.

Each step of the interpretation $\mathcal{I}_T, \mathcal{I}_U, \mathcal{I}_W$ will be explained in detail in the following subsections.

6.3.1 Step 1: actions to basic actions

The first step of the AADL behavior actions/transitions interpretation involves the decomposition of original transition $T \in \mathscr{T}$ into sets of basic transitions $U \in \mathscr{U}$.

A transition from a source state s when condition (guard) c is satisfied, to a destination state t, with attached general action S, noted as $s - [c] \rightarrow t\{S\}$, can be decomposed into a set of intermediate transitions $U \in \mathscr{U}$, in which the actions in each new transition are basic actions $B \in \mathscr{B}$.

We use $\mathcal{I}_T(T)$ to represent this interpretation from a general transition $T \in \mathscr{T}$ to basic transition $U \in \mathscr{U}$.

$$\mathcal{I}_T(T) : T \mapsto U \quad \text{where} \quad \begin{array}{l} T \in \mathscr{T} \\ U \in \mathscr{U} \end{array}$$

The interpretation of the transitions can be performed in parallel. The operator ";" notes the composition of transitions.

$$\mathcal{I}_T(T_1; T_2) \;=\; \mathcal{I}_T(T_1); \mathcal{I}_T(T_2)$$

For the decomposition of an original transition T into basic transitions U, we only need to decompose the non-basic actions (i.e., if and loop statement) into basic actions with intermediate transitions. For this purpose, first, processing the sequences, the non-basic actions need to be separated from the original actions; second, the decomposition of non-basic actions to basic actions is performed.

6.3.1.1 Processing sequences

We use an intermediate recursive function $\mathcal{I}_{TS}$ to make explicit the separation of the actions of a transition T as the sequential composition of a first block of processed basic actions, noted B, and a second block, noted S, where the actions are not yet transformed. The application of this function is denoted $\mathcal{I}_{TS}(B, S)_{s,\ c,\ t}$, for of a transition $T = s - [c] \rightarrow t\ \{B; S\}$, where B consists of the basic actions from the beginning of actions, S includes the remaining actions, and $s,\ c,\ t$ are respectively the source state, the guard and the exit state.

The basic rule of the recursive transformation is that, from the beginning of the original actions, if an action is a basic action, it is moved to B, until a non basic action occurs. B is separated, and an intermediate transition is introduced with actions B, from the source state s to an intermediate state s_1. For the remaining actions S, first the non basic actions will be decomposed using the rules that will be presented in section 6.3.1.2, then for the rest, the separation will continue following the same rules.

1. For a transition $T = s - [c] \rightarrow t\ \{S\}$, at the very beginning of the interpretation, there is no basic action before S. We set B to be empty, represented by ϕ.

$$\mathcal{I}_T(s - [c] \rightarrow t\ \{S\}) \;=\; \mathcal{I}_{TS}(\phi, S)_{s,c,t}$$

Example 1 Take the first transition presented in Figure 6.1 for example. The actions attached to this transition are noted as S. At this stage, no basic action is set, B is empty, noted as $B = \phi$.

$B = \phi$
S = {warn_diff_pres := **true**;
door_info := locked;
if (on_ground) door_locked := **false**; **else** door_locked := **true**; **end if**;}

2. If there is no actions following B in the interpretation, which means that all the actions are basic actions, then the resulting transition is the same as the original one.

$$\mathcal{I}_{TS}(B, \phi)_{s,c,t} = s - [c] \rightarrow t \{B\}$$

Example 2 The actions in the following example are two assignments, both are basic actions, and there is no action after them. So the resulting transition is the same as the one presented here.

$$s - [c] \rightarrow t \{\text{warn\_diff\_pres} := \textbf{true}; \text{ door\_info} := \text{locked};\}$$

3. If the action to be processed is a basic action $B' \in \mathscr{B}$, it can be merged to the previous basic action set B. So that, $B; B'$ becomes the new basic action set, and the remaining actions S will continue the processing.

$$\mathcal{I}_{TS}(B, B'; S)_{s,c,t} = \mathcal{I}_{TS}(B; B', S)_{s,c,t} \quad \text{if } B' \in \mathscr{B}$$

Example 3 Take the same example as the one in Example 1. The first action B' = {warn_diff_pres := **true**} is an assignment, which is a basic action. So B' is merged to B, and the remaining actions will continue applying the rules.

$B = \phi$
B' = {warn_diff_pres := **true**}
S = {door_info := locked;
if (on_ground) door_locked := **false**; **else** door_locked := **true**; **end if**;}

4. If the action to be processed is not a basic action, $S' \notin \mathscr{B}$, separate the actions B, S' and the remaining actions S in three transitions, and continue the separation for the last transition with the actions S. The non basic action S' will be decomposed applying the rules that will be given in section 6.3.1.2.

$$\mathcal{I}_{TS}(B, S'; S)_{s,c,t} = \left(\begin{array}{l} s - [c] \rightarrow s' \{B\}; \\ s' - [] \rightarrow s'' \{S'\}; \\ \mathcal{I}_{TS}(\phi, S)_{s'', \mathbf{true}, t} \{S\} \end{array} \right), \quad \text{if } S' \notin \mathscr{B}$$

Example 4 The conditional action S' = {**if** (on_ground) door_locked := **false**; **else** door_locked := **true**; **end if**;} is not a basic action. Since B is not empty, and there is no action after S' ($S = \phi$), we will introduce an intermediate state s' for a new transition with the action B. From state s' to the destination state t, a transition with the conditional action S' is generated. This transition will continue be decomposed into basic actions using the rules in the next subsection.

B = {warn_diff_pres := **true**; door_info := locked;}
S' = {**if** (on_ground) door_locked := **false**; **else** door_locked := **true**; **end if**;}
$S = \phi$

Applying these rules, a transition T is separated into such new transitions that for each new transition, either the actions are all basic actions, or the action is a single non-basic action. In the next subsection, the non-basic actions will be decomposed into basic actions.

6.3.1.2 Decomposition

An intermediate function $\mathcal{I}_{TD}$ is used for decomposing a composite action S ($S \in \mathscr{S}$ and $S \notin \mathscr{B}$) of a transition T, which is a condition or a loop or timing actions (**computation**() and **delay**()). The application of this function is denoted $\mathcal{I}_{TD}(S)_{s,\ c,\ t}$, for S the composite action and $s,\ c,\ t$ the source state, guard and exit state (respectively) of the transition. In that case: $\mathcal{I}_{TS}(\phi, S)_{s,\ c,\ t} = \mathcal{I}_{TD}(S)_{s,\ c,\ t}$. It returns an intermediate transition $U \in \mathscr{U}$ (which may be a composition of transitions) where the actions in U are basic actions.

1. **Condition**

   A condition statement **if** x **then** S_1 **else** S_2 evaluates S_1 with the condition (**when** x) and evaluates S_2 with condition (**when not** x).

$$\mathcal{I}_{TD}(\mathbf{if}\ x\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2)_{s,c,t} = \left(\begin{array}{l} s - [c\ \mathbf{and}\ x] \rightarrow t \{S_1\}; \\ s - [c\ \mathbf{and\ not}\ x] \rightarrow t \{S_2\} \end{array} \right)$$

   The two generated transitions need to be processed recursively. If S_1 or S_2 still contains non-basic actions, continue applying the rules (separation and decomposition), until only basic actions exist. The same for the other composite actions.

Example 5 Figure 6.5 (upper) shows a transition with a condition action attached. Each branch is decomposed into a new transition (Figure 6.5 (bottom)): starts from the source state *s*, guarded with the condition (*on_ground* or **not** *on_ground*), attached with the corresponding actions in the branch, and enters in the destination state *t*.

```
s-[]->t { if (on_ground)
          door_locked := false;
          else door_locked := true;
          end if; }
```

```
s-[on on_ground]->t {door_locked := false; };
s-[on not on_ground]->t {door_locked := true; };
```

Figure 6.5: Conditional action decomposition example

2. **Loop**

A loop statement **for** (*x* **in** *X*) {*S*} allows iterations over finite integer ranges or over unparameterized enumerated types, which are specified by a data classifier.

Integer range For an integer range, *i* **in** *M*..*N*, which means that *M* and *N* are two integers and $M < N$, the loop statement can be refined as:

$$\mathcal{I}_{TD}(\textbf{for } (i \textbf{ in } M..N)\ \{S\})_{s,c,t} = \begin{pmatrix} s - [c] \rightarrow s'\ \{i := M\}; \\ s' - [\textbf{on } (i \leq N)] \rightarrow s'\{S\,; i := i + 1\}; \\ s' - [\textbf{on } (i > N)] \rightarrow t\ \{\ \} \end{pmatrix}$$

Example 6

```
s-[]->t {for (i in 1..3)    |   s-[]->s1 {i:=1};
            { r:= r+1;}}    |   s1-[on (i<=3)]->s1 {r:= r+1; i:= i+1};
                            |   s1-[on (i>3)]->t {}
```

The original transition (left) has one loop action attached. The corresponding decomposed transitions are represented on the right. First, initialize the iterator *i*. The second transition includes the actions ($r := r + 1$) in the body of the loop and increases the iterator. The last transition controls the automaton to enter in the destination state *t*, if the loop condition is no longer satisfied.

Enumeration range For an iteration over unparameterized enumeration, *x* **in** *X*, in which *X* is a data classifier of enumerated type $X = \{x_1, x_2, ..., x_n\}$, the loop statement can be translated as:

$$\mathcal{I}_{TD}(\textbf{for } (x \textbf{ in } X)\ \{S\})_{s,c,t} = \left(\begin{array}{l} s - [c] \rightarrow s'\{i := 1\}; \\ s' - [\textbf{on } (i \leq n)] \rightarrow s'\ \{x = x_i; S; i := i + 1\}; \\ s' - [\textbf{on } (i > n)] \rightarrow t\ \{\} \end{array} \right)$$

where $X = \{x_1, x_2, ..., x_n\}$ and $|X| = n$

3. **Computation(m,n)**

   Computation(m, n) expresses the use of CPU for a possibly non-deterministic period of time between m and n. For this non-deterministic choice, we introduce a function **random**(m, n) to choose a random period r $(m \leq r \leq n)$ while translating. For simple, we consider here a conversion of the time unit into integers.

$$\mathcal{I}_{TD}(\textbf{computation } (m, n))_{s,c,t} = \left(\begin{array}{l} s - [c] \rightarrow s'\ \{i := 1\} \\ s' - [\textbf{on } (i < r)] \rightarrow s'\ \{i := i + 1\} \\ s' - [\textbf{on } (i = r)] \rightarrow t \end{array} \right)$$

where $r =$ **random**(m, n)

4. **Delay(m,n)**

   The difference between **delay** and **computation** is that **computation** consumes CPU, while **delay** does not, which means that, when a thread delays some time interval, other threads can execute during this time.

   The interpretation of **delay** is more complicated, in reason of rescheduling request. The **delay** service request suspends execution of the requesting thread for an amount of elapsed time. The current thread state is set to WAITING, and a time counter is initialized with a duration which is a random value between the delay lower bound m and upper bound n. A rescheduling is performed. The scheduler will select the highest priority process in the READY state. When the counter is expired, the thread will return to the READY state, and a rescheduling is required. A simple example is given in Figure 6.6.

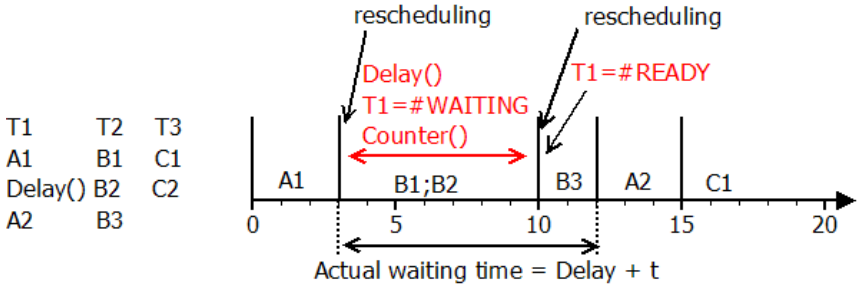


Figure 6.6: Delay

Let us consider a transition $s - [c] \rightarrow t$ {**delay**(m, n)} that calls a **delay**. In the resulting Signal translation, the corresponding transition (at the instant at which the source state is s and the guard c is true) will be viewed as the activation clock of a **delay** system call. This may be represented by an output signal `delay`, that encodes the clock at which control has to be returned to the scheduler, due to **delay** system calls.

In this solution, some specific input and output signals, called *control signals*, are added. These signals include:

input signal `tick` : This signal specifies the clock at which the process is running. The final Signal equations are sampled by the signal `tick`.

output signals corresponding to system calls: For example, a signal `delay` is produced corresponding to the clock of the **delay**, and signals `dmin, dmax` correspond to the lower and upper bound of the delay. If these signals are not used in the process, their clock is the empty clock.

$$\mathcal{I}_{TD}(\textbf{delay}\ (m,n))_{s,c,t} \ = \ s - [c] \rightarrow t\ \{delay := c;\ dmin := m;\ dmax := n\}$$

In [101, 61], a translation of C/SystemC and FairThreads system calls in Signal is presented. Our approach is similar to these works.

Using the rules presented, the non-basic actions can now be decomposed into basic actions. Next step, the basic actions will be interpreted into SSA forms.

6.3.2 Step 2: basic actions to SSA form actions

In SSA form, a variable x is defined only once within a given sample of time or instant. New versions of x are used to uniquely index each assignment of x. Each variable in the SSA form program appears only once on the left hand side of an assignment.

Since the basic actions are recursively a sequence of three types of actions: assignment, message reception and message emission, in this step, only these types of actions (resp. transitions) need to be interpreted. The interpretation in this step is addressed into two stages: first, the interpretation of the basic actions in SSA form; second, the processing of sequences of basic transitions in SSA form.

The function $\mathcal{I}_U$ is used to represent the interpretation of intermediate transition $U \in \mathscr{U}$ with action $B \in \mathscr{B}$, to be represented in SSA form transition $W \in \mathscr{W}$ with action $A \in \mathscr{A}$. In this step, an intermediate transition $U = s - [c] \rightarrow t$ {B} (from source state s to destination state t, guarded with condition c, and attached with the basic actions B) will be interpreted into SSA form transition $W \in \mathscr{W}$, in which the actions A are all in SSA form.

$$\mathcal{I}_U : U \rightarrow W \quad \text{where} \quad \begin{array}{l} U \in \mathscr{U} \\ W \in \mathscr{W} \end{array}$$

We use an environment E to associate each variable $x \in X$ with its definitions (new versions of x), $E : X \to 2^X$. The environment of x is noted $E(x)$, and the domain of E is noted $\mathcal{V}(E)$.

Example 8 A variable x is renamed with x_1, x_2, x_3, and variable y is renamed with y_1, y_2. The environment E associates the variable x, y to their corresponding definitions.

$$
\begin{aligned}
E : & \begin{pmatrix} x & \mapsto & (x_1, x_2, x_3) \\ y & \mapsto & (y_1, y_2) \end{pmatrix} \\
E(x) &= (x_1, x_2, x_3) \\
E(y) &= (y_1, y_2) \\
\mathcal{V}(E) &= (x, y)
\end{aligned}
$$

We write $use_E(x)$ for the expression that returns the current definition of the variable x defined in environment E. If x is defined in the domain of E, $x \in \mathcal{V}(E)$, it will return the current version of x, otherwise, x is not defined in the current environment, then the previous instant of x will be returned. The notation $new_E(x)$ is used to return a new version x' of variable x, which has not been defined in environment E.

$$
\begin{aligned}
use_E(x) &= \begin{cases} x_n & \text{if } x \in \mathcal{V}(E),\ E(x) = (x_1, \ldots, x_n) \\ x\$ & \text{otherwise} \end{cases} \\
new_E(x) &= x' \quad \text{where } x' \notin \mathcal{V}(E)
\end{aligned}
$$

The following two subsections present the two stages, respectively: interpret the basic actions to SSA form, and interpret the basic transitions to SSA form.

6.3.2.1 Interpretation of basic actions in SSA form

The notation $\mathcal{I}_{UA}(B)^E = A^F$ is used to note the interpretation of the basic action $B \in \mathscr{B}$ defined in environment E to SSA form action $A \in \mathscr{A}$ with an updated environment F.

1. For an assignment basic action $x := f(y, z)$ in an environment E, the new action will take its SSA form assignment: x is renamed with a new version, and y, z use their current version in E. F is an updated environment.

$$
\begin{aligned}
\mathcal{I}_{UA}(x := f(y,z))^E = (a := f(b,c))^F \quad \text{where} \quad & a = new_E(x) \\
& b = use_E(y) \\
& c = use_E(z) \\
& F = E \biguplus (x \mapsto a)
\end{aligned}
$$

- The definition of x will take a new version a that has not been defined in the environment E.
- The SSA form of y and z will use their current version defined in environment E, represented by $use_E(y)$ and $use_E(z)$.
- The new environment F stores the new version of x defined in E: $F = E \biguplus (x \mapsto new_E(x))$.

Example 9 An assignment $x := y + z$ with an environment $E = (y \mapsto (y_1, y_2))$ is interpreted in SSA form: the definition of x takes a new version x_1, and y uses its current version y_2. Since no z is defined in environment E, it takes the previous value $z\$$.

$$\mathcal{I}_{UA}(x := f(y,z))^E = (a := f(y_2, z\$))^F \quad \text{where} \quad \begin{aligned} E &= (y \mapsto (y_1, y_2)) \\ a &= new_E(x) \\ F &= (x \mapsto a,\ y \mapsto (y_1, y_2)) \end{aligned}$$

2. For a receiving action $p?(x)$ with an environment E, the interpretation takes its SSA form as follows. The action $p?(x)$ transfers a data received from port p to the variable x. The received data from port p is written to variable x, represented as $x := p$. The representation needs to be conformed to SSA form in the environment E.

$$\mathcal{I}_{UA}(p?(x))^E = (a := b)^F \quad \text{where} \quad \begin{aligned} a &= new_E(x) \\ b &= use_E(p) \\ F &= E \biguplus (x \mapsto a) \end{aligned}$$

Example 10 The interpretation of $p?(x)$ with an environment $E = (p \mapsto p_1)$ returns its SSA form transition:

$$\mathcal{I}_{UA}(p?(x))^E = (a := p_1)^F \quad \text{where} \quad \begin{aligned} a &= new_E(x) \\ E &= (p \mapsto p_1) \\ F &= (p \mapsto p_1,\ x \mapsto a) \end{aligned}$$

3. The sending message action $p!(x)$ writes data x to event or event data port p, and calls the *Send* service. A new unique version p' of p ($\{p \mapsto p'\}$) is added to update

the original environment E, $F = E \biguplus \{p \mapsto p'\}$. This action can be considered as an assignment of p with the value x. So the interpretation will take the SSA form of the assignment $p := x$. The definition of p will be a new version p', which is not in the domain of E. The current definition of x defined in environment E ($use_E(x)$), is used for the assignment of p.

$$\mathcal{I}_{UA}(p!(x))^E = (a := b)^F \quad \text{where} \quad \begin{array}{l} a = new_E(p) \\ b = use_E(x) \\ F = E \biguplus (p \mapsto a) \end{array}$$

Example 11 An action $p!(x)$ with an environment $E = (x \mapsto (x_1, x_2, x_3))$ is interpreted as follows. The port p takes a new version, and stores it in the new environment F, which takes the environment E and the new version of p ($\{p \mapsto a\}$) as a plus.

$$\mathcal{I}_{UA}(p!(x))^E = (a := x_3)^F \quad \text{where} \quad \begin{array}{l} a = new_E(p) \\ E = (x \mapsto (x_1, x_2, x_3)) \\ F = (x \mapsto (x_1, x_2, x_3),\ p \mapsto a) \end{array}$$

4. For a sequence of basic actions $B_1; B_2$, the interpretation is the composition of the interpretations of B_1 and B_2. Here F is the environment that merges E and the new updates introduced by B_1 (noted $E(B_1)$).

$$\mathcal{I}_{UA}(B_1; B_2)^E = A_1^F; \mathcal{I}_{UA}(B_2)^F \quad \text{where} \quad \begin{array}{l} \mathcal{I}_{UA}(B_1)^E = A_1^F \\ F = E \biguplus E(B_1) \end{array}$$

6.3.2.2 Interpretation of basic transitions in SSA form

Normally, the basic actions attached in the same transition can be performed in the same instant, except for the sending message action $p!(x)$ (refer to Section 6.2.3). Each time instant, the port p can only send out one message and writes data to output port once. Therefore, in the interpretation of a transition, if there is a second (or third, ...) occurrence of a $p!(x)$, it has to be separated in two SSA form transitions.

We use an intermediate recursive function $\mathcal{I}_{UT}$ to make explicit the decomposition of the basic actions attached to a basic transition into successive actions in time. The application of this function in an environment E to a basic transition $s - [c] \rightarrow t \{B_1; B_2\}$ is denoted $\mathcal{I}_{UT}(B_1, B_2)^E_{s,c,t}$, where B_1 consists of the actions from the beginning that can

be executed in the same instant, and B_2 includes the remaining actions. The function $\mathcal{I}_{UT}$ uses the function $\mathcal{I}_{UA}$ defined in the previous subsection for basic actions.

1. For a transition $U = s - [c] \rightarrow t\ \{B\}$, at the very beginning of the interpretation, B_1 is empty, which is represented by ϕ, and the environment E is set to be empty. Then:

$$\mathcal{I}_U(s - [c] \rightarrow t\ \{B\}) = \mathcal{I}_{UT}(\phi, B)^{\phi}_{s,c,t}$$

2. If all the actions can be executed in the same instant, the resulting transition will the a transition attached with the actions represented in SSA form.

$$\mathcal{I}_{UT}(B, \phi)^{E}_{s,c,t} = s - [c] \rightarrow t\ \{\mathcal{I}_{UA}(B)^{E}\}$$

3. An assignment $x := f(y, z)$ or receiving message action $p?(x)$ can be merged to the previous basic action B_1.

$$\mathcal{I}_{UT}(B_1,\ B'; B_2)^{E}_{s,c,t} = \mathcal{I}_{UT}(B_1; B',\ B_2)^{E}_{s,c,t} \quad \text{if } B' = x := f(y, z) \quad | \quad p?(x)$$

4. For a transition $U = s - [c] \rightarrow t\ \{B_1; p!(x); B_2\}$, if p has not been defined in this transition, $p!(x)$ can be merged to the previous basic action set B_1. The notation $\mathcal{V}(U)$ notes the domain of variables that are defined in transition U.

$$\mathcal{I}_{UT}(B_1,\ p!(x); B_2)^{E}_{s,c,t} = \mathcal{I}_{UT}(B_1; p!(x),\ B_2)^{E}_{s,c,t} \quad \text{if } p \notin \mathcal{V}(U)$$

Example 12 In a transition $U = s - [c] \rightarrow t\ \{x := 1; p!(x); p?(x);\}$, no p has been defined before $p!(x)$. Therefore, $p!(x)$ can be merged to the previous actions B_1.

$$B_1 = \{x := 1;\}$$
$$B_2 = p?(x);$$
$$\mathcal{I}_{UT}(x := 1,\ p!(x); p?(x);)^{E}_{s,c,t} = \mathcal{I}_{UT}(x := 1; p!(x),\ p?(x);)^{E}_{s,c,t}$$

Otherwise, there has been already a definition of p in this transition, $p \in \mathcal{V}(U)$, since in each instant, the port p can only call the *Send* service once, so this $p!(x)$ will be separated from B_1. The transition is separated.

$$\mathcal{I}_{UT}(B_1,\ p!(x); B_2)^{E}_{s,c,t} = s - [c] \rightarrow s'\ \{\mathcal{I}_{UA}(B_1)^{E}\}\ ;\ \mathcal{I}_{UT}(p!(x), B_2)^{F}_{s',c,t} \quad \text{if}\ \ p \in \mathcal{V}(U)$$
$$\text{with}\ \ F = E \biguplus E(B_1)$$

In case of $p \in \mathcal{V}(U)$, the action B_1 is interpreted in SSA form, and an intermediate state s' is introduced for a transition from state s. The remaining actions $p!(x); B_2$ will continue applying the transformation rules, from the intermediate state s' to the destination state t, with an updated environment F which merges E and the new environment introduced by B_1 (noted as $E(B_1)$).

Example 13 In a transition $U = s - [c] \rightarrow t\ \{x := 1; p!(x); p!(x); y := x + z\}$, there is already a p defined in this transition before the second one occurs. The actions in the first part ($x := 1; p!(x);$) will be separated, and form an intermediate transition. The remaining actions ($p!(x);$) will continue the interpretation.

$$B_1 = \{x := 1; p!(x);\}$$
$$B_2 = \{y := x + z;\}$$
$$\mathcal{I}_{UT}(x := 1; p!(x),\ p!(x);\ y := x + z)^{E}_{s,c,t} = \begin{pmatrix} s - [c] \rightarrow s'\ \{\mathcal{I}_{UA}(x := 1; p!(x))^{E}\}; \\ \mathcal{I}_{UT}(p!(x), y := x + z)^{F}_{s',c,t} \end{pmatrix}$$
$$F = E \biguplus E(B_1) = E \biguplus (x \mapsto new_E(x),\ p \mapsto new_E(p))$$

Applying the rules in this step, the basic actions, sequence of assignments and sending/receiving message actions can be represented in SSA form, and the corresponding basic transitions are interpreted in SSA form too. In the next step, these SSA form transitions/actions will be interpreted into Signal.

6.3.3 Step 3: SSA to Signal

From the interpretation of step 2, all the transitions/actions can be represented in SSA form. In this section, the SSA form transitions/actions are interpreted in Signal processes.

6.3.3.1 SSA form action interpretation

The interpretation $\mathcal{I}_A(A)^g = P$ of an SSA form action A takes as parameter the guard g that leads to it, and returns a Signal equation P.

The only action in SSA form is the assignment, or sequence of assignments. Therefore, the interpretation in this step will only take the assignment action and the sequence in consideration.

- For a single assignment $x := f(y, z)$, the signals to be used (on the right side of the assignment) will be sampled by the guard g: **when** g. A partial definition ::= is used

here since there may be other definitions of x with other guards. A **cell** is introduced for the operands to make them available at their minimal common clock.

$$\mathcal{I}_A(x := f(y,z))^g \;=\; (x ::= f(y \textbf{ cell } h,\ z \textbf{ cell } h) \textbf{ when } g \mid h := y\hat{+}z)/h$$

- For a sequence of assignments, the interpretation can be performed in parallel. The notation "|" is used to note the parallel composition in Signal.

$$\mathcal{I}_A(A_1; A_2)^g \;=\; \mathcal{I}_A(A_1)^g \mid \mathcal{I}_A(A_1)^g$$

Next step, the transitions will be interpreted in Signal.

6.3.3.2 SSA form transition interpretation

The notation $\mathcal{I}_W(W)^{st,nst}$ is used to represent the interpretation of the SSA form transition $W \in \mathscr{W}$ to a Signal process. It is parameterized by variables st, nst representing respectively the current state and next state of the transition system.

$$\mathcal{I}_W(W_1; W_2)^{st,nst} = \mathcal{I}_W(W_1)^{st,nst} \mid \mathcal{I}_W(W_2)^{st,nst}$$

$$\mathcal{I}_W(W)^{st,nst} \;=\; (P \mid nst ::= t \textbf{ when } g) \quad \text{where} \quad \begin{array}{l} W = s - [c] \rightarrow t\ \{A\} \\ g = \textbf{when } (st = s) \textbf{ when } c \\ P = \mathcal{I}_A(A)^g \end{array}$$

The equation $nst ::= t$ **when** g expresses that when the guard g is true, which means when the current state st is s and when the condition c is satisfied, the next state nst of the transition will be t. When a transition completes, the exit state becomes the current state.

Example 14 A transition W, from state s to t, guarded by c, attached with an action $A \;=\; x := y + z$, is interpreted into the following Signal process. The assignment of x is sampled by the guard c and the current state variable st. The signal nst represents the next state: when the current state st is in state s, and the guard c is satisfied, the next state will be t.

$$\mathcal{I}_W(s - [c] \rightarrow t\ \{A\})^{st,nst} \;=\; (P \mid nst ::= t \textbf{ when } g)$$
where $A \;=\; x := y + z$
$P \;= \mathcal{I}_A(A)^{\textbf{when } g} \;= (x ::= (y \textbf{ cell } h) + (z \textbf{ cell } h) \textbf{ when } g \mid h := y\hat{+}z)/h$
$g = \textbf{when } (st = s) \textbf{ when } c$

6.3.4 Global interpretation

Finally, following the three steps' interpretation, the global transformation $\mathcal{I}$ of a transition system $T = (T_1; \ldots; T_n)^{s_0,\ tick}$ has to take into account the initial state s_0 of the transition system and the clock, denoted *tick*, representing the instants at which the considered thread is active—this clock is defined by the scheduler. The input signal *tick*, which is a *control signal* of type `event`, connects the operating system to the thread to notify it that it is selected to be actually running.

$$\mathcal{I}(T_1; T_2)^{s_0,\ tick} = \mathcal{I}(T_1)^{s_0,\ tick} \mid \mathcal{I}(T_2)^{s_0,\ tick}$$

$$\mathcal{I}(T)^{s_0,\ tick} = (P \mid st := nst\ \$\ \textbf{init}\ s_0 \mid st\ \hat{}\ = tick)\ /st, nst \quad \text{where} \quad P = \mathcal{I}_W(W)^{st,nst}$$

$$W = \mathcal{I}_U(U)$$

$$U = \mathcal{I}_T(T)$$

In the next section, a case study will be presented to help the explanation of the interpretation steps.

6.4 Case study

Figure 6.7 is a graphical representation of the example previously shown in Figure 6.1, which is a behavior specification of a *door_handler* thread from a SDSCS (Simplified Doors and Slides Control System). There is only one state in this automaton: s_0, which acts as both **initial** and **complete** state. When the source state is s_0 and the guard ((dps $>$ 3) **and** handle) is satisfied, then the first transition is triggered, so that the two assignments and the condition action are executed (see Figure 6.1). After the completion of the computation, the transition enters state s_0 again. If the guard condition **not**(dps $>$ 3) **and** handle is true, then the second transition will be executed.

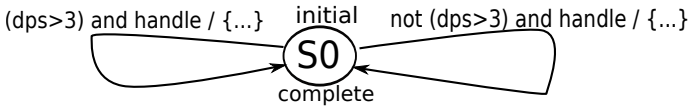


Figure 6.7: Automata of the door_handle thread

Step 1: $T \overset{\mathcal{I}_T}{\mapsto} U$**.** From original actions/ transitions to basic actions/ transitions.

The intermediate generated code of the basic transitions/actions for the *door_handler* thread is depicted below. It contains a transformation of the transitions as well as its attached actions. The interpretation introduces two intermediate states: *STATE_0* and *STATE_1*.

```
annex behavior_specification{**
  states
    s0 : initial complete state;
```

```
  STATE_0 : state;
  STATE_1 : state;
transitions
  s0 -[ on (dps>3) and handle ] -> STATE_0 {
      warn_diff_pres := true;
      door_info := locked; };
  STATE_0 -[ on on_ground ] -> s0 {
      door_locked := false; };
  STATE_0 -[ on not on_ground ] -> s0 {
      door_locked := true; };
  s0 -[ on not (dps>3) and handle ] -> STATE_1 {
      warn_diff_pres := false;
      door_info := locked; };
  STATE_1 -[ on in_flight ] -> s0 {
      door_locked := true; };
  STATE_1 -[ on not in_flight ] -> s0 {
      door_locked := false; };
**}
```

The first three transitions rewrite the first original one. The new transition first enters an intermediate state *STATE_0*, when the guard (**on** dps > 3 **and** handle) is true. The two assignments ({warn_diff_pres := **true**; door_info := locked; }) are interpreted in the same transition. For the condition action, depending on the condition, a transition is introduced for each branch. The two branches both end with state $s0$.

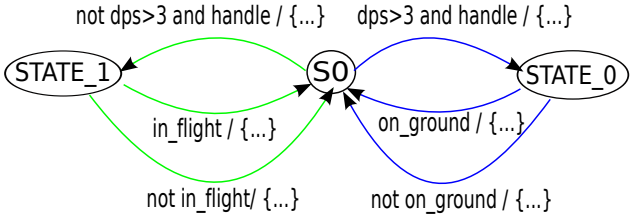


Figure 6.8: Automata for the intermediate Behavior code of the door_handler thread

The automaton including these intermediate transitions is also shown in Figure 6.8. The three transitions on the right represent the first transition of the original specification. From the initial state s_0, when guard (dps > 3 **and** handle) is true, the transition will enter state *STATE_0*, attached with the two assignment actions. The two transitions from *STATE_0* to state s_0 are the two branches of the condition action. The others on the left represent the second one.

Step 2: $U \overset{\mathcal{I}_U}{\mapsto} W$**.** From basic actions/ transitions to SSA form actions/ transitions.

In this example, the SSA form actions and transitions are the same as depicted in step one, since all the variables are already uniquely defined in each of the transitions.

Step 3: $W \overset{\mathcal{I}_W}{\mapsto} P$**.** From SSA form actions/ transitions to Signal process.

Each transition will then be interpreted into Signal equations. The generated Signal code is shown in Figure 6.9. Two state variables (st, nst) are added: st represents the

```
process door_handler =
(? integer dps;
   boolean handle, locked;
   event tick;
 ! boolean warn_diff_pres, door_info;
   event delay;
   dreal  dmin, dmax;)
(| warn_diff_pres := true when (st=s0) when ((dps>3) and handle)
             default false when (st=s0) when ((not (dps>3)) and handle)
 | door_info := locked when (st=s0) when ((dps>3) and handle)
       default locked when (st=s0) when ((not (dps>3)) and handle)
 | door_locked := false when (st=STATE_0) when on_ground
          default true when (st=STATE_0) when not on_ground
          default true when (st=STATE_1) when in_flight
          default false when (st=STATE_1) when not in_flight
 | nst:= STATE_0 when (st=s0) when ((dps>3) and handle)
        default s0 when (st=STATE_0) when on_ground
        default s0 when (st=STATE_0) when not on_ground
        default STATE_1 when (st=s0) when ((not (dps>3)) and handle)
        default s0 when (st=STATE_1) when in_flight
        default s0 when (st=STATE_1) when not in_flight
        default st
 | st:= nst$ init s0
 | st^= tick
 | delay:=^0
 | dmin^= dmax^= delay
 |)
where
state st, nst;
end;
```

Figure 6.9: Signal code for the door handler example

current state, and *nst* represents the next state; *st* takes the previous value of *nst*, and it is initialized by the **initial** state s_0:

$$\text{st} := \text{nst} \$ \textbf{ init } s_0$$

The next state *nst* takes the destination state of each transition, sampled with the current state and the guard. For example, the first transition in step two (the same as in step one in this example) starts from state s_0, ends with state *STATE_0*, guarded by (dps > 3 **and** handle). So the Signal equation for this state transition is defined as:

$$\text{nst} := \text{ STATE\_0 } \textbf{when } (\text{st} = s_0) \textbf{ when } ((\text{dps} > 3) \textbf{ and } \text{handle})$$

Each assignment is sampled with the source state and the guard:

$$\text{warn\_diff\_pres} := \textbf{true when } (\text{st} = s_0) \textbf{ when } ((\text{dps} > 3) \textbf{ and } \text{handle})$$

The variable *warn_diff_pres* has two partial assignments in two different transitions. We use a **default** operation to merge these two definitions:

warn_diff_pres := **true when** (st = s_0) **when** ((dps > 3) **and** handle
default false when (st = s_0) **when** ((**not** (dps > 3)) **and** handle

The state variables *st* and *nst* are declared as type of *state*, which is an enumerated type, including all the state instances:

$$\text{type state} = \textbf{enum}\ (s_0,\ STATE\_0,\ STATE\_1)$$

The program is translated into Signal following the general rules described above. The thread is activated at the instants defined by the scheduler. Depending on the scheduling policy, the scheduler will set one of the enabled threads to be active at each time instant, denoted by an associated tick event. Then simulation code (for example, C code) is generated with the help of the Polychrony toolset. Traces can be added to be able to follow the simulation. Properties can be checked using Sigali [126] which is a Signal companion model-checker. Similar experiments have been described in [61] for Signal code obtained through SSA from C/C++ parallel programs.

6.5 Conclusion

This chapter presents the principle and implementation of an interpretation of the AADL Behavior Annex into a synchronous data-flow and multi-clocked model of computation. This interpretation is based on the use of SSA as an intermediate format. It gives a thorough description of an inductive SSA transformation algorithm across a hierarchy of transitions that produce synchronous equations.

We obtain an effective method for transforming a hierarchy of behavior specifications consisting of transitions and actions into a set of synchronous equations.

Our approach and tools are based on the studies and experimental results on the translation of C/C++ parallel codes into synchronous formalism using SSA transformation [61]. In the ANR project SPACIFY, [154] proposes an approach to model notations of the Synoptic language and to embed them in a suitable model of computation for the purpose of formal verification and code generation. It consists of an inductive SSA transformation algorithm across a hierarchy of blocks that produces synchronous equations.

With Chapter 5 and this chapter, a system specified in AADL can be modeled in a polychronous framework, including both the architecture and behaviors. In next chapter, a distribution of the system will be considered.

Chapter 7

Distributed simulation model generation

Contents

7.1 Distributed code generation in Polychrony 167

7.1.1 Mapping 169

7.1.2 Scheduler 171

7.1.3 Adding communications 173

7.2 An example 175

7.2.1 System description 177

7.2.2 Modeling and Distributing the example in Signal 177

7.3 Conclusion 181

In this chapter, a distribution model will be presented. Distribution will provide the possibility to build complex but powerful systems. The previous two chapters have shown how to model partially asynchronous applications from system-level AADL specifications, this chapter will present how to generate distributed simulation code.

We are interested in the system allocation and communication between them. For this purpose, we address the distribution in three steps: mapping, scheduling and communication. The Signal pragma "RunOn" is used for the partitioning mapping information. MPI is a language independent communication protocol used to program parallel computer. We use MPI to communicate between distributed programs. An example is given to illustrate the distribution application.

7.1 Distributed code generation in Polychrony

Given the combined capabilities of the distributed components, a distributed system [44, 116] can be much larger and more powerful than combinations of stand-alone systems. Polychrony provides general compilation methods particularly for distributed code generation [47, 58].

The purpose of our distribution is, given a centralised program and the distribution specifications, to build the program on different processors. These programs must be able to communicate harmoniously, such that their combined behavior will be functionally equivalent to the behavior of the initial centralised source program.

In Chapter 5 and Chapter 6, we have given the transformation of the AADL components and functional behaviors to Signal. They can be represented in Signal programs. For a complete transformation, we are interested in two specifications. The first one is the Signal program which includes all the computation components and depicts flows of data, translated from the AADL components. The other one is the hardware architecture specification: in the AADL architecture specification, it is clearly defined how the system should be distributed, for example, which process should be executed and scheduled by which processor, and how the processor will periodically or sporadically schedule the processes.

In this chapter, our goal is to obtain automatic distributed code generation from:

1. the software architecture of the application,

2. a representation of the distributed target architecture,

3. a manual mapping of the software modules onto the hardware components (given by an AADL/Signal description).

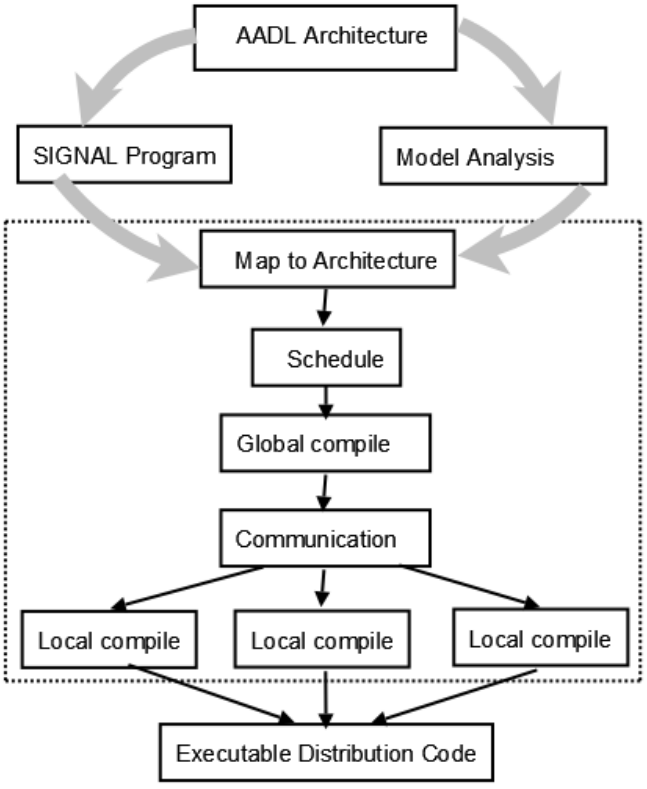


Figure 7.1: Distributed code generation

To automatically distribute a Signal program, Figure 7.1 illustrates the steps of this distribution. First, the specified components are mapped on the target architecture, a scheduler is added for each processor, and the clock synchronizations between the partitioned components are synthesized. Then add communications between these components. And finally generate executable distribution code [123].

7.1.1 Mapping

```
system system1
end system1;

system implementation system1.impl
  subcomponents
    CLIENT: process client;
    SERVERS: process servers;
    BUS1: bus bus1;
    Processor1: processor CPU;
    Processor2: processor CPU;
  connections
    c1: data port CLIENT.x -> SERVERS.x;
    c2: data port SERVERS.y -> CLIENT.y;
  properties
    Actual_Processor_Binding => reference Processor1 applies to CLIENT;
    Actual_Processor_Binding => reference Processor2 applies to SERVERS;
end system1.impl;

process client
  features
    x: out data port;
    y: in data port;
end client;

process servers
  features
    x: in data port;
    y: out data port;
end servers;

bus bus1
end bus1;

processor CPU
end CPU;
```

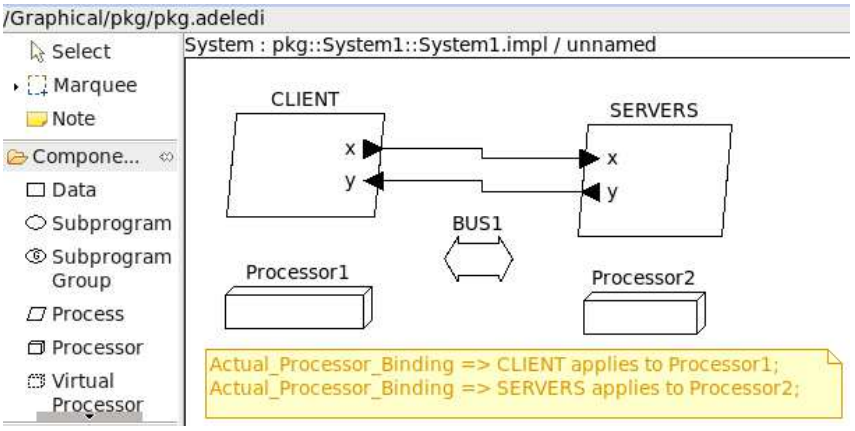


Figure 7.2: AADL distribution specification

In this section, we illustrate the mapping by an example with two processors (Figure 7.2). The system `system1` contains two processes: `CLIENT` and `SERVERS`, and two

processors: `Processor1` and `Processor2`. Each process is distributed on a different processor. From the *properties* definition of the system implementation, we can see that process `CLIENT` is bound to processor `Processor1`, and `SERVERS` is bound to processor `Processor2`.

By analyzing the AADL architecture, the system is composed of two processes. According to Chapter 5, we have two *partitions* translated for the processes from this AADL model.

Then we need to distribute the *partitions* to different machines/processors. Because each *partition* is translated from a process, which is bound to a different processor as specified in the *properties*, the two *partitions* do not share the same processor, so we can assign each *partition* to a different processor.

Each *partition* is paced by its own clock. The Signal pragma "RunOn" [58] in the Polychrony environment is used for partitioning information: when a partitioning based on the use of the pragma "RunOn" is applied to an application, the global application is partitioned according to the different values of the pragma so as to obtain sub-models. The tree of clocks (the root of the tree represents the most frequent clock) and the interface of these sub-models may be completed in such a way that they represent endochronous processes [153] (an endochronous process is mostly insensitive to internal and external propagation delays).

The mapping principles are the following:

1. Description of the software data flow program, using the Signal language (noted as program *P*). This step is already done following Chapter 5.

2. Description of the target architecture. This is specified in the AADL model, especially in the *properties*.

3. Mapping of the software program on the target architecture. So, the program *P* is rewritten as `(|P1|...|Pn|)`, where n is the number of "processors". This step is only a syntactic restructuration of the original program. The Signal process *Pi* is annotated with a pragma "RunOn i". This pragma will be used for partitioning.

Here we give a simple example for distributing two *partitions* run on different processors, of which the AADL model is described in Figure 7.2:

```
process system1 =
(? event tick; )
pragmas
  target "MPI"
  RunOn {e1} "1"
  RunOn {e2} "2"
end pragmas
(|e1::(| x:= CLIENT(y,ptick1)
       |(ptick1):= SCHEDULE1(tick)
       |)
 |e2::(| y:= SERVERS(x,ptick2)
       |(ptick2):= SCHEDULE2(tick)
       |)
 |)
where
```

```
   label e1,e2;
   Message_type x,y;
   event ptick1, ptick2;
end;
```

The Signal processes `CLIENT()` and `SERVERS()` are implemented as explained in Chapter 5. We put the `SCHEDULE()` implementation details off until later refinement stages, and focus on distribution in this step: the two *partitions* are assigned to two different labels: `CLIENT()` with label $e1$ and `SERVERS()` with label $e2$. The pragma **RunOn** is used: `RunOn {e1} "1"` and `RunOn {e2} "2"`. So that, the programs labeled with $e1$ (resp. $e2$) will run on processor 1 (resp. 2). Therefore, the two *partitions* are distributed to two different processors.

7.1.2 Scheduler

The purpose of scheduling is to be able to structure the code into pieces of sequential code, and guarantee the ordered execution of these codes. To express the mapping of execution tasks to platform processors, we will first recall the scheduler modeling in our transformation.

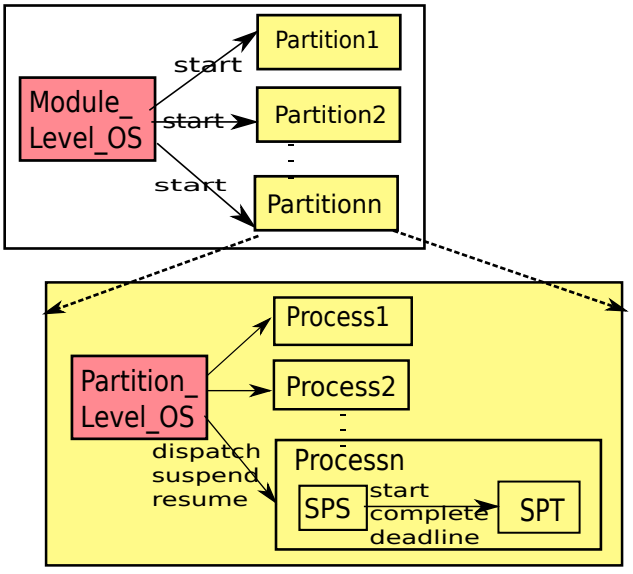


Figure 7.3: Scheduling

The scheduler selects enabled tasks for execution according to the scheduling policy. In APEX-ARINC model, the *Module_Level_OS* is the scheduler for scheduling the *partitions* within the same module (see Figure 7.3). Scheduling *partitions* is strictly deterministic over time in the ARINC653 standard. Based upon the configuration of partitions within the core module, overall resource requirements/availability and specific partition requirements, a time-based activation schedule is generated that identifies the partition windows allocated to the individual *partitions*.

Each *partition* is then scheduled according to its respective partition window. The schedule is fixed for the particular configuration of partitions on the processor. Since *partitions* have no priority, the scheduling algorithm is predetermined, repetitive with a

fixed periodicity, and is configurable by the system integrator only. At least one *partition* window is allocated to each *partition* during each cycle.

Within the *partition*, there is a *Partition_Level_OS* (in Figure 7.3, zoom of Partition n) for scheduling the *processes* (translated from the AADL threads), which is already implemented during the transformation phase in Chapter 5.

In this scheduling step, the principles are the following:

1. Scheduling the *partitions*. In our transformation rules, each processor may apply to several processes, which means that a processor may run several *partitions*. Since the scheduler for the *partitions* running on a processor is deterministic, it is static and fixed according to the configuration tables. We will create a static scheduler for simple, refer to the *Module_Level_OS* in APEX.

   We give a scheduler example below, named $SCHEDULE()$. The processes $INIT()$ and $GET\_ACTIVE\_PARTITION()$ are used for initializing and activating the partitions. Although our mock-up implementation considers only one partition (as it is the case for the example of the previous subsection), they can be implemented for several partitions. The implementation can be written as an external C program.

```
process SCHEDULE =
(? event tick;
 ! integer active_partition;)
(|(| initialize := INIT(tick)
   |)
 |(| active_partition := GET_ACTIVE_PARTITION()
   | b_init := (not initialize)$1 init true
   | active_partition ^= when (not b_init)
   |)
 |)
where
  event initialize;
  boolean b_init;
end;
```

2. Global compiling. This phase makes explicit all the synchronizations of the application, then synthesizes, if it exists, the master clock of the program, and detects synchronization constraints.

   After adding the scheduler to the program, now we can make a global compiling to separate the parts for further development. This phase makes explicit all the synchronizations of the application and detects synchronization constraints.

   The compiler uses *clock calculus* to statically analyze every program statement, identify the structure of the clocks of variables, and schedule the overall computations. If the collection of the statements as a whole contains clock constraints violation, then synchronization constraints exist.

   The distributed program is automatically generated in the Polychrony environment, applying the *distribution* compiling. Partial distributed code of the example shown in previous subsection is given here:

```
process system1_1 =   |    process system1_2 =
(? Mess_type y;       |    (? Mess_type x;
   event ptick;       |       event ptick;
 ! Mess_type x;)      |     ! Mess_type x;)
pragmas               |    pragmas
  RunOn "1"           |      RunOn "2"
end pragmas           |    end pragmas
  (|...|);            |      (|...|);
```

The program is separated into two parts. Each part is made of the code pieces that have the same label. The part is labeled as XX_i (XX represents the name of the original Signal program, "system1" in this example, and *i* is the name of the RunOn processor), and the internal inputs/outputs from the other part are automatically added. In our example, the two parts *system1_1* and *system1_2* are generated after the global compiling, message x is transfered from *system1_1* to *system1_2*, and *system1_1* will receive message y from *system1_2*.

Compared with the original program, a rewriting has been performed for the separated parts. All the computation properties are preserved, and a scheduler is added for each of the processors.

7.1.3 Adding communications

A distributed program consists of a set of processes interacting with the environment and communicating between themselves. Reasons for the communication are to send data or signal to another process, to synchronize with other processes, or to request an action from another process.

A common distributed processing environment is constituted by several parts that are interconnected forming a network and they communicate and coordinate their actions by passing messages. Usually a real-time distributed executive provides services such as time and resource management, allocation and scheduling of processes and inter-processes communication and synchronization. Among these services, one of the most important is the communication support. We emphasize this issue because of the peculiar role of inter-processor communications in distributed memory multi-processors.

In distributed computing, a common assumption is that, when a task sends a message to some other task, it should not need to know where this task is situated, making the message communication transparent. Some commercial operating systems (e.g. VAX/ELN) provide a distributed kernel, which directly supports this transparency in the message communication. If this property is not available then a Distributed Task Manager (DTM) is usually developed to provide this transparency. This middleware is a layer of software that stands above each operating system on each node. Therefore some form of communication and synchronization mechanism which guarantees that the semantics of synchronous communication will be preserved needs to be added.

There are two basic solutions for communications: shared variables and message passing.

Shared variables do not allow synchronization between parallel processes, unless some complex mechanism is built on top of them. Moreover, they make formal verification harder.

The other solution is message passing. Message passing in distributed systems can be synchronous or asynchronous. In our implementation, we use MPI (Message Passing Interface) [159]. MPI is a language-independent communication protocol used to program parallel computers. It is a message passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation.

MPI technology tends to provide an efficient and portable standard for message passing communication programs used in distributed memory and parallel computing. The target platforms are systems which consist of massively parallel computing (the programmer is responsible for identifying the parallelism) such as workstation clusters or heterogeneous networks. There are currently several MPI implementations such as MPI/Pro, IBM MPI, Open MPI [16] and LAM [13]. These implementations provide different communication modes such as asynchronous communication, virtual topologies and efficient message buffer management.

The problem of timed synchronous communication between two processes, P and Q, can be stated as follows:

- Send. To send a message to process Q, process P executes **MPI_Send** as displayed in Table 7.1. A unique *tag* is assigned to the message.

int **MPI_Send** (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
Input Parameters **buf** initial address of sending buffer (choice) **count** number of elements in sending buffer (non-negative integer) **datatype** data type of each sending buffer element (handle) **dest** rank of destination (integer) **tag** message tag (integer) **comm** communicator (handle)

Table 7.1: Synopsis of MPI_Send

- Receive. Meanwhile, process Q executes a receive command as presented in Table 7.2. The parameters in receive command are almost the same as the ones in send. The *tag* is labeled the same as the one assigned in the send command. The *source* parameter specifies the rank of source.

int **MPI_recv** (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM comm, MPI_STATUS *status)

Table 7.2: Synopsis of MPI_recv

MPI preserves the ordering and the integrity of the messages (for example, by ACK schemes and sequence numbers). This will ensure that values are not mixed up or out of sequence, provided that the *send* actions are executed on one location in the same order as the corresponding *recv* actions in the other location.

Based upon the distribution specification, a unique tag is assigned to each common variable of the program. Figure 7.4 shows the communications in the case of the same example. The message *x* from part *system1_1* is tagged with *tag 0*. On the receiving side, it reserves the same tag. The same for the other message, *y* (with *tag 1*) in the reverse direction.

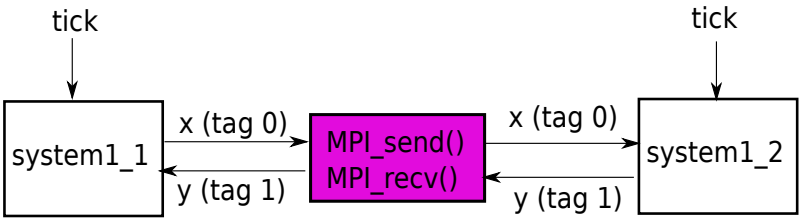


Figure 7.4: MPI Communication

To execute the distributed parts, we need to configure the actual physical machines for their execution. In our example, we configure the hosts as follows: the generated executable bin file *system1_1* (resp. *system1_2*) which is located in DIRECTORY0 (resp. DIRECTORY1) will be executed on node0 (resp. node1). The two node machines are specified by the host address in the "lamhosts" file. The detailed configuration is given in Table 7.3.

lamhosts: 10.0.1.3 %the first machine% 10.0.1.5 %the second machine%
c0 -wd DIRECTORY0 10.0.1.3 % node0 runs on 10.0.1.3% c1 -wd DIRECTORY1 10.0.1.5 % node1 runs on 10.0.1.5%

Table 7.3: Configuration of the example

Finally the parts can be compiled locally and executed on different machines respectively. The programs are run on distributed processors.

7.2 An example

Our approach is a practical application of formal results that have been proved in the polychronous MoC. It has been shown that if `P=(|P1|...|Pn|)` is such that `Pi`'s are endo-isochronous (all of them are endochronous and the pairwise compositions of their restrictions on their common variables are endochronous) [71], assume the deployment is simply performed by using an asynchronous mode of communication between the different processes, then the original semantics of each individual process of the deployed GALS architecture is preserved (the sequence of values are preserved for all flows).

In this section, we illustrate this methodology by a real example, showing how the Signal program is obtained from the AADL specification, and then how the distribution code is generated in the Polychrony framework.

```
system door_management
  features
    cll1 : out data port;
    cll2 : out data port;
end door_management;

system implementation door_management.impl
  subcomponents
    door1 : device Door;
    door2 : device Door;
    doors_process1 : process doors_process;
    doors_process2 : process doors_process;
    cpiom1 : processor CPIOM;
    cpiom2 : processor CPIOM;
    afdx : bus AFDX;
  connections
    c1 : data port door1.closed -> doors_process1.closed1;
    c2 : data port door2.closed -> doors_process1.closed2;
    c3 : data port door1.closed -> doors_process2.closed1;
    c4 : data port door2.closed -> doors_process2.closed2;
    c5 : data port doors_process1.cll -> cll1;
    c6 : data port doors_process2.cll -> cll2;
  properties
    Actual_Processor_Binding => reference cpiom1 applies to doors_process1;
    Actual_Processor_Binding => reference cpiom2 applies to doors_process2;
end door_management.impl;

device Door
  features
    closed : out data port;
end Door;

process doors_process
  features
    closed1 : in data port;
    closed2 : in data port;
    cll     : out data port;
end doors_process;

processor CPIOM
end CPIOM;

bus AFDX
end AFDX;
```

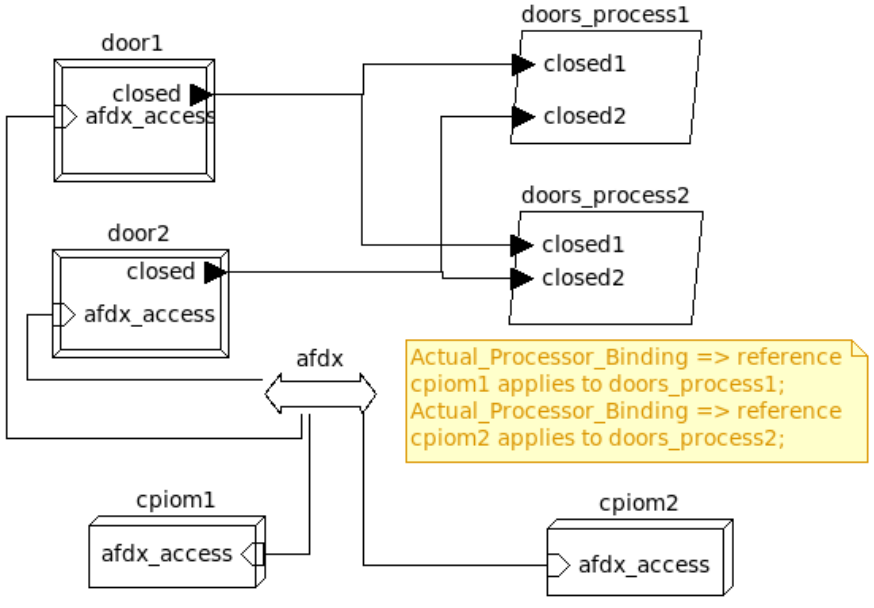


Figure 7.5: An example of process allocation

7.2.1 System description

AADL allows the specification of functionality allocation on hardware. The allocation includes: binding systems/processes/threads on processors; mapping connections on bus. They are specified in the **properties**.

A thread is bound to the processor specified by the **Actual_Processor_Binding** property. Connections between different execution platforms are bound to the bus specified by the **Actual_Connection_Binding** property.

Figure 7.5 is a simplified door management system example. The system is aimed to monitor on the flight status, control the pressurisation and the residual pressure warning lights. This system is present on all the Airbus aircrafts. Here, since we only focus on the distribution specification in this chapter, the detailed functional component implementations are ignored, and the complex door system is simplified as a device.

The basic architecture is shown in Figure 7.5. There are a *door_management* system type and a system implementation declaration. The **subcomponents** subclause declares two **devices**, two **processes**, two **processors** and a **bus** contained in the **system** implementation. The two **devices** are connected to the two **processes** through **data port connections**. These connections are bound to the **bus** *afdx*, seen in the graphical representation of Figure 7.5 (bottom). The two **processes** *doors_process1* and *doors_process2* are applied to different **processors**, which is specified in the **properties**. The process *doors_process1* is bound to processor *cpiom1*, and *doors_process2* is bound to *cpiom2*:

```
Actual_Processor_Binding => reference cpiom1 applies to doors_process1;
Actual_Processor_Binding => reference cpiom2 applies to doors_process2;
```

7.2.2 Modeling and Distributing the example in Signal

In this subsection, we will show the Signal model of the example, and its distribution code following the steps we have presented.

7.2.2.1 Modeling in Signal

The example is modeled in Signal following the transformation rules we have presented in Chapter 5. The **process** is modeled into an ARINC *partition* (a Signal process), which contains a *scheduler* for scheduling the **threads** inside the **process**. The **device** is translated into a Signal process, outside the *partition*, that provides some external environment information. The **bus** is transformed to a Signal process, connecting the communications which are specified bound to it. The whole **system** is a Signal process, including all the processes translated from the subcomponents.

Figure 7.6 shows partial generated Signal code that is transformed from the example in Figure 7.5. The detailed implementations are omitted, only the interface is shown here.

The processes *PART_cpiom1* and *PART_cpiom2* are the two *partitions* translated from the AADL **processes** *doors_process1* and *doors_process2*. Each *partition* contains a scheduler *Scheduler_CPIOM*, which corresponds to the **processor** that it is bound to. The process *BUS_AFDX* represents the **bus** connection. The communication between

the **device** *Device_Door* and the *partition* is through the **bus** *BUS_AFDX* (can be refined using the same bus instead of using two bus instances).

```
module MODULE_door_management =
 process SYSTEM_door_management =
      ( ? event tick;
        ! boolean cll1, cll2;)
     (| PARTITION_INSTANCE_cpiom1 :: cll1 := PART_cpiom1{}(tick, closed1_o, closed2_o)
      | PARTITION_INSTANCE_cpiom2 :: cll2 := PART_cpiom2{}(tick, closed1_o, closed2_o)
      | closed1 := Device_Door()
      | closed2 := Device_Door()
      | closed1_o := BUS_AFDX(closed1)
      | closed2_o := BUS_AFDX(closed2)
      |)
     where
     boolean closed1, closed2, closed1_o, closed2_o;
     label PARTITION_INSTANCE_cpiom1, PARTITION_INSTANCE_cpiom2;

     process BUS_AFDX =
          ( ? busii;
            ! busoo;); %bus%

      process Device_Door =
          ( ! boolean closed;); %device%

     process PART_cpiom1 =
          ( ? event tick;
              boolean closed1, closed2;
            ! boolean cll; )
        (| ... |)
        where
        process Scheduler_CPIOM =
             ( ! event tick;);
        end; %partition 1%

     process PART_cpiom2 =
          ( ? event tick;
              boolean closed1, closed2;
            ! boolean cll;)
        (| ... |)
        where
        process Scheduler_CPIOM =
             ( ! event tick; ) ;
        end; %partition 2%

     end;
end %MODULE_door_management%;
```

Figure 7.6: Signal model of the door management system

Since we are interested in the distribution specification in this chapter, the detailed implementation for each of the processes is not shown here.

7.2.2.2 Distribution

From the AADL specification, we know that the two **processes** are bound to different **processors**. We can distribute the generated *partitions* to different physical machines in Step 1.

Step 1: Mapping. Since each *partition* is bound to a unique **processor**, we can distribute it directly, by applying the "RunOn" pragma.

Figure 7.7 rewrites the program in Figure 7.6. The **processes** *doors_process1* and *doors_process2* are explicitly specified in the AADL specification to run on different **processors**. So the generated programs *PART_cpiom1* and *PART_cpiom2* are assigned respectively a label *e1* and *e2*. The **devices** and **bus** are not declared to be bound to some **processor**. In such a case, we suppose that they run on some other **processor**, labeled with *e3*. So that, the program is distributed to run on three machines.

```
process SYSTEM_door_management =
      ( ? event tick;
        ! boolean cll1, cll2;)
  pragmas
    target "MPI"
    RunOn {e1} "1"
    RunOn {e2} "2"
    RunOn {e3} "3"
  end pragmas
    (|e1 :: ( | cll1 := PART_cpiom1{}(tick, closed1_o, closed2_o, ptick1)
              | ptick1 := SCHEDULE1(tick)
              |)
     |e2 :: ( | cll2 := PART_cpiom2{}(tick, closed1_o, closed2_o, ptick2)
              | ptick2 := SCHEDULE2(tick)
              |)
     |e3 :: ( | closed1 := Device_Door()
              | closed2 := Device_Door()
              | closed1_o := BUS_AFDX(closed1)
              | closed2_o := BUS_AFDX(closed2)
              |)
    |)
  where
    label e1,e2,e3;
    event ptick1, ptick2;
  end;
```

Figure 7.7: Distribution of Signal model

The Signal pragma "RunOn" is applied to this application. The application is partitioned according to the different values of the pragma. For example, the pragma `RunOn {e1}` "1" specifies that the program labeled with *e1* will run on processor "1".

```
process SCHEDULE1 =
  (? event tick;
   ! event ptick;)
  (| activation_partition := GET_ACTIVE_PARTITION(tick)
   | ptick := when ^active_partition
   |)
 where
   integer active_partition;

   process GET_ACTIVE_PARTITION =
     ( ? event tick;
       ! integer active_partition; )
       (| active_partition := 1 when tick
        |);
 end;
```

Figure 7.8: A simple scheduler for one partition

Step 2: Scheduler. The code in Figure 7.7 has introduced a process *SCHEDULE()* both in labels *e1* and *e2*. This scheduler is used for selecting the correct *partition* during

execution within the *module* (a *module* is composed of the **processes** that are bound to the same **processor**).

In our example, each *module* contains only one **process**. So the scheduler *SCHEDULE()* will allocate to the *partition* one partition window during each cycle. Figure 7.8 gives a simple implementation of the scheduler with static scheduling (it is trivial in that case and has to be refined for scheduling several *partitions*).

The process *SCHEDULE1* selects the *partition PART_cpiom1* as the active *partition* each time the scheduler receives a signal *tick*, which represents the main clock of the system. We suppose that the **processor** *cpiom1* has the same frequency as the main clock. Therefore, the activation of the *partition* is synchronized with the main clock. The same for the other scheduler *SCHEDULE2*.

```
process MODULE_door_management_1 =
(? event tick;
   boolean closed1, closed2;
 ! boolean cll)
 pragmas
   RunOn "1"
 end pragmas
 (|...|);
```

Figure 7.9: Distributed signal code

The distributed program is automatically generated, applying the *distribution* compiling. The original program is compiled into three parts. Figure 7.9 shows part of the generated distributed code of the first node, `MODULE_door_management_1`, that runs on the first machine (denoted by "1"). The other two are similar to `MODULE_door_management_1`. The internal inputs and outputs from other distributed parts are automatically added.

lamhosts:
10.0.1.3 %the first machine%
10.0.1.5 %the second machine%
10.0.1.8 %the third machine%
run.config:
c0 -wd /home/user/Polychrony-V4.17/example/MODULE_door_management_1
10.0.1.3 % node0 runs on the first machine%
c1 -wd /home/user/Polychrony-V4.17/example/MODULE_door_management_2
10.0.1.5 % node1 runs on the second machine%
c2 -wd /home/user/Polychrony-V4.17/example/MODULE_door_management_3
10.0.1.8 % node2 runs on the third machine%

Table 7.4: Configuration table

Step 3: Adding communications. The communication will be added automatically, since we declared in the pragma: `target` "`MPI`", for using MPI as message passing communication protocol in this distributed computing. The messages between the distributed

programs are tagged with a unique integer tag. So that the sequence and the integrity of the message are guaranteed.

The final physical configuration is shown in Table 7.4. Each distributed node (program) is configured to a different machine. So that, the distributed programs can be compiled locally, and run on independent physical processors.

7.3 Conclusion

This chapter presents the principles that generate distributed Signal code from the AADL specification. The centralized Signal model can be obtained following Chapter 5. The hardware allocation information is specified in the AADL model. The distribution is based on the **processor** binding. The translated *partitions* that are applied to the same **processor** as described in AADL specification, are distributed to the same physical machine. This distribution is implemented by using the Signal pragma "RunOn". The Message Passing Interface (MPI) is used for the communication between the distributed programs. The application is then explained by a real case of door management system.

As a conclusion, from a complete representation of an application, including its virtual distribution on a target architecture, it is possible to make a global compilation, partitioning, insertion of communication features, and to simulate the application on the considered architecture.

Part IV

Validation

Chapter 8

Validation of GALS systems

Contents

8.1 Formal verification 186

8.1.1 Case study of a dual flight guidance system 187

8.1.2 FGS Modeling in AADL 188

8.1.3 Interpreting the model in Signal 190

8.1.4 Checking safety properties with Sigali 190

8.2 Simulation 193

8.2.1 Case study of a door management system 193

8.2.2 System Modeling in AADL 194

8.2.3 Interpreting the model in Signal 196

8.2.4 Other models and system integration 197

8.2.5 Profiling 199

8.2.6 VCD-based simulation 200

8.3 Conclusion 201

Hardware/software co-design of embedded systems needs to meet restricted time, cost and performance requirements. Many improvements of classical design approaches have been adopted to satisfy these requirements, including parallel development of hardware and software, high-level modeling, early validation, component-based methods, GALS architecture, etc. In this chapter, we briefly show how these improvements are integrated into our work to tackle the problems of time, cost and performance in the system co-design of avionics.

Our design approach is mainly inspired by the following projects: the Ptolemy project [78], MoBIES project [43], SML-Sys modeling frameworks [144], ForSyDe modeling frameworks [127], etc. In these projects, heterogeneous models or components, based on different models of computation are integrated into a system either through agent-based methods or through translating them into certain common formalism. We adopt a similar system-level co-design approach, which is also adopted by Polychrony for the tool demonstration in the framework of CESAR project [36].

The proposed approach consists of high-level and/or heterogeneous modeling, semantic-preserving transformation and integration of models, formal verification, and simulation with regard to performance evaluation. Polychrony is adopted as a common development platform to bridge heterogeneity between modeling, formal verification and simulation. The advantages of this approach is that high-level models and transformations are developed in parallel to reduce time cost; early validation and performance analysis are carried out on high-level models; Polychrony and its associated tools allow the fast validation with no need for additional translations to other formalisms.

Two avionic case studies: simplified flight guidance system and simplified door management system, are used in this chapter to illustrate our approach. Formal verification and simulation, particularly profiling and value change dump (VCD) based demonstration, are also presented with these two case studies.

8.1 Formal verification

In recent years, AADL has been increasingly used for embedded safety-critical systems for demonstration purpose in the avionics domain. The main objective is not only related to architecture exploration and analysis. Formal verification of functional aspects becomes a very interesting validation approach. Validation of AADL models via formal methods has been studied in [132, 67, 164, 139], where AADL specifications are associated with formal semantics so that formal verification is performed.

Our approach is involved in model checking in the framework of Polychrony. AADL specifications are translated into Signal, then the model checking tool associated with Polychrony, called Sigali [126], is used to check safety properties. Sigali adopts *polynomial dynamical systems* (PDS) [125] over $\mathbb{Z}/_{3\mathbb{Z}}$ = {-1, 0, 1}) as the formal model. Signal programs can be transformed into polynomial equations, which are manipulated by Sigali. The three states of a Boolean signal can be encoded as: present $\wedge$ true $\Leftrightarrow$ 1; present $\wedge$ false $\Leftrightarrow$ -1; absent $\Leftrightarrow$ 0. In case of non-Boolean signals, only their clocks are encoded, i.e., their values are ignored.

PDS can be seen as an implicit representation of an automaton. Each set of states can be represented by its associated polynomials. By manipulating polynomial equations, Sigali avoids the enumeration of the state space. In Sigali, a polynomial is represented by a Ternary Decision Diagram (TDD), which is an extension of Binary Decision Diagram (BDD).

Model checking can be carried out on PDS [125] with regard to the following properties: *liveness*, *invariance*, *reachability*, and so on. Liveness implies the system can always evolve. This property can be checked through the Sigali command: `alive(system)`, where `system` is the system to be checked. Invariance indicates that a set of states is invariant when all the recursive reachable states are also in this set. This property can be checked through the Sigali command: `invariant(system, proposition)`. Each state in an automaton is associated with `propositions`. In previous command, `proposition` can be considered as a set of states whose `proposition` is true. When checking reachability, iff the system has always a trajectory towards a state starting from the initial states, this state is considered reachable. This property can be checked through the command: `reachable(system, proposition)`.

8.1.1 Case study of a dual flight guidance system

To illustrate our approach to formal verification, we take a subsystem of a Flight Control System (FCS) [130] as an example, whose main architecture is shown in Figure 8.1. The subsystem is mainly composed of Flight Guidance System (FGS), which has two symmetric and redundant implementations (FGS_L and FGS_R in Figure 8.1). Each implementation is situated at one of the two sides of an aircraft. Each FGS implementation consists of two parts: mode logic and control laws. Mode logic is used to determine the activation of FGS operation mode. And the operation mode is then used to select right control laws, which compare the measured state of the aircraft (position, speed, and attitude) with the desired state in order to generate pitch and roll guidance commands to minimize the difference between the measured and desired state.

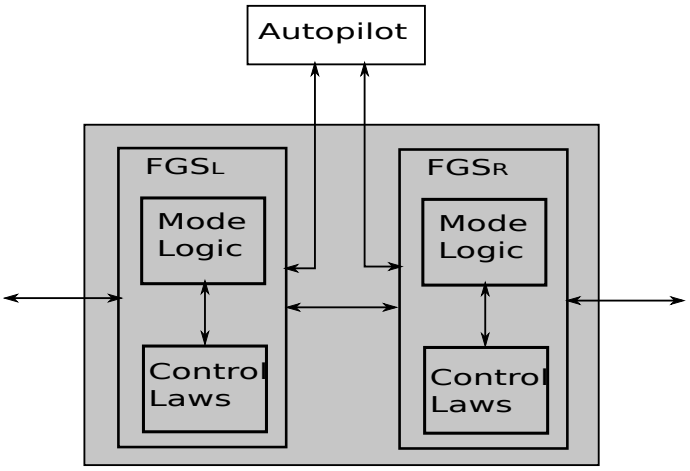


Figure 8.1: The simplified flight control system

FGS_L and FGS_R communicate with each other over a cross-channel bus. These two FGSs work in two modes: *dependent* and *independent*. In normal conditions, only one FGS (the pilot flying side) is active, while the other one operates as a silent, hot spare. This corresponds to *dependent* mode of operation. In this mode, the active FGS provides guidance information to other systems. Moreover, the pilot and copilot can switch which side is the pilot flying side manually. Whereas in some critical conditions, both sides are active and independently generate guidance information. This corresponds to *independent* mode. In this mode, both sets of guidance information are provided to the *Autopilot*, which is in charge to verify if they agree with regard to a predefined tolerance value. In case of agreement, the values are averaged and are taken into account. Otherwise, the situation is sent to the pilots for further decision. A more detailed description of the FGS can be found in [130].

The desired safety requirements for the dual FGS system are presented informally here according to [130]: 1) *At least one FGS is always active*. When the system is on, at least one FGS always provides guidance information. 2) *One and only one side is always identified as pilot flying side*. It ensures to avoid that, in the *dependent* mode of operation, both sides provide guidance to the pilots and/or autopilot at the same time. 3) *Both FGS are active in the independent mode of operation*. This mode is considered as critical phases of flight, during which system safety requirements shall be satisfied.

8.1.2 FGS Modeling in AADL

We only consider a critical part of FGS that deals with the determination of FGS side activation in the formal verification. In addition, the dual FGS system is assumed that neither FGS implementation can fail, that they execute synchronously, and they communicate over a channel that does not lose data.

In the first step of modeling, components, connections, and inputs/outputs of the overall system are specified. Based on the system specification, the modeled system consists of a left and right FGS, and a cross channel bus that connects them (Figure 8.2). System inputs include sensor readings and pilot commands.

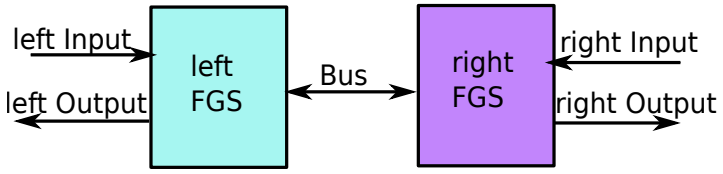


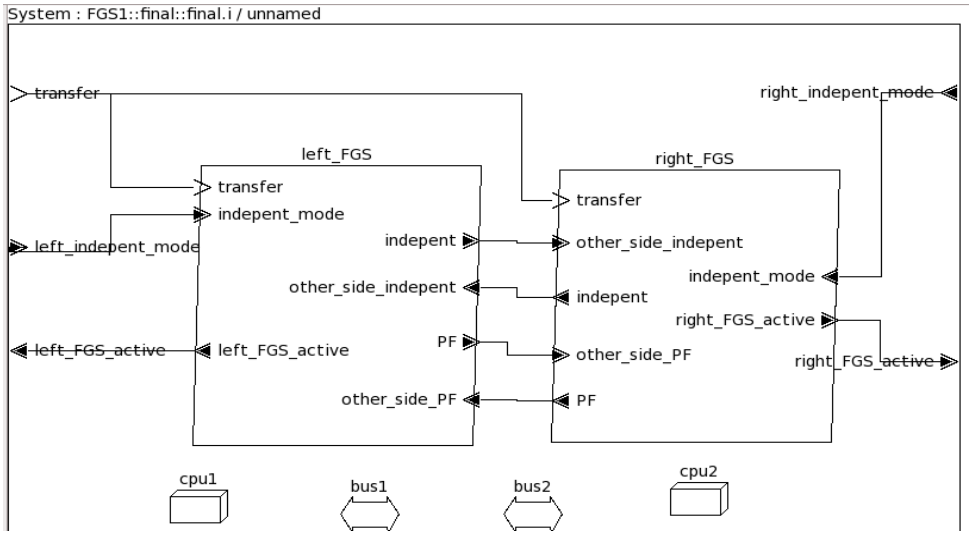
Figure 8.2: Architecture for the FGS system

An AADL model of the simplified system and partial code is shown in Figure 8.3. Each FGS is modeled as a thread executed within a separate protected virtual address space of a process. Each process binds to its own processor so that two FGS work independently. The system inputs are the *transfer*, *left_independent_mode* and *right_independent_mode*. The system outputs are *left_FGS_active* and *right_FGS_active*, which indicate the activation of corresponding side.

The *pilot_flying* and *independent* status of each FGS is communicated with the other via a bus. These two values convey each FGS' determination of whether it is the pilot flying side and whether it is in independent mode. In the synchronous modeling, the bus holds communication data for one step and then send them to their destination.

Partial code of the *thread* of FGS is shown below. The AADL behavior annex is used to specify the computation. The FGS thread transits between *primary* and *backup* states. When it transits to *primary* state, the *pilot_flying* and *active* status is set to be true.

```
thread FGS_thread
  features
    ts: in event data port behavior::boolean;
    inde_mode: in event data port behavior::boolean;
    other_pf: in event data port behavior::boolean;
    other_inde: in event data port behavior::boolean;
    pf: out event data port behavior::boolean;
    tindependent: out event data port behavior::boolean;
    tactive: out event data port behavior::boolean;
  properties
    Period => 100 Ms;
end FGS_thread;

thread implementation FGS_thread.impl
  annex behavior_specification {**
    states
      primary: state;
      backup: state;
```




```
1  system DFGSsys
2    features
3      transfer: in event data port behavior::boolean;
4      L_inde_mode: in event data port behavior::boolean;
5      R_inde_mode: in event data port behavior::boolean;
6      Lactive: out event data port behavior::boolean;
7      Ractive: out event data port behavior::boolean;
8  end DFGSsys;
9
10 system implementation DFGSsys.impl
11   subcomponents
12     cpu1: processor CPU.impl;
13     LFGS: process FGS.impl;
14     cpu2: processor CPU.impl;
15     RFGS: process FGS.impl;
16     bus1: bus net.impl;
17     bus2: bus net.impl;
18   connections
19     c1: event data port transfer -> LFGS.transfer;
20     c2: event data port transfer -> RFGS.transfer;
21     c3: event data port L_inde_mode -> LFGS.inde_mode;
22     c4: event data port R_inde_mode -> RFGS.inde_mode;
23     c5: event data port LFGS.pactive -> Lactive;
24     c6: event data port RFGS.pactive -> Ractive;
25     c7: event data port LFGS.pilot_flying -> RFGS.other_pilot_flying;
26     c8: event data port LFGS.independent -> RFGS.other_independent;
27     c9: event data port RFGS.pilot_flying -> LFGS.other_pilot_flying;
28     c10:event data port RFGS.independent -> LFGS.other_independent;
29   properties
30     Actual_Processor_Binding => reference cpu1 applies to LFGS;
31     Actual_Processor_Binding => reference cpu2 applies to RFGS;
32     Actual_Connection_Binding =>  reference bus1 applies to c7;
33     Actual_Connection_Binding =>  reference bus1 applies to c8;
34     Actual_Connection_Binding =>  reference bus2 applies to c9;
35     Actual_Connection_Binding =>  reference bus2 applies to c10;
36 end DFGSsys.impl;
```

Figure 8.3: AADL model for the ideal FGS system

```
transitions
  primary -[other_pf?(true) ]-> backup
           { pf!(false);  tindependent := inde_mode;};
  backup -[ts?]-> primary { pf!(true); tactive!(true);
                            tindependent := inde_mode;};
```

```
    **};
end FGS_thread.impl;
```

8.1.3 Interpreting the model in Signal

Each FGS *process* is interpreted as an ARINC *partition* (*PART_cpu1, PART_cpu2*) illustrated in Figure 8.4. They communicate with each other through two buses: *bus1* and *bus2*.

```
process DFGS = (? boolean Transfer, R_inde_mode, L_inde_mode;
                ! boolean Ractive, Lactive, RPF, LPF, LIndependent, RIndependent;)
(| (LIndependent,LPF,Lactive) := PART_cpu1{true,false}
                                 (Transfer,ORPF,ORI,L_inde_mode) % LeftFGS %
 | (RIndependent,RPF,Ractive) := PART_cpu2{false,true}
                                 (Transfer,OLPF,OLI,R_inde_mode) % RightFGS%
 | OLI  := bus1{false}(LIndependent)
 | OLPF := bus1{true}(LPF)
 | ORI  := bus2{true}(RIndependent)
 | ORPF := bus2{false}(RPF)
 |)
```

Figure 8.4: The Signal code for the FGS system

In order to make it more clear and more readable, we simplify and refine the generated FGS *thread* Signal code as shown below. Appendix B shows the complete generated Signal code of the FGS system.

```
process ThreadT_tFGS =
 {boolean Ini_PF;boolean Ini_OPF;} % init for pilot flying and other side pilot flying%
 ( ? boolean Transfer,Other_PF,Other_Independent,Inde_mode;
   ! boolean Independent,PF,Active;)
(| ZPF := PF$ init Ini_PF   % previous pilot flying %
 | ZOPF := Other_PF$ init Ini_OPF  % previous other side pilot flying %
 | FC  := Transfer and (not ZPF)  % when transfer and previous pilot flying is not true %
 | PF  := (true when FC)
          default (false when (Other_PF and (not ZOPF)) when not FC)
                          % when other side becomes pilot flying from not pilot flying %
          default ZPF
 | Independent   := Inde_mode   % independent mode %
 | Active   := PF or (Other_Independent and Inde_mode)  % active state %
|)
where
   boolean ZOPF, ZPF, FC;
end;
```

8.1.4 Checking safety properties with Sigali

Polynomial dynamical systems, as symbolic represention of the Sigali-based model checking, are obtained by compiling the Signal programs with the `-z3z` parameter. For instance, the following command is used to translate the generated Signal programs (named `DFGS.sig` file) to polynomial dynamical systems:

```
signal -z3z DFGS.sig
```

The z3z file `DFGS.z3z` is generated as output after the compiling. This file, together with library files, is loaded in the verification system by using the **load** or **read** command. The following commands are used for loading the files for our example:

```
%sigali
*-----------------------------------------------*
* Sigali - version 2.3 (Dec 2005)               *
*-----------------------------------------------*
Sigali: read("DFGS.z3z");
-----------------------------------------------
Sigali: read("Creat_SDP.lib");
-----------------------------------------------
Sigali: read("Bibli.lib");
-----------------------------------------------
```

In the next, previously presented safety properties are formalized with Sigali property specifications, which are mainly based on computational tree logic (CTL). With these commands, Sigali verifies if the properties hold for the overall system in an interactive way:

- **Property 1: At least one FGS is always active.**

  To show this, we first define Property 1 in Sigali as follows (S represents the polynomial dynamical system of Signal program `DFGS.sig` generated by Sigali, `Lactive` and `Ractive` are the two outputs in the Signal program representing respectively left and right side active):

  ```
  Property1 := B_Or(B_True(S, Lactive),B_True(S, Ractive)),
  ```

  and the property 1 in CTL to be checked:

  ```
  AG(Property1)
  ```

  which states that `Property 1` is always globally true.

  The result of the verification is illustrated in the next. The Property 1 is always satisfied according to the result.

  ```
  Sigali : subset(initial(S),AG(B_Or(B_True(S,Lactive),
  B_True(S,Ractive))));
  -----------------------------------------------
  True
  -----------------------------------------------
  ```

- **Property 2: One and only one side is always identified as pilot flying side**

  We define Property 2 in Sigali as follows (`LPF` and `RPF` are the two outputs in the Signal program representing respectively the left and right pilot flying):

  ```
  Property2 := B_Or(Equal(B_True(S,LPF), B_False(S,RPF)),
               Equal(B_False(S,LPF),B_True(S,RPF))),
  ```

and the property in CTL to be checked:

```
AG(Property2)
```

We check the property with Sigali:

```
Sigali : subset(initial(S),AG(B_Or(Equal(B_True(S,LPF),B_False(S,RPF)),
Equal(B_False(S,LPF),B_True(S,RPF)))));
------------------------------------------------------------------------
False
------------------------------------------------------------------------
```

The verification result states that this property is not always satisfied. Analysis has been carried out to find counter-examples. According to the analysis, in certain cases as illustrated in Table 8.1, when pilot flying side is switched by pilot command, the new side takes effect immediately. However, the previous pilot flying side is switched off until the next tick because of one tick communication delay.

Inputs	Transfer	1
	L_inde_mode	1
	R_inde_mode	1
Outputs	LPF	1
	RPF	1
	Lactive	1
	Ractive	1

Table 8.1: A counter-example exhibited by one single tick of analysis

Actually, even this property is not hold for one tick, it is still acceptable. Hence we relax the property:

```
AG(!Property2 -> AX(Property2)).
```

It implies that for all states, the property can be false for at most one tick. This property is checked by Sigali and it always holds:

```
Sigali : subset(initial(S),AG(B_Or(B_Or(Equal(B_True(S,LPF),
B_False(S,RPF)),Equal(B_False(S,LPF),B_True(S,RPF))),AX(B_Or
(Equal(B_True(S,LPF),B_False(S,RPF)),Equal(B_False(S,LPF),
B_True(S,RPF)))))));
------------------------------------------------------------
True
------------------------------------------------------------
```

- **Property 3: Both FGS are active in the independent mode of operation**.

  We define Property 3 in Sigali as follows (`LIndependent` and `RIndependent` are the two outputs in the Signal program representing respectively the left and right independent mode):

```
Property3 := B_Or(Complementary(
B_And(B_True(S,LIndependent),B_True(S,RIndependent))),
B_And(B_True(S,Lactive),B_True(S,Ractive))).
```

Property 3 does not always hold according to the verification result. Similar to Property 2, Property 3 is relaxed by one tick, and we retry the check:

```
AG(!Property3 -> AX(Property3)).
```

This turns out to be true and acceptable.

8.2 Simulation

In this section, we depict the design process for the co-modeling and co-simulation of a CESAR case study in the framework of Polychrony. This design process is based on a co-design approach at the system level [166]. Polychrony is served as a common development platform. Simulation is carried out for the purpose of verification and timing-related analysis at the system level.

8.2.1 Case study of a door management system

SDSCS (Simplified Doors and Slides Control System) is a generic simplified version of the system that allows managing doors on Airbus series aircrafts. It is a safety-critical system since incorrect door closing or opening during flight may lead to fatal crashes. The reliable system design and validation are therefore very important. In addition to the fulfillment of safety objectives, high-level modeling and component-based development are also expected for fast and efficient design. SDSCS has been chosen for the demonstration of capabilities developed in the CESAR project [35].

An Airbus aircraft has several kinds of doors, such as passenger, cargo and emergency doors. Here we focus on the management of two passenger doors. Each passenger door has a software handler, which achieves the following four tasks:

- monitor door status via door sensors;
- control flight lock actuators;
- manage the residual pressure of the cabin by controlling the outflow valves, visual status indication and an aural warning;
- and inhibit cabin pressurization if any external door is not closed, latched and locked.

The four tasks are implemented with simple logic that determines the status of monitors, actuators, etc., according to the sensor readings. In addition to sensors and actuators, SDSCS is equipped with other hardware components, such as processing units, communication link, and concentrators. The SDSCS is implemented on the IMA (Integrated Modular Avionics) platform, in which CPIOMs (Core Processing Input/Output Modules)

and RDCs (Remote Data Concentrators) are connected via the AFDX (Aircraft Full DupleX) network (Figure 8.5). Sensors and actuators are also connected to RDCs via AFDX. CPIOMs receive sensor readings via RDCs and communicate also with other systems via AFDX.

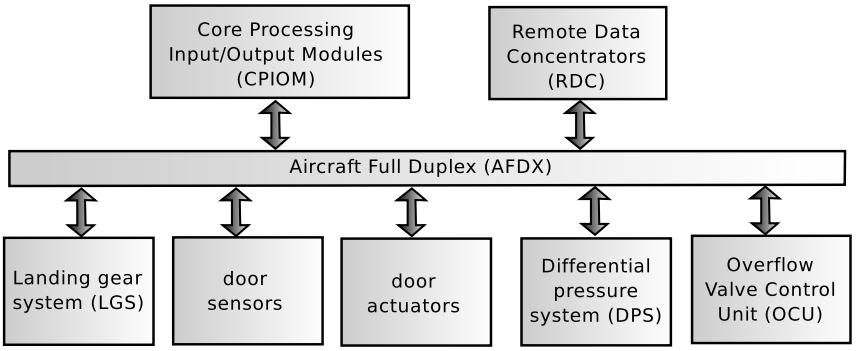


Figure 8.5: A simple illustration of the SDSCS system architecture.

This case study is simple yet complete to illustrate the effectiveness of our proposed AADL-based system-level simulation. In the next, the AADL modeling of this avionic system, model transformation and integration with other high-level models, and system co-simulation are presented.

8.2.2 System Modeling in AADL

Figure 8.6 shows an overview of the SDSCS modeled in AADL. The whole system is presented as an AADL *system*. The two doors, *door1* and *door2*, are modeled as *subsystems*. They are controlled by two processes *doors_process1* and *doors_process2* respectively. These processes are bound to two *processors*: *CPIOM1* and *CPIOM2*, to perform the computation independently. *Sensors* and *Actuators*, such as *LGS*, *DPS*, *OCU*, etc., are modeled as AADL *devices* that interface with external environment of the system. The sensors and actuators inside a door are gathered either in *door1* or *door2*. All the communication between the *devices* and *processors* is through the *bus*: *AFDX1*. SDSCS has three *threads* to manage doors: *door_handler1*, *door_handler2*, and *doors_mix*. These threads are implemented by Simulink models. In addition, each *processor* runs one *doors_process*. These two components are placed into one ARINC *partition*. Each *processor* is associated with an ARINC *partition_level_OS*, which is responsible for scheduling all the *processes* in the same *partition*. In this example, all the threads and devices are periodic, and share the same periodicity. The detailed AADL textual model can be found in Appendix A.

Figure 8.7 gives a more precise description of the *doors_process*. It contains three threads for the door management: *door_handler1*, *door_handler2*, and *doors_mix*. A thread is a schedulable functional unit that executes instructions in the virtual space assigned to the *process* owning it. In this case study, AADL is mainly used for the modeling of system architecture. The functional modeling is expected to be open for other high-level modeling languages. Simulink is adopted in this case study, i.e., the behavior of three threads are modeled in Simulink.

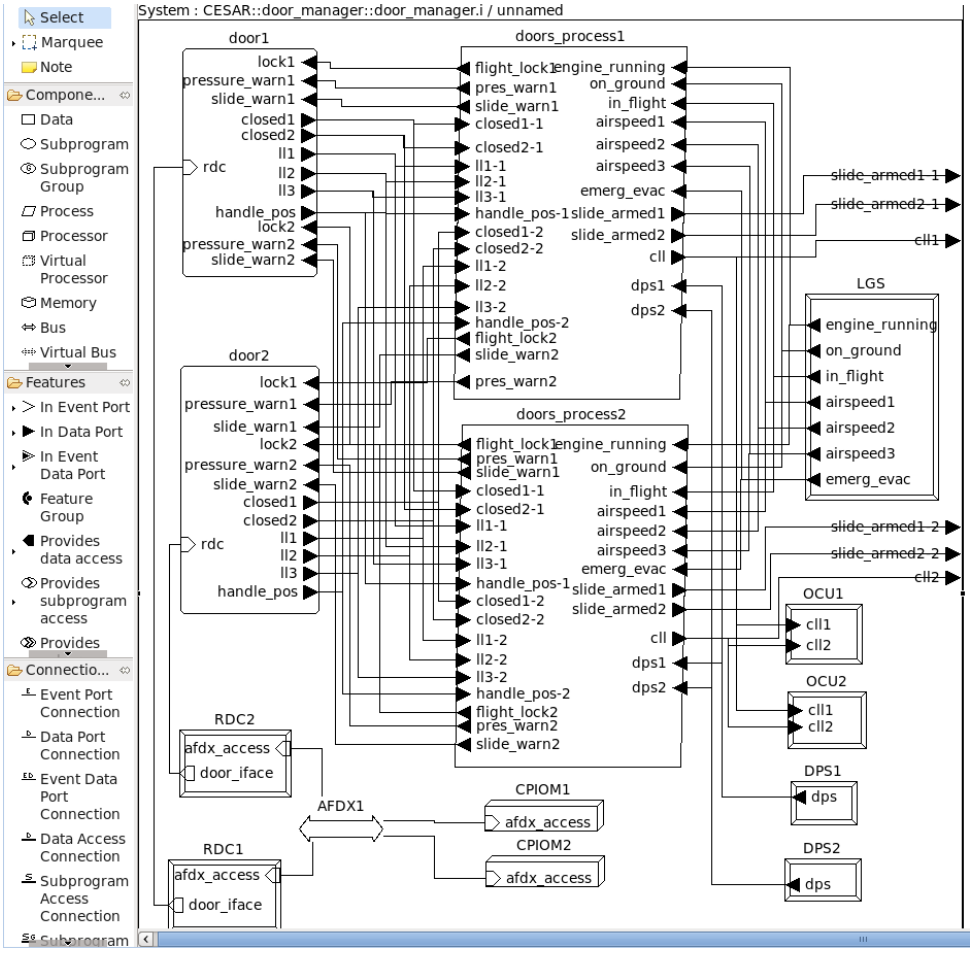


Figure 8.6: An overview of the AADL modeling of SDSCS

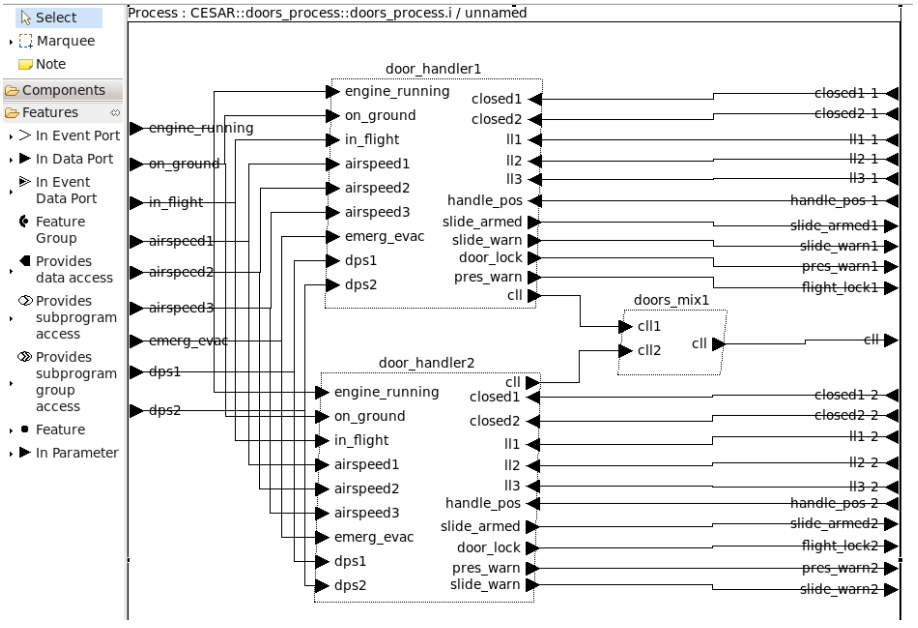


Figure 8.7: The threads in the doors_process

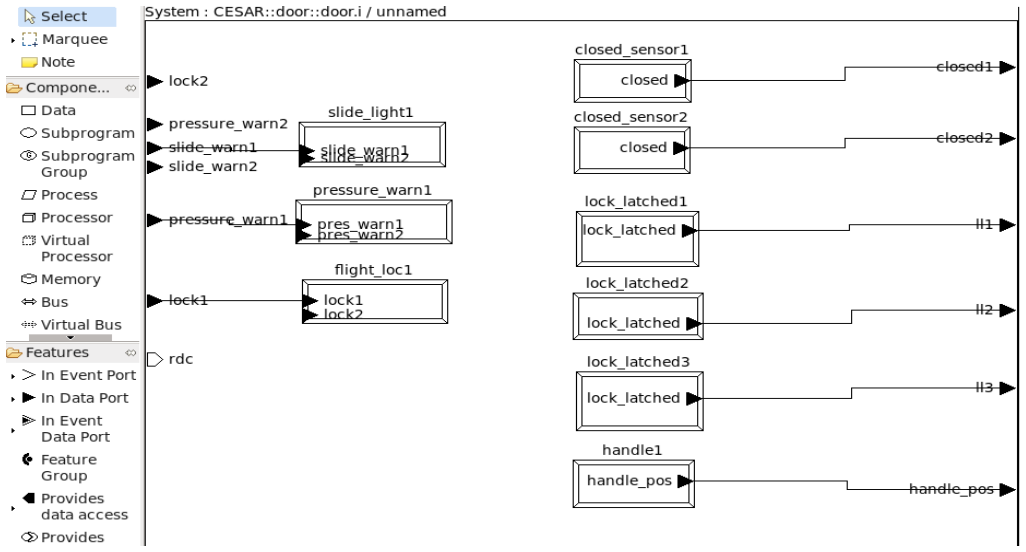


Figure 8.8: Graphical representation of an AADL door subsystem

A detailed modeling of the *door1*, as an AADL *subsystem*, is illustrated in Figure 8.8. It contains nine *devices* as its subcomponents.These devices are models of door sensors and actuators.

AADL	SME
a: *component* :: *ThreadType* and b:*component* :: *ThreadImpl* and b.*compType* = a	*sme* :: *ModelDeclaration* and its instance (*sme* :: *ModelInstance*)
instance name	*ModelDeclaration.name*
ProcessType.features.dataPort *ProcessType.features.eventPort* *ProcessType.features.eventDataPort*	*ModelDeclaration.inputs* or *ModelDeclaration.outputs* depending on the port *direction* attribute
For each of its subcomponents defined in *ProcessImpl.subcomponents*	*sme* :: *ModelDeclaration* and its instance (*sme* :: *ModelInstance*) included in the parent's attribute *modelDeclarations*
ProcessImpl.connections	*sme* :: *DataflowConnection*

Table 8.2: Thread Transformation

8.2.3 Interpreting the model in Signal

This model contains two *processes*. Each *process* has three periodic *threads*. The *process* is transformed into an ARINC *partition*, and thread is transformed into an ARINC *process*. Table 8.2 summarizes the AADL Ecore model and its corresponding transformation in SME model. SME is the Signal meta under Eclipse, and it includes the meta-model of the Signal language extended with mode automata concepts.

Figure 8.9 gives a partial code specification of the *process doors_process* in Signal. The Signal process *Scheduler_CPIOM()* is a scheduler that schedules the three

threads. Each *thread* is translated into a timing environment process (e.g. *PropertyT_door_handler()*) and a functional process (e.g. *ThreadT_door_handler()*). The detailed implementation of these processes can be found in Appendix A.

```
process PART_cpiom1 =
    ( ? boolean engine_running, on_ground, in_flight,emerg_evac, ll1_1, ll2_1, ll3_1;
        dreal airspeed1, airspeed2, airspeed3;
        boolean  closed1_1, closed2_1,  handle_pos1,closed1_2;
        boolean  closed2_2, dps1, dps2, handle_pos2, slide_armed_in1, slide_armed_in2;
      ! boolean cll, slide_armed1, slide_armed2, flight_lock1, pres_warn1, slide_warn1;
        boolean flight_lock2, pres_warn2, slide_warn2, cll1, cll2;
    )
    unsafe
  (| Scheduler_cpiom1 :: (O_tick,door_handler1_start, door_handler1_resume,door_handler1_stop,
                          door_handler2_start,door_handler2_resume, door_handler2_stop,
                          door_mix_start,door_mix_resume,door_mix_stop) := Scheduler_CPIOM{}()

   | P_cpiom1_doors_process1_door_handler1 :: (O_door_handler1_dispatch,O_door_handler1_deadline,
                          O_door_handler1_scomplete,O_door_handler1_start)
                          := PropertyT_door_handler{}(O_tick,door_handler1_start,
                             door_handler1_resume,door_handler1_stop)
   | cpiom1_doors_process1_door_handler1 :: (slide_armed1,cll1,slide_warn1, flight_lock1,pres_warn1)
                           := ThreadT_door_handler{}(O_door_handler1_dispatch,
                           O_door_handler1_deadline,O_door_handler1_scomplete,O_door_handler1_start,
                           handle_pos1,ll1_1,ll2_1,ll3_1,closed1_1,closed2_1,in_flight,on_ground,
                           engine_running,airspeed1,airspeed2,airspeed3, emerg_evac,
                           dps1,dps2,slide_armed_in1)

   | P_cpiom1_doors_process1_door_handler2 :: (O_door_handler2_dispatch,O_door_handler2_deadline,
                          O_door_handler2_scomplete,O_door_handler2_start)
                          := PropertyT_door_handler{}(O_tick,door_handler2_start,
                             door_handler2_resume,door_handler2_stop)
   | cpiom1_doors_process1_door_handler2 :: (slide_armed2,cll2,slide_warn2,flight_lock2,pres_warn2)
                          := ThreadT_door_handler{}(O_door_handler2_dispatch,O_door_handler2_deadline,
                             O_door_handler2_scomplete, O_door_handler2_start,handle_pos2,
                             ll1_2,ll2_2,ll3_2,closed1_2,closed2_2,in_flight,on_ground,engine_running,
                             airspeed1,airspeed2,airspeed3,emerg_evac,dps1,dps2,slide_armed_in2)

   | P_cpiom1_doors_process1_door_mix :: (O_door_mix_dispatch,O_door_mix_deadline,
                           O_door_mix_scomplete,O_door_mix_start):= PropertyT_doors_mix{}
                           (O_tick,door_mix_start,door_mix_resume,door_mix_stop)

   | cpiom1_doors_process1_door_mix :: cll := ThreadT_doors_mix{}(O_door_mix_dispatch,
                           O_door_mix_deadline,O_door_mix_scomplete,O_door_mix_start,cll1,cll2)


  |)
  where
   ...
  end;
```

Figure 8.9: Partial Signal code of doors_process

8.2.4 Other models and system integration

As mentioned previously, functional part of SDSCS can be modeled with other modeling languages. In this case study, Simulink and/or Stateflow in the Matlab family are adopted. Dataflow models and state machines are common models of computation adopted in the system design of avionics, automotive applications, etc. One of the most popular tools that accept these models is Simulink/Stateflow [26]. Gene-Auto is a framework for code generation from a safe subset of Simulink and Stateflow models for safety critical embedded systems [155]. This safe subset is also adopted in our work. From now on, Simulink is used for short to indicate the subset of Simulink and/or Stateflow languages that is adopted by Gene-Auto.

Only discrete time of Simulink is taken into account here. Moreover, each block of Simulink is associated with a specific activation clock. A global activation clock is used to synchronize the activation clocks of Simulink blocks. From this point of view, our Simulink model is synchronous. The Simulink model is shown in Figure 8.10. Sensors, such as *flight_status*, *dps*, and *door_io_in*, are connected to four Simulink blocks, each of which implements a SDSCS task as mentioned in the previous subsection. The model transformation chain from functional models in Simulink to Signal which is similar to the transformation from AADL to Signal, is under development by other colleagues.

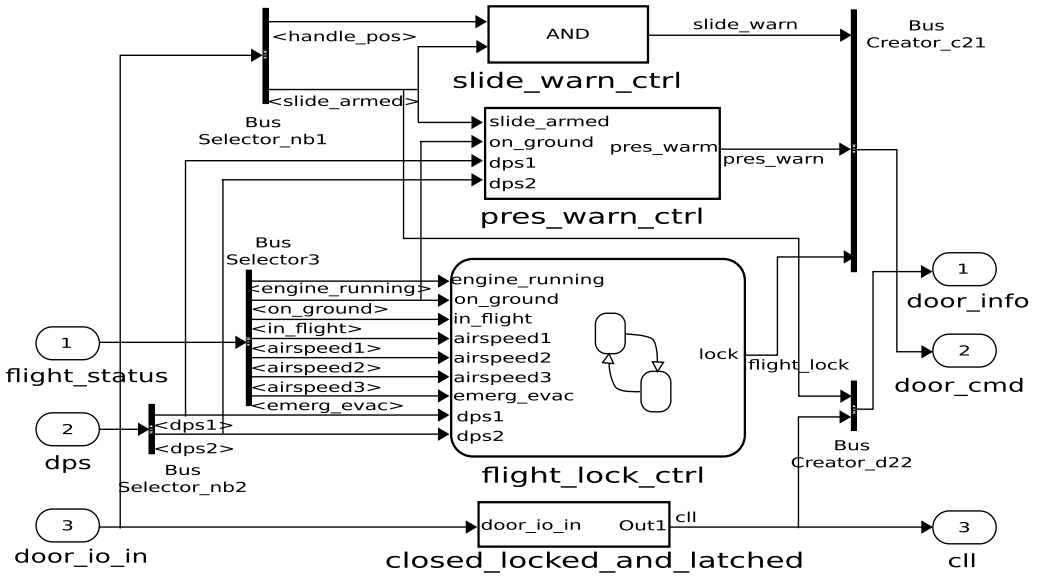


Figure 8.10: The door handler component of the SDSCS modeled in Simulink

In addition to the high-level Simulink and AADL models, additional models are also needed in the SDSCS for the complete simulation. They include an allocation model, a scheduler model and an environment model.

In this case study, the allocation of functionality onto architecture is specified in the AADL model. In the AADL to Signal transformation, all the threads mapped on to the same processor (CPIOM) are placed in the same partition. The generated Signal programs are annotated with allocation information. All the Signal processes translated from the same partition have the same Signal pragma `RunOn` "`i`" [59], which enables the distribution of these processes onto the same processor `i` (as presented in Chapter 7).

According to the AADL specification, a partition-level scheduler is needed for the simulation. In this case study, a simple static scheduler is implemented. This scheduler takes events, such as *dispatch*, *start*, *completion*, etc., into account for the scheduling of threads in the same partition. Simple non-preemption partition-level schedulers have been coded in Signal manually for the simulation. More sophisticated schedulers, such as that provided by Cheddar [146] are expected to be integrated into the system.

Sensors and actuators are the media between SDSCS and its environment. The environment with regard to SDSCS includes the aircraft system outside SDSCS as well as the environment outside the aircraft which provides flight altitude, air speed, etc. The environment modeling is carried out directly in Polychrony. Several Signal processes are added as environment models. For instance, a process detects the *close* status of physi-

cal doors and sends the *close* signals to door sensors, another process provides air speed readings to aircraft speed signals sensors. These Signal processes that model the environment are composed with other Signal processes that are transformed from the high-level Simulink and AADL models.

Once all the needed models are obtained, the composition of these models is possible. All the parts, such as system behavior, hardware architecture, environment, and schedulers, are expressed by Signal processes, thus a composition of these processes implies system integration. *Functional models* in Figure 8.11 imply the composition of all processes translated from Simulink. The communication of distributed processes is implemented by MPI (Message Passing Interface). *Architecture models* indicate the composition or connection of hardware interfaces and hardware implementations. *Simulation clocks* are used to provide reference clocks and periodical clocks for simulation. The integrated system is then used for C or Java code generation via the Signal compiler for simulation purpose.

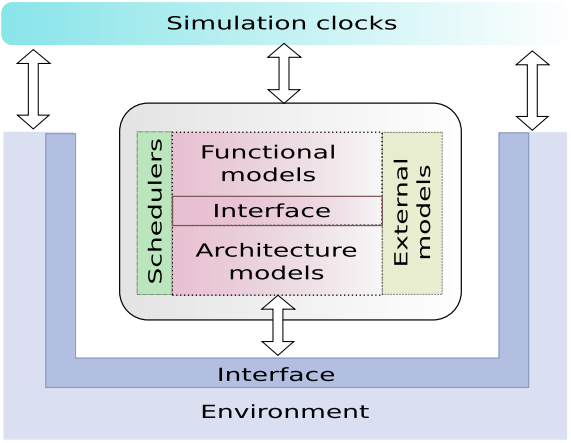


Figure 8.11: The system integration from the point of view of Signal processes

8.2.5 Profiling

Software profiling is considered as a kind of dynamic program analysis through the information gathered when the program executes. This analysis is always involved in performance improvement. Profiling is also adopted in Polychrony for the performance evaluation of Signal programs [106] [82]. The profiling process includes: **temporal properties specification**, **temporal homomorphism**, and **co-simulation**.

In the framework of Polychrony, profiling refers to timing analysis through associating *date* and *duration* information to Signal programs. Each signal x in the program is associated with a date signal, *date_x*, to indicate its availability time. This date signal may be specified with metric clock, logical clock or clock cycles. In the first case, date signals are positive real numbers, and in other cases they are positive integers. Each operation in Signal programs is associated with the *duration* information, which has the same data type as date signals. The *duration* is represented by a pair of numbers corresponding to the worst and best case. Furthermore, it is a function of several parameters related to operation type, data types, and implementations on different architectures, etc.

A *morphism* of Signal processes represents a series of transformations of Signal processes without changing their synchronous semantics. The set of homomorphic Signal processes exhibits different behavioral aspects of original Signal processes. Temporal properties are introduced in the morphism of Signal processes so that they are used to reveal the timing aspect of these processes. A Signal process can be considered as a directed graph of signals and operations, where signals are arcs and operations are nodes. A temporal morphism of Signal process preserves the graph structure. However, nodes are replaced by operation durations and arcs are replaced by date signals.

In addition, *duration* parameters of Signal operations allow the parameterization of homomorphism. These parameters together with their values allow specifying the execution of operations on specific architecture (particularly processing elements). They also enable to import and use specific libraries of cost functions in the homomorphism.

As the temporal homomorphism preserves the synchronous semantics, the homomorphic Signal processes can be composed together for the co-simulation. The latter exhibits the timing behavior with regard to previously mentioned temporal properties. Figure 8.12 illustrates a schema of the co-simulation that has been carried out successfully. *SDSCS* is the original Signal program, whose inputs *I* are provided by *Inputs*. *T(SDSCS)* is the temporal homomorphism of *SDSCS* with regard to specified *Temporal properties* and a parameterization of *Library of cost functions*. *Date* provides date signals to *T(SDSCS)* according to *I*. The input signals are synchronized to their corresponding date signals. Control values of *SDSCS*, which decide specific traces of execution, are sent to *T(SDSCS)* so that they have the same execution traces. Date signals of inputs and outputs of *T(SDSCS)* are finally sent to *Observer* in order to obtain the simulation result *V*.

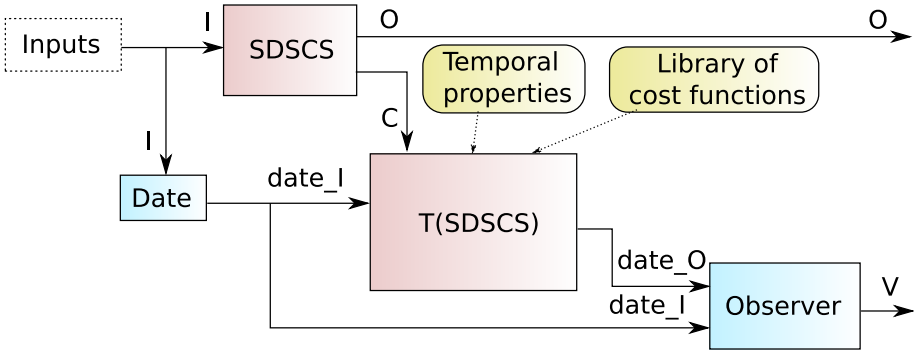


Figure 8.12: The co-simulation of Signal programs with regard to their temporal behavior

8.2.6 VCD-based simulation

In addition to profiling, another simulation has also been carried out. It aims at the visualization of value change during the execution of programs via VCD. VCD files are generally generated by EDA (Electronic Design Automation) logic simulation tools, and they adopt an ASCII-based file format, whose simplicity and compact structure allows a wide spread in application simulation. Moreover, the four-value VCD format has been defined as IEEE Standard [33] together with VHDL (Verilog Hardware Description Language). In our simulation, traces are recorded in VCD format. The VCD files are used for the visualization of simulation results through graphical VCD viewers, such as GTKWave

[34]. Figure 8.13 shows a visualization result of the simulation. In this figure, the change of signal values with regard to the fastest clock is shown. The visualization can also be done dynamically at the same time as the execution.

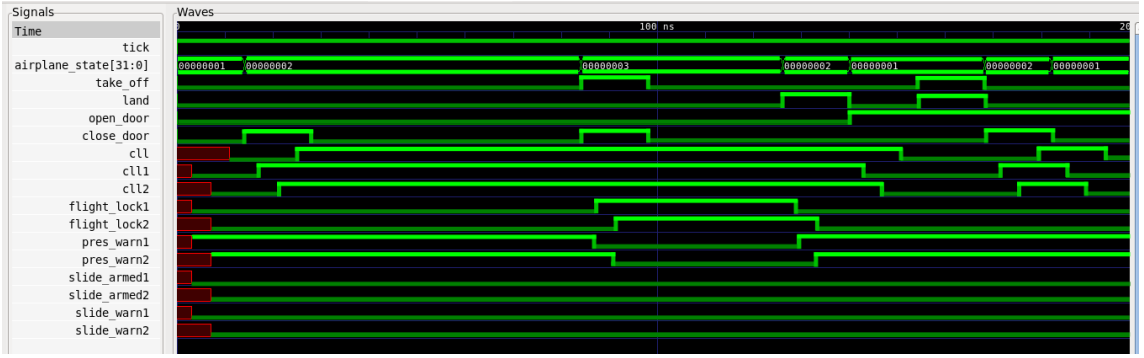


Figure 8.13: Simulation by a VCD viewer: GTKWave

8.3 Conclusion

In this chapter, we describe a formal verification and simulation tool chain for AADL. We give a high-level view of the tools involved and illustrate the successive transformations required by our verification process. Two case studies are presented for the illustration of AADL application validation, particularly, formal verification and simulation.

With the first case study, model checking of AADL models via Polychrony is briefly illustrated. AADL is used to model a simplified flight guidance system (FGS). Then the AADL models are translated into a symbolic representation, i.e., polynomial dynamical systems, manipulated by the model checker Sigali. With Sigali, safety properties expressed in CTL are checked on the system.

The second case study is called Simplified Doors and Slides Control System (SDSCS) from avionics. In addition to the AADL simulation, this case study also illustrates an approach to address high-level and heterogeneous model based system co-design, based on the globally asynchronous locally synchronous system architecture (GALS). In this case study, distributed hardware architecture, together with their asynchronous aspects, is modeled in AADL, whereas functional behavior, expressed with the synchronized dataflow model, is modeled in Simulink. SME/Polychrony is adopted as a common formalism to bridge between two heterogeneous models. The AADL modeling and transformation results in Signal are given. An early phase simulation via profiling in consideration of timing constraints and also a simulation demonstration by VCD viewers are enabled in the framework of Polychrony.

As a perspective, connections to more simulation tools with regard to timing analysis are expected, such as Syndex [25] and RT-Builder [22].

Conclusion

AADL (Architecture Analysis and Design Language), as an SAE (Society of Automotive Engineers) standard, is dedicated to the high-level design and evaluation for embedded systems. It is particularly used in the modeling and analysis of avionic applications. It allows describing both system structure and functional aspects via a component-based approach, e.g., locally synchronous processes are allocated onto a distributed architecture and communicate in a globally asynchronous manner (GALS system). In order to enable early-phase validation and step-wise refinement, our objective of this thesis is to model, transform and validate such systems in a synchronous programming framework, and then automatically generate its GALS implementation.

The polychronous model of computation (MoC), based on the concepts of the synchronous approach, allows specifying a system whose components can have their own working frequency, or their local activation clock. It is well adapted to support a GALS design methodology. In addition, formal verification, simulation and analysis technologies have been developed over this model for the design of embedded systems. Polychrony, based on the polychronous model, is a unified framework of modeling, transformation, refinement, validation of embedded systems design.

The main motivation of the work presented in this thesis is to bridge the gaps between AADL and Polychrony: a proposition of a methodology for system-level modeling and validation of embedded systems specified in AADL via the polychronous MoC. This methodology includes system-level modeling via AADL, automatic transformations from the high-level AADL model to the polychronous model, code distribution, formal verification, simulation of the obtained polychronous model, and model refinement according to the validation results.

Contributions

Our system-level design takes into account both the system architecture, particularly described in Integrated Modular Avionics (IMA), and functional aspects, e.g., software components implemented in the synchronous programming language Signal. Starting from high-level AADL specifications, automatic code distribution for simulation has been experimented via the polychronous model and Signal distribution pragmas. In addition, AADL Behavior annex, an extension for the specifications of the actual behaviors, is also involved in the translation from AADL to Signal. It relies on the use of SSA (Static Single Assignment) as intermediate formalism. High-level simulation, validation and verification are then carried out by associated technologies and tools, such as Sigali, profiling, and value change demonstration, to formal check and analyze the result model. Existing

techniques and libraries of Polychrony, which consist of a model of the APEX-ARINC real-time operating system services, have been used. The main contributions are detailed in the following:

Synchronous modeling of a subset of the AADL model. We have proposed a formal methodology to model a subset of the AADL model into synchronous formalism in the IMA architecture [121]. This modeling bridges the gaps between AADL and synchronous languages with the help of Polychrony. By analyzing the time domains, the AADL architectures and components with abstract logical time are expressed in the polychronous framework with more concrete simulation time. Since we use a modular approach to define the AADL model, it supports a structured design of the semantics of the AADL application. The resulting Signal model from the transformation of the AADL model preserves the structured semantics of the original design. It turns out to be an effective approach to transform a hierarchy of AADL applications into a set of synchronous equations. On one hand, this transformation preserves the structural and semantic properties of original AADL specifications, which insures that verifications carried out on the Signal model are also valid for the AADL model. On the other hand, it keeps the characteristics associated with synchronous languages, enabling a formal and clear semantic definition.

Formal interpretation of AADL Behavior Annex. AADL Behavior Annex specifies the local functional behaviors of the AADL components, without requiring external target languages. We have presented an interpretation of the Behavior Annex into a synchronous data-flow and multi-clocked MoC [122]. In this interpretation, SSA is used as an intermediate formalism. A thorough description of an inductive SSA transformation algorithm across a hierarchy of transitions that produces synchronous equations is presented. With regard to verification requirements, this interpretation minimizes the number of states across automata, hence provides good model checking performance. The interpretation is expressed by a set of flexible transformation rules, which simplifies the interpretation.

Distributed code generation. We use Polychrony to distribute the centralized code to different physical processors. We have presented, from a complete representation of an AADL model, including its virtual distribution on target hardware components, how it is possible to have distributed Signal programs, making a partitioning of the application on the target physical architecture [123]. Our approach is a practical application of formal results that have been proved by the polychronous MoC, for which it has been shown that if the distributed programs or processes are endo-isochronous (assume that the deployment is simply performed by using an asynchronous mode of communication), then the original semantics of each individual process of the deployed GALS architecture is preserved.

Formal verification, simulation and analysis. Besides that the synchronous model is suitable to express real-time applications, the code generated from this model allows one to reason about safety-critical behavior of these applications. In that way, formal verification and simulation, based on the generated Signal code, are possibly carried out to check the reliable design of AADL applications. With the model checker Sigali, a model

checker associated with Polychrony, properties and functional correctness of the application can be checked. The verification results help to reveal safety-related problems in the original design in AADL. Early-phase simulation, including profiling and value change demonstration, in consideration of timing constraints are also carried out in our work. Profiling, together with parameters that represent chararcteristics of specific architecture, is performed to illustrate performance-related analysis, for example, worst case execution time. Another simulation is involved in value change demonstration via value change dump (VCD). It shows the signal value change during the system execution.

Perspectives

Extension of the modeling. A subset of AADL has been handled in the synchronous modeling in this thesis, which includes periodic threads, immediate and delayed connections, buses, Bebavior Annex, etc. The accomplished modeling and transformations show an interesting and promising result. An extension of this modeling is expected to cover a larger scope of AADL, e.g., different types of threads, data, subprogram calls, client-server protocol [165], semi-synchronous protocol, as well as more temporal aspects that are related to GALS design and enable more sophisticated timing analysis. Indeed, this extension of the modeling would enable to verify more complex system behavior of applications specified in AADL. Another extension consists in providing a model library in Signal, which addresses standard component modeling. This library includes canonical models, which can be parameterized to model similar components of AADL, e.g., aperiodic and sporadic threads. Models of AADL hardware components, e.g., bus and processor, can be included in this library. This library helps to reduce the repetitive modeling task.

Extension of validation. We are currently using Sigali in the formal verification process. One possible perspective of this work is to perform timed model checking, which requires a translation of the Signal model into other timed model for verification. In that case, it would be possible to formally define best-case and worst-case time behavior of the AADL model [119]. In addition to the verification and simulation, connections to more simulation tools with regard to timing analysis are expected, such as SynDEx [25], RT-Builder [22], etc. Starting with modules described in AADL architecture description language, we are seeking for an approach for the verification of GALS systems. One could also model and verify GALS systems using a combination of synchronous languages and other languages, such as Promela, process calculi, etc., for communication channels and asynchronous concurrency [75]. We can extend this work as the refinement of a synchronous component into a set of asynchronous modules. Insert desynchronization buffers into such a refinement and model check the correctness of the refinement using flow preservation refinement and real-time constraints are desired. Moreover, we can study different types of interconnections between the synchronous components. This would allow for verification of more complex GALS systems going beyond point-to-point connections.

Extension of the scheduling model. Scheduling specified in AADL has been addressed in a simple and static way in this thesis. Extensions of our scheduling model is also a perspective. First extension is involved in the integration of complete scheduling state machines into threads. Certain scheduling states have already been introduced in our modeling, but it is not complete compared to AADL specification. Then a partition-level scheduler, in accordance with ARINC, can be integrated into each processor. Second extension concerns the integration of standard or off-the-shelf schedulers into the system for simulation purpose, e.g., an existing scheduler has been written in Signal for ARINC [85], or Cheddar [8] provides schedulers that implement standard scheduling strategies.

Bibliography

[1] AADL. http://www.aadl.info/.

[2] AADL Users. http://www.aadl.info/aadl/currentsite/start/who.html.

[3] AADL2SYNC, Verimag. http://www-verimag.imag.fr/~synchron/index.php?page=aadl2sync.

[4] ACME. http://www.cs.cmu.edu/~acme/.

[5] Ada. http://www.adaworld.com/.

[6] AIRBUS. http://www.airbus.com/en/.

[7] BIP, Verimag. http://www-verimag.imag.fr/BIP,196.html.

[8] Cheddar Project. http://beru.univ-brest.fr/~singhoff/cheddar.

[9] EADS, European Aeronautic Defense and Space company. http://www.eads.com.

[10] Ellidiss Software. http://http://www.ellidiss.com/.

[11] ESA, European Space Agency. http://www.esa.int/esaCP/index.html.

[12] Honeywell. http://www51.honeywell.com/honeywell/.

[13] LAM MPI. http://www.lam-mpi.org/.

[14] MARTE. http://www.omgmarte.org/.

[15] Ocarina Project. http://ocarina.enst.fr/.

[16] Open MPI. http://www.open-mpi.org/.

[17] OpenEmbeDD. http://openembedd.org/home_html.

[18] OSATE. http://gforge.enseeiht.fr/projects/osate/.

[19] POK. http://penelope.enst.fr/aadl/wiki/PokUsageWithOcarina.

[20] PolyORB-HI-C. http://penelope.enst.fr/aadl/wiki/PolyorbhicPresentation.

[21] Rockwell Collins. http://www.rockwellcollins.com/.

[22] RT-Builder. http://www.geensoft.com/.

[23] SEI. http://www.sei.cmu.edu/.

[24] SME, ESPRESSO, INRIA. http://www.irisa.fr/espresso/Polychrony/.

[25] Syndex. http://www-rocq.inria.fr/syndex/.

[26] The MathWorks Simulink. http://www.mathworks.com/products/simulink/.

[27] The MathWorks Stateflow. http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow.

[28] Topcased-aadl. http://gforge.enseeiht.fr/projects/topcased-aadl.

[29] Topcased Project. http://www.topcased.org.

[30] Unified Model Language. http://www.omg.org/.

[31] ADL, 2005. http://www.itl.nist.gov/div897/ctg/adl/adl_info.html.

[32] Cinderella SDL 1.3, 2005. http://www.cinderella.dk/.

[33] IEEE Standard for Verilog Hardware Description Language (VHDL), 2006. IEEE Std 1364 -2005.

[34] GTKWave, 2010. http://gtkwave.sourceforge.net/.

[35] T.C. Project, Cost-efficient methods and processes for safety relevant embedded systems, 2010. http://www.cesarproject.eu.

[36] The CESAR project, Cost-efficient methods and processes for safety relevant embedded systems. http://www.cesarproject.eu, 2010.

[37] Avionics Application Software Standard Interface. *Airlines Electronic Engineering Committee, ARINC Specification 653*, January 1997.

[38] Architecture Analysis and Design Language, AS5506. *Society of Automotive Engineers, SAE standard, V2.0*, January 2009.

[39] Design Guidance for Integrated Modular Avionics. *Airlines Electronic Engineering Committee, ARINC Report 651-1*, November 1997.

[40] AADL. Carnegie Mellon University. http://www.aadl.info/aadl/currentsite/newsbriefs.html.

[41] SAE Aerospace. Architecture Analysis and Design Language (AADL). *SAE AS5506*, 2004.

[42] SAE Aerospace. Annex Behavior Language Compliance and Application Program Interface. *SAE AS5506*, 2006.

[43] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, I. Lee, P. Mishra, G. J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proc. IEEE*, 91(1):11–28, 2003.

[44] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, Computer Laboratory, University of Cambridge, January 2001.

[45] Charles André, Frédéric Mallet, and Robert de Simone. Modeling Time(s). In *LNCS*, pages 559–573. Springer, 2007.

[46] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of Synchronous Versus Asynchronous Distributed Systems. *J. ACM*, 30(3):449–456, 1983.

[47] P. Aubry, P. Le Guernic, and S. Machard. Synchronous Distribution of SIGNAL Programs. *Hawaii International Conference on System Sciences*, 0:656, 1996.

[48] Felice Balarin and Alberto Sangiovanni-Vincentelli. Schedule Validation for Embedded Reactive Real-Time Systems, 1997.

[49] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, pages 16:103–149, 1991.

[50] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991.

[51] Albert Benveniste, Benoït Caillaud, and Paul Le Guernic. From synchrony to asynchrony. In *CONCUR'99, Concurrency Theory, 10th International Conference, Volume 1664 of Lecture Notes In Computer Science*, pages 162–177. Springer, 1999.

[52] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[53] Albert Benveniste, Paul Caspi, Paul Le Guernic, Hervé March, Jean-Pierre Talpin, and Stavros Tripakis. A protocol for loosely time-triggered architectures. In *In Embedded Software Conference*, pages 252–266. Springer, 2002.

[54] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[55] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation, 1992.

[56] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse France, 2008.

[57] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse France, 2008.

[58] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL V4-Inria Version: Reference manual. http://www.irisa.fr/espresso/Polychrony.

[59] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. Compilation of polychronous data flow equations. In Sandeep Shukla and Jean-Pierre Talpin, editors, *Correct-by-Construction Embedded Software Synthesis: Formal Frameworks, Methodologies, and Tools*, 2010.

[60] Loïc Besnard, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Compilation of polychronous data flow equations. *Correct-by-construction embedded software design*, 2010.

[61] Loïc Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson, and Florence Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form. In *Ninth International Workshop on Automated Verification of Critical Systems (AVOCS'09)*. Electronic Communications of the EASST, September 2009.

[62] Egon Börger. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag Berlin Heidelberg New York, 2003.

[63] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, Nov 1994.

[64] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software—Practice and Experience*, 28(8):859–881, Jul 1998.

[65] Matthias Brun, Jero Delatour, and Yvon Trinquet. Code Generation from AADL to a Real-Time Operating System: An Experimentation Feedback on the Use of Model Transformation. *Engineering of Complex Computer Systems, IEEE International Conference*, pages 257–262, 2008.

[66] Jonas Carlsson, Kent Palmkvist, and Lars Wanhammar. Synchronous Design Flow for Globally Asynchronous Locally Synchronous Systems.

[67] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-Time Systems. In *International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES'08)*, Toulouse, France, September 2008. Springer-Verlag.

[68] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-Time Systems. *Models in Software Engineering: Workshops and Symposia at MODELS 2008*, pages 5–19, 2009.

[69] J.-D. Choi, V. Sarkar, and E. Schonberg. Incremental computation of static single assignment form. In *Sixth International Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 223–237, Apr 1996.

[70] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[71] M. Chapiro Daniel. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.

[72] B. de Simone. The ESTEREL language. In *Proceedings IEEE, 79-9*, 1991.

[73] Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, Simulate and Implement ARINC653 Systems using the AADL. *ACM SIGAda Ada Letters. Volume 29. Number 3, Pages 31-44. ISSN:1094-3641. Also published in the proceedings of the international ACM SIGAda conference, Tampa Bay, Florida, USA*, November 2009.

[74] P. Dissaux, J.P. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat. AADL Behavioral annex, 2006.

[75] Frédéric Doucet, Massimiliano Menarini, Ingolf Kruger, Jean-Pierre Talpin, and Rajesh Gupta. A verification approach for gals integration of synchronous components. In *Proceedings of the International Workshop on Formal Methods for Globally Asynchronous Locally Synchronous Design (FMGALS)*, Verona, Italy, July 2005.

[76] Bruce Powel Douglass. *Doing Hard Time: Developing Real-Time Systems With UML, Objects, Frameworks, And Patterns* . Addison-Wesley Professional, May, 1999.

[77] William R. Dunn. Designing Safety-Critical Computer Systems. *Computer*, 36:40–46, 2003.

[78] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, J. Ludwig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity: The Ptolemy approach. *Proc. IEEE*, 91(1):127–144, 2003.

[79] Mariusz A. Fecko, M. Ümit Uyar, Paul D. Amer, Adarshpal S. Sethi, Ted Dzik, Raymond Menell, and Mike Mcmahon. A Success Story of Formal Description Techniques: Estelle Specification and Test Generation for MIL-STD 188-220, 1999.

[80] P.H. Feiler, D.P. Gluch, and J.J. Hudak. The Architecture Analysis & Design Language (aadl): An Introduction. *Technical Note CMU/SEI-2006-TN-011*, February 2006.

[81] Ricardo Bedin Franca, Jean-Paul Bodeveix, Mamoun Filali, Jean-Francois Rolland, David Chemouil, and Dave Thomas. The AADL behaviour annex – experiments and roadmap. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 377–382, Washington, DC, USA, 2007. IEEE Computer Society.

[82] A. Gamatié, T. Gautier, and L. Besnard. Modeling of Avionics Applications and Performance Evaluation Techniques using the Synchronous Language SIGNAL. In *SLAP'03*. Elsevier Science B.V., 2003.

[83] Abdoulaye Gamatié. *Modélisation polychrone et évaluation de systèmes temps réel*. PhD thesis, IFSIC/IRISA, May 2004.

[84] Abdoulaye Gamatié and Thierry Gautier. Modeling of avionics applications and performance evaluation techniques using the synchronous language Signal. In *Proceedings of SLAP'03, volume 88 of ENTCS*. Elsevier, 2003.

[85] Abdoulaye Gamatié and Thierry Gautier. Synchronous Modeling of Avionics Applications using the SIGNAL Language. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 144, Washington, DC, USA, 2003. IEEE Computer Society.

[86] Abdoulaye Gamatié and Thierry Gautier. The SIGNAL Approach to the Design of System Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 21:641–657, August 2009.

[87] Abdoulaye Gamatié, Thierry Gautier, and Paul Le Guernic. An Example of Synchronous Design of Embedded Real-Time Systems based on IMA. In *10th International Conference on Real-time and Embedded Computing Systems and Applications (RTCSA'2004)*, Gothenburg, Sweden, August 2004.

[88] Abdoulaye Gamatié, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. Polychronous design of embedded real-time applications. *ACM Trans. Softw. Eng. Methodol.*, 16(2):9, 2007.

[89] Gamatié, A. *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2009.

[90] Thierry Gautier, Paul Le Guernic, and Olivier Maffeïs. For a New Real-Time Methodology. In *IRISA PUBLICATION INTERNE NO. 870*, 1994.

[91] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, Grenoble, oct 2002. LNCS 2491, Springer Verlag.

[92] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[93] Nicholas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publisher, 1993.

[94] Nicolas Halbwachs and Siwar Baghdadi. Synchronous Modelling of Asynchronous Systems. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 240–251. Springer-Verlag, 2002.

[95] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE, 1994.

[96] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for real-time system specification*. Dorset House Publishing Co., Inc., New York, NY, USA, 1987.

[97] Erwan Jahier, Nicolas Halbwachs, and Pascal. Raymond. Synchronous Modeling and Validation of Priority Inheritance Schedulers. In *Fundamental Approaches to Software Engineering Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 140–154, York Royaume-Uni, 2009. Springer Verlag.

[98] Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond. Synchronous modeling and validation of schedulers dealing with shared resources. Technical Report, Verimag, July 2008. http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2008-10.

[99] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, and David Lesens. Virtual execution of aadl models via a translation into synchronous programs. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 134–143, New York, NY, USA, 2007. ACM.

[100] Erwan Jahier, Louis Mandel, Nicolas Halbwachs, and Pascal Raymond. The aadl2sync User Guide. Technical Report, Verimag, June 2007. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/aadl2sync/aadl2sync-man.pdf.

[101] Kenneth Johnson, Loïc Besnard, Thierry Gautier, and Jean-Pierre Talpin. A Synchronous Approach to Threaded Program Verification. Technical Report RR-7320, INRIA.

[102] G. Kahn. The semantics of a simple language for parallel programming. *J. L. Rosenfeld*, pages 471–475, 1974.

[103] Alan Kaminsky. Real-Time Systems and Their Programming Languages. *Computer*, 24:150–151, 1991.

[104] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. In *Proceedings of the IEEE, Vol. 91, No.1*, pages 145–164, January, 2003.

[105] John C. Knight. Safety Critical Systems: Challenges and Directions. In *ICSE 2002*, 2002.

[106] A. Kountouris and P. Le Guernic. Profiling of Signal Programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, HP Labs, Bristol, UK, 1996.

[107] J.W. Krueger, S. Vestal, and B. Lewis. Fitting the pieces together: system/software analysis and code integration using METAH. In *Digital Avionics Systems Conference, 1998. Proceedings., 17th DASC. The AIAA/IEEE/SAE*, Bellevue, WA , USA, November 1998.

[108] Phillip A. Laplante. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, Piscataway, NJ, USA, 1992.

[109] O. Laurent and F. Pouzolz. Airbus generic pilot application Aircraft doors management system, May 2009.

[110] P. Le Guernic. SIGNAL: Description algébrique des flots de signaux. *Architecture des machines et systèmes informatiques*, pages 243–252, November 1982.

[111] P. Le Guernic and A. Benveniste. Real-time, synchronous, data-flow programming: the language SIGNAL and its mathematical semantics. Technical Report Technical Report 533 (revised version: 620), INRIA, June 1986.

[112] P. Le Guernic and T. Gautier. Data-flow to von Neumann: the SIGNAL approach. *Advanced Topics in Data-Flow Computing*, pages 413–438, 1991.

[113] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with Signal. In *Proceeding of the IEEE, v.79*, September 1997.

[114] Paul Le Guernic, Thierry Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. In *Proceedings of the IEEE*, pages 1321–1336, 1991.

[115] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.

[116] Gérard Le Lann. Distributed Systems - Towards A Formal Approach. pages 12–25. Information Processing 77, B.Gilchrist, Editor, North-Holland Publishing Company, 1977.

[117] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, 1998.

[118] Insup Lee, Patrice Brémond-Grégoire, and Richard Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. In *Proceedings of the IEEE*, pages 158–171, 1994.

[119] Su-Young Lee, Frédéric Mallet, and Robert de Simone. Dealing with AADL End-to-End Flow Latency with UML MARTE. *Engineering of Complex Computer Systems, IEEE International Conference*, pages 228–233, 2008.

[120] Kristina Lundqvist and Lars Asplund. A Formal Model of the Ada Ravenscar Tasking Profile. pages 12–25. Springer-Verlag, 1999.

[121] Yue Ma, Jean-Pierre Talpin, and Thierry Gautier. Virtual prototyping AADL architectures in a polychronous model of computation. *MEMOCODE08*, 2008.

[122] Yue Ma, Jean-Pierre Talpin, and Thierry Gautier. Interpretation of AADL Behavior Annex into Synchronous Formalism Using SSA. *Computer and Information Technology, International Conference*, pages 2361–2366, 2010.

[123] Yue Ma, Jean-Pierre Talpin, Sandeep K. Shukla, and Thierry Gautier. Distributed Simulation of AADL Specifications in a Polychronous Model of Computation. *Embedded Software and Systems, Second International Conference*, pages 607–614, 2009.

[124] Frédéric Mallet, Charles André, and Julien De Antoni. Executing AADL Models with UML/MARTE. In *ICECCS '09: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 371–376, Washington, DC, USA, 2009. IEEE Computer Society.

[125] H. Marchand, E. Rutten, M. Le Borgne, and M. Samaan. Formal Verification of programs specified with Signal: Application to a Power Transformer Station Controller. *Science of Computer Programming*, 41(1):85–104, August 2001.

[126] Hervé Marchand and Michel Le Borgne. Synthesis of discrete-event controllers based on the Signal environment. In *In Discrete Event Dynamic System: Theory and Applications*, pages 325–346, 2000.

[127] D. Mathaikutty, H. Patel, and S. Shukla. A Functional Programming Framework of Heterogeneous Model of Computation for System Design. In *Forum on Specification and Design Languages (FDL)*, 2004.

[128] Nenad Medvidovic and David S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages, 1997.

[129] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000.

[130] Steven P. Miller and Jan Duffy. FGS Partitioning Final Report. Research Report NCC-01001, Rockwell Collins, Advanced Technology Center, 2004.

[131] Prabhat Mishra, Aviral Shrivastava, and Nikil Dutt. Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs. *ACM Trans. Des. Autom. Electron. Syst.*, 11(3):626–658, 2006.

[132] Daniel Monteverde, Alfredo Olivero, Sergio Yovine, and Victor Braberman. VTS-based Specification and Verification of Behavioral Properties of AADL Models. In *International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES'08)*. Springer-Verlag, 2008.

[133] Mohammadreza Mousavi, Paul Le Guernic, Jean-Pierre Talpin, Sandeep Kumar Shukla, and Twan Basten. Modeling and Validation of Globally Asynchronous Design in Synchronous Framework. In *Digital Automation and Test Europe (Date'04)*, 2003.

[134] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, 2000.

[135] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07*, 2007.

[136] Martin Ouimet and Kristina Lundqvist. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07*, 2007.

[137] Lei Pi, J-P Bodeveix, and M. Filali. A comparative study of different formalisms to define aadl data communication. www.cert.fr/feria/svf/FAC/2009/Papiers/11.pdf.

[138] Lei Pi, Jean-Paul Bodeveix, and Mamoun Filali. Modeling AADL Data Communication with BIP. In *Ada-Europe '09: Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 192–206, Berlin, Heidelberg, 2009. Springer-Verlag.

[139] Lei Pi, Jean-Paul Bodeveix, and Mamoun Filali. Modeling AADL data communication with BIP. In *Ada-Europe'09*, pages 192–206. Springer-Verlag, 2009.

[140] Lei Pi, Zhibin Yang, J-P Bodeveix, M. Filali, Kai Hu, and Dianfu Ma. A comparative study of FIACRE and TASM to define AADL real time concepts. In *ICECCS'09*, 2009.

[141] Mónica Pinto, Lidia Fuentes, and Jose María Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 118–137, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[142] Jan Rothe, Hendrik Tews, and Bart Jacobs. The Coalgebraic Class Specification Language CCSL. *Journal of Universal Computer Science*, 7:175–193, 2000.

[143] J. Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Research Report, NASA Langley Research Center, June 1999.

[144] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Trans. Comput. Aided Design Integr. Circuits Syst.*, 23(1):17–32, 2004.

[145] Hauck Scott. Asynchronous Design Methodologies: An Overview. In *PROCEEDINGS OF THE IEEE*, pages 69–93, 1995.

[146] Frank Singhoff and Alain Plantec. AADL modeling and analysis of hierarchical schedulers. In *ACM international conference on Ada (SIGAda'07)*, 2007.

[147] I. Smarandache. *Transformations affines d'horloges: application au codesign de systèmes temps réel en utilisant les langages Signal et Alpha.* PhD thesis, Universite de Rennes 1, October 1998.

[148] Irina M. Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints. In *World Congress on Formal Methods (2)*, pages 1364–1383, 1999.

[149] Oleg Sokolsky, Insup Lee, and Duncan Clark. Schedulability Analysis of AADL models. In *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, 2006.

[150] John A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *Computer*, 21(10):10–19, 1988.

[151] John A. Stankovic. Strategic Directions in Real-Time and Embedded Systems. *ACM Computing Surveys*, 28(4):751–763, 1996.

[152] Jean-Pierre Talpin, Paul Le Guernic, Sandeep Kumar Shukla, Rajesh Gupta, and Frédéric Doucet. Polychrony for Formal Refinement-Checking in a System-Level Design Methodology, 2003.

[153] Jean-Pierre Talpin, Julien Ouy, Loïc Besnard, and Paul Le Guernic. Compositional design of isochronous systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 928–933, New York, NY, USA, 2008. ACM.

[154] Jean-Pierre Talpin, Julien Ouy, Thierry Gautier, Loïc Besnard, and Cortier Alexandre. Modular interpretation of heterogeneous modeling diagrams into synchronous equations using static single assignment. Research Report RR-7036, INRIA, 2009.

[155] Andres Toom, Tonu Naks, Marc Pantel, Marcel Gandriau, and Indra Wati. GeneAuto: An Automatic Code Generator for a Safe Subset of SimuLink/StateFlow and Scicos. In *European Conference on Embedded Real-Time Software (ERTS'08)*, 2008.

[156] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[157] Roberto Varona-Gomez and Eugenio Villar. AADL Simulation and Performance Analysis in SystemC. *Engineering of Complex Computer Systems, IEEE International Conference on*, 0:323–328, 2009.

[158] Steve Vestal and Jonathan Krueger. Technical and Historical Overview of MetaH. In *Honeywell Technology Center*, 2000.

[159] David Walker and Jack J. Dongarra. Mpi: A Standard Message Passing Interface. *Supercomputer*, 12:56–68, 1996.

[160] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Prentice Hall Professional Technical Reference, 1991.

[161] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr 1991.

[162] Daniel Weil, Valérie Bertin, Etienne Closse, Michel Poize, Patrick Venier, and Jacques Pulou. Efficient compilation of ESTEREL for real-time embedded systems. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 2–8, New York, NY, USA, 2000. ACM.

[163] Doran K. Wilde. The ALPHA Language. Technical Report, IRISA-INRIA, Rennes, 1994. www.irisa.fr/bibli/publi/pi/1994/827/827.html.

[164] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a formal semantics for the AADL behavior annex. In *Design, Automation and Test in Europe (DATE'09)*, pages 1166–1171, 2009.

[165] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a Formal Semantics for the AADL Behavior Annex. In *DATE'09, Toulouse, France, 2009*, 2009.

[166] H.F. Yu, Y. Ma, Y. Glouche, J.-P. Talpin, L. Besnard, T. Gautier, P. Le Guernic, A. Toom, and O. Laurent. System-level Co-simulation of Integrated Avionics Using Polychrony. *the 26th ACM Symposium On Applied Computing*, 2011.

Appendix A

SDSCS example

A.1 AADL specifications

```
system door_management
  features
    engine_running: in data port behavior::boolean;
    in_flight: in data port behavior::boolean;
    on_ground: in data port behavior::boolean;
    airspeed1: in data port behavior::float;
    airspeed2: in data port behavior::float;
    airspeed3: in data port behavior::float;
    emerg_evac: in data port behavior::boolean;
    slide_armed1_1: out data port behavior::boolean;
    slide_armed1_2: out data port behavior::boolean;
    slide_armed2_1: out data port behavior::boolean;
    slide_armed2_2: out data port behavior::boolean;
    cll1: out data port behavior::boolean;
    cll2: out data port behavior::boolean;
    cll1_1: out data port behavior::boolean;
    cll2_1: out data port behavior::boolean;
    cll1_2: out data port behavior::boolean;
    cll2_2: out data port behavior::boolean;
end door_management;

system implementation door_management.imp
  subcomponents
    door1: system Door.imp;
    door2: system Door.imp;
    dps1: device DPS;
    dps2: device DPS;
    ocu1: device OCU;
    ocu2: device OCU;
    doors_process1: process doors_process.imp;
    doors_process2: process doors_process.imp;
    cpiom1: processor CPIOM.imp;
    cpiom2: processor CPIOM.imp;
    rdc1: device RDC;
    rdc2: device RDC;
    afdx: bus AFDX.imp;
  connections
```

```
c1: data port engine_running -> doors_process1.engine_running;
c2: data port on_ground -> doors_process1.on_ground;
c3: data port in_flight -> doors_process1.in_flight;
c4: data port airspeed1 -> doors_process1.airspeed1;
c5: data port airspeed2 -> doors_process1.airspeed2;
c6: data port airspeed3 -> doors_process1.airspeed3;
c7: data port emerg_evac -> doors_process1.emerg_evac;
c8: data port engine_running -> doors_process2.engine_running;
c9: data port on_ground -> doors_process2.on_ground;
c10: data port in_flight -> doors_process2.in_flight;
c11: data port airspeed1 -> doors_process2.airspeed1;
c12: data port airspeed2 -> doors_process2.airspeed2;
c13: data port airspeed3 -> doors_process2.airspeed3;
c14: data port emerg_evac -> doors_process2.emerg_evac;
c15: data port doors_process1.cll -> ocu1.cll1;
c16: data port doors_process1.cll -> ocu2.cll1;
c17: data port doors_process2.cll -> ocu1.cll2;
c18: data port doors_process2.cll -> ocu2.cll2;
c19: data port dps1.dps -> doors_process1.dps1;
c20: data port dps2.dps -> doors_process1.dps2;
c21: data port dps1.dps -> doors_process2.dps1;
c22: data port dps2.dps -> doors_process2.dps2;
c23: data port doors_process1.slide_armed1 -> slide_armed1_1;
c24: data port doors_process1.slide_armed2 -> slide_armed2_1;
c25: data port doors_process1.cll -> cll1;
c26: data port doors_process2.slide_armed1 -> slide_armed1_2;
c27: data port doors_process2.slide_armed2 -> slide_armed2_2;
c28: data port doors_process2.cll -> cll2;
c29: data port doors_process1.flight_lock1 -> door1.lock1;
c30: data port doors_process1.pres_warn1 -> door1.pressure_warn1;
c31: data port doors_process1.slide_warn1 -> door1.slide_warn1;
c32: data port doors_process1.flight_lock2 -> door2.lock1;
c33: data port doors_process1.pres_warn2 -> door2.pressure_warn1;
c34: data port doors_process1.slide_warn2 -> door2.slide_warn1;
c35: data port door1.closed1 -> doors_process1.closed1_1;
c36: data port door1.closed2 -> doors_process1.closed2_1;
c37: data port door1.ll1 -> doors_process1.ll1_1;
c38: data port door1.ll2 -> doors_process1.ll2_1;
c39: data port door1.ll3 -> doors_process1.ll3_1;
c40: data port door1.handle -> doors_process1.handle_pos1;
c41: data port door2.closed1 -> doors_process1.closed1_2;
c42: data port door2.closed2 -> doors_process1.closed2_2;
c43: data port door2.ll1 -> doors_process1.ll1_2;
c44: data port door2.ll2 -> doors_process1.ll2_2;
c45: data port door2.ll3 -> doors_process1.ll3_2;
c46: data port door2.handle -> doors_process1.handle_pos2;
c47: data port doors_process2.flight_lock1 -> door1.lock2;
c48: data port doors_process2.pres_warn1 -> door1.pressure_warn2;
c49: data port doors_process2.slide_warn1 -> door1.slide_warn2;
c50: data port doors_process2.flight_lock2 -> door2.lock2;
c51: data port doors_process2.pres_warn2 -> door2.pressure_warn2;
c52: data port doors_process2.slide_warn2 -> door2.slide_warn2;
c53: data port door1.closed1 -> doors_process2.closed1_1;
c54: data port door1.closed2 -> doors_process2.closed2_1;
c55: data port door1.ll1 -> doors_process2.ll1_1;
```

```
    c56: data port door1.ll2 -> doors_process2.ll2_1;
    c57: data port door1.ll3 -> doors_process2.ll3_1;
    c58: data port door1.handle -> doors_process2.handle_pos1;
    c59: data port door2.closed1 -> doors_process2.closed1_2;
    c60: data port door2.closed2 -> doors_process2.closed2_2;
    c61: data port door2.ll1 -> doors_process2.ll1_2;
    c62: data port door2.ll2 -> doors_process2.ll2_2;
    c63: data port door2.ll3 -> doors_process2.ll3_2;
    c64: data port door2.handle -> doors_process2.handle_pos2;
    c65: data port door1.slide_armed -> doors_process1.slide_armed_in1;
    c66: data port door2.slide_armed -> doors_process1.slide_armed_in2;
    c67: data port door1.slide_armed -> doors_process2.slide_armed_in1;
    c68: data port door2.slide_armed -> doors_process2.slide_armed_in2;
    c69: bus access rdc1.door_iface -> door1.rdc;
    c70: bus access rdc1.door_iface -> door2.rdc;
    c71: data port doors_process1.cll1 -> cll1_1;
    c72: data port doors_process1.cll2 -> cll2_1;
    c73: data port doors_process2.cll1 -> cll1_2;
    c74: data port doors_process2.cll2 -> cll2_2;
    c75: bus access afdx -> rdc1.afdx_access;
    c76: bus access afdx -> rdc2.afdx_access;
    c77: bus access afdx -> cpiom1.afdx_access;
    c78: bus access afdx -> cpiom2.afdx_access;
  properties
    Actual_Processor_Binding =>  reference cpiom1 applies to doors_process1;
    Actual_Processor_Binding =>  reference cpiom2 applies to doors_process2;
    Actual_Connection_Binding =>  reference afdx applies to c29;
    Actual_Connection_Binding =>  reference afdx applies to c30;
    Actual_Connection_Binding =>  reference afdx applies to c31;
    Actual_Connection_Binding =>  reference afdx applies to c32;
    Actual_Connection_Binding =>  reference afdx applies to c33;
    Actual_Connection_Binding =>  reference afdx applies to c34;
    Actual_Connection_Binding =>  reference afdx applies to c35;
    Actual_Connection_Binding =>  reference afdx applies to c36;
    Actual_Connection_Binding =>  reference afdx applies to c37;
    Actual_Connection_Binding =>  reference afdx applies to c38;
    Actual_Connection_Binding =>  reference afdx applies to c39;
    Actual_Connection_Binding =>  reference afdx applies to c40;
    Actual_Connection_Binding =>  reference afdx applies to c41;
    Actual_Connection_Binding =>  reference afdx applies to c42;
    Actual_Connection_Binding =>  reference afdx applies to c43;
    Actual_Connection_Binding =>  reference afdx applies to c44;
    Actual_Connection_Binding =>  reference afdx applies to c45;
    Actual_Connection_Binding =>  reference afdx applies to c46;
    Actual_Connection_Binding =>  reference afdx applies to c47;
    Actual_Connection_Binding =>  reference afdx applies to c48;
    Actual_Connection_Binding =>  reference afdx applies to c49;
    Actual_Connection_Binding =>  reference afdx applies to c50;
    Actual_Connection_Binding =>  reference afdx applies to c51;
    Actual_Connection_Binding =>  reference afdx applies to c52;
    Actual_Connection_Binding =>  reference afdx applies to c53;
    Actual_Connection_Binding =>  reference afdx applies to c54;
    Actual_Connection_Binding =>  reference afdx applies to c55;
    Actual_Connection_Binding =>  reference afdx applies to c56;
    Actual_Connection_Binding =>  reference afdx applies to c57;
```

```
    Actual_Connection_Binding =>  reference afdx applies to c58;
    Actual_Connection_Binding =>  reference afdx applies to c59;
    Actual_Connection_Binding =>  reference afdx applies to c60;
    Actual_Connection_Binding =>  reference afdx applies to c61;
    Actual_Connection_Binding =>  reference afdx applies to c62;
    Actual_Connection_Binding =>  reference afdx applies to c63;
    Actual_Connection_Binding =>  reference afdx applies to c64;
    Actual_Connection_Binding =>  reference afdx applies to c65;
    Actual_Connection_Binding =>  reference afdx applies to c66;
    Actual_Connection_Binding =>  reference afdx applies to c67;
    Actual_Connection_Binding =>  reference afdx applies to c68;
end door_management.imp;

system Door
  features
    handle: out data port behavior::boolean;
    ll1: out data port behavior::boolean;
    ll2: out data port behavior::boolean;
    ll3: out data port behavior::boolean;
    closed1: out data port behavior::boolean;
    closed2: out data port behavior::boolean;
    slide_armed: out data port behavior::boolean;
    slide_warn1: in data port behavior::boolean;
    slide_warn2: in data port behavior::boolean;
    pressure_warn1: in data port behavior::boolean;
    pressure_warn2: in data port behavior::boolean;
    lock1: in data port behavior::boolean;
    lock2: in data port behavior::boolean;
    rdc: requires bus access AFDX.imp;
end Door;

system implementation Door.imp
  subcomponents
    handle1: device Handle;
    locklatched1: device LockLatched;
    locklatched2: device LockLatched;
    locklatched3: device LockLatched;
    closedsensor1: device ClosedSensor;
    closedsensor2: device ClosedSensor;
    pressurewarn: device PressureWarn;
    slidelight: device SlideLight;
    flightlock: device FlightLock;
    slidearmed: device SlideArmed;
  connections
    con1: data port handle1.handle -> handle;
    con2: data port closedsensor1.closed -> closed1;
    con3: data port closedsensor2.closed -> closed2;
    con4: data port locklatched1.ll -> ll1;
    con5: data port locklatched2.ll -> ll2;
    con6: data port locklatched3.ll -> ll3;
    con7: data port slide_warn1 -> slidelight.slide_warn1;
    con8: data port slide_warn2 -> slidelight.slide_warn2;
    con9: data port pressure_warn1 -> pressurewarn.pressure_warn1;
    con10: data port pressure_warn2 -> pressurewarn.pressure_warn2;
    con11: data port lock1 -> flightlock.lock1;
```

```
    con12: data port lock2 -> flightlock.lock2;
    con13: data port slidearmed.slide_armed -> slide_armed;
end Door.imp;

device FlightLock
  features
    lock1: in data port behavior::boolean;
    lock2: in data port behavior::boolean;
end FlightLock;

device SlideLight
  features
    slide_warn1: in data port behavior::boolean;
    slide_warn2: in data port behavior::boolean;
end SlideLight;

device Handle
  features
    handle: out data port behavior::boolean;
end Handle;

device LockLatched
  features
    ll: out data port behavior::boolean;
end LockLatched;

device ClosedSensor
  features
    closed: out data port behavior::boolean;
end ClosedSensor;

device PressureWarn
  features
    pressure_warn1: in data port behavior::boolean;
    pressure_warn2: in data port behavior::boolean;
end PressureWarn;

device SlideArmed
  features
    slide_armed: out data port behavior::boolean;
end SlideArmed;

device DPS
  features
    dps: out data port behavior::boolean;
end DPS;

device OCU
  features
    cll1: in data port behavior::boolean;
    cll2: in data port behavior::boolean;
end OCU;

processor CPIOM
  features
```

```
    afdx_access: requires bus access AFDX.imp;
end CPIOM;

processor implementation CPIOM.imp
end CPIOM.imp;

bus AFDX
end AFDX;

bus implementation AFDX.imp
end AFDX.imp;

device RDC
  features
    afdx_access: requires bus access AFDX.imp;
    door_iface: provides bus access AFDX.imp;
end RDC;

process doors_process
  features
    flight_lock1: out data port behavior::boolean;
    flight_lock2: out data port behavior::boolean;
    pres_warn1: out data port behavior::boolean;
    pres_warn2: out data port behavior::boolean;
    slide_warn1: out data port behavior::boolean;
    slide_warn2: out data port behavior::boolean;
    closed1_1: in data port behavior::boolean;
    closed2_1: in data port behavior::boolean;
    closed1_2: in data port behavior::boolean;
    closed2_2: in data port behavior::boolean;
    ll1_1: in data port behavior::boolean;
    ll2_1: in data port behavior::boolean;
    ll3_1: in data port behavior::boolean;
    ll1_2: in data port behavior::boolean;
    ll2_2: in data port behavior::boolean;
    ll3_2: in data port behavior::boolean;
    handle_pos1: in data port behavior::boolean;
    handle_pos2: in data port behavior::boolean;
    slide_armed_in1: in data port behavior::boolean;
    slide_armed_in2: in data port behavior::boolean;
    engine_running: in data port behavior::boolean;
    in_flight: in data port behavior::boolean;
    on_ground: in data port behavior::boolean;
    airspeed1: in data port behavior::float;
    airspeed2: in data port behavior::float;
    airspeed3: in data port behavior::float;
    emerg_evac: in data port behavior::boolean;
    dps1: in data port behavior::boolean;
    dps2: in data port behavior::boolean;
    slide_armed1: out data port behavior::boolean;
    slide_armed2: out data port behavior::boolean;
    cll: out data port behavior::boolean;
    cll1: out data port behavior::boolean;
    cll2: out data port behavior::boolean;
end doors_process;
```

```
process implementation doors_process.imp
  subcomponents
    door_handler1: thread door_handler.imp;
    door_handler2: thread door_handler.imp;
    door_mix: thread doors_mix.imp;
  connections
    conn1: data port engine_running -> door_handler1.engine_running;
    conn2: data port on_ground -> door_handler1.on_ground;
    conn3: data port in_flight -> door_handler1.in_flight;
    conn4: data port airspeed1 -> door_handler1.airspeed1;
    conn5: data port airspeed2 -> door_handler1.airspeed2;
    conn6: data port airspeed3 -> door_handler1.airspeed3;
    conn7: data port emerg_evac -> door_handler1.emerg_evac;
    conn8: data port dps1 -> door_handler1.dps1;
    conn9: data port dps2 -> door_handler1.dps2;
    conn10: data port closed1_1 -> door_handler1.closed1;
    conn11: data port closed2_1 -> door_handler1.closed2;
    conn12: data port ll1_1 -> door_handler1.ll1;
    conn13: data port ll2_1 -> door_handler1.ll2;
    conn14: data port ll3_1 -> door_handler1.ll3;
    conn15: data port handle_pos1 -> door_handler1.handle_pos;
    conn16: data port door_handler1.slide_armed -> slide_armed1;
    conn17: data port door_handler1.slide_warn -> slide_warn1;
    conn18: data port door_handler1.flight_lock -> flight_lock1;
    conn19: data port door_handler1.pres_warn -> pres_warn1;
    conn20: data port door_handler1.cll -> door_mix.cll1;
    conn21: data port engine_running -> door_handler2.engine_running;
    conn22: data port on_ground -> door_handler2.on_ground;
    conn23: data port in_flight -> door_handler2.in_flight;
    conn24: data port airspeed1 -> door_handler2.airspeed1;
    conn25: data port airspeed2 -> door_handler2.airspeed2;
    conn26: data port airspeed3 -> door_handler2.airspeed3;
    conn27: data port emerg_evac -> door_handler2.emerg_evac;
    conn28: data port dps1 -> door_handler2.dps1;
    conn29: data port dps2 -> door_handler2.dps2;
    conn30: data port closed1_2 -> door_handler2.closed1;
    conn31: data port closed2_2 -> door_handler2.closed2;
    conn32: data port ll1_2 -> door_handler2.ll1;
    conn33: data port ll2_2 -> door_handler2.ll2;
    conn34: data port ll3_2 -> door_handler2.ll3;
    conn35: data port handle_pos2 -> door_handler2.handle_pos;
    conn36: data port door_handler2.slide_armed -> slide_armed2;
    conn37: data port door_handler2.slide_warn -> slide_warn2;
    conn38: data port door_handler2.flight_lock -> flight_lock2;
    conn39: data port door_handler2.pres_warn -> pres_warn2;
    conn40: data port door_handler2.cll -> door_mix.cll2;
    conn41: data port door_mix.cll -> cll;
    conn42: data port slide_armed_in1 -> door_handler1.slide_armed_in;
    conn43: data port slide_armed_in2 -> door_handler2.slide_armed_in;
    conn44: data port door_handler1.cll -> cll1;
    conn45: data port door_handler2.cll -> cll2;
end doors_process.imp;

thread door_handler
```

```
  features
    handle_pos: in data port behavior::boolean;
    ll1: in data port behavior::boolean;
    ll2: in data port behavior::boolean;
    ll3: in data port behavior::boolean;
    closed1: in data port behavior::boolean;
    closed2: in data port behavior::boolean;
    in_flight: in data port behavior::boolean;
    on_ground: in data port behavior::boolean;
    engine_running: in data port behavior::boolean;
    airspeed1: in data port behavior::float;
    airspeed2: in data port behavior::float;
    airspeed3: in data port behavior::float;
    emerg_evac: in data port behavior::boolean;
    dps1: in data port behavior::boolean;
    dps2: in data port behavior::boolean;
    slide_armed_in: in data port behavior::boolean;
    slide_armed: out data port behavior::boolean;
    cll: out data port behavior::boolean;
    slide_warn: out data port behavior::boolean;
    flight_lock: out data port behavior::boolean;
    pres_warn: out data port behavior::boolean;
end door_handler;

thread implementation door_handler.imp
  properties
    Dispatch_Protocol => Periodic;
    Period => 50 Ms;
end door_handler.imp;

thread doors_mix
  features
    cll1: in data port behavior::boolean;
    cll2: in data port behavior::boolean;
    cll: out data port behavior::boolean;
end doors_mix;

thread implementation doors_mix.imp
  properties
    Dispatch_Protocol => Periodic;
    Period => 50 Ms;
end doors_mix.imp;
```

A.2 Signal specifications

Model *door_handler* thread in Signal

```
process ThreadT_door_handler =
(? event dispatch, deadline,scomplete, start;
   boolean handle_pos,ll1,ll2,ll3,closed1,closed2,
           in_flight,on_ground,engine_running;
   dreal airspeed1,airspeed2,airspeed3;
   boolean emerg_evac,dps1,dps2,slide_armed_in;
 ! boolean slide_armed,cll,slide_warn,flight_lock,pres_warn;)
```

```
(|(l_slide_armed, l_cll, l_slide_warn, l_flight_lock, l_pres_warn)
    := SPT_door_handler{}(dispatch, deadline, scomplete, start, handle_pos,
    ll1, ll2, ll3, closed1,closed2, in_flight, on_ground, engine_running,
    airspeed1, airspeed2, airspeed3,emerg_evac, dps1, dps2, slide_armed_in)
 | slide_armed := at{}(l_slide_armed,scomplete)
 | cll := at{}(l_cll,scomplete)
 | slide_warn := at{}(l_slide_warn,scomplete)
 | flight_lock := at{}(l_flight_lock,scomplete)
 | pres_warn := at{}(l_pres_warn,scomplete)
 |)
 where  ...
 end;

process SPT_door_handler =
( ? event dispatch, deadline,scomplete,start;
    boolean handle_pos,ll1,ll2, ll3,closed1,closed2,
            in_flight, on_ground,engine_running;
    dreal airspeed1, airspeed2,airspeed3;
    boolean emerg_evac,dps1,dps2,slide_armed_in;
  ! boolean slide_armed,cll,slide_warn,flight_lock,pres_warn;)
(| l_handle_pos := at{}(handle_pos,start)
 | l_ll1 := at{}(ll1,start)
 | l_ll2 := at{}(ll2,start)
 | l_ll3 := at{}(ll3,start)
 | l_closed1 := at{}(closed1,start)
 | l_closed2 := at{}(closed2,start)
 | l_in_flight := at{}(in_flight,start)
 | l_on_ground := at{}(on_ground,start)
 | l_engine_running := at{}(engine_running,start)
 | l_airspeed1 := at{}(airspeed1,start)
 | l_airspeed) := at{}(airspeed2,start)
 | l_airspeed3 := at{}(airspeed3,start)
 | l_emerg_evac := at{}(emerg_evac,start)
 | l_dps1 := at{}(dps1,start)
 | l_dps2 := at{}(dps2,start)
 | l_slide_armed_in := at{}(slide_armed_in,start)
 | (slide_armed, cll, slide_warn, flight_lock, pres_warn) := SP_door_handler{}
     (l_handle_pos, l_ll1, l_ll2, l_ll3, l_closed1, l_closed2, l_in_flight,
      l_on_ground, l_engine_running, l_airspeed1, l_airspeed2, l_airspeed3,
      l_emerg_evac, l_dps1, l_dps2, l_slide_armed_in)
 |)
where  ...
end;

process SP_door_handler =
( ? boolean handle_pos, ll1,ll2, ll3, closed1, closed2,
            in_flight, on_ground, engine_running;
    dreal airspeed1, airspeed2, airspeed3;
    boolean emerg_evac, dps1, dps2, slide_armed_in;
  ! boolean slide_armed, cll, slide_warn, flight_lock, pres_warn; )
(| ...|) ;
```

Model *doors_mix* thread in Signal

```
process ThreadT_doors_mix =
```

```
( ? event dispatch,deadline,scomplete,start;
    boolean cll1,cll2;
  ! boolean cll;)
(| l_cll := SPT_doors_mix{}(dispatch,deadline, scomplete,start, cll1, cll2)
 | cll := at{}(l_cll,scomplete)
 |);

process SPT_doors_mix =
( ? event dispatch,deadline,scomplete,start;
    boolean cll1,boolean cll2;
    ! boolean cll;)
 (| l_cll1 := at{}(cll1,start)
  | l_cll2 := at{}(cll2,start)
  | cll:= SP_doors_mix{}(l_cll1,l_cll2)
  |)
where  ...
end;

process SP_doors_mix =
( ? boolean cll1, boolean cll2;
  ! boolean cll;)
(|cll := doors_process_nb1_ii(cll1,cll2)
 |)  ;
```

A simple partition *scheduler* model

```
process partition_scheduler =
  (? event tick, ptick;
   ! event dispatch_1, dispatch_2, dispatch_3, start_handler_1_1,
           start_handler_1_2,start_mix_1, complete_handler_1_1,
           complete_handler_1_2, complete_mix_1, deadline_handler_1_1,
           deadline_handler_1_2, deadline_mix_1;)
  (| cnt := 1 when ptick when tick default pre_cnt + 1
   | pre_cnt := cnt $ 1 init 0
   | tick ^= cnt ^= pre_cnt
   | dispatch_1           := when ( cnt = 2 )
   | start_handler_1_1    := when ( cnt = 3 )
   | complete_handler_1_1 := when ( cnt = 4 )
   | deadline_handler_1_1 := when ( cnt = 5 )
   | dispatch_2           := when ( cnt = 6 )
   | start_handler_1_2    := when ( cnt = 7 )
   | complete_handler_1_2 := when ( cnt = 8 )
   | deadline_handler_1_2 := when ( cnt = 9 )
   | dispatch_3           := when ( cnt = 10 )
   | start_mix_1          := when ( cnt = 11 )
   | complete_mix_1       := when ( cnt = 12 )
   | deadline_mix_1       := when ( cnt = 13 )
   |)
   where integer cnt, pre_cnt;
end;
```

Appendix B

FGS example

B.1 AADL specifications

```
system DFGSsys
  features
    transfer: in event data port behavior::boolean;
    L_inde_mode: in event data port behavior::boolean;
    R_inde_mode: in event data port behavior::boolean;
    Lactive: out event data port behavior::boolean;
    Ractive: out event data port behavior::boolean;
end DFGSsys;

system implementation DFGSsys.impl
  subcomponents
    cpu1: processor CPU.impl;
    LFGS: process FGS.impl;
    cpu2: processor CPU.impl;
    RFGS: process FGS.impl;
    bus1: bus net.impl;
    bus2: bus net.impl;
  connections
    c1: event data port transfer -> LFGS.transfer;
    c2: event data port transfer -> RFGS.transfer;
    c3: event data port L_inde_mode -> LFGS.inde_mode;
    c4: event data port R_inde_mode -> RFGS.inde_mode;
    c5: event data port LFGS.pactive -> Lactive;
    c6: event data port RFGS.pactive -> Ractive;
    c7: event data port LFGS.pilot_flying -> RFGS.other_pilot_flying;
    c8: event data port LFGS.independent -> RFGS.other_independent;
    c9: event data port RFGS.pilot_flying -> LFGS.other_pilot_flying;
    c10:event data port RFGS.independent -> LFGS.other_independent;
  properties
    Actual_Processor_Binding => reference cpu1 applies to LFGS;
    Actual_Processor_Binding => reference cpu2 applies to RFGS;
    Actual_Connection_Binding =>  reference bus1 applies to c7;
    Actual_Connection_Binding =>  reference bus1 applies to c8;
    Actual_Connection_Binding =>  reference bus2 applies to c9;
    Actual_Connection_Binding =>  reference bus2 applies to c10;
end DFGSsys.impl;
```

```
processor CPU
end CPU;

processor implementation CPU.impl
end CPU.impl;

bus net
end net;

bus implementation net.impl
end net.impl;

process FGS
  features
    transfer: in event data port behavior::boolean;
    inde_mode: in event data port behavior::boolean;
    other_pilot_flying: in event data port behavior::boolean;
    other_independent: in event data port behavior::boolean;
    pilot_flying: out event data port behavior::boolean;
    independent: out event data port behavior::boolean;
    pactive: out event data port behavior::boolean;
end FGS;

process implementation FGS.impl
  subcomponents
    tFGS: thread FGS_thread.impl;
  connections
    c11: event data port transfer -> tFGS.ts;
    c12: event data port inde_mode -> tFGS.inde_mode;
    c13: event data port other_pilot_flying -> tFGS.other_pf;
    c14: event data port other_independent -> tFGS.other_inde;
    c15: event data port pilot_flying -> tFGS.pf;
    c16: event data port independent -> tFGS.tindependent;
    c17: event data port pactive -> tFGS.tactive;
end FGS.impl;

thread FGS_thread
  features
    ts: in event data port behavior::boolean;
    inde_mode: in event data port behavior::boolean;
    other_pf: in event data port behavior::boolean;
    other_inde: in event data port behavior::boolean;
    pf: out event data port behavior::boolean;
    tindependent: out event data port behavior::boolean;
    tactive: out event data port behavior::boolean;
  properties
    Period => 100 Ms;
end FGS_thread;

thread implementation FGS_thread.impl
  annex behavior_specification {**
    states
      primary: state;
      backup: state;
    transitions
```

```
      primary -[other_pf?(true) ]-> backup { pf!(false);};
      backup -[ts?]-> primary { pf!(true); tactive!(true);};
    **};
end FGS_thread.impl;
```

B.2 Signal specifications

```
process DFGS =
(? boolean Transfer, R_inde_mode, L_inde_mode;
 ! boolean Ractive, Lactive, RPF, LPF, LIndependent, RIndependent;)
(| (LIndependent,LPF,Lactive) := PART_cpu1{true,false}
                                 (Transfer,ORPF,ORI,L_inde_mode)
 | (RIndependent,RPF,Ractive) := PART_cpu2{false,true}
                                 (Transfer,OLPF,OLI,R_inde_mode)
 | OLI  := bus1{false}(LIndependent)
 | OLPF := bus1{true}(LPF)
 | ORI  := bus2{true}(RIndependent)
 | ORPF := bus2{false}(RPF)
 |)
where
use SIGALI;
boolean ZTS,OLI,ORI,OLPF,ORPF;

process PART_cpu1 = {boolean Ini_PF;boolean Ini_OPF;}
    (? boolean Transfer,Other_PF,Other_Independent,Inde_mode;
     ! boolean Independent,PF,Active;)
(| ZPF := PF$ init Ini_PF
 | ZOPF := Other_PF$ init Ini_OPF
 | FC  := Transfer and (not ZPF)
 | PF  := (true when FC)
          default (false when (Other_PF and (not ZOPF)) when not FC)
          default ZPF
 | Independent   := Inde_mode
 | Active   := PF or (Other_Independent and Inde_mode)
|)
where
   boolean ZOPF, ZPF, FC;
end;

process PART_cpu2 = {boolean Ini_PF;boolean Ini_OPF;}
    (? boolean Transfer,Other_PF,Other_Independent,Inde_mode;
     ! boolean Independent,PF,Active;)
(| ZPF := PF$ init Ini_PF
 | ZOPF := Other_PF$ init Ini_OPF
 | FC  := Transfer and (not ZPF)
 | PF  := (true when FC)
          default (false when (Other_PF and (not ZOPF)) when not FC)
          default ZPF
 | Independent   := Inde_mode
 | Active   := PF or (Other_Independent and Inde_mode)
 |)
where
   boolean ZOPF, ZPF, FC;
end;
```

```
process bus1 = {boolean IniS;}
  (? boolean S; ! boolean DS;)
(| DS := S$1 init IniS |)

process bus2 = {boolean IniS;}
  (? boolean S; ! boolean DS;)
(| DS := S$1 init IniS |)
end;
end;

process ThreadT_tFGS = {boolean Ini_PF;boolean Ini_OPF;}
           ( ? boolean Transfer,Other_PF,Other_Independent,Inde_mode;
             ! boolean Independent,PF,Active;)
           (| ZPF := PF$ init Ini_PF
 | ZOPF := Other_PF$ init Ini_OPF
 | FC  := Transfer and (not ZPF)
 | PF  := (true when FC)
          default (false when (Other_PF and (not ZOPF)) when not FC)
          default ZPF
 | Independent   := Inde_mode
 | Active   := PF or (Other_Independent and Inde_mode)
|)
where
   boolean ZOPF, ZPF, FC;
end;
```

Modélisation compositionnelle d'architectures globalement asynchrones - localement synchrones (GALS) dans un modèle de calcul polychrone

AADL est dédié à la conception de haut niveau et l'évaluation de systèmes embarqués. Il permet de décrire la structure d'un système et ses aspects fonctionnels par une approche à base de composants. Des processus localement synchrones sont alloués sur une architecture distribuée et communiquent de manière globalement asynchrone (système GALS). Une spécificité du modèle polychrone est qu'il permet de spécifier un système dont les composants peuvent avoir leur propre horloge d'activation : il est bien adapté à une méthodologie de conception GALS. Dans ce cadre, l'atelier Polychrony fournit des modèles et des méthodes pour la modélisation, la transformation et la validation de systèmes embarqués. Cette thèse propose une méthodologie pour la modélisation et la validation de systèmes embarqués spécifiés en AADL via le langage synchrone multi-horloge Signal. Cette méthodologie comprend la modélisation de niveau système en AADL, des transformations automatiques du modèle AADL vers le modèle polychrone, la distribution de code, la vérification formelle et la simulation du modèle polychrone. Notre transformation prend en compte l'architecture du système, décrite dans un cadre IMA, et les aspects fonctionnels, les composants logiciels pouvant être mis en œuvre en Signal. Les composants AADL sont modélisés dans le modèle polychrone en utilisant une bibliothèque de services ARINC. L'annexe comportementale d'AADL est interprétée dans ce modèle via SSA. La génération de code distribué est obtenue avec Polychrony. La vérification formelle et la simulation sont effectuées sur deux études de cas qui illustrent notre méthodologie pour la conception fiable des applications AADL.

Mots clefs: AADL, Signal, GALS, IMA, ARINC, modèle polychrone, modèle synchrone, simulation, vérification.

Compositional modeling of globally asynchronous locally synchronous (GALS) architectures in a polychronous model of computation

AADL is dedicated to high-level design and evaluation of embedded systems. It allows describing both system structure and functional aspects via a component-based approach, e.g., GALS system. The polychronous model of computation stands out from other synchronous specification models by the fact that it allows one specifying a system whose components can have their own activation clocks. It is well adapted to support a GALS design methodology. Its framework Polychrony provides models and methods for modeling, transformation and validation of embedded systems. This thesis proposes a methodology for modeling and validation of embedded systems specified in AADL via the multi-clock synchronous programming language Signal. This methodology includes system-level modeling via AADL, automatic transformations from the high-level AADL model to the polychronous model, code distribution, formal verification and simulation of the obtained polychronous model. Our transformation takes into account both the system architecture, particularly described in Integrated Modular Avionics (IMA), and functional aspects, e.g., software components implemented in the polychronous language Signal. AADL components are modeled into the polychronous MoC within the IMA architecture using a library of ARINC services. The AADL Behavior Annex is interpreted into the multi-clocked MoC using SSA as an intermediate formalism. Distributed code generation is obtained with Polychrony. Formal verification and simulation are carried out on two case studies that illustrate our methodology for the reliable design of AADL applications.

Keywords: AADL, Signal, GALS, IMA, ARINC, polychronous model, synchronous model, simulation, verification.